Network Driver Design Guide

Article • 09/27/2024

This Network Driver Design Guide describes how to design and create network device drivers for Windows operating systems beginning with Windows Vista.

This guide includes the following sections:

Introduction to Network Drivers Topics

Article • 09/27/2024

This section discusses introductory concepts for kernel-mode network drivers and includes the following topics:

- Roadmap for Developing NDIS Drivers
- Using the Network Driver Design Guide
- Network Architecture for Kernel-Mode Drivers
- Network Driver Programming Considerations
- Driver Stack Management
- NET_BUFFER Architecture
- Introduction to NDIS PDPI

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Roadmap for Developing NDIS Drivers

Article • 03/14/2023

To create a Network Driver Interface Specification (NDIS) driver package, follow these steps:

- Step 1: Learn about Windows architecture and drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- Step 2: Learn about NDIS.

  For general information about NDIS and NDIS drivers, see the following topics:

  Windows Network Architecture and the OSI Model

  Network Driver Programming Considerations

  Driver Stack Management

  NET_BUFFER Architecture

- Step 3: Determine additional Windows driver design decisions.

  For more information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- Step 4: Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user-mode application. For more information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Lab Kit (HLK) testing, see Building, Debugging, and Testing Drivers. For more information about building, testing, verifying, and debugging tools, see Driver Development Tools.

- Step 5: Select the type of NDIS driver you will implement.

  For more information about types of NDIS drivers, see Using the Network Driver Design Guide.

  Follow the roadmaps for the type of driver.

Roadmap for Developing NDIS Miniport Drivers

Roadmap for Developing NDIS Protocol Drivers

Roadmap for Developing NDIS Filter Drivers

Roadmap for Developing NDIS Intermediate Drivers

Roadmap to Develop Mobile Broadband Miniport Drivers

Roadmap for Developing Windows Filtering Platform Callout Drivers

- Step 6: Review the Network driver samples in the Windows driver samples repository on GitHub.

- Step 7: Develop (or port), build, test, and debug your NDIS driver.

  See the porting guides if you are porting an existing driver:

  - Porting NDIS 6.x Drivers to NDIS 6.40

  - Porting NDIS 6.x Drivers to NDIS 6.30

  - Porting NDIS 6.x Drivers to NDIS 6.20

  - Porting NDIS 5.x Drivers to NDIS 6.0

    For more information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process will help ensure that you build a driver that works.

- Step 8: Create a driver package for your driver.

  For more information about how to install drivers, see Providing a Driver Package. For more information about how to install an NDIS driver, see Components and Files Used for Network Component Installation and Notify Objects for Network Components.

- Step 9: Sign and distribute your driver.

  The final step is to sign and distribute the driver. If your driver meets the quality standards that are defined for the Windows Hardware Lab Kit (HLK), you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual driver.

Navigating the Network Driver Design Guide

Article • 12/15/2021

Microsoft Windows-based operating systems support several types of kernel-mode network drivers. The Network section of the Windows Driver Kit (WDK) documentation describes how to write these network drivers. This topic briefly describes the supported types of network drivers and explains which sections of the Network section you should read before writing each type of network driver.

This network driver design guide documents the following Network Driver Interface Specification (NDIS) interfaces:

- NDIS 6.40, which is supported on Windows 8.1, Windows Server 2012 R2, and later versions of Windows. NDIS 6.30 includes support for Network Direct Kernel Provider Interface (NDKPI) 1.12.

  For more information about NDIS 6.30, see Introduction to NDIS 6.40.

- NDIS 6.30, which is supported on Windows 8, Windows Server 2012, and later versions of Windows. NDIS 6.30 includes support for single root/I/O virtualization (SR-IOV), Hyper-V extensible switch, Network Direct Kernel Provider Interface (NDKPI) 1.1, and other services.

  For more information about NDIS 6.30, see Introduction to NDIS 6.30.

- NDIS 6.20, which is supported on Windows 7, Windows Server 2008 R2, and later versions of Windows. NDIS 6.20 includes support for Virtual Machine Queue (VMQ), receive side throttle, and other services.

  For more information about NDIS 6.20, see Introduction to NDIS 6.20.

- NDIS 6.1, which is supported on Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of Windows. NDIS 6.1 includes support for header-data split, direct OID requests, and other services.

  For more information about NDIS 6.1, see Introduction to NDIS 6.1.

- NDIS 6.0, which is supported on Windows Vista and later versions of Windows. NDIS 6.0 includes support for filter drivers and many additional services that were not provided by earlier NDIS versions. NDIS 6.0 includes major updates to driver initialization and network data management including required support for driver reconfiguration at runtime and the **NET_BUFFER** architecture for handling network packet data. For more information about supporting runtime reconfiguration, see Driver Stack Management. For more information about how to handle network packet data in NDIS 6.0 see NET_BUFFER Architecture.

For more information about NDIS 6.0, see Introduction to NDIS 6.0.

Windows Vista and later operating system versions support the following types of kernel-mode NDIS-based network drivers:

Miniport Drivers

A *miniport driver* manages miniport adapters and provides an interface to the adapters for higher-level drivers. A *miniport adapter* is a conceptual entity that can represent either a physical device or a virtual device. For example, a miniport adapter can represent a network interface card (NIC) or a virtual device that is associated with an intermediate driver.

There are many variations of miniport drivers, such as a *connection-oriented miniport call manager (MCM)*, a *Windows Driver Model (WDM) miniport driver,* and the upper edge of an intermediate driver.

Protocol Drivers

A *protocol driver* provides high-level services in a driver stack. A protocol driver binds to underlying miniport adapters. An *upper-level protocol driver* implements an interface, possibly an application-specific interface, at its upper edge to provide services to users of the network. At its lower edge, a protocol driver provides a protocol interface to pass network data to and receive incoming data from the next-lower driver.

There are many variations of protocol drivers, such as a *connection-oriented call manager (MCM), a connection-oriented client,* and the lower edge of an intermediate driver.

Filter Drivers

A *filter driver* filters information on the interface between protocol drivers and miniport drivers. *Filter modules* are attached in the binding between the protocol driver and the miniport adapter and are generally transparent to the other drivers. Filter drivers can implement *modifying or monitoring filters*. For example, a filter driver can enhance the services that the underlying miniport adapter provides or simply collect statistics.

Intermediate Drivers

An *intermediate driver* interfaces between upper-level protocol drivers and miniport drivers. Intermediate drivers provide a miniport driver interface at their upper-edge to bind to overlying protocol drivers. Intermediate drivers provide a protocol driver interface at their lower edge to bind to underlying miniport adapters. Intermediate drivers are typically used to implement *n* to *m* multiplexer services. For example, an intermediate driver can implement load balance and failover solutions.

Intermediate drivers can also manage hardware when they are configured as a *miniport-intermediate driver*.

For more information about the Windows network architecture and programming considerations, see Network Architecture for Kernel-Mode Drivers and Network Driver Programming Considerations.

For more information about network INF files, which are used to install network components, see Installing Network Components. If your network driver requires a notify object--for example, to control bindings--also see Notify Objects for Network Components.

The following additional driver models are available to use particular hardware technologies and architectures.

Technology	Description

Technology	Description
Scalable Networking	Networking technologies that support the offload of tasks to a network adapter, such as the following: - Header-Data Split, a service that splits the header and the data in received Ethernet frames into separate buffers. - Receive Side Scaling, a network driver technology that improves network performance on multiprocessor systems. - TCP Chimney Offload, an offload of the data-transfer part of the TCP protocol processing to a network adapter that has the appropriate capabilities. - TCP/IP Offload, an offload of tasks or connections to a network adapter that has the appropriate capabilities. - Network Direct Kernel Provider Interface (NDKPI), which enables kernel-mode Windows components, such as SMB server and client, to use remote direct memory access (RDMA) functionality that is provided by independent hardware vendors (IHVs). - Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload, which makes it possible to use Generic Routing Encapsulation (GRE)-encapsulated packets with: ○ Large Send Offload (LSO) ○ Virtual Machine Queue (VMQ) ○ Transmit (Tx) checksum offload ○ Receive (Rx) checksum offload
Virtualized Networking	Networking technologies that support Hyper-V virtualization environments, such as the following: - Single Root I/O Virtualization (SR-IOV) - Virtual Machine Queue (VMQ) - Hyper-V Extensible Switch
Wireless Networking	Networking capabilities that include Native 802.11 Wireless LAN.

Technology	Description
Network Module Registrar	A system facility that allows a driver to attach network modules to one another.
Winsock Kernel	A kernel-mode Network Programming Interface (NPI).
IP Helper	A set of utility functions that enable drivers to retrieve and modify information about the network configuration of the local computer.
Windows Filtering Platform Callout Drivers	A kernel-mode interface that enables deep inspection, packet modification, stream modification, and logging of network data.
System Area Networks	A type of network connection that uses Windows Sockets Direct to support a high-performance, connection-oriented network.
Remote NDIS (RNDIS)	A class specification that defines a system-provided, bus-independent message set over a USB bus.

Learning About Miniport Drivers

Article • 12/15/2021

There are several types of miniport drivers. The following list describes which sections of the WDK documentation you should read, depending on the type of miniport driver that you are writing:

Connectionless miniport drivers
If you are writing a miniport driver that controls a network interface card (NIC) for connectionless network media (such as Ethernet), read:

- Introduction to NDIS Miniport Drivers
- NDIS Miniport Drivers

Connection-oriented miniport drivers
If you are writing a miniport driver for connection-oriented network media (such as ISDN), read:

- All of the sections that are listed earlier in this topic under "Connectionless miniport drivers"
- Connection-Oriented NDIS

WAN miniport drivers
If you are writing a miniport driver that controls a wide area network (WAN) NIC, read:

- All of the sections that are listed earlier in this topic under "Connectionless miniport drivers"
- WAN Miniport Drivers

Miniports with a WDM lower interface
If you are writing a miniport driver that interfaces to other kernel drivers and has a Microsoft Windows Driver Model (WDM) lower interface, read:

- All of the sections that are listed earlier in this topic under "Connectionless miniport drivers"
- Miniport Drivers with a WDM Lower Interface

IrDA miniport drivers
If you are writing a miniport driver that controls an IrDA adapter, read:

- All of the sections that are listed earlier in this topic under "Connectionless miniport drivers"

Miniport drivers that support scalable networking

To learn about miniport drivers that support scalable networking, read:

- Scalable Networking

Miniport drivers that support offloading TCP/IP to hardware

To learn about miniport drivers that offload TCP/IP to hardware, read:

- TCP/IP Offload

Learning About Protocol Drivers

Article • 12/15/2021

You can write a protocol driver that has either a connectionless or a connection-oriented lower edge. In addition, your protocol driver can provide Winsock support. The following list describes which sections of the WDK documentation you should read, depending on the type of protocol driver that you are writing:

Protocol drivers that have a connectionless lower edge
If you are writing a protocol driver whose lower edge provides an interface to connectionless miniport drivers, read:

- NDIS Protocol Drivers

Protocol drivers that are connection-oriented clients or that are connection-oriented providers of call manager services
If you are writing a connection-oriented client, which provides an interface to connection-oriented miniport drivers, or if you are writing a connection-oriented call manager, read:

- NDIS Protocol Drivers
- Connection-Oriented NDIS

Protocol drivers that have Winsock support
If you are writing a protocol that provides Winsock support, read:

- NDIS Protocol Drivers
- Transport Helper DLLs for Windows Sockets

Learning About Filter Drivers

Article • 12/15/2021

You can write a filter driver that has a connectionless interface. The following list describes which sections of the WDK documentation you should read, depending on the type of filter driver that you are writing:

Filter drivers
If you are writing a filter driver whose lower edge provides an interface to connectionless miniport drivers, read:

- NDIS Filter Drivers

Learning About Intermediate Drivers

Article • 12/15/2021

You can write an intermediate driver that has either a connectionless or a connection-oriented lower edge. The following list describes which sections of the WDK documentation you should read, depending on the type of intermediate driver that you are writing:

Intermediate drivers that have a connectionless lower edge
If you are writing an intermediate driver whose lower edge provides an interface to connectionless miniport drivers, read:

- NDIS Intermediate Drivers

Intermediate drivers that have a connection-oriented lower edge
If you are writing an intermediate driver whose lower edge provides an interface to connection-oriented miniport drivers, read:

- NDIS Intermediate Drivers
- Connection-Oriented NDIS

Windows network architecture and the OSI model

Article • 12/16/2023

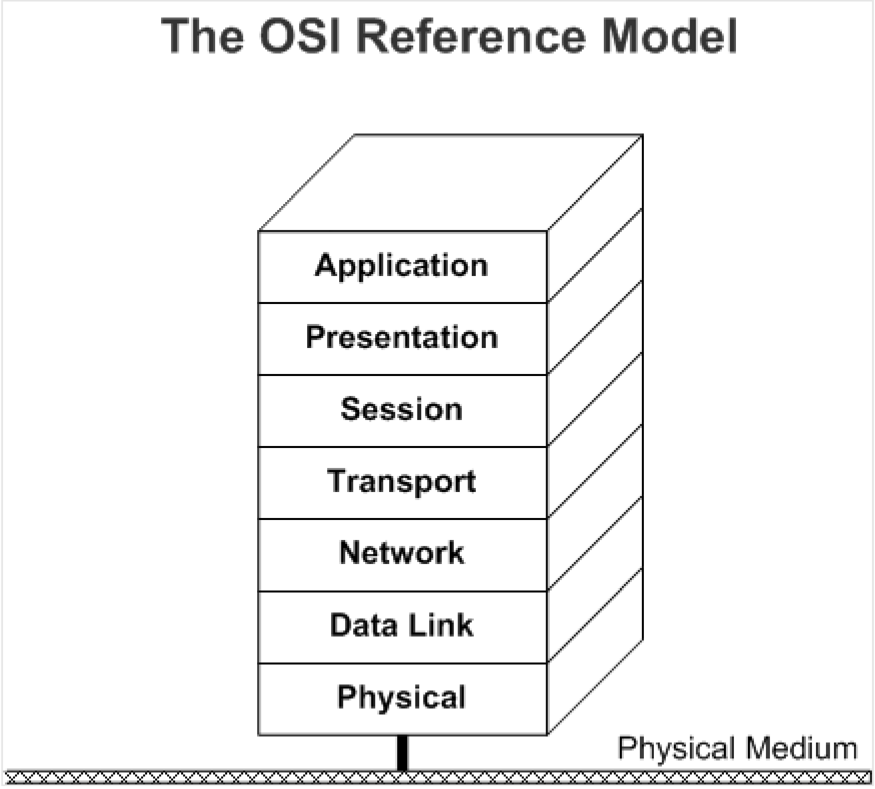
This article explores the Windows network architecture and how Windows network drivers implement the *bottom four* layers of the OSI model.

For general information on all seven layers of the model, see the OSI model.

The Microsoft Windows operating systems use a network architecture that is based on the seven-layer networking model developed by the International Organization for Standardization (ISO) in 1978.

The ISO Open Systems Interconnection (OSI) Reference model describes networking as "a series of protocol layers with a specific set of functions allocated to each layer. Each layer offers specific services to higher layers while shielding these layers from the details of how the services are implemented. A well-defined interface between each pair of adjacent layers defines the services offered by the lower layer to the higher one and how those services are accessed."

The following diagram illustrates the OSI model.



Windows network drivers implement the bottom four layers of the OSI model.

Physical layer

The physical layer is the lowest layer of the OSI model. This layer manages the reception and transmission of the unstructured raw bit stream over a physical medium. It describes the electrical/optical, mechanical, and functional interfaces to the physical medium. The physical layer carries the signals for all of the higher layers.

In Windows, the network interface card (NIC) implements the physical layer, its transceiver, and the medium to which the NIC is attached.

Data link layer

The data link layer sends frames between physical addresses and is responsible for error detection and recovery occurring in the physical layer.

The data link layer is further divided by the Institute of Electrical and Electronics Engineers (IEEE) into two sublayers: media access control (MAC) and logical link control (LLC).

MAC

The MAC sublayer manages access to the physical layer, checks frame errors, and manages address recognition of received frames.

In the Windows network architecture, the MAC sublayer is implemented in the NIC. The NIC is controlled by a software device driver called the miniport driver. Windows supports several variations of miniport drivers including WDM miniport drivers, miniport call managers (MCMs), and miniport intermediate drivers.

LLC

The LLC sublayer provides error-free transfer of data frames from one node to another. The LLC sublayer establishes and terminates logical links, controls frame flow, sequences frames, acknowledges frames, and retransmits unacknowledged frames. The LLC sublayer uses frame acknowledgment and retransmission to provide virtually error free transmission over the link to the layers above.

In Windows, a software driver known as a protocol driver implements the LLC sublayer.

Network layer

The network layer controls the operation of the subnet. This layer determines the physical path that the data should take, based on the following:

- Network conditions
- Priority of service
- Other factors, such as routing, traffic control, frame fragmentation and reassembly, logical-to-physical address mapping, and usage accounting

A protocol driver implements the network layer.

Transport layer

The transport layer ensures that messages are delivered error free, in sequence, and with no loss or duplication. This layer relieves the higher-layer protocols from being concerned about data transfer with their peers.

A minimal transport layer is required in protocol stacks that include a reliable network or LLC sublayer that provides virtual circuit capability. For example, because the NetBEUI transport driver for Windows is an OSI-compliant LLC sublayer, its transport layer functions are minimal. If the protocol stack doesn't include an LLC sublayer, and if the network layer is unreliable or supports datagrams (as with TCP/IP's IP layer or NWLink's IPX layer), the transport layer should include frame sequencing and acknowledgment, as well as retransmission of unacknowledged frames.

In the Windows network architecture, a protocol driver, sometimes referred to as a *transport driver*, implements the transport layer.

Overview of NDIS driver types

Article • 09/27/2024

The Network Driver Interface Specification (NDIS) library abstracts the network hardware from network drivers. NDIS also specifies a standard interface between layered network drivers, thereby abstracting lower-level drivers that manage hardware from upper-level drivers, such as network transports. NDIS also maintains state information and parameters for network drivers, including pointers to functions, handles, and parameter blocks for linkage, and other system values.

NDIS supports the following primary types of network drivers:

- Miniport drivers
- Protocol drivers
- Filter drivers
- Intermediate drivers

Note: These topics detail each type of NDIS driver individually. For more information about the NDIS driver stack and a diagram showing the relationship between all four NDIS driver types, see NDIS Driver Stack.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Miniport drivers

Article • 09/27/2024

An *NDIS miniport driver* has two basic functions:

- Managing a network interface card (NIC), including sending and receiving data through the NIC.
- Interfacing with higher-level drivers, such as filter drivers, intermediate drivers, and protocol drivers.

A miniport driver communicates with its NICs and with higher-level drivers through the NDIS library. The NDIS library exports a full set of functions (**NdisM*Xxx*** and other **Ndis*Xxx*** functions) that encapsulate all of the operating system functions that a miniport driver must call. The miniport driver, in turn, must export a set of entry points (*MiniportXxx* functions) that NDIS calls for its own purposes, or on behalf of higher-level drivers, to access the miniport driver.

> ⓘ **Note**
>
> For more information about the NDIS driver stack and a diagram showing the relationship between all four NDIS driver types, see NDIS Driver Stack.

The following send and receive operations illustrate the interaction of miniport drivers with NDIS and with higher-level drivers:

- When a transport driver has a packet to transmit, it calls an **Ndis*Xxx*** function exported by the NDIS library. NDIS then passes the packet to the miniport driver by calling the appropriate *MiniportXxx* function exported by the miniport driver. The miniport driver then forwards the packet to the NIC for transmission by calling the appropriate **Ndis*Xxx*** functions.
- When a NIC receives a packet addressed to itself, it can post a hardware interrupt that is handled by NDIS or the NIC's miniport driver. NDIS notifies the NIC's miniport driver by calling the appropriate *MiniportXxx* function. The miniport driver sets up the transfer of data from the NIC and then indicates the presence of the received packet to bound higher-level drivers by calling the appropriate **Ndis*Xxx*** function.

Connectionless and Connection-Oriented Miniport Drivers

NDIS supports miniport drivers for both connectionless environments and connection-oriented environments.

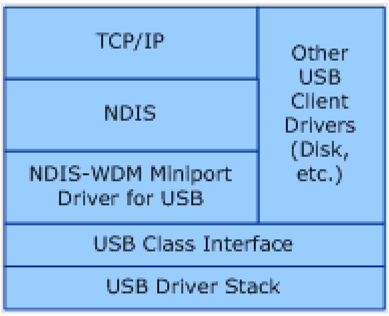
Connectionless miniport drivers control NICs for connectionless network media, such as Ethernet. Connectionless miniport drivers are further divided into deserialized and serialized drivers:

Note All NDIS 6.0 and later drivers are deserialized.

- *Deserialized drivers* serialize the operation of their own *MiniportXxx* functions and that internally queue all incoming send packets. This results in significantly better full-duplex performance, provided that the driver's critical sections (code that only a single thread at a time can run) are kept small.
- *Serialized drivers* rely on NDIS to serialize calls to their *MiniportXxx* functions and to manage their send queues.

Connection-oriented miniport drivers control NICs for connection-oriented network media, such as ISDN. Connection-oriented miniport drivers are always deserialized -- they always serialize the operation of their own *MiniportXxx* functions and queue internally all incoming send packets.

An NDIS miniport driver can have a non-NDIS lower edge (see the following figure).


TCP/IP
NDIS
NDIS-WDM Miniport Driver for USB
Other USB Client Drivers (Disk, etc.)
USB Class Interface
USB Driver Stack

Through its non-NDIS lower edge, a miniport driver uses the class interface for a bus, such as the Universal Serial Bus (USB) to control a device on the bus. The miniport driver communicates with the device by sending I/O request packets (IRPs) either to the bus or directly to remote devices that are attached to the bus. At its upper edge, the miniport driver exposes a standard NDIS miniport driver interface, which enables the miniport driver to communicate with overlying NDIS drivers.

Related topics

NDIS Miniport Drivers

NDIS Miniport Driver Reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Protocol drivers

Article • 12/15/2021

A network protocol, which is the highest driver in the NDIS hierarchy of drivers, is often used as the lowest-level driver in a transport driver that implements a transport protocol stack, such as a TCP/IP stack. A *transport protocol driver* allocates packets, copies data from the sending application into the packet, and sends the packets to the lower-level driver by calling NDIS functions. A protocol driver also provides a protocol interface to receive incoming packets from the next lower-level driver. A transport protocol driver transfers received data to the appropriate client application.

At its lower edge, a protocol driver interfaces with intermediate network drivers and miniport drivers. The protocol driver calls **Ndis*Xxx*** functions to send packets, read and set information that is maintained by lower-level drivers, and use operating system services. The protocol driver also exports a set of entry points (*ProtocolXxx* functions) that NDIS calls for its own purposes or on behalf of lower-level drivers to indicate up receive packets, indicate the status of lower-level drivers, and to otherwise communicate with the protocol driver.

At its upper edge, a transport protocol driver has a private interface to a higher-level driver in the protocol stack.

> ⓘ **Note**
>
> For more information about the NDIS driver stack and a diagram showing the relationship between all four NDIS driver types, see **NDIS Driver Stack**.

Related topics

NDIS Protocol Drivers

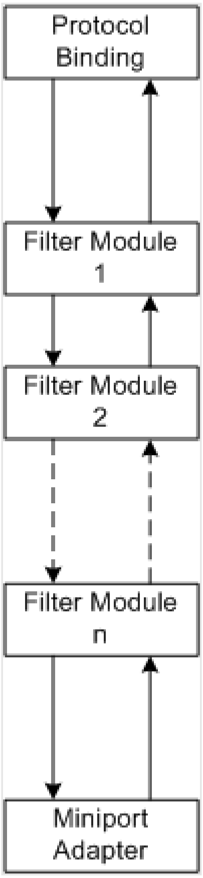
NDIS Protocol Driver Reference

Filter drivers

Article • 09/27/2024

NDIS 6.0 introduced NDIS filter drivers. Filter drivers can monitor and modify the interaction between protocol drivers and miniport drivers. Filter drivers are easier to implement and have less processing overhead than NDIS intermediate drivers.

A *filter module* is an instance of a filter driver. As the following figure illustrates, filter modules are typically layered between miniport adapters and protocol bindings.



A filter driver communicates with NDIS and other NDIS drivers through the NDIS library. The NDIS library exports a full set of functions (**NdisF*Xxx*** and other **Ndis*Xxx*** functions) that encapsulate all of the operating system functions that a filter driver must call. The filter driver, in turn, must export a set of entry points (*FilterXxx* functions) that NDIS calls for its own purposes, or on behalf of other drivers, to access the filter driver.

ⓘ **Note**

For more information about the NDIS driver stack and a diagram showing the relationship between all four NDIS driver types, see NDIS Driver Stack.

Related topics

NDIS Filter Drivers

NDIS Filter Driver Reference

Feedback

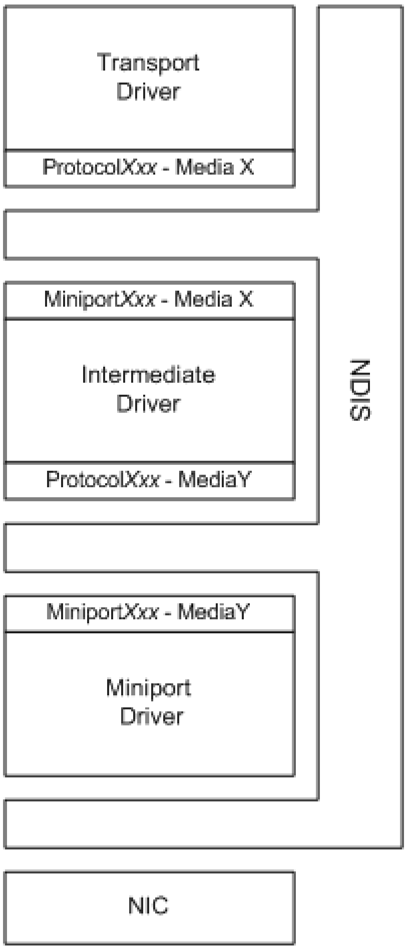
Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

Intermediate drivers

Article • 12/15/2021

As the following figure illustrates, intermediate drivers are typically layered between miniport drivers and transport protocol drivers.



ⓘ **Note**

For more information about the NDIS driver stack and a diagram showing the relationship between all four NDIS driver types, see **NDIS Driver Stack**.

Because of its intermediate position in the driver hierarchy, an intermediate driver must communicate with both overlying protocol drivers and underlying miniport drivers in order to expose:

- Protocol entry points.

  At its lower edge, NDIS calls the *ProtocolXxx* functions to communicate requests from underlying miniport drivers. The intermediate driver looks like a protocol driver to an underlying miniport driver.

- Miniport driver entry points.

  At its upper edge, NDIS calls the *MiniportXxx* functions to communicate the requests of one or more overlying protocol drivers. The intermediate driver looks like a miniport driver to an overlying protocol driver.

An intermediate driver exports a subset of the *MiniportXxx* functions at its upper edge. It also exports one or more virtual adapters, to which overlying protocol drivers can bind. To a protocol driver, a virtual adapter that was exported by an intermediate driver appears to be a physical NIC. When a protocol driver sends packets or requests to a virtual adapter, the intermediate driver propagates these packets and requests to the underlying miniport driver. When the underlying miniport driver indicates received packets, responds to a protocol driver's requests for information, or indicates status, the intermediate driver propagates such packets, responses, and status up to the protocol drivers that are bound to the virtual adapter.

You can use intermediate drivers to:

- Translate between different network media.

- Balance packet transmission across more than one NIC. A load balancing driver exposes one virtual adapter to overlying transport protocols and distributes send packets across more than one NIC.

Related topics

NDIS Intermediate Drivers

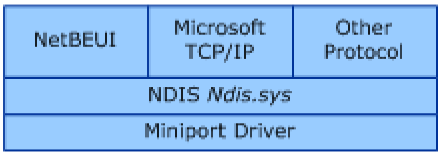
NDIS Intermediate Driver Reference

Connectionless Environment for Network Drivers

Article • 12/15/2021

The connectionless environment is the standard network driver environment for connectionless media, such as Ethernet. For a description of the drivers in this environment, see NDIS Drivers and NDIS Miniport Drivers.

The following figure shows the NDIS environment for connectionless network drivers.



Related topics

Connection-Oriented Environment for Network Drivers

NDIS Drivers

NDIS Miniport Drivers

Connection-Oriented Environment for Network Drivers

Article • 12/15/2021

Connection-oriented drivers are supported by the Connection-Oriented NDIS (CoNDIS) interface.

For detailed information about the CoNDIS architecture, see Connection-Oriented Environment

Network Driver Programming Considerations

Article • 12/15/2021

Microsoft Windows network drivers share similar design goals. Network drivers should be written to be portable and scalable, to provide simple configuration of hardware and software, to use object-based interfaces, and to support asynchronous I/O. This section describes how to apply these general design goals to the network drivers that you write for Microsoft Windows Vista and later operating systems.

This section includes the following topics:

- Performance in Network Drivers
- Performance in Network Adapters
- Portability in Network Drivers
- Multiprocessor Support in Network Drivers
- IRQLs in Network Drivers
- Synchronization and Notification in Network Drivers
- Packet Structures in Network Drivers
- Using Shared Memory in Network Drivers
- Asynchronous I/O and Completion Functions in Network Drivers
- Security Issues for Network Drivers

Performance in Network Drivers

Article • 12/15/2021

Minimizing send and receive path length

Although the send and receive paths differ from driver to driver, there are some general rules for performance optimizations:

- Optimize for the common paths. The Kernprof.exe tool is provided with the developer and IDW builds of Windows that extracts the needed information. The developer should look at the routines that consume the most CPU cycles and attempt to reduce the frequency of these routines being called or the time spent in these routines.

- Reduce time spent in DPC so that the network adapter driver does not use excessive system resources, which would cause overall system performance to suffer.

- Make sure that debugging code is not compiled into the final released version of the driver; this avoids executing excess code.

Partitioning data and code to minimize sharing across processors

Partitioning is needed to minimize shared data and code across processors. Partitioning helps reduce system bus utilization and improves the effectiveness of processor cache. To minimize sharing, driver writers should consider the following:

- Implement the driver as a deserialized miniport as described in Deserialized NDIS Miniport Drivers.

- Use per-processor data structures to reduce global and shared data access. This allows you to keep statistic counters without synchronization, which reduces the code path length and increases performance. For vital statistics, have per-processor counters that are added together at query time. If you must have a global counter, use interlocked operations instead of spin locks to manipulate the counter. See Using Locking Mechanisms Properly below for information about how to avoid using spin locks.

  To facilitate this, KeGetCurrentProcessorNumberEx can be used to determine the current processor. To determine the number of processors when allocating per-processor data structures, KeQueryGroupAffinity can be used.

  The total number of bits set in the affinity mask indicates the number of active processors in the system. Drivers should not assume that all the set bits in the mask will be contiguous because the processors might not be consecutively numbered in the future releases of the operating system. The number of processors in an SMP machine is a zero-based value.

  If your driver maintains per-processor data, you can use the KeQueryGroupAffinity function to reduce cache-line contention.

Avoiding false sharing

False sharing occurs when processors request shared variables that are independent from each other. However, because the variables are on the same cache line, they are shared among the processors. In such situations, the cache line will travel back and forth between processors for every access to any of the variables in it, causing an increase in cache flushes and reloads. This increases the system bus utilization and reduces overall system performance.

To avoid false sharing, align important data structures (such as spin locks, buffer queue headers, singly linked lists) to cache-line boundaries by using NdisGetSharedDataAlignment.

Using locking mechanisms properly

Spin locks can reduce performance if not used properly. Drivers should minimize their use of spin locks by using interlocked operations wherever possible. However, in some cases, a spin lock might be the best choice for some purposes. For example, if a driver acquires a spin lock while handling the reference count for the number of packets that have not been indicated back to the driver, it is not necessary to use an interlocked operation. For more information, see Synchronization and Notification in Network Drivers.

Here are some tips for using locking mechanisms effectively:

- Use NDIS singly-linked list functions such as the following for managing resource pools:

  NdisInitializeSListHead

  NdisInterlockedPushEntrySList

  NdisInterlockedPopEntrySList

  NdisQueryDepthSList

- If you need to use spin locks, use them to only protect data, not code. Don't use one lock to protect all data used in common paths. For example, separate the data used in the send and receive paths into two data structures so that when the send path needs to lock its data, the receive path is not affected.

- If you are using spin locks and the path is already at DPC level, use the **NdisDprAcquireSpinLock** and **NdisDprReleaseSpinLock** functions to avoid extra code when acquiring and releasing the locks.

- To minimize the number of spin lock acquires and releases, use these NDIS RWLock functions:

  NdisAllocateRWLock

  NdisAcquireRWLockRead

  NdisAcquireRWLockWrite

  NdisReleaseRWLock

Using 64-bit DMA

64-Bit DMA If the network adapter supports 64-bit DMA, steps must be taken to avoid extra copies for addresses above the 4 GB range. When the driver calls **NdisMRegisterScatterGatherDma**, the **NDIS_SG_DMA_64_BIT_ADDRESS** flag must be set in the *Flags* parameter.

Ensuring proper buffer alignment

Buffer alignment on a cache-line boundary improves performance when copying data from one buffer to another. Most network adapter receive buffers are properly aligned when they are first allocated, but the user data that must eventually be copied into the application buffer is misaligned due to the header space consumed. In the case of TCP data (the most common scenario), the shift due to the TCP, IP and Ethernet headers results in a shift of 0x36 bytes. To resolve this problem, we recommend that drivers allocate a slightly larger buffer and insert packet data at an offset of 0xA bytes. This will ensure that, after the buffers are shifted by 0x36 bytes for the header, the user data is properly aligned. For more information about cache-line boundaries, see the Remarks section for **NdisMAllocateSharedMemory**.

Using Scatter-Gather DMA

NDIS Scatter/Gather DMA provides the hardware with support to transfer data to and from noncontiguous ranges of physical memory. Scatter-Gather DMA uses a **SCATTER_GATHER_LIST** structure, which includes an array of **SCATTER_GATHER_ELEMENT** structures and the number of elements in the array. This structure is retrieved from the packet descriptor passed to the driver's send function. Each element of the array provides the length and starting physical address of a physically contiguous Scatter-Gather region. The driver uses the length and address information for transferring the data.

Using the Scatter-Gather routines for DMA operations can improve utilization of system resources by not locking these resources down statically, as would occur if map registers were used. For more information, see NDIS Scatter/Gather DMA.

If the network adapter supports TCP Segmentation Offload (Large Send Offload), then the driver will need to pass in the maximum buffer size it can get from TCP/IP into the *MaximumPhysicalMapping* parameter within **NdisMRegisterScatterGatherDma** function. This will guarantee that the driver has enough map registers to build the Scatter-Gather list and eliminate any possible buffer allocations and copying. For more information, see these topics:

- Determining Task Offload Capabilities
- Offloading the Segmentation of Large TCP Packets

Supporting Receive Side Throttle

To minimize disruptions during media playback in multimedia applications, NDIS 6.20 and later drivers must support Receive Side Throttle (RST) in processing receive interrupts. For more information, see:

Receive Side Throttle in NDIS 6.20 "Send and Receive Code Paths" in Summary of Changes Required to Port a Miniport Driver to NDIS 6.20

Performance in Network Adapters

Article • 09/27/2024

There are always tradeoffs in deciding which hardware functions to implement on a network adapter. It's becoming increasingly important to consider adding task offload features that allow for interrupt moderation, dynamic tuning on the hardware, improving the use of the PCI bus, and supporting Jumbo Frames. These offload features are important for the high-end network adapter that is used in configurations requiring top performance.

- Supporting TCP and IP checksum offload
- Supporting large send offload (LSO)
- Supporting IP security (IPSec) offload
- Improving interrupt moderation
- Using the PCI bus efficiently
- Supporting jumbo frames

Supporting TCP and IP checksum offload

For most common network traffic, offloading checksum calculation to the network adapter hardware offers a significant performance advantage by reducing the number of CPU cycles required per byte. Checksum calculation is the most expensive function in the networking stack for two reasons:

- It contributes to long path length.
- It causes cache churning effects (typically on the sender).

Offloading checksum calculation to the sender improves the overall system performance by reducing the load on the host CPU and increasing cache effectiveness.

In the Windows Performance Lab, we have measured TCP throughput improvements of 19% when checksum was offloaded during network-intensive workloads. Analysis of this improvement shows that 11% of the total improvement is due to the path length reduction, and 8% is due to increasing the caches effectiveness.

Offloading checksum on the receiver has the same advantages as offloading checksum on the sender. Increased benefit can be seen on systems that act as both client and server, such as a sockets proxy server. On systems where the CPU isn't necessarily busy, such as a client system, the benefit of offloading checksum may be seen in better network response times, rather than in noticeably improved throughput.

Supporting large send offload (LSO)

Windows offers the ability for the network adapter/driver to advertise a larger Maximum Segment Size (MSS) than the MTU to TCP up to 64K. This allows TCP to allocate a buffer of up to 64K to the driver, which divides the large buffer into packets that fit within the network MTU.

The TCP segmenting work is done by the network adapter/driver hardware instead of the host CPU . This results in a significant performance improvement if the network adapter CPU is able to handle the additional work.

For many of the network adapters tested, there was little improvement seen for pure networking activities when the host CPU was more powerful than the network adapter hardware. However, for typical business workloads, an overall system performance improvement of up to 9% of the throughput has been measured, because the host CPU uses most of its cycles to execute transactions. In these cases, offloading TCP segmentation to the hardware frees the host CPU from the load of segmentation, allowing it extra cycles to perform more transactions.

Supporting IP security (IPSec) offload

Windows offers the ability to offload the encryption work of IPSec to the network adapter hardware. Encryption, especially 3 DES (also known as triple DES), has a very high cycles/byte ratio. Therefore, it's no surprise that offloading IPSec to the network adapter hardware measured a 30% performance boost in secure Internet and VPN tests.

Improving interrupt moderation

A simple network adapter generates a hardware interrupt on the host upon the arrival of a packet or to signal completion of a packet send request. Interrupt latency and resulting cache churning effects add overhead to the overall networking performance. In many scenarios (for example, heavy system usage or heavy network traffic), it's best to reduce the cost of the hardware interrupt by processing several packets for each interrupt.

With heavy network workloads, up to 9% performance improvement in throughput has been measured over network-intensive workloads. However, tuning Interrupt Moderation parameters only for throughput improvements may result in a performance hit on the response time. To maintain optimum settings and accommodate for different workloads, it's best to allow for dynamically adjusted parameters as described in the Auto-tuning later in this article.

Using the PCI bus efficiently

One of the most important factors in the network adapter hardware performance is how efficiently it uses the PCI bus. Further, the network adapter's DMA performance affects the performance of all PCI cards that are on the same PCI bus. The following guidelines must be considered when optimizing PCI usage:

- Streamline DMA transfers by aggregating target pages where appropriate.
- Reduce PCI protocol overhead by performing DMA in large chunks (at least 256 bytes). If possible, time the flow of data so that entire packets are transferred in a single PCI transaction. However, consider how the transfer should take place. For example, don't wait for all of the data to arrive before initiating transfers, because waiting will increase latency and consume additional buffer space.
- It's better to pad the DMA packet transfer with additional bytes, rather than requiring a short extra transfer to "clean up" by transferring the last few bytes of the packet.
- Use the Memory Read, Memory Read Line, and Memory Read Multiple transactions as recommended by the PCI specification.
- The network adapter bus interface hardware should detect limitations in the host memory controller and adjust behavior accordingly. In example, the network adapter bus interface hardware should detect memory-controller pre-fetch limitations on a DMA Memory Reads and wait for a short period before attempting the transaction again. The hardware should detect excessive retries on the part of the network adapter and increase the time before the first retry on future transactions when cut off by the host. There's no point in continuing to submit transactions to the memory controller when you're certain that it's still busy fetching the next sequential set of data.
- Minimize the insertion of wait states, especially during data transfers. It's better to relinquish the bus and let another PCI adapter using the bus get some work done if more than one or two wait states are going to be inserted.
- Use Memory Mapped I/O instead of Programmed I/O. This is also true for drivers.

Supporting jumbo frames

Supporting larger Maximum Transmission Units (MTUs) and thus larger frame sizes, specifically Jumbo Frames, reduces the network stack overhead incurred per byte. A 20% TCP throughput increase has been measured when the MTU was changed from 1514 to

9000. Also, a significant reduction of CPU utilization is obtained due to the fewer number of calls from the network stack to the network driver.

Portability in Network Drivers

Article • 12/15/2021

NDIS drivers should be written so that they are easily portable across all platforms that support Microsoft Windows operating systems. In general, porting from one hardware platform to another should only require recompilation with a system-compatible compiler.

Follow these guidelines when you write NDIS drivers:

- Avoid calling operating system-specific functions. Instead, use the NDIS equivalent functions. NDIS exports a rich set of support functions for writing drivers, and if you call these support functions, you can port the code between Microsoft operating systems that support NDIS.

- Write drivers in C (specifically, the ANSI C Standard). Avoid using any language features that other system-compatible compilers do not support. Do not use any features that the ANSI C standard designates as "implementation defined."

- Avoid dependencies on data types whose size and layout vary across platforms. For example, do not write driver code that calls any C Run-Time Library functions instead of NDIS-provided functions.

- Do not use floating-point operations in kernel mode. If you attempt such operations, a fatal error will occur.

- Use **#ifdef** and **#endif** statements to encapsulate code that is used to support platform-specific features.

Multiprocessor Support in Network Drivers

Article • 12/15/2021

To write a portable driver for all Microsoft Windows versions, you need to write code to safely run on computers with multiple concurrently running processors. A network driver must be multiprocessor-safe and must use the provided NDIS library functions.

In a uniprocessor environment, a single processor runs only one computer instruction at a time, even though it is possible for a network interface card (NIC) or other device to interrupt the current execution stream when packets arrive or as timer interrupts occur. Typically, when manipulating data structures such as packet queues, a driver disables interrupts on the NIC, performs the manipulation, and then reenables interrupts. Many threads in a uniprocessor environment appear to run simultaneously but actually run in interleaved time slices.

In a multiprocessor environment, processors simultaneously run several computer instructions. A driver must synchronize so that when one driver function manipulates common data structures, the same or another driver function on another processor does not attempt to modify shared data at the same time. All driver code is reentrant in a symmetric multiprocessor (SMP) computer. To eliminate this resource protection problem, Windows device drivers use spin locks. For more information, see Synchronization and Notification in Network Drivers.

IRQLs in Network Drivers

Article • 12/15/2021

Every driver function called by NDIS runs at a system-determined IRQL (one of PASSIVE_LEVEL < DISPATCH_LEVEL < DIRQL). For example, a miniport driver's initialization function, halt function, reset function, and shutdown function commonly run at PASSIVE_LEVEL, although the reset and shutdown functions can be invoked at a higher IRQL if the system requires it. Interrupt code runs at DIRQL, so an NDIS intermediate or protocol driver never runs at DIRQL. All other NDIS driver functions run at or below IRQL = DISPATCH_LEVEL.

The IRQL at which a driver function runs affects which NDIS functions it can call. Certain functions can be called only at IRQL = PASSIVE_LEVEL. Others can be called at DISPATCH_LEVEL or lower. You should check every NDIS function for IRQL restrictions.

Any driver function that shares resources with the driver's interrupt service routine (ISR) must be able to raise its IRQL to DIRQL to prevent race conditions. NDIS provides such a mechanism.

Synchronization and Notification in Network Drivers

Article • 12/15/2021

Whenever two threads of execution share resources that can be accessed at the same time, either in a uniprocessor computer or on a symmetric multiprocessor (SMP) computer, they need to be synchronized. For example, on a uniprocessor computer, if one driver function is accessing a shared resource and is interrupted by another function that runs at a higher IRQL, such as an ISR, the shared resource must be protected to prevent race conditions that leave the resource in an indeterminate state. On an SMP computer, two threads could be running simultaneously on different processors and attempting to modify the same data. Such accesses must be synchronized.

NDIS provides spin locks that you can use to synchronize access to shared resources between threads that run at the same IRQL. When two threads that share a resource run at different IRQLs, NDIS provides a mechanism for temporarily raising the IRQL of the lower IRQL code so that access to the shared resource can be serialized.

When a thread depends on the occurrence of an event outside the thread, the thread relies on notification. For example, a driver might need to be notified when some time period has passed so that it can check its device. Or a network interface card (NIC) driver might have to perform a periodic operation such as polling. Timers provide such a mechanism.

Events provide a mechanism that two threads of execution can use to synchronize operations. For example, a miniport driver can test the interrupt on a NIC by writing to the device. The driver must wait for an interrupt to notify the driver that the operation was successful. You can use events to synchronize an operation between the thread waiting for the interrupt to complete and the thread that handles the interrupt.

The following subsections in this topic describe these NDIS mechanisms.

- Spin Locks
- Avoiding Spin Lock Problems
- Timers
- Events

Spin Locks

A *spin lock* provides a synchronization mechanism for protecting resources shared by kernel-mode threads running at IRQL > PASSIVE_LEVEL in either a uniprocessor or a multiprocessor computer. A spin lock handles synchronization among various threads of execution that are running concurrently on an SMP computer. A thread acquires a spin lock before accessing protected resources. The spin lock keeps any thread but the one holding the spin lock from using the resource. On a SMP computer, a thread that is waiting on the spin lock loops attempting to acquire the spin lock until it is released by the thread that holds the lock.

Another characteristic of spin locks is the associated IRQL. Attempted acquisition of a spin lock temporarily raises the IRQL of the requesting thread to the IRQL associated with the spin lock. This prevents all lower IRQL threads on the same processor from preempting the executing thread. Threads, on the same processor, running at a higher IRQL can preempt the executing thread, but these threads cannot acquire the spin lock because it has a lower IRQL. Therefore, after a thread has acquired a spin lock, no other threads can acquire the spin lock until it has been released. A well-written network driver minimizes the amount of time a spin lock is held.

A typical use for a spin lock is to protect a queue. For example, the miniport driver send function, *MiniportSendNetBufferLists*, might queue packets passed to it by a protocol driver. Because other driver functions also use this queue, *MiniportSendNetBufferLists* must protect the queue with a spin lock so that only one thread at a time can manipulate the links or contents. *MiniportSendNetBufferLists* acquires the spin lock, adds the packet to the queue and then releases the spin lock. Using a spin lock ensures that the thread holding the lock is the only thread modifying the queue links while the packet is safely added to the queue. When the miniport driver takes the packets off the queue, such an access is protected by the same spin lock. When running instructions that modify the head of the queue or any of the link fields making up the queue, the driver must protect the queue with a spin lock.

A driver must take care not to overprotect a queue. For example, the driver can perform some operations (for example, filling in a field containing the length) in the network driver-reserved field of a packet before it queues the packet. The driver can do this outside the code region protected by the spin lock, but must do it before queuing the packet. After the packet is on the queue and the running thread releases the spin lock, the driver must assume that other threads can dequeue the packet immediately.

Avoiding Spin Lock Problems

To avoid a possible deadlock, an NDIS driver should release all NDIS spin locks before calling an NDIS function other than an **Ndis*Xxx*Spinlock** function. If an NDIS driver does not comply with this requirement, a deadlock could occur as follows:

1. Thread 1, which holds NDIS spin lock A, calls an **Ndis*Xxx*** function that attempts to acquire NDIS spin lock B by calling the NdisAcquireSpinLock function.

2. Thread 2, which holds NDIS spin lock B, calls an **Ndis*Xxx*** function that attempts to acquire NDIS spin lock A by calling the **NdisAcquireSpinLock** function.

3. Thread 1 and thread 2, which are each waiting for the other to release its spin lock, become deadlocked.

Microsoft Windows operating systems do not restrict a network driver from simultaneously holding more than one spin lock. However, if one section of the driver attempts to acquire spin lock A while holding spin lock B, and another section attempts to acquire spin lock B while holding spin lock A, deadlock results. If it acquires more than one spin lock, a driver should avoid deadlock by enforcing an order of acquisition. That is, if a driver enforces acquiring spin lock A before spin lock B, the situation described above will not occur.

Acquiring a spin lock raises the IRQL to DISPATCH_LEVEL and stores the old IRQL in the spin lock. Releasing the spin lock sets the IRQL to the value stored in the spin lock. Because NDIS sometimes enters drivers at PASSIVE_LEVEL, problems can arise with the following code sequence:

syntax

```
NdisAcquireSpinLock(A);
NdisAcquireSpinLock(B);
NdisReleaseSpinLock(A);
NdisReleaseSpinLock(B);
```

A driver should not access spin locks in this sequence for the following reasons:

- Between releasing spin lock A and releasing spin lock B, the code is running at PASSIVE_LEVEL instead of DISPATCH_LEVEL and is subject to inappropriate interruption.

- After releasing spin lock B, the code is running at DISPATCH_LEVEL which could cause the caller to fault at much later time with an IRQL_NOT_LESS_OR_EQUAL stop error.

Using spin locks impacts performance and, in general, a driver should not use many spin locks. Occasionally, functions that are usually distinct (for example, send and receive functions) have minor overlaps for which two spin locks can be used. Use of more than one spin lock might be a worthwhile tradeoff in order to allow the two functions to operate independently on separate processors.

Timers

Timers are used for polling or timing out operations. A driver creates a timer and associates a function with the timer. The associated function is called when the period specified in the timer expires. Timers can be one-shot or periodic. Once a periodic timer is set, it will continue to fire at the expiration of every period until explicitly cleared. A one-shot timer must be reset each time it fires.

Timers are created and initialized by calling **NdisAllocateTimerObject** and set by calling **NdisSetTimerObject**. If a nonperiodic timer is used, it must reset by calling **NdisSetTimerObject**. A timer is cleared by calling **NdisCancelTimerObject**.

Events

Events are used to synchronize operations between two threads of execution. An event is allocated by a driver and initialized by calling **NdisInitializeEvent**. A thread running at IRQL = PASSIVE_LEVEL calls **NdisWaitEvent** to put itself into a wait state. When a driver thread waits on an event, it specifies a maximum time to wait as well as the event to be waited on. The thread's wait is satisfied when **NdisSetEvent** is called causing the event to be signaled, or when the specified maximum wait-time interval expires, whichever occurs first.

Typically, the event is set by a cooperating thread that calls **NdisSetEvent**. Events are unsignaled when they are created and must be set in order to signal waiting threads. Events remain signaled until **NdisResetEvent** is called.

Related topics

Multiprocessor Support in Network Drivers

Packet Structures in Network Drivers

Article • 12/15/2021

In NDIS 6.0 and later versions, a higher layer driver allocates **NET_BUFFER** and **NET_BUFFER LIST** structures to hold network packet information, and sends the structures to the next lower NDIS driver so that the data can be sent on the network. Lower-level drivers allocate NET_BUFFER and NET_BUFFER_LIST structures to hold received data and pass the structures up to interested higher-layer drivers. Sometimes, a higher layer driver allocates structures and passes them to a lower layer driver with a request for the lower layer driver to copy received data into the buffers provided. NDIS provides functions for allocating and manipulating the substructures that make up the NET_BUFFER and NET_BUFFER_LIST structures.

For more information about the structure of network data buffers in NDIS drivers, see NET_BUFFER Architecture.

Using Shared Memory in Network Drivers

Article • 12/15/2021

Miniport drivers for bus-master direct memory access (DMA) devices allocate shared memory for use by the network interface card (NIC) and the miniport driver.

NdisMAllocateSharedMemory can be called by a bus-master miniport driver to allocate memory for permanent sharing between the network adapter and the miniport driver. This function returns a virtual address and a physical address for the shared memory. The addresses are valid until a call to NdisMFreeSharedMemory frees the memory.

Asynchronous I/O and Completion Functions in Network Drivers

Article • 12/15/2021

Latency is inherent in some network operations. Because of this latency, many of the upper-edge functions provided by a miniport driver and the lower-edge functions of a protocol driver are designed to support asynchronous operation. Rather than wasting CPU cycles waiting in a loop for some time-consuming task to finish or a hardware event to signal, network drivers rely on the ability to handle most operations asynchronously.

Asynchronous network I/O is supported by using a *completion* function. The following example illustrates using a completion function for a network *send* operation, but this same mechanism exists for many other operations that are performed by a protocol or miniport driver.

When a protocol driver calls NDIS to send a packet, resulting in a call to the miniport driver's *MiniportSendNetBufferLists* function, the miniport driver can try to complete this request immediately and return an appropriate status value as a result. For synchronous operation, the possible responses are NDIS_STATUS_SUCCESS for successful completion of the send, NDIS_STATUS_RESOURCES, and NDIS_STATUS_FAILURE indicating a failure of some kind.

But a send operation can take some time to complete while the miniport driver (or NDIS) queues the packet and waits for the NIC to indicate the result of the send operation. The miniport driver *MiniportSendNetBufferLists* function can handle this operation asynchronously by returning a status value of NDIS_STATUS_PENDING. When the miniport driver completes the send operation, it calls the completion function, **NdisMSendNetBufferListsComplete**, passing a pointer to the packet descriptor that was sent. This information is passed to the protocol driver, signaling completion.

Most driver operations that can require an extended time to complete support asynchronous operation with a similar completion function. Such functions have names of the form **NdisM*Xxx*Complete**.

Completion functions are also provided to:

- Set and querying configuration.
- Reset hardware.
- Indicate status.

- Indicate received data.
- Transfer received data.

Security Issues for Network Drivers

Article • 12/15/2021

For a general discussion on writing secure drivers, see Creating Reliable Kernel-Mode Drivers.

Beyond following safe coding practices and the general device driver guidance, network drivers should do the following to enhance security:

- All network drivers should validate values that they read from the registry. Specifically, the caller of **NdisReadConfiguration** or **NdisReadNetworkAddress** must not make any assumptions about values read from the registry and must validate each registry value that it reads. If the caller of **NdisReadConfiguration** determines that a value is out of bounds, it should use a default value instead. If the caller of **NdisReadNetworkAddress** determines that a value is out of bounds, it should use the permanent medium access control (MAC) address or a default address instead.

OID-specific issues

- A miniport driver, in its *MiniportOidRequest* or **MiniportCoOidRequest** functions, should validate any object identifier (OID) value that the driver is requested to set. If the driver determines that the value to be set is out of bounds, it should fail the set request. For more information about object identifiers, see Obtaining and Setting Miniport Driver Information and NDIS Support for WMI.

- If an intermediate driver's *MiniportOidRequest* function does not pass a set operation to an underlying miniport driver, the function should validate the OID value. For more information, see Intermediate Driver Query and Set Operations.

Query OID security guidelines

Most Query OIDs can be issued by any usermode application on the system. Follow these specific guidelines for Query OIDs.

1. Always validate the size of the buffer is large enough for the output. Any query OID handler without an output buffer size check has a security bug.

```C++
if (oid->DATA.QUERY_INFORMATION.InformationBufferLength <
sizeof(ULONG)) {
    oid->DATA.QUERY_INFORMATION.BytesNeeded = sizeof(ULONG);
    return NDIS_STATUS_INVALID_LENGTH;
}
```

2. Always write a correct and minimal value to BytesWritten. It is a red flag to assign `oid->BytesWritten = oid->InformationBufferLength` like the following example does.

C++

```
// ALWAYS WRONG
oid->DATA.QUERY_INFORMATION.BytesWritten =
DATA.QUERY_INFORMATION.InformationBufferLength;
```

The OS will copy BytesWritten bytes back to a usermode application. If BytesWritten is larger than the number of bytes the driver actually wrote, then the OS might end up copying back uninitialized kernel memory to usermode, which would be an information disclosure vulnerability. Instead, use code similar to this:

C++

```
oid->DATA.QUERY_INFORMATION.BytesWritten = sizeof(ULONG);
```

3. Never read values back from the buffer. In some cases, the output buffer of an OID is directly mapped into a hostile usermode process. The hostile process can change your output buffer after you've written to it. For example, the code below can be attacked, because an attacker can change NumElements after it is written:

C++

```
output->NumElements = 4;
for (i = 0 ; i < output->NumElements ; i++) {
    output->Element[i] = . . .;
}
```

To avoid reading back from the buffer, keep a local copy. For example, to fix the above example, introduce a new stack variable:

C++

```
ULONG num = 4;
output->NumElements = num;
for (i = 0 ; i < num; i++) {
    output->Element[i] = . . .;
}
```

With this approach, the for loop reads back from the driver's stack variable `num` and not from its output buffer. The driver should also mark the output buffer with the `volatile` keyword, to prevent the compiler from silently undoing this fix.

Set OID security guidelines

Most Set OIDs can be issued by a usermode application running in the Administrators or System security groups. Although these are generally trusted applications, the miniport driver still must not permit memory corruption or injection of kernel code. Follow these specific rules for Set OIDs:

1. Always validate the input is large enough. Any OID set handler without an input buffer size check has a security vulnerability.

   C++

   ```
   if (oid->DATA.SET_INFORMATION.InformationBufferLength < sizeof(ULONG))
   {
       return NDIS_STATUS_INVALID_LENGTH;
   }
   ```

2. Whenever validating an OID with an embedded offset, you must validate that the embedded buffer is within the OID payload. This requires several checks. For example, OID_PM_ADD_WOL_PATTERN may deliver an embedded pattern, that needs to be checked. Correct validation requires checking:

   a. InformationBufferSize >= sizeof(NDIS_PM_PACKET_PATTERN)

   C++

   ```
   PmPattern = (PNDIS_PM_PACKET_PATTERN) InformationBuffer;
   if (InformationBufferLength < sizeof(NDIS_PM_PACKET_PATTERN))
   {
       Status = NDIS_STATUS_BUFFER_TOO_SHORT;
       *BytesNeeded = sizeof(NDIS_PM_PACKET_PATTERN);
       break;
   }
   ```

   b. Pattern->PatternOffset + Pattern->PatternSize does not overflow

   C++

   ```
   ULONG TotalSize = 0;
   if (!NT_SUCCESS(RtlUlongAdd(Pattern->PatternOffset, Pattern->PatternSize, &TotalSize) ||
   ```

```
    TotalSize > InformationBufferLength)
{
    return NDIS_STATUS_INVALID_LENGTH;
}
```

These two checks can be combined using code like the following example:

C++

```
ULONG TotalSize = 0;
if (InformationBufferLength < sizeof(NDIS_PM_PACKET_PATTERN) ||
    !NT_SUCCESS(RtlUlongAdd(Pattern->PatternSize, Pattern-
>PatternOffset, &TotalSize) ||
    TotalSize > InformationBufferLength)
{
    return NDIS_STATUS_INVALID_LENGTH;
}
```

c. InformationBuffer + Pattern->PatternOffset + Pattern->PatternLength does not overflow

C++

```
ULONG TotalSize = 0;
if (!NT_SUCCESS(RtlUlongAdd(Pattern->PatternOffset, Pattern-
>PatternLength, &TotalSize) ||
    (!NT_SUCCESS(RtlUlongAdd(TotalSize, InformationBuffer,
&TotalSize) ||
    TotalSize > InformationBufferLength)
{
    return NDIS_STATUS_INVALID_LENGTH;
}
```

d. Pattern->PatternOffset + Pattern->PatternLength <= InformationBufferSize

C++

```
ULONG TotalSize = 0;
if(!NT_SUCCESS(RtlUlongAdd(Pattern->PatternOffset, Pattern-
>PatternLength, &TotalSize) ||
    TotalSize > InformationBufferLength))
{
    return NDIS_STATUS_INVALID_LENGTH;
}
```

Method OID security guidelines

Method OIDs can be issued by a usermode application running in the Administrators or System security groups. They are a combination of a Set and a Query, so both preceding lists of guidance also apply to Method OIDs.

Other network driver security issues

- Many NDIS miniport drivers expose a control device by using NdisRegisterDeviceEx. Those that do this must audit their IOCTL handlers, with all the same security rules as a WDM driver. For more information, see Security Issues for I/O Control Codes.

- Well-designed NDIS miniport drivers should not rely on being called in a particular process context, nor interact very closely with usermode (with IOCTLs & OIDs being the exception). It would be a red flag to see a miniport that opened usermode handles, performed usermode waits, or allocated memory against usermode quota. That code should be investigated.

- Most NDIS miniport drivers should not be involved in parsing packet payloads. In some cases, though, it may be necessary. If so, this code should be audited very carefully, as the driver is parsing data from an untrusted source.

- As is standard when allocating kernel-mode memory, NDIS drivers should use appropriate NX Pool Opt-In Mechanisms. In WDK 8 and newer, the `NdisAllocate*` family of functions are properly opted in.

Driver Stack Management

Article • 12/15/2021

NDIS 6.0 introduced the ability to pause and restart a driver stack. To support the stack management features that NDIS 6.0 provides, you must rewrite legacy drivers.

NDIS 6.0 also introduced NDIS filter drivers. Filter drivers can monitor and modify the interaction between protocol drivers and miniport drivers. Filter drivers are easier to implement and have less processing overhead than NDIS 5.*x* intermediate drivers. For these reasons, you should use filter drivers instead of filter intermediate drivers.

A driver stack contains the following logical elements:

Miniport Adapter
A *miniport adapter* is an adapter instance of an NDIS miniport driver or intermediate driver. The virtual miniport of an intermediate driver is a miniport adapter. NDIS configures the other elements of a driver stack over a miniport adapter after a device becomes available.

Protocol Binding
A *protocol binding* is a binding instance of a protocol driver. A protocol binding binds an NDIS protocol driver to a miniport adapter. Multiple protocol drivers can bind to a miniport adapter.

Filter Module
A *filter module* is an instance of a filter driver. NDIS can pause a driver stack to insert, remove, or reconfigure a filter module. Filter modules can monitor and modify the behavior of a miniport adapter.

The following topics provide more information about the driver stack, driver states, and driver stack operations:

- NDIS Driver Stack
- Adapter States of a Miniport Driver
- Binding States of a Protocol Driver
- Module States of a Filter Driver
- NDIS Stack Operations

Related topics

NDIS Filter Drivers

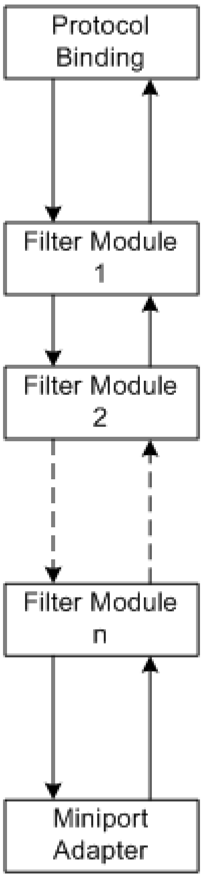
NDIS Intermediate Drivers

NDIS Driver Stack

Article • 03/14/2023

Basic Stack Configuration

The following figure shows a basic configuration of the logical elements in an NDIS 6.0 driver stack. The figure illustrates a driver stack with an unspecified number of filter modules. The arrows represent information flow between the elements of the stack.



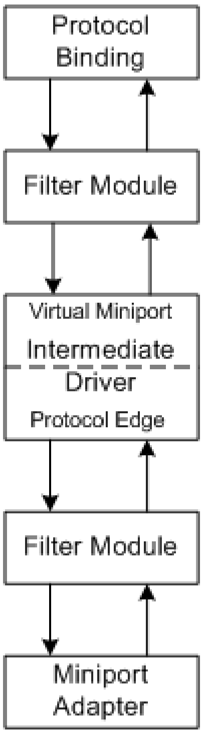
As the preceding figure shows, you can stack any number of filter modules over a miniport adapter. These modules can be instances of different filter drivers and/or multiple instances of the same filter driver. If a miniport driver manages more than one miniport adapter, a separate driver stack can exist over each miniport adapter.

Protocol drivers bind to miniport adapters. Therefore, underlying filter modules in a driver stack are transparent to protocol drivers. To obtain information about underlying filter modules, protocol drivers can enumerate the filter modules in a driver stack.

If more than one protocol driver binds to an miniport adapter, the filter modules are the same for both protocol drivers. Based upon the binding, NDIS routes requests to the correct protocol driver.

NDIS 6.0 Stack with Intermediate Driver

The following figure shows an NDIS 6.0 driver stack with an intermediate driver.



If you include an NDIS intermediate driver in the driver stack, the stack is essentially two stacks: one above the other.

The intermediate driver's virtual miniport provides the miniport adapter for the upper stack, whereas the intermediate driver's protocol edge provides the protocol binding for the lower stack.

A virtual miniport has the same states as any other miniport adapter. For more information about miniport adapter states, see Adapter States of a Miniport Driver.

The protocol edge of the intermediate driver should implement the same binding states as a protocol driver. For more information about binding states, see Binding States of a Protocol Driver.

Related topics

Adapter States of a Miniport Driver

Binding States of a Protocol Driver

Driver Stack Management

NDIS Filter Drivers

NDIS Intermediate Drivers

NDIS Miniport Drivers

NDIS Protocol Drivers

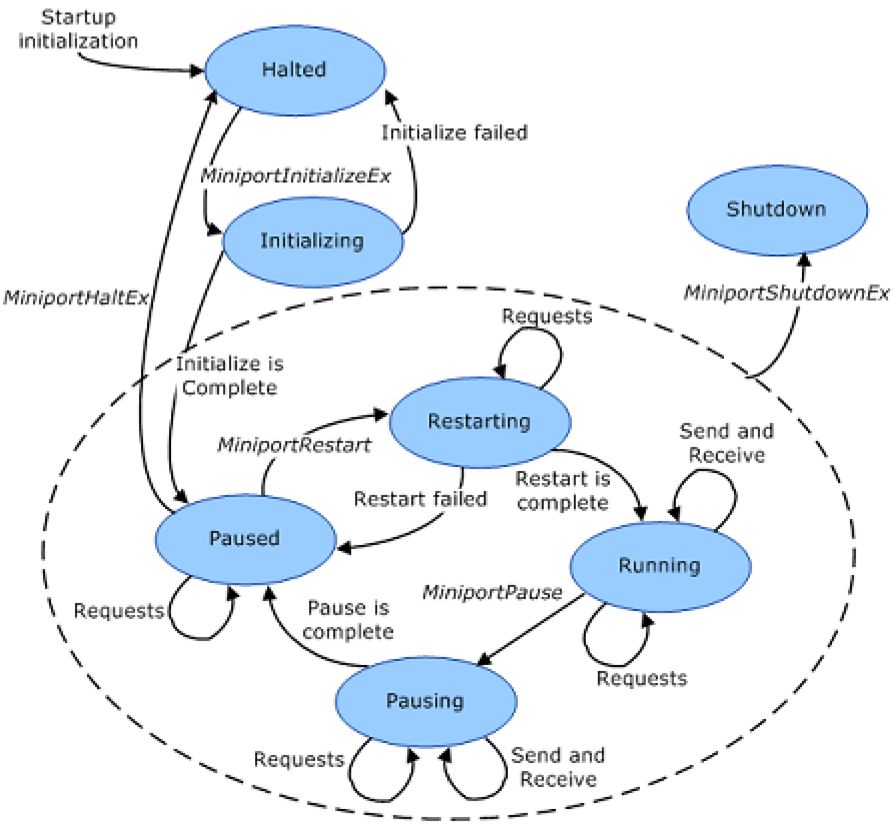
Adapter States of a Miniport Driver

Article • 12/15/2021

For each miniport adapter that it manages, an NDIS miniport driver must support the following set of operational states:

- Halted
- Shutdown
- Initializing
- Paused
- Restarting
- Running
- Pausing

The following figure shows the interrelationships between these states.



Note The reset operation does not affect miniport adapter operational states. Also, the state of the adapter might change while a reset operation is in progress. For example, NDIS might call a driver's pause handler when there is a reset operation in progress. In this case, the driver can complete either the reset or the pause operation in any order while following the normal requirements for each operation. For a reset operation, the driver can fail transmit request packets or it can keep them queued and complete them later. However, you should note that an overlying driver cannot complete a pause operation while its transmit packets are pending.

The following defines the adapter states:

Halted
Halted is the initial state of all miniport adapters. When a miniport adapter is in the Halted state, and NDIS calls the driver's *MiniportInitializeEx* function to initialize the miniport adapter, the miniport adapter enters the Initializing state. If *MiniportInitializeEx* fails, the miniport adapter returns to the Halted state. When the miniport adapter is in the Paused state and NDIS calls the *MiniportHaltEx* function, the miniport adapter returns to the Halted state.

Shutdown
A miniport adapter in the *Shutdown* state cannot be used until the system is shut down and restarted. When the miniport adapter is in the Paused, Restarting, Running, or Pausing state and NDIS calls the miniport driver's *MiniportShutdownEx* function, the miniport adapter enters the Shutdown state.

Initializing
In the *Initializing* state, a miniport driver completes any operations that are required to initialize a miniport adapter. When a miniport adapter is in the Halted state and the NDIS calls the miniport driver's *MiniportInitializeEx* function, the miniport adapter enters the Initializing state. If *MiniportInitializeEx* succeeds, the miniport adapter enters the Paused state. If *MiniportInitializeEx* fails, the miniport adapter returns to the Halted state.

Paused
When a miniport adapter is in the *Paused* state, a miniport driver does not indicate received network data or accept send requests. When a miniport adapter is in the Pausing state and the pause operation is complete, the miniport adapter enters the Paused state. When a miniport adapter is in the Initializing state and *MiniportInitializeEx* is successful, the miniport adapter enters the Paused state. When NDIS calls the miniport driver's **MiniportRestart** function, the miniport adapter transitions from the Paused state to the Restarting state. When NDIS calls the miniport driver's *MiniportHaltEx* function, the miniport adapter transitions from the Paused state to the Halted state.

Restarting
In the *Restarting* state, a miniport driver completes any operations that are required to restart send and receive operations for a miniport adapter. When a miniport adapter is in the Paused state and NDIS calls the driver's *MiniportRestart* function, the miniport

adapter enters the Restarting state. If the restart fails, the miniport adapter returns to the Paused state. If the restart is successful, the miniport adapter enters the Running state.

Running
In the *Running* state, a miniport driver performs normal send and receive processing for a miniport adapter. When the miniport adapter is in the Restarting state and the driver is ready to perform send and receive operations, the miniport adapter enters the Running state.

Pausing
In the *Pausing* state, a miniport driver completes any operations that are required to stop send and receive operations for a miniport adapter. The driver must wait for NDIS to return all outstanding receive indications. When a miniport adapter is in the Running state and NDIS calls the driver's *MiniportPause* function, the miniport adapter enters the Pausing state. A miniport driver cannot fail a pause operation. When the pause operation is complete, the miniport adapter enters the Paused state.

Related topics

Driver Stack Management

NDIS Miniport Drivers

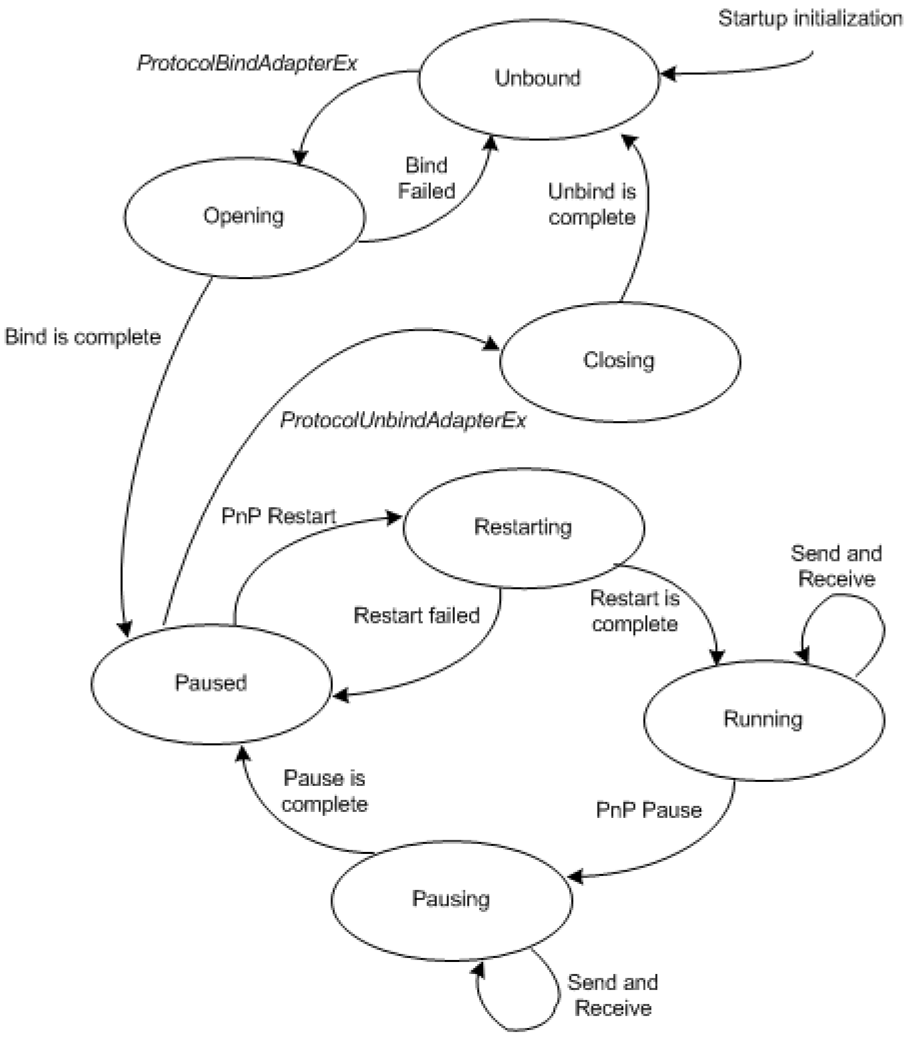
Binding States of a Protocol Driver

Article • 12/15/2021

An NDIS protocol driver must support the following operational states for each binding that the driver manages:

- Unbound
- Opening
- Running
- Closing
- Pausing
- Paused
- Restarting

The following figures shows the relationship between these states.



The following defines the protocol driver binding states:

Unbound
The *Unbound* state is the initial state of a binding. In this state, the protocol driver waits for NDIS to call the *ProtocolBindAdapterEx* function. After NDIS calls *ProtocolBindAdapterEx*, the binding enters the Opening state. After an unbind operation is complete, a binding returns to the Unbound state from the Closing state.

Opening
In the *Opening* state, a protocol driver allocates resources for the binding and attempts to open the miniport adapter. After NDIS calls the driver's *ProtocolBindAdapterEx* function, the binding enters the Opening state. If the protocol driver fails to bind to the miniport adapter, the binding returns to the Unbound state. If the driver successfully binds to the miniport adapter, the binding enters the Paused state.

Running
In the *Running* state, a protocol driver performs normal send and receive processing for a binding. When the binding is in the Restarting state and the driver is ready to perform send and receive operations, the binding enters the Running state.

Closing
In the *Closing* state, the protocol driver closes the binding to the miniport adapter and then releases the resources for the binding. After NDIS calls the protocol driver's *ProtocolUnbindAdapterEx* function, the binding enters the Closing state. After the protocol driver completes the unbind operations, the binding enters the Unbound state.

Pausing
In the *Pausing* state, a protocol driver completes any operations that are required to stop send and receive operations for a binding. When a binding is in the Running state and NDIS sends the protocol driver a PnP pause notification, the binding enters the Pausing state. The protocol driver must wait for all its outstanding send request to complete. A protocol driver cannot fail a pause operation. After the pause operation is complete, the binding enters the Paused state.

Paused
In the *Paused* state, the protocol driver does not perform send or receive operations for a binding. When a binding is in the Pausing state and a pause operation is complete, the binding enters the Paused state. When a binding is in the Opening state and a open operation completes successfully, the binding enters the Paused state. If NDIS sends the protocol driver a PnP restart notification for the binding, the binding enters the Restarting state. If NDIS calls the driver's *ProtocolUnbindAdapterEx* function, the binding enters the Closing state.

Restarting
In the *Restarting* state, a protocol driver completes any operations that are required to restart send and receive operations for a binding. When a binding is in the Paused state and NDIS sends the protocol driver a PnP restart notification, the binding enters the Restarting state. If the restart fails, the binding returns to the Paused state. If the restart is successful, the binding enters the Running state.

Related topics

Driver Stack Management

NDIS Protocol Drivers

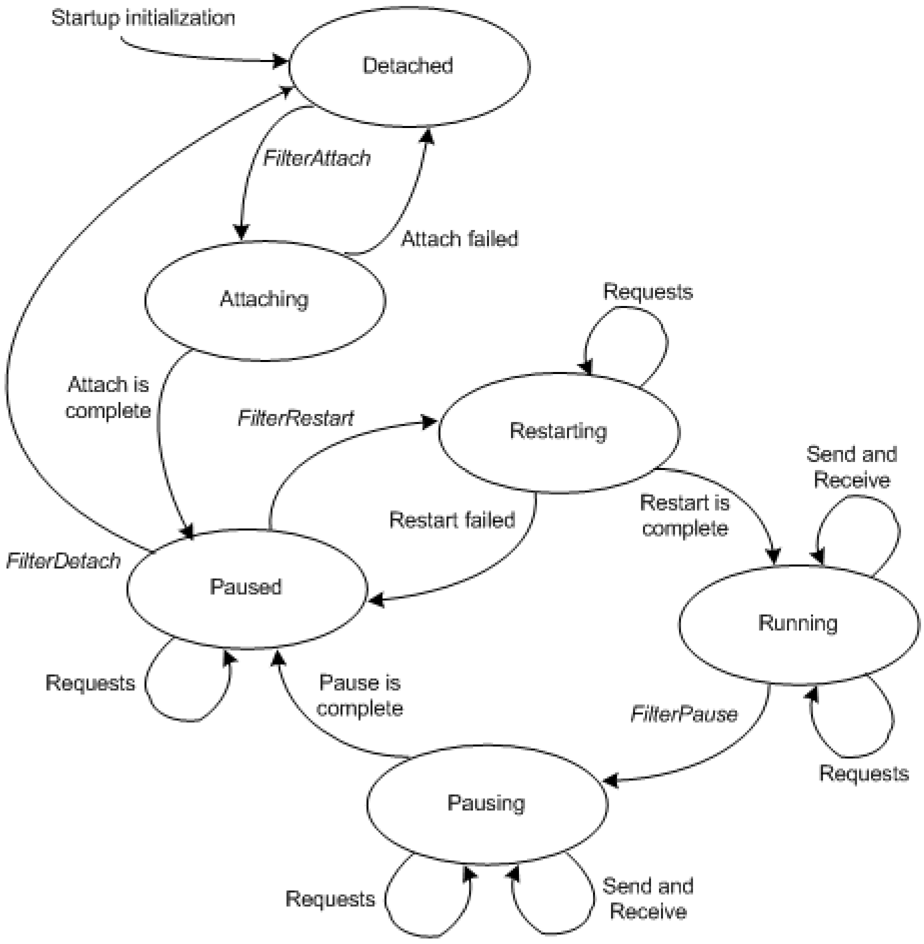
Module States of a Filter Driver

Article • 12/15/2021

An NDIS filter driver must support the following operational states for each filter module (instance of a filter driver) that the driver manages:

- Detached
- Attaching
- Paused
- Restarting
- Running
- Pausing

The following figure shows the relationships between these states.



The following defines the filter module states:

Detached

The *Detached* state is the initial state of a filter module. When a filter module is in this state, NDIS can call the filter driver's *FilterAttach* function to attach the filter module to the driver stack. When NDIS calls a filter driver's *FilterAttach* function, the filter module enters the Attaching state. If the attach operation fails, the filter module returns to the Detached state. When the module is in the Paused state and NDIS calls the *FilterDetach* function, the module returns to the Detached state.

Attaching
When a filter module is in the *Attaching* state, a filter driver prepares to attach the module to the driver stack. After the filter module preparation is complete, the filter module enters the Paused state. If a failure occurs (for example, because the required resources are not available), the filter module returns to the Detached state.

Paused
When a filter module is in the *Paused* state, the filter module does not perform send or receive operations. When a filter module is in the *Attaching* state and *FilterAttach* is successful, the filter module enters the *Paused* state. When a filter module is in the *Pausing* state and the pause operation completes, the filter module enters the *Paused* state. When a filter module is in the *Paused* state and NDIS calls the filter driver's **FilterRestart** function, the filter module enters the *Restarting* state. When a filter module is in the *Paused* state and NDIS calls the driver's *FilterDetach* handler, the filter module enters the *Detached* state.

Restarting
In the *Restarting* state, a filter driver completes any operations that are required to restart send and receive operations for a filter module. When a filter module is in the Paused state and NDIS calls the driver's *FilterRestart* function, a filter module enters the Restarting state. If the restart fails, the filter module returns to the Paused state. If the restart is successful, the filter module enters the Running state.

Running
In the *Running* state, a filter driver performs normal send and receive processing for a filter module. When the filter module is in the Restarting state and the driver is ready to perform send and receive operations, the filter module enters the Running state.

Pausing
In the *Pausing* state, a filter driver completes any operations that are required to stop send and receive operations for a filter module. The filter driver must wait for all its outstanding send requests to complete and for NDIS to return all its outstanding receive indications. When a filter module is in the Running state and NDIS calls the driver's *FilterPause* function, the filter module enters the Pausing state. A filter driver cannot fail a pause operation. After the pause operation is complete, the filter module enters the Paused state.

Related topics

Driver Stack Management

NDIS Filter Drivers

Starting a Driver Stack

Article • 12/15/2021

After the system detects a networking device, the system starts an NDIS driver stack for the device. The device can be a virtual device or a physical device. In either case, a driver stack start operation proceeds as follows:

1. The system loads and initializes the drivers if they are not already loaded.

   It does not load the drivers in any particular order.

2. The system calls each driver's **DriverEntry** function.

   After **DriverEntry** returns:

   - The miniport adapter for the device is in the Halted state.
   - The filter modules are in the Detached state.
   - The protocol binding is in the Unbound state.

3. The system requests NDIS to start the miniport adapter.

   To initialize the miniport adapter, NDIS calls the miniport driver's *MiniportInitializeEx* function. If *MiniportInitializeEx* is successful, the miniport adapter enters the Paused state.

4. NDIS attaches the filter modules, beginning with the module that is closest to the miniport driver and progressing to the top of the driver stack.

   To request the driver to attach a filter module to the driver stack, NDIS calls a filter driver's *FilterAttach* function. If each attach operation is successful, the filter module enters the Paused state.

5. After all the underlying drivers are in the Paused state, NDIS calls the protocol driver's *ProtocolBindAdapterEx* function.

   Then the protocol driver binding enters the Opening state. The protocol driver calls the **NdisOpenAdapterEx** function to open the binding with the miniport adapter.

6. NDIS allocates the necessary resources for the binding and calls the protocol driver's *ProtocolOpenAdapterCompleteEx* function.

   The binding enters the Paused state.

7. To complete the bind operation, the protocol driver calls the **NdisCompleteBindAdapterEx** function.

8. NDIS restarts the driver stack. For more information about restarting the driver stack, see Restarting a Driver Stack.

Stopping a Driver Stack

Article • 12/15/2021

If a device is removed, NDIS stops a driver stack. A driver stack stop operation proceeds as follows:

1. NDIS pauses the driver stack. For more information about pausing the driver stack, see Pausing a Driver Stack.

2. NDIS calls the protocol driver's *ProtocolUnbindAdapterEx* function.

   The binding enters the Closing state. After outstanding OID and send requests are complete and all receive data is returned, the binding enters the Unbound state.

3. NDIS detaches all the filter modules, beginning from the top of the stack and progressing down to the miniport driver.

   After NDIS calls a filter driver's *FilterDetach* function and the filter driver releases all the resources for a filter module, the filter module is in the Detached state.

4. NDIS halts the miniport adapter.

   After NDIS calls the miniport driver's *MiniportHaltEx* function, the miniport driver releases all the resources for the miniport adapter and the miniport adapter is in the Halted state.

5. If all of a filter driver's modules are detached, the system can unload the filter driver.

6. If all the miniport adapters that a miniport driver manages are halted, the system can unload the miniport driver.

Pausing a Driver Stack

Article • 12/15/2021

NDIS pauses a driver stack to complete operations such as inserting a filter module or adding a binding. In general, a driver stack pause operation proceeds as follows:

1. NDIS sends a PnP pause event to the protocol driver.

   The binding enters the Pausing state. After all outstanding send requests are complete, the protocol driver completes the PnP event. The binding is in the Paused state.

2. NDIS pauses all the filter modules, beginning at the top of the stack and progressing down to the miniport driver.

   After NDIS calls the filter driver's *FilterPause* function, the filter module enters the Pausing state. After NDIS returns all outstanding receive indications, and all outstanding send operations are complete, the filter module enters the Paused state.

3. NDIS pauses the miniport adapter.

   After NDIS calls the miniport driver's *MiniportPause* function, the miniport adapter enters the Pausing state. After NDIS returns all outstanding receive indications, the miniport adapter enters the Paused state.

Note NDIS drivers cannot fail a pause request. You should log any errors that occur.

Restarting a Driver Stack

Article • 12/15/2021

NDIS restarts a driver stack after operations such as inserting a filter module or adding a binding. A driver stack restart operation proceeds as follows:

1. NDIS restarts the miniport adapter.

   After NDIS calls the miniport driver's **MiniportRestart** function, the miniport adapter enters the Restarting state. The miniport driver prepares to resume send and receive operations. If the preparation fails, the miniport adapter returns to the Paused state. After the driver is ready to resume send and receive operations, the miniport adapter enters the Running state.

2. NDIS restarts the filter modules, beginning at the bottom of the driver stack and progressing up to the protocol driver.

   After NDIS calls a filter driver's **FilterRestart** function, the filter module enters the Restarting state. The filter driver prepares to resume send and receive operations. If the preparation fails, the module returns to the Paused state. After the driver is ready to resume send and receive operations, the filter module enters the Running state.

3. NDIS sends a PnP restart event to the protocol driver.

   The binding enters the Restarting state. The protocol driver prepares to resume send and receive operations. If the preparation fails, the binding returns to the Paused state. After the protocol driver is ready to resume send and receive operations, the binding enters the Running state.

Modifying a Running Driver Stack

Article • 12/15/2021

NDIS modifies a driver stack for operations such as inserting, removing, or reconfiguring a filter module. NDIS can activate or deactivate the bypass mode in a filter module. For more information about bypass mode in filter drivers, see Data Bypass Mode.

Note If filter driver entry points change (that is, because of bypass mode), NDIS pauses and restarts the driver stack. Pause and restart could cause some network packets to be dropped on the transmit path, or receive path. Network protocols that provide a reliable transport mechanism might retry the network I/O operation in the case of a lost packet, but other protocols that do not guarantee reliability do not retry the operation.

NDIS modifies a running driver stack as follows:

1. NDIS pauses the driver stack.

   For more information, see Pausing a Driver Stack.

2. NDIS modifies the stack.

   For example, to add a filter module, NDIS determines where to insert the new filter module into the stack and creates, inserts, and attaches the filter module.

3. When a filter module is inserted or deleted, the characteristics of the driver stack might change. In this case, NDIS sends a Plug and Play event notification to all of the protocol bindings and filter modules in the driver stack to notify the drivers of this change.

4. NDIS restarts the driver stack.

   For more information, see Restarting a Driver Stack.

NET_BUFFER Architecture

Article • 12/15/2021

This section provides high level information about **NET_BUFFER** structures and related data structures and functions. NET_BUFFER structures provide an efficient means to package and manage network data.

The following topics are included in this section:

Network Data Structures

Retreat and Advance Operations

Obtaining Pool Handles

Dispatch IRQL Tracking

Send and Receive Operations

Ethernet Send and Receive Operations

Derived NET_BUFFER_LIST Structures

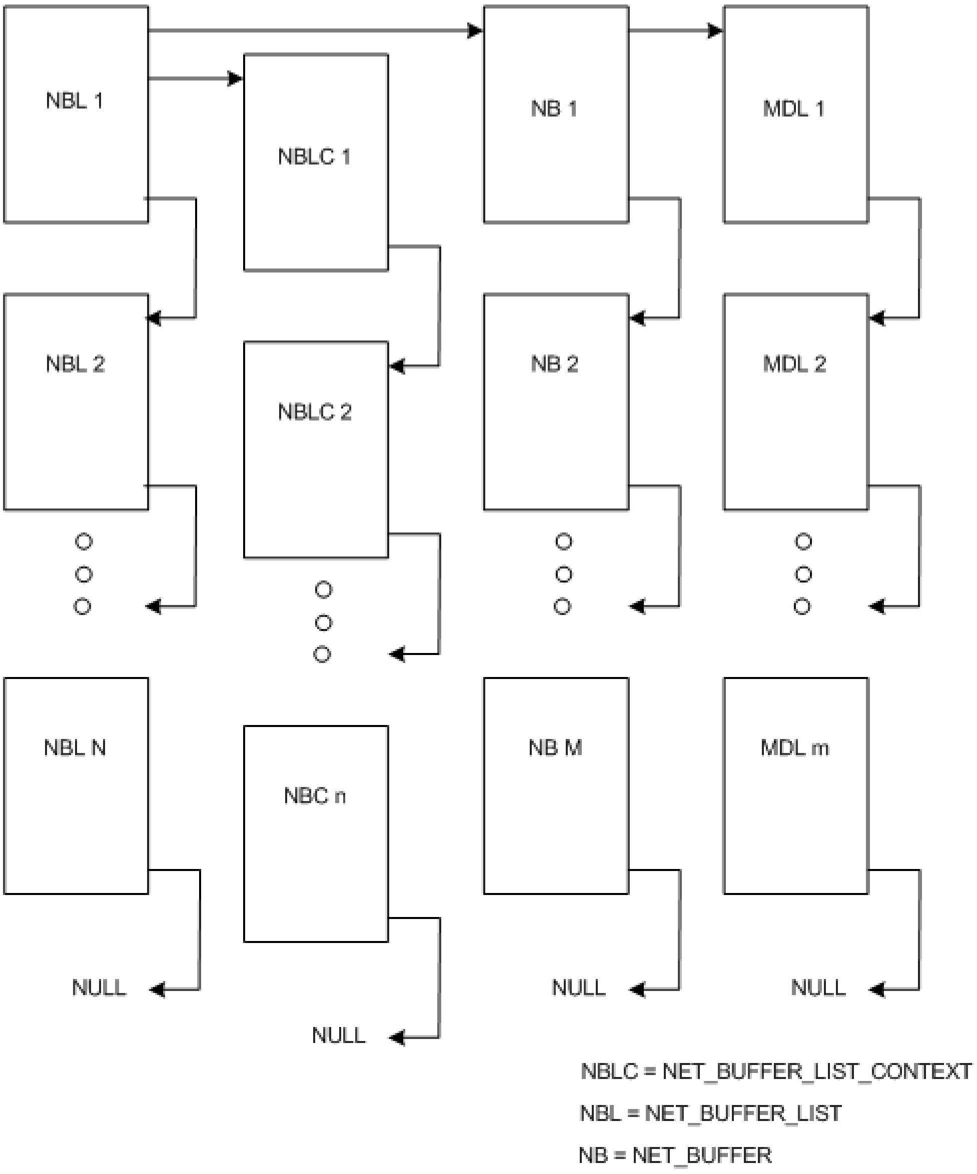
Network Data Structures

Article • 12/15/2021

Network data consists of packets of data that are sent or received over the network. NDIS provides data structures to describe and organize such data. The primary network data structures for NDIS 6.0 and later are:

- NET_BUFFER
- NET_BUFFER LIST
- NET_BUFFER_LIST_CONTEXT

The following figure illustrates the relationships between these structures.



In NDIS 6.0 and later, the NET_BUFFER is the basic building block for packaging network data. Each NET_BUFFER structure has an MDL chain. The MDLs map the addresses of data buffers to the data space that the NET_BUFFER structures specify. This data mapping is identical to the MDL chains that NDIS 5.*x* and earlier drivers use in the **NDIS_PACKET** structure. NDIS provides functions to manipulate the MDL chain.

Multiple NET_BUFFER structures can be attached to a NET_BUFFER_LIST structure. The NET_BUFFER structures are organized as a NULL-terminated singly linked list. Only the driver that originates a NET_BUFFER_LIST structure, or NDIS, should modify the linked list directly to insert and delete NET_BUFFER structures.

NET_BUFFER LIST structures contain information that describes all the **NET_BUFFER** structures that are attached to a list. If a driver requires additional space for context information, the driver can store such information in the NET_BUFFER_LIST_CONTEXT structures. NDIS provides functions to allocate, free and access the data in the NET_BUFFER_LIST_CONTEXT structures.

Multiple NET_BUFFER_LIST structures can be attached to form a list of NET_BUFFER_LIST structures. The NET_BUFFER_LIST structures are organized as a NULL-terminated singly linked list. Drivers can modify the linked list directly to insert and delete NET_BUFFER_LIST structures.

Related topics

NET_BUFFER

NET_BUFFER Structure

NET_BUFFER LIST

NET_BUFFER_LIST Structure

NET_BUFFER_LIST_CONTEXT

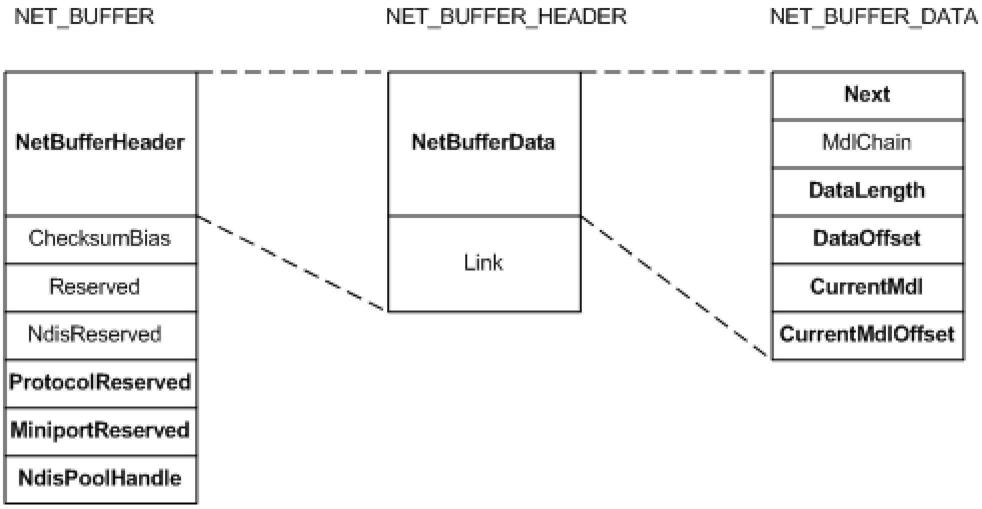
NET_BUFFER_LIST_CONTEXT Structure

NET_BUFFER Structure

Article • 12/15/2021

NDIS 6.0 and later NET_BUFFER structures are analogous to the NDIS_PACKET structures used by NDIS 5.*x* and earlier drivers. Each NET_BUFFER structure packages a packet of network data.

The following figure shows the fields in a NET_BUFFER structure.



The NET_BUFFER structure includes a NET_BUFFER_HEADER structure in the **NetBufferHeader** member. The NET_BUFFER_HEADER structure includes a NET_BUFFER_DATA structure in the **NetBufferData** member. You should use NDIS macros to access NET_BUFFER structure members. For a complete list of these macros, see the NET_BUFFER structure reference page.

Some of the NET_BUFFER structure members are only used by NDIS. The members that drivers typically use are:

ProtocolReserved
Reserved for use by protocol drivers.

MiniportReserved
Reserved for use by miniport drivers.

NdisPoolHandle
Specifies a pool handle that identifies the NET_BUFFER pool from which the NET_BUFFER structure was allocated.

Next
Specifies a pointer to the next NET_BUFFER structure in a linked list of NET_BUFFER structures. If this is the last NET_BUFFER structure in the list, this member is **NULL**.

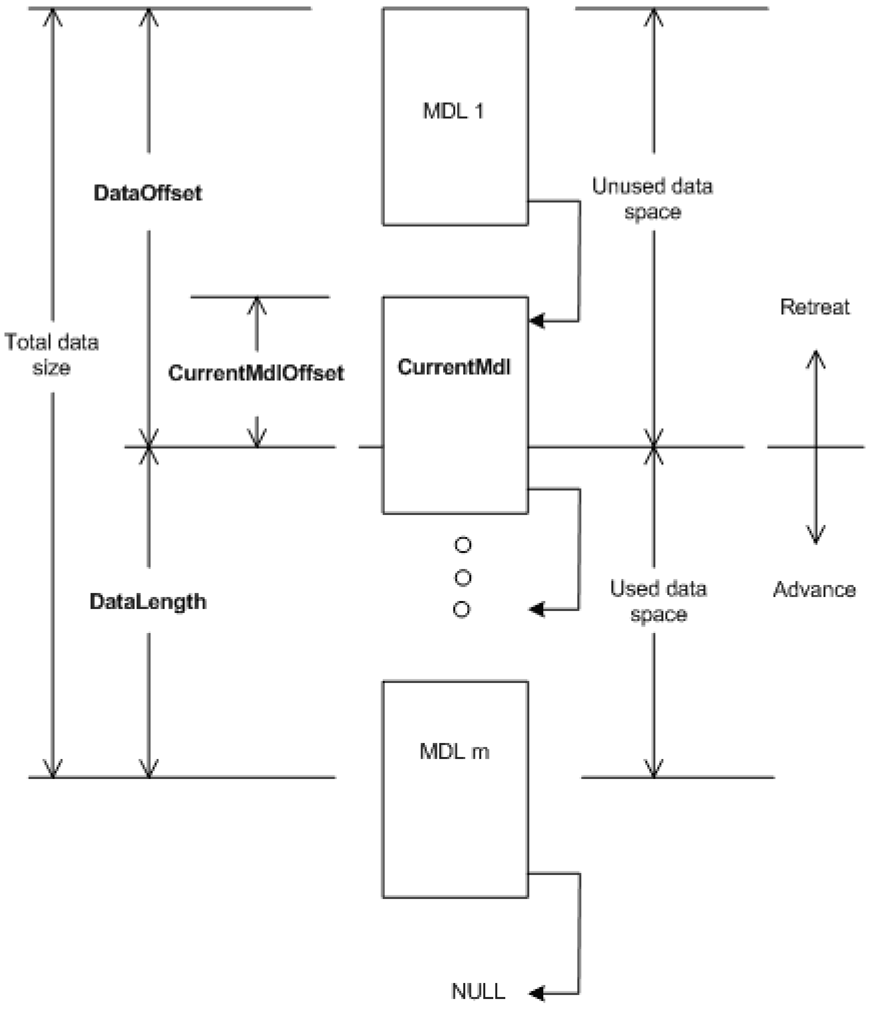
DataLength
Specifies the length in bytes of the network data in the MDL chain.

DataOffset
Specifies the offset, in bytes, from the start of memory in the MDL chain to the start of the network data in the MDL chain.

CurrentMdl
Specifies a pointer to the first MDL that the current driver is using. This pointer provides an optimization that improves performance by skipping over any MDLs that the current driver is not using.

CurrentMdlOffset
Specifies the offset, in bytes, to the beginning of the used data space in the MDL that is specified by the **CurrentMdl** member of the NET_BUFFER structure.

The following figure shows the relationship between the **CurrentMdl**, **CurrentMdlOffset**, **DataOffset**, and **DataLength** members and the data space.



NDIS provides functions to manage the data space in the MDL chain. How drivers use the data space changes dynamically with the current driver. Sometimes there is data space that is currently unused by the current driver. Although the *unused data space* is currently unused, it can contain valid data. For example, on the receive path, the *unused data space* can contain header information that was used by a lower level driver.

Drivers perform retreat and advance operations to increase and decrease the *used data space*. For more information about retreat and advance operations, see Retreat and Advance Operations.

The following terms and definitions describe elements of the **NET_BUFFER** data space:

Used data space
Used data space contains data that the current driver is using at the current time. Drivers increase *used data space* with retreat operations and reduce *used data space* with advance operations.

Unused data space
The current driver is not using this data space at the current time.

Total data size
The total data size is the sum of the size of the *used data space* and *unused data space*. To calculate the total size, add the **DataOffset** to the **DataLength** .

Retreat
Retreat operations increase the size of the *used data space*.

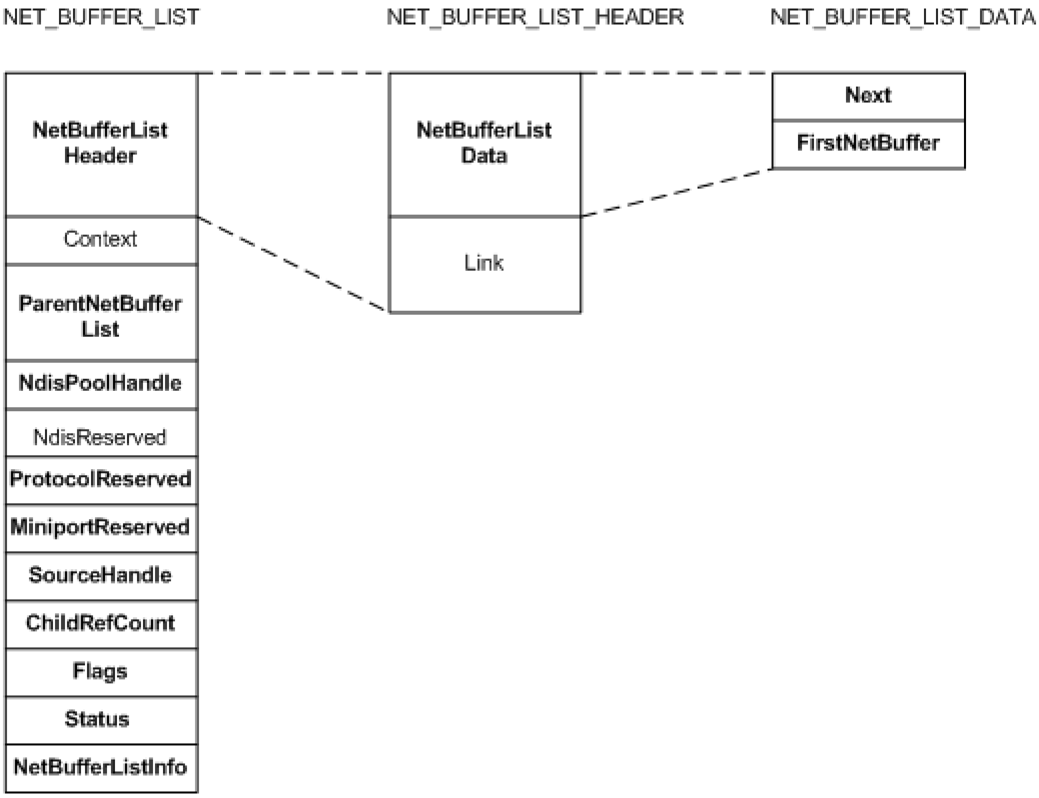
Advance
Advance operations decrease the size of the *used data space*.

NET_BUFFER_LIST Structure

Article • 12/15/2021

A NET_BUFFER_LIST structure packages a linked list of NET_BUFFER structures.

The following figure shows the fields in a NET_BUFFER_LIST structure.



The NET_BUFFER_LIST structure includes a NET_BUFFER_LIST_HEADER structure in the **NetBufferListHeader** member. The NET_BUFFER_LIST_HEADER structure includes a NET_BUFFER_LIST_DATA structure in the **NetBufferListData** member. You should use NDIS macros to access NET_BUFFER_LIST structure members. For more information about these macros, see the NET_BUFFER_LIST structure reference page.

Some of the members are only used by NDIS. The members that drivers are most likely to use are defined in the following list:

ParentNetBufferList

If a NET_BUFFER_LIST structure is a child that was derived from a parent(cloned, fragmented, or reassembled), **ParentNetBufferList** specifies a pointer to the parent NET_BUFFER_LIST structure. Otherwise, this parameter is **NULL**.

NdisPoolHandle

Specifies a pool handle that identifies the NET_BUFFER_LIST pool from which the NET_BUFFER_LIST structure was allocated.

ProtocolReserved
Reserved for use by protocol drivers.

MiniportReserved
Reserved for use by miniport drivers.

SourceHandle
A handle that NDIS provided to the driver in a binding or attaching operation by using one of the following driver-supplied routines:

Miniport Driver
MiniportInitializeEx

Protocol Driver
ProtocolBindAdapterEx

Filter Driver
FilterAttach

NDIS uses **SourceHandle** to return the NET_BUFFER_LIST structure to the driver that sent the NET_BUFFER_LIST structure. NDIS drivers should not read this handle.

ChildRefCount
If a **NET_BUFFER_LIST** structure is a parent (has children derived by clone, fragment, or reassemble operations), **ChildRefCount** specifies the number of existing children. Otherwise, this parameter is zero.

Flags
Reserved for future specification of attributes for the NET_BUFFER_LIST structure. There are currently no flags available to drivers.

Status
Specifies the final completion status of a network data operation for this NET_BUFFER_LIST structure. Miniport drivers write this value before completing a send operation.

NetBufferListInfo
Specifies **NET_BUFFER_LIST** structure information that is common to all **NET_BUFFER** structures in the list. This information is often referred to as "out-of-band (OOB) data."

Next
Specifies a pointer to the next NET_BUFFER_LIST structure in a linked list of NET_BUFFER_LIST structures. If a NET_BUFFER_LIST structure is the last structure in the list, this member is **NULL**.

FirstNetBuffer
Specifies a pointer to the first NET_BUFFER structure in a linked list of NET_BUFFER structures that is associated with this NET_BUFFER_LIST structure.

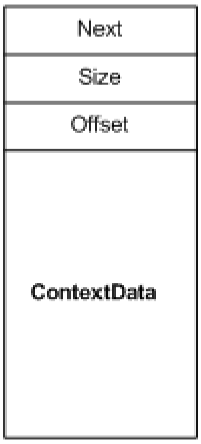
Note **Context** is a pointer to a **NET_BUFFER_LIST_CONTEXT** structure. NDIS provides macros and functions to manipulate the data at **Context** . For more information about the NET_BUFFER_LIST_CONTEXT structure, see NET_BUFFER_LIST_CONTEXT Structure.

NET_BUFFER_LIST_CONTEXT Structure

Article • 06/15/2022

NDIS drivers use NET_BUFFER_LIST_CONTEXT structures to store additional data that is associated with a NET_BUFFER_LIST structure. The **Context** member of the NET_BUFFER_LIST structure is a pointer to a NET_BUFFER_LIST_CONTEXT structure. The information stored in the NET_BUFFER_LIST_CONTEXT structures is opaque to NDIS and other drivers in the stack.

The following figure shows the fields in a NET_BUFFER_LIST_CONTEXT structure.



The NET_BUFFER_LIST_CONTEXT structure includes **ContextData** member that contains the context data. This data can be any context information that a driver requires for the NET_BUFFER_LIST structure.

Drivers should use the following NDIS macros and functions to access and manipulate members in a NET_BUFFER_LIST_CONTEXT structure:

NdisAllocateNetBufferListContext

NdisFreeNetBufferListContext

NET_BUFFER_LIST_CONTEXT_DATA_START

NET_BUFFER_LIST_CONTEXT_DATA_SIZE

Retreat and Advance Operations

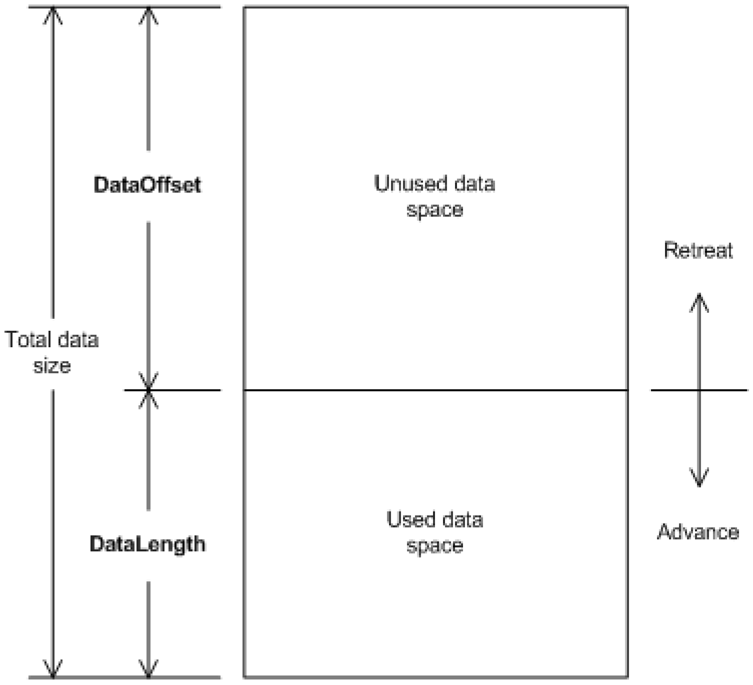
Article • 12/15/2021

NDIS provides retreat and advance functions to manipulate **NET_BUFFER** structures. Retreat operations make more *used data space* available to the current driver. Advance operations release *used data space*.

Retreat operations are required during send operations or when a driver returns received data to an underlying driver. For example, during a send operation, a driver can call the **NdisRetreatNetBufferDataStart** function to make room for header data.

Advance operations are required when a send operation is complete or when a driver receives data from an underlying driver. For example, during a receive operation, a driver can call the **NdisAdvanceNetBufferDataStart** function to skip over the header data that was used by a lower level driver. In this case, the header data remains in the buffer in the *unused data space*.

The following figure shows the relationship between the network data and these operations.



The following topics provide more information about advance and retreat operations:

Retreat Operations

Advance Operations

Retreat Operations

Article • 12/15/2021

Retreat operations can increase the size of the used data space in a NET_BUFFER structure or in all of the NET_BUFFER structures in a NET_BUFFER_LIST structure.

NDIS provides the following retreat functions:

NdisRetreatNetBufferDataStart

NdisRetreatNetBufferListDataStart

Retreat operations can sometimes allocate MDLs that are associated with a NET_BUFFER structure. To provide the mechanism for allocating MDLs, a driver can provide an optional entry point for a NetAllocateMdl function. If the entry point is **NULL**, NDIS uses a default method to allocate MDLs. MDLs must be freed within a NetFreeMdl function that provides the reciprocal of the mechanism that was used to allocate the MDL.

To obtain the new **DataLength**, NDIS adds the driver-specified *DataOffsetDelta* to the current **DataLength** . If the size of the *unused data space* is greater than the *DataOffsetDelta*, a retreat operation reduces the **DataOffset** . In this case, the new **DataOffset** is the current **DataOffset** minus the *DataOffsetDelta* .

If the *DataOffsetDelta* is greater than **DataOffset**, a retreat operation allocates new data space. In this case, NDIS adjusts the **DataOffset** accordingly.

For send operations, NDIS allocates memory if there isn't enough *unused data space* to satisfy a retreat request. If no memory allocation is required, NDIS simply adjusts the **DataOffset** and **DataLength** . For better performance, drivers should allocate enough total data size before sending to accommodate the retreat operations of all the underlying drivers.

For the receive return case, NDIS simply adjusts the **DataOffset** and **DataLength** accordingly. The retreat operation reverses the advance operation that took place during receive processing. After the retreat operation, the *used data space* contains the header data that underlying drivers used during receive processing.

Advance Operations

Article • 12/15/2021

Advance operations decrease the size of the used data space in a NET_BUFFER structure or in all of the NET_BUFFER structures in a NET_BUFFER_LIST structure.

Drivers use the following advance functions:

NdisAdvanceNetBufferDataStart

NdisAdvanceNetBufferListDataStart

Advance operations can sometimes free MDLs that are associated with a NET_BUFFER structure. To provide the mechanism for freeing MDLs, a driver can provide an optional entry point for a NetFreeMdl functions. If the entry point is **NULL**, NDIS uses a default method to allocate MDLs. MDLs must only be freed within a *NetFreeMdl* using that reciprocal of the mechanism that was used to allocate the MDL in the NetAllocateMdl function.

To obtain the new **DataLength**, NDIS subtracts the driver-specified *DataOffsetDelta* from the current **DataLength** . If a previous retreat operation allocated new data space, the advance operation can free such previously allocated memory. If an advance operation does not free memory, NDIS simply adds the *DataOffsetDelta* to the current **DataOffset** to obtain the new **DataOffset** . If the advance operation freed memory, NDIS adjusts the **DataOffset** accordingly.

For the send complete case, advance operations can free memory that was allocated in previous retreat operations. For better performance, drivers should allocate enough total data size before sending to accommodate the retreat operations of all the underlying drivers.

For the receive indication case, advance operations simply adjust the **DataOffset** and **DataLength** accordingly. After the advance operation, the headers of lower layers remain in the *unused data space*.

Obtaining Pool Handles

Article • 12/15/2021

The following NDIS pool allocation functions require a handle to allocate resources:

- NdisAllocateNetBufferPool
- NdisAllocateNetBufferListPool

NDIS 6.0 drivers obtain a handle as follows:

Protocol drivers
Protocol drivers call the NdisRegisterProtocolDriver function to obtain a handle.

Miniport drivers
NDIS calls the *MiniportInitializeEx* function to pass the handle to the miniport driver.

Intermediate drivers
Intermediate drivers call the NdisRegisterProtocolDriver function to obtain a handle for pools used in send operations and NDIS calls *MiniportInitializeEx* to pass the handle to the intermediate driver for pools used in receive operations.

Filter drivers
NDIS calls the *FilterAttach* function to pass the handle to the filter driver.

Other drivers
If a driver cannot obtain a handle through one of the preceding methods, the driver can call the NdisAllocateGenericObject function to get a handle.

Dispatch IRQL Tracking

Article • 12/15/2021

To improve system performance, some NDIS functions (for example, the *MiniportSendNetBufferLists* function) include a dispatch level flag that indicates the current IRQL. The proper use of the dispatch level flag can help to avoid unnecessary attempts to set the IRQL.

There are other flags that control other attributes, but the names for the dispatch level flags are:

NDIS_SEND_FLAGS_DISPATCH_LEVEL

NDIS_SEND_COMPLETE_FLAGS_DISPATCH_LEVEL

NDIS_RECEIVE_FLAGS_DISPATCH_LEVEL

NDIS_RETURN_FLAGS_DISPATCH_LEVEL

NDIS_RWL_AT_DISPATCH_LEVEL

The caller must determine the dispatch level flag setting from the known current IRQL, not by testing the IRQL. For example, you know the IRQL because it is a fixed characteristic of the driver design, or the driver saved the current IRQL.

If the known current IRQL is DISPATCH_LEVEL, the caller should set this flag. If the current IRQL is unknown, or the caller is not running at DISPATCH_LEVEL, the caller should clear this flag. If the caller is NDIS, the called function should test this flag to avoid changing the IRQL.

Drivers should not test for the IRQL to determine the value for the dispatch level flag. Testing would defeat the purpose of the flag. If necessary, the called function can simply do the testing itself. How a driver determines that it should or should not set the flag is left to the design of the particular driver.

Send and Receive Operations

Article • 12/15/2021

In a single function call, NDIS 6.0 drivers can send multiple **NET_BUFFER_LIST** structures with multiple **NET_BUFFER** structures on each NET_BUFFER_LIST structure. Also, NDIS drivers can indicate completed send operations for multiple NET_BUFFER_LIST structures with multiple NET_BUFFER structures on a NET_BUFFER_LIST structure.

In the receive path, miniport drivers can use a list of NET_BUFFER_LIST structures to indicate receives. Each NET_BUFFER_LIST indicated by a miniport driver contains one NET_BUFFER structure. However, Native 802.11 drivers can have more than one NET_BUFFER structure. Because a different protocol binding can process each NET_BUFFER_LIST structure, NDIS can return each NET_BUFFER_LIST structure to the miniport driver independently.

To support NDIS 5.*x* and earlier drivers, NDIS provides a translation layer between the **NDIS_PACKET**-based and NET_BUFFER-based interfaces. NDIS performs the necessary conversion between **NET_BUFFER** structures and NDIS_PACKET structures. To avoid performance degradation due to translation, NDIS drivers must be updated to use NET_BUFFER structures and should support multiple **NET_BUFFER_LIST** structures in all data paths.

This section includes the following topics:

Sending Network Data

Canceling a Send Operation

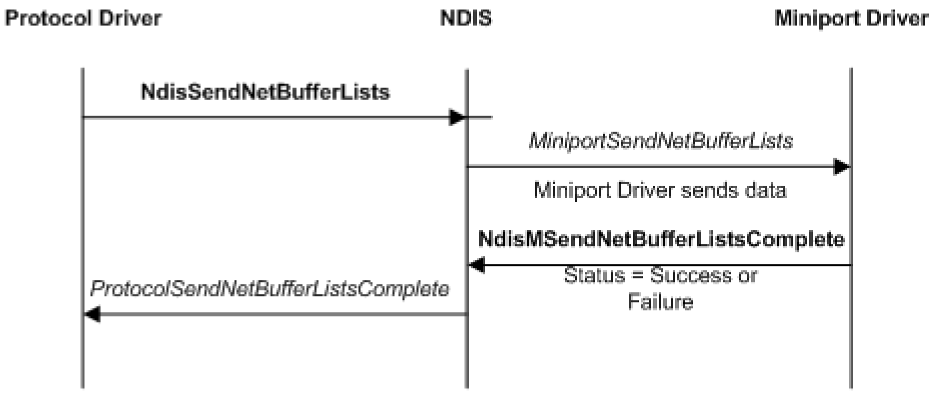
Receiving Network Data

Looping Back NDIS Packets

Sending Network Data

Article • 12/15/2021

The following figure illustrates a basic send operation, which involves a protocol driver, NDIS, and a miniport driver.



Protocol Driver
NDIS
Miniport Driver
NdisSendNetBufferLists
MiniportSendNetBufferLists
Miniport Driver sends data
NdisMSendNetBufferListsComplete
Status = Success or Failure
ProtocolSendNetBufferListsComplete

Protocol drivers call the **NdisSendNetBufferLists** function to send **NET_BUFFER_LIST** structures on a binding. NDIS calls the miniport driver's *MiniportSendNetBufferLists* function to forward the NET_BUFFER_LIST structures to an underlying miniport driver.

All NET_BUFFER-based send operations are asynchronous. The miniport driver calls the **NdisMSendNetBufferListsComplete** function with an appropriate status code when it is done. The sending of each NET_BUFFER_LIST structure can be completed individually. NDIS calls the protocol driver's **ProtocolSendNetBufferListsComplete** function each time the miniport driver calls **NdisMSendNetBufferListsComplete**.

Protocol drivers can reclaim the ownership of the NET_BUFFER_LIST structures and all associated structures and data as soon as the NDIS calls the protocol driver's *ProtocolSendNetBufferListsComplete* function.

The miniport driver or NDIS can return the **NET_BUFFER_LIST** structures in any order. Protocol drivers are guaranteed that the list of **NET_BUFFER** structures attached to each NET_BUFFER_LIST structure has not been modified.

Any NDIS driver can separate the NET_BUFFER structures in a NET_BUFFER_LIST structure. Any NDIS driver can also separate the MDLs in a NET_BUFFER structure. However, the driver must always return the NET_BUFFER_LIST structures with the NET_BUFFER structures and MDLs in the original form. For example, an intermediate driver might separate a NET_BUFFER_LIST into two new NET_BUFFER_LIST structures and pass on part of the original data to the next driver. However, when the intermediate driver completes the processing of the original NET_BUFFER_LIST it must return the complete NET_BUFFER_LIST with the original NET_BUFFER structures and MDLs.

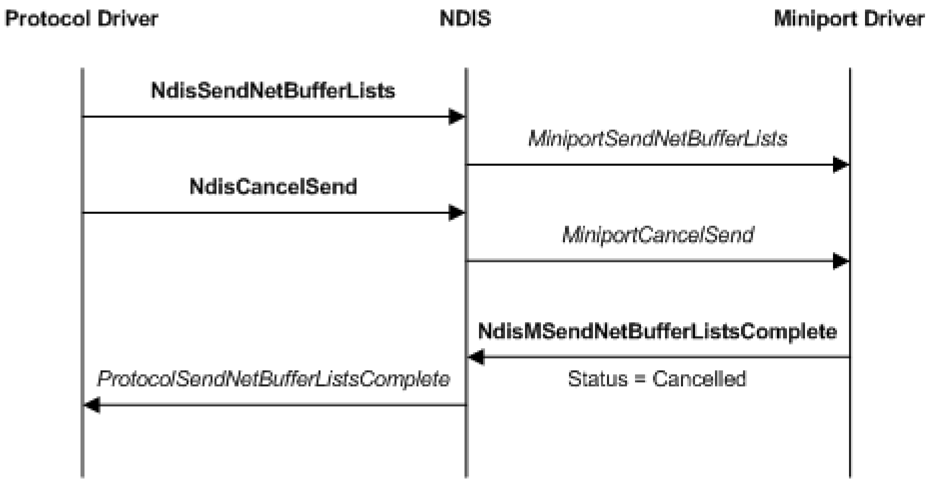
Protocol drivers set the **SourceHandle** member in the NET_BUFFER_LIST structure to the *NdisBindingHandle* that NDIS provided in a call to the **NdisOpenAdapterEx** function. NDIS uses the **SourceHandle** member to return the NET_BUFFER_LIST structures to the protocol driver that sent the NET_BUFFER_LIST structures.

Intermediate drivers also set the **SourceHandle** member in the NET_BUFFER_LIST structure to the *NdisBindingHandle* value that NDIS provided in a call to **NdisOpenAdapterEx**. If an intermediate driver forwards a send request, the driver must save the **SourceHandle** value that the overlying driver provided before it writes to the **SourceHandle** member. When NDIS returns a forwarded NET_BUFFER_LIST structure to the intermediate driver, the intermediate driver must restore the **SourceHandle** that it saved.

Canceling a Send Operation

Article • 12/15/2021

The following figure illustrates canceling a send operation.



A driver calls the NDIS_SET_NET_BUFFER_LIST_CANCEL_ID macro for each NET_BUFFER_LIST structure that it passes to lower-level drivers for transmission. The NDIS_SET_NET_BUFFER_LIST_CANCEL_ID function marks the specified packet with a cancellation identifier.

Before assigning cancellation IDs to packets, a driver should call NdisGeneratePartialCancelId to obtain the high-order byte of each cancellation ID that it assigns. This ensures that the driver does not duplicate cancellation IDs assigned by other drivers in the system. Drivers typically call **NdisGeneratePartialCancelId** once from the **DriverEntry** routine; however, drivers can obtain more than one partial cancellation identifier by calling **NdisGeneratePartialCancelId** more than once.

To cancel the pending transmission of data in a marked NET_BUFFER_LIST structure, a driver passes the cancellation ID to the NdisCancelSendNetBufferLists function. Drivers can obtain a NET_BUFFER_LIST structure's cancellation ID by calling the NDIS_GET_NET_BUFFER_LIST_CANCEL_ID macro.

If a driver marks all NET_BUFFER_LIST structures with the same cancellation identifier, it can cancel all pending transmissions with a single call to **NdisCancelSendNetBufferLists**. If a driver marks all NET_BUFFER_LIST structures within a subgroup of NET_BUFFER_LIST structures with a unique identifier, it can cancel all pending transmissions within that subgroup with a single call to **NdisCancelSendNetBufferLists**.

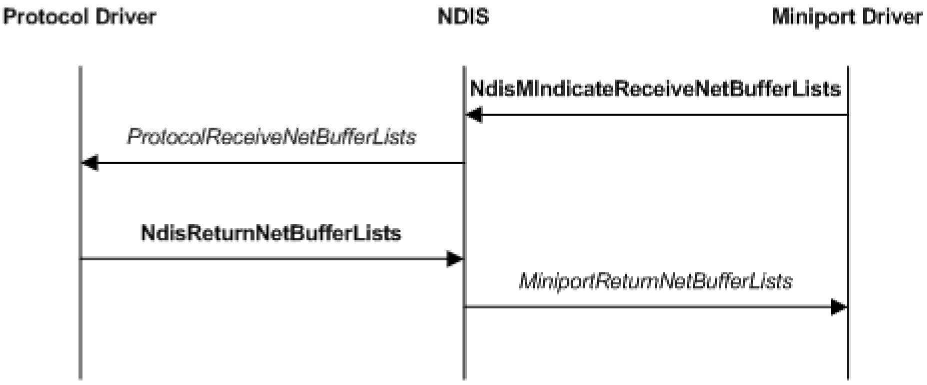
NDIS calls the *MiniportCancelSend* function of the appropriate lower-level driver on the binding. After aborting the pending transmission, the underlying miniport driver calls the NdisMSendNetBufferListsComplete function, to return the NET_BUFFER_LIST structures and a completion status of NDIS_STATUS_SEND_ABORTED. NDIS, in turn, calls the appropriate driver's **ProtocolSendNetBufferListsComplete** function.

In its *ProtocolSendNetBufferListsComplete* function, a protocol driver can call NDIS_SET_NET_BUFFER_LIST_CANCEL_ID with *CancelId* set to **NULL**. This prevents the NET_BUFFER_LIST from inadvertently being used again with a stale cancellation ID.

Receiving Network Data

Article • 12/15/2021

The following figure illustrates a basic receive operation, which involves a miniport driver, NDIS, and a protocol driver.


Protocol Driver
NDIS
Miniport Driver
NdisMIndicateReceiveNetBufferLists
ProtocolReceiveNetBufferLists
NdisReturnNetBufferLists
MiniportReturnNetBufferLists

Miniport drivers call the NdisMIndicateReceiveNetBufferLists function to indicate NET_BUFFER structures to higher level drivers. Every NET_BUFFER structure should usually be attached to a separate NET_BUFFER_LIST structure. This allows protocol drivers to create a subset of the original list of NET_BUFFER_LIST structures and forward them to different clients. Some drivers, for example native IEEE 802.11 miniport drivers, might attach more than one NET_BUFFER structure to a NET_BUFFER_LIST structure.

After linking all the NET_BUFFER_LIST structures, a miniport driver passes a pointer to the first NET_BUFFER_LIST structure in the list to the **NdisMIndicateReceiveNetBufferLists** function. NDIS examines the NET_BUFFER_LIST structures and it calls the ProtocolReceiveNetBufferLists function of each protocol driver that is associated with the NET_BUFFER_LIST structures. NDIS passes a subset of the list that includes only the NET_BUFFER_LIST structures that are associated with the correct binding to each protocol driver. NDIS matches the **NetBufferListFrameType** value that is specified in the NET_BUFFER_LIST structure to the frame type that each protocol driver registers.

If the NDIS_RECEIVE_FLAGS_RESOURCES flag in the *ReceiveFlags* parameter that is passed to a protocol driver's *ProtocolReceiveNetBufferLists* function is set, NDIS regains the ownership of the NET_BUFFER_LIST structures immediately after the *ProtocolReceiveNetBufferLists* call returns.

Note If the NDIS_RECEIVE_FLAGS_RESOURCES flag is set, the protocol driver must retain the original set of NET_BUFFER_LIST structures in the linked list. For example, when this flag is set the driver might process the structures and indicate them up the stack one at a time but before the function returns it must restore the original linked list.

If the NDIS_RECEIVE_FLAGS_RESOURCES flag in the *ReceiveFlags* parameter that is passed to a protocol driver's *ProtocolReceiveNetBufferLists* function is not set, the protocol driver can retain ownership of the NET_BUFFER_LIST structures. In this case, the protocol driver must return the NET_BUFFER_LIST structures by calling the **NdisReturnNetBufferLists** function.

If a miniport driver is running low on receive resources, it can set the NDIS_RECEIVE_FLAGS_RESOURCES flag in the *ReceiveFlags* parameter in the call to **NdisMIndicateReceiveNetBufferLists**. In that case, the driver can reclaim the ownership of all the indicated **NET_BUFFER_LIST** structures and embedded **NET_BUFFER** structures as soon as **NdisMIndicateReceiveNetBufferLists** returns. Indicating NET_BUFFER structures with the NDIS_RECEIVE_FLAGS_RESOURCES flag set forces the protocol drivers to copy the data and therefore should be avoided. A miniport driver should detect when it is about to run out of receive resources and take any steps that are necessary to avoid this situation.

NDIS calls a miniport driver's *MiniportReturnNetBufferLists* function after the protocol driver calls **NdisReturnNetBufferLists**.

Note If a miniport driver indicates a NET_BUFFER_LIST structure with the NDIS_RECEIVE_FLAGS_RESOURCES flag set, that does not mean that NDIS will indicate the NET_BUFFER_LIST structure to the protocol driver with the same status. For example, NDIS could copy a NET_BUFFER_LIST structure with the NDIS_RECEIVE_FLAGS_RESOURCES flag set and indicate the copy to the protocol driver with the flag cleared.

NDIS can return **NET_BUFFER_LIST** structures to the miniport driver in any arbitrary order and in any combination. That is, the linked list of NET_BUFFER_LIST structures returned back to a miniport driver by a call to its *MiniportReturnNetBufferLists* function, can have NET_BUFFER_LIST structures from different previous calls to **NdisMIndicateReceiveNetBufferLists**.

Miniport drivers should set the **SourceHandle** member in the NET_BUFFER_LIST structures to the *MiniportAdapterHandle* that NDIS provided to the miniport driver in the *MiniportInitializeEx* function. Filter drivers must set the **SourceHandle** member of each NET_BUFFER_LIST structure that the filter driver originated to the filter's **NdisFilterHandle** that NDIS provided to the filter driver in the *FilterAttach* function. Filter drivers must not modify the **SourceHandle** member in any NET_BUFFER_LIST structures that were not originated by the filter driver.

Intermediate drivers also set the **SourceHandle** member in the NET_BUFFER_LIST structure to the *MiniportAdapterHandle* value that NDIS provided to the intermediate driver in the *MiniportInitializeEx* function. If an intermediate driver forwards a receive indication, the driver must save the **SourceHandle** value that the underlying driver provided before it writes to the **SourceHandle** member. When NDIS returns a forwarded NET_BUFFER_LIST structure to the intermediate driver, the intermediate driver must restore the **SourceHandle** that it saved.

Looping Back NDIS Packets

Article • 03/14/2023

If the NDIS_NBL_FLAGS_IS_LOOPBACK_PACKET flag in the **NblFlags** member of the NET_BUFFER_LIST structure is set, the packet is a loopback packet. Protocol drivers and filter drivers can check this flag to determine if a packet is a loopback packet.

NDIS loops packets back if all of the following three conditions are satisfied:

1. The underlying miniport adapter media type is **NdisMedium802_3** or **NdisMedium802_5**.

2. Any one of the following three conditions is satisfied:

   a. A protocol binding set the NDIS_PACKET_TYPE_PROMISCUOUS setting with the OID_GEN_CURRENT_PACKET_FILTER OID to specify its packet filter (and, for Windows 8 and later, did not set NDIS_PACKET_TYPE_NO_LOCAL in the same OID) and either of the following is true:

      - There is more than one binding to the miniport adapter.
      - There is a filter module attached to the miniport adapter and the filter module registered a receive handler.

   b. A protocol binding set the NDIS_PACKET_TYPE_ALL_LOCAL setting with the OID_GEN_CURRENT_PACKET_FILTER OID to specify its packet filter and either of the following is true.

      - There is more than one binding to the miniport adapter.
      - There is a filter module attached to the miniport adapter and the filter module registered a receive handler.

   c. The caller sets the NDIS_SEND_FLAGS_CHECK_FOR_LOOPBACK flag in the *SendFlags* parameter of the **NdisSendNetBufferLists** function.

3. The packet is acceptable as determined by the packet filter set with the OID_GEN_CURRENT_PACKET_FILTER OID for the miniport adapter. The following are some examples:

   - If the packet is a direct packet, the destination address in the packet must match the MAC address of the miniport adapter.
   - If the packet is a multicast packet, the packet filter must have NDIS_PACKET_TYPE_ALL_MULTICAST set or the destination address matches

one of the multicast address in the miniport adapter's multicast address list and the packet filter has NDIS_PACKET_TYPE_MULTICAST set.
- If the packet is a broadcast packet, the miniport adapter's packet filter must have NDIS_PACKET_TYPE_BROADCAST set.
- The miniport adapter's packet filter has NDIS_PACKET_TYPE_PROMISCUOUS or NDIS_PACKET_TYPE_ALL_LOCAL set.

A protocol binding receives loopback packets if either of the following is true:

1. The protocol binding is the original sender of the packet and NDIS_SEND_FLAGS_CHECK_FOR_LOOPBACK is set.

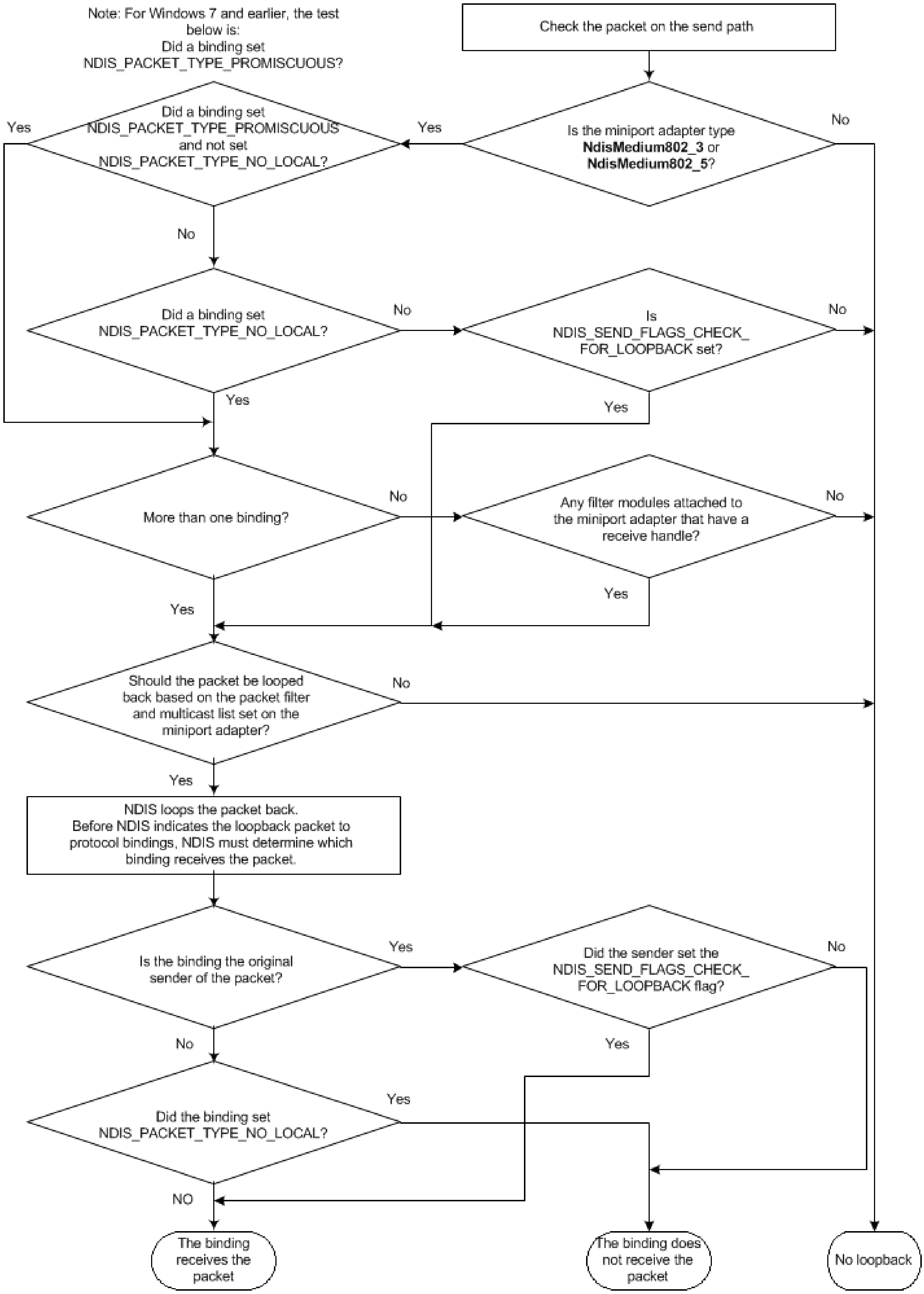
2. The protocol binding does not set NDIS_PACKET_TYPE_NO_LOCAL in the packet filter.

A protocol binding will not receive loopback packets if either of the following is true:

1. The protocol binding sets NDIS_PACKET_TYPE_NO_LOCAL in the packet filter and it is not the original sender for the packet.

2. The protocol binding is the original sender but NDIS_SEND_FLAGS_CHECK_FOR_LOOPBACK is not set in the *SendFlags* parameter in a call to the **NdisSendNetBufferLists** function.

The following figure shows the loopback algorithm logic flow.


Note: For Windows 7 and earlier, the test below is: Did a binding set NDIS_PACKET_TYPE_PROMISCUOUS?
Check the packet on the send path
Is the miniport adapter type **NdisMedium802_3** or **NdisMedium802_5**?
Yes
No
Did a binding set NDIS_PACKET_TYPE_PROMISCUOUS and not set NDIS_PACKET_TYPE_NO_LOCAL?
Yes
No
Did a binding set NDIS_PACKET_TYPE_NO_LOCAL?
No
Yes
Is NDIS_SEND_FLAGS_CHECK_ FOR_LOOPBACK set?
No
Yes
More than one binding?
No
Yes
Any filter modules attached to the miniport adapter that have a receive handle?
No
Yes
Should the packet be looped back based on the packet filter and multicast list set on the miniport adapter?
No
Yes
NDIS loops the packet back. Before NDIS indicates the loopback packet to protocol bindings, NDIS must determine which binding receives the packet.
Is the binding the original sender of the packet?
Yes
No
Did the sender set the NDIS_SEND_FLAGS_CHECK_ FOR_LOOPBACK flag?
No
Yes
Did the binding set NDIS_PACKET_TYPE_NO_LOCAL?
Yes
NO
The binding receives the packet
The binding does not receive the packet
No loopback

Ethernet Send and Receive Operations

Article • 12/15/2021

This section defines send and receive requirements for Ethernet drivers. For general information about send and receive operations, see Send and Receive Operations.

This section includes the following topics:

Sending Ethernet Frames

Indicating Received Ethernet Frames

Sending Ethernet Frames

Article • 12/15/2021

The Windows TCP/IP transport supports a set of requirements for sending Ethernet frames. Any driver (for example, a MUX intermediate driver or filter driver) that originates send requests or modifies the send requests of overlying drivers must support the requirements that the TCP/IP transport implements.

Note If any driver in a driver stack does not follow these requirements, underlying miniport drivers, MUX intermediate drivers, and filter drivers might behave unpredictably.

For Ethernet send requests, drivers must support these requirements:

- If a driver originates a send request, the driver should allocate a NET_BUFFER_LIST structure for the Ethernet frames. The **NetBufferListInfo** member in each NET_BUFFER_LIST structure must include the out-of-band (OOB) data that is required for the particular use. The OOB data applies to all of the NET_BUFFER structures that are associated with a NET_BUFFER_LIST structure.

- If a driver originates a send request, the driver should allocate one or more NET_BUFFER structures for the Ethernet frames and link these structures to the NET_BUFFER_LIST structure. Each NET_BUFFER structure that is linked to a NET_BUFFER_LIST structure describes a single Ethernet frame. The driver may chain multiple NET_BUFFER_LIST structures in a send request.

- All NET_BUFFER structures that are associated with a NET_BUFFER_LIST structure must have the same Ethernet frame type and IP protocol version (IPv4 or IPv6).

- All NET_BUFFER structures that are associated with a NET_BUFFER_LIST structure must have the same source and destination MAC addresses.

- If a driver is sending TCP or UDP frames, all of the NET_BUFFER structures that are associated with a NET_BUFFER_LIST structure must be associated with same TCP or UDP connection. **Note** Subject to the following requirements, transmitted Ethernet frames can be split. That is, multiple memory descriptor lists (MDLs) can be associated with a NET_BUFFER structure in a send request.

- Do not split the MAC header of the transmit Ethernet frame across multiple MDLs. Treat the Virtual LAN (VLAN) (or Priority) flag, if present, as part of the MAC header. Therefore, this flag must be in the same MDL as the rest of the MAC header.

- If a driver changes the links in the MDL chain in a NET_BUFFER structure or the NET_BUFFER chain in a NET_BUFFER_LIST structure, the driver must restore the links to the original configuration before it returns ownership of the NET_BUFFER_LIST to an overlying driver. However, drivers are not required to restore the links between NET_BUFFER_LIST structures.

Indicating Received Ethernet Frames

Article • 12/15/2021

The Windows TCP/IP protocol driver imposes a set of requirements for receiving Ethernet frames. Any driver that originates receive indications of Ethernet frames or modifies receive indications of underlying drivers must support the general requirements that TCP/IP imposes. These drivers include Ethernet miniport drivers, MUX intermediate drivers, and filter drivers.

Note If a driver does not follow these requirements, overlying drivers (such as the TCP/IP transport, MUX intermediate drivers, and filter drivers) might behave unpredictably.

Drivers that originate Ethernet receive indications must support the following requirements:

- The driver must allocate a **NET_BUFFER_LIST** structure for the received Ethernet frame. Each **NET_BUFFER_LIST** structure must include the out-of-band (OOB) data that is defined in the **NetBufferListInfo** member of the **NET_BUFFER_LIST** required for the particular use.

- The driver must allocate a **NET_BUFFER** structure for the frame and link it to a **NET_BUFFER_LIST** structure. The Ethernet miniport must assign exactly one **NET_BUFFER** structure to a **NET_BUFFER_LIST** structure when indicating received data. This restriction applies only to the Ethernet receive path. It is not applicable to the other media types, such as the native 802.11 wireless LAN interface. or NDIS in general.

- Starting with NDIS 6.1, under certain scenarios, a **NET_BUFFER** structure can be associated with multiple memory descriptor lists (MDLs) for the received Ethernet frame. Even though a **NET_BUFFER_LIST** structure must contain a single **NET_BUFFER** structure, using multiple MDLs allows the driver to split the received packet data into separate buffers.

  For example, Ethernet drivers that support the header-data split interface split a received Ethernet frame by using a linked list of multiple MDLs that are associated with a single **NET_BUFFER** structure. For more information, see Header-Data Split.

  For simplicity and performance reasons, we highly recommend that drivers that don't support header-data split use only one MDL for each **NET_BUFFER** structure.

  Note In NDIS 6.0 for Windows Vista, each **NET_BUFFER** structure must contain only one MDL.

- Drivers must not split received Ethernet frames in the middle of the IP header, IPv4 options, IPsec headers, IPv6 extension headers, or upper-layer protocol headers, unless the first MDL contains at least as many bytes as NDIS specified for the lookahead size.

NDIS protocol and filter drivers must support split Ethernet frames in receive indications if such split frames comply with the restrictions that are defined in the preceding list item. The restrictions ensure that the protocol and filter drivers are compatible with future Windows versions.

Derived NET_BUFFER_LIST Structures

Article • 12/15/2021

NDIS provides functions that drivers can use to manage **NET_BUFFER_LIST** structures that are derived from other NET_BUFFER_LIST structures. These functions are typically used by intermediate drivers.

The following NDIS functions can create derived NET_BUFFER_LIST structures from an existing NET_BUFFER_LIST structure:

NdisAllocateCloneNetBufferList

NdisAllocateFragmentNetBufferList

NdisAllocateReassembledNetBufferList

These functions improve system performance because NDIS creates the derived structures without copying the network data. There are three types of **NET_BUFFER_LIST** structures that can be derived from an existing NET_BUFFER_LIST structure:

Clone
A cloned NET_BUFFER_LIST structure is a duplicate that references the original data. Drivers can use this type of structure to efficiently transfer the same data to multiple paths.

Fragment
A fragment **NET_BUFFER_LIST** structure includes a set of **NET_BUFFER** structures that reference the original data; however, the data is divided into units that do not exceed a maximum size. Drivers can use this type of structure to efficiently break up large buffers into smaller buffers.

Reassembled
A reassembled NET_BUFFER_LIST structure contains a NET_BUFFER structure that references the original data from multiple source NET_BUFFER structures. Drivers can use this type of structure to efficiently combine many smaller buffers into a single large buffer.

This following topics provide more information about derived NET_BUFFER_LIST structures:

- Relationships Between NET_BUFFER_LIST Generations
- Cloned NET_BUFFER_LIST Structures
- Fragmented NET_BUFFER_LIST Structures

- Reassembled NET_BUFFER_LIST Structures

Relationships Between NET_BUFFER_LIST Generations

Article • 12/15/2021

Driver writers should understand and maintain the relationship between the parent (original) NET_BUFFER_LIST structures and the child (derived) structures that result from clone, fragment, and reassemble operations.

The caller of a clone/fragment/reassemble function maintains the parent/child relationship, including the parent pointer in the child NET_BUFFER_LIST structure and a child count. The child count ensures that the caller frees the parent after all the children have been freed. The following rules apply:

- After a driver creates child structures from a NET_BUFFER_LIST structure, it should retain the ownership of the parent structure and should pass the child structures to other drivers. The driver should never pass the parent NET_BUFFER_LIST structure to another driver.

- A driver should only update the child count in the parent NET_BUFFER_LIST structure. Because the parent structure is never passed to another driver, there is no risk that the value of the child count could be overwritten. The driver should set the parent pointer in the child structures to point to the parent structure.

- When a driver receives a NET_BUFFER_LIST from another driver, the driver must not overwrite the parent pointer. If the received NET_BUFFER_LIST structure is a child, its parent pointer should be set already. The driver can use the NET_BUFFER_LIST received from another driver as a parent structure.

- NDIS does not enforce the preceding rules. The current owner of a NET_BUFFER_LIST structure must manage the child count and parent pointer. For example, if the current owner will both clone and fragment a NET_BUFFER_LIST structure, it must manage the parent pointer and child counter.

- NDIS sets the child count to zero and the parent pointer to **NULL** when it allocates a NET_BUFFER_LIST structure. NDIS does not change these fields each time a driver passes a NET_BUFFER_LIST structure to another driver.

Related topics

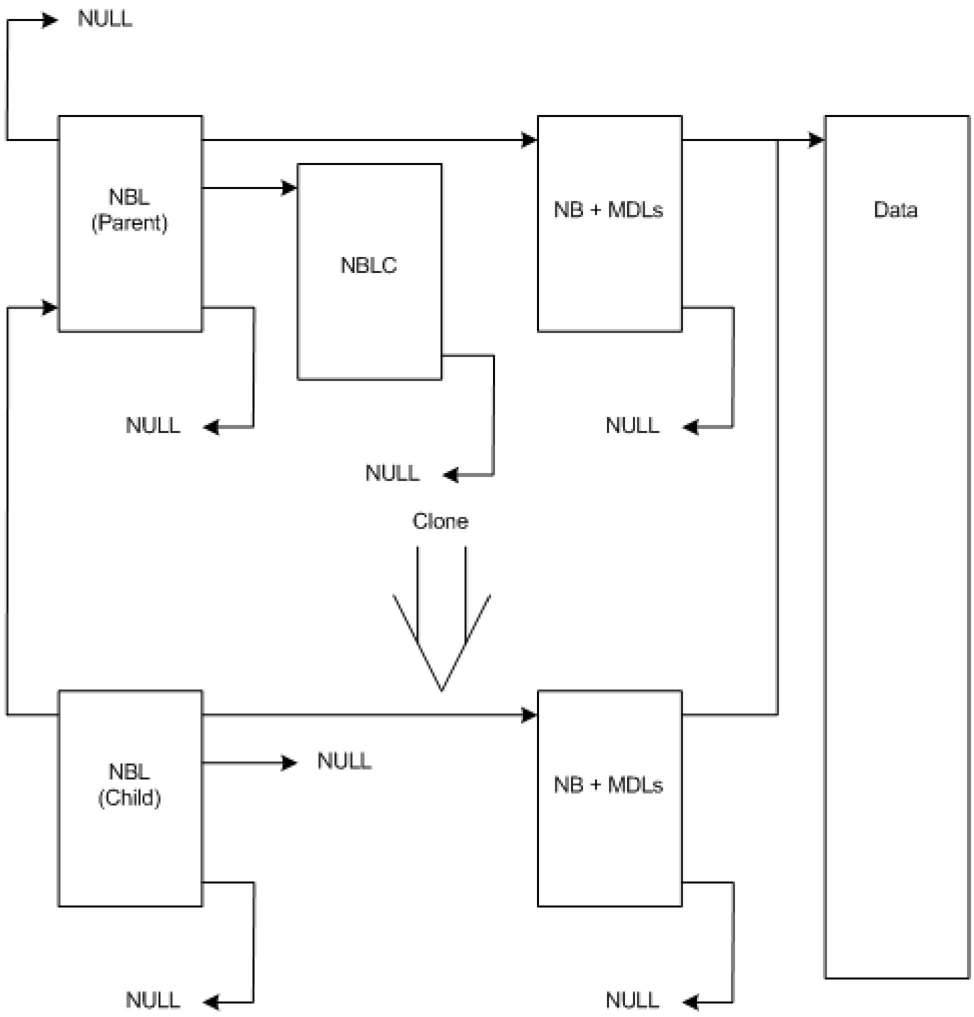
Derived NET_BUFFER_LIST Structures

Cloned NET_BUFFER_LIST Structures

Article • 12/15/2021

An NDIS driver creates a cloned NET_BUFFER_LIST structure from an existing NET_BUFFER_LIST structure. The cloned structure references the original structures data. Drivers can use this type of structure to efficiently transfer the same data to multiple paths.

The following figure shows the relationship between a parent NET_BUFFER_LIST structure and a cloned child structure.



NBLC = NET_BUFFER_LIST_CONTEXT

NBL = NET_BUFFER_LIST

NB = NET_BUFFER

The preceding figure contains a parent NET_BUFFER_LIST structure and a child structure that was derived from that parent. The parent structure has one NET_BUFFER_LIST_CONTEXT structure and one NET_BUFFER structure with MDLs attached. The parent structure's parent pointer is **NULL** indicating that it is not a derived structure.

The child NET_BUFFER_LIST structure has one NET_BUFFER structure with MDLs attached. The child NET_BUFFER_LIST has a pointer to the parent structure. The **NULL** where a NET_BUFFER_LIST_CONTEXT structure pointer would be indicates that the child has no NET_BUFFER_LIST_CONTEXT structure.

Drivers call the **NdisAllocateCloneNetBufferList** function to create a clone **NET_BUFFER_LIST** structure. NDIS allocates new **NET_BUFFER** structures and MDLs with the cloned NET_BUFFER_LIST structure. NDIS does not allocate a **NET_BUFFER_LIST_CONTEXT** structure for the cloned structure. The new NET_BUFFER structures and MDLs describe the same data as in the parent structure. The data is not copied.

Drivers call the **NdisFreeCloneNetBufferList** function to free a NET_BUFFER_LIST structure and all associated NET_BUFFER structures and MDL chains that were previously allocated by calling **NdisAllocateCloneNetBufferList**.

Related topics

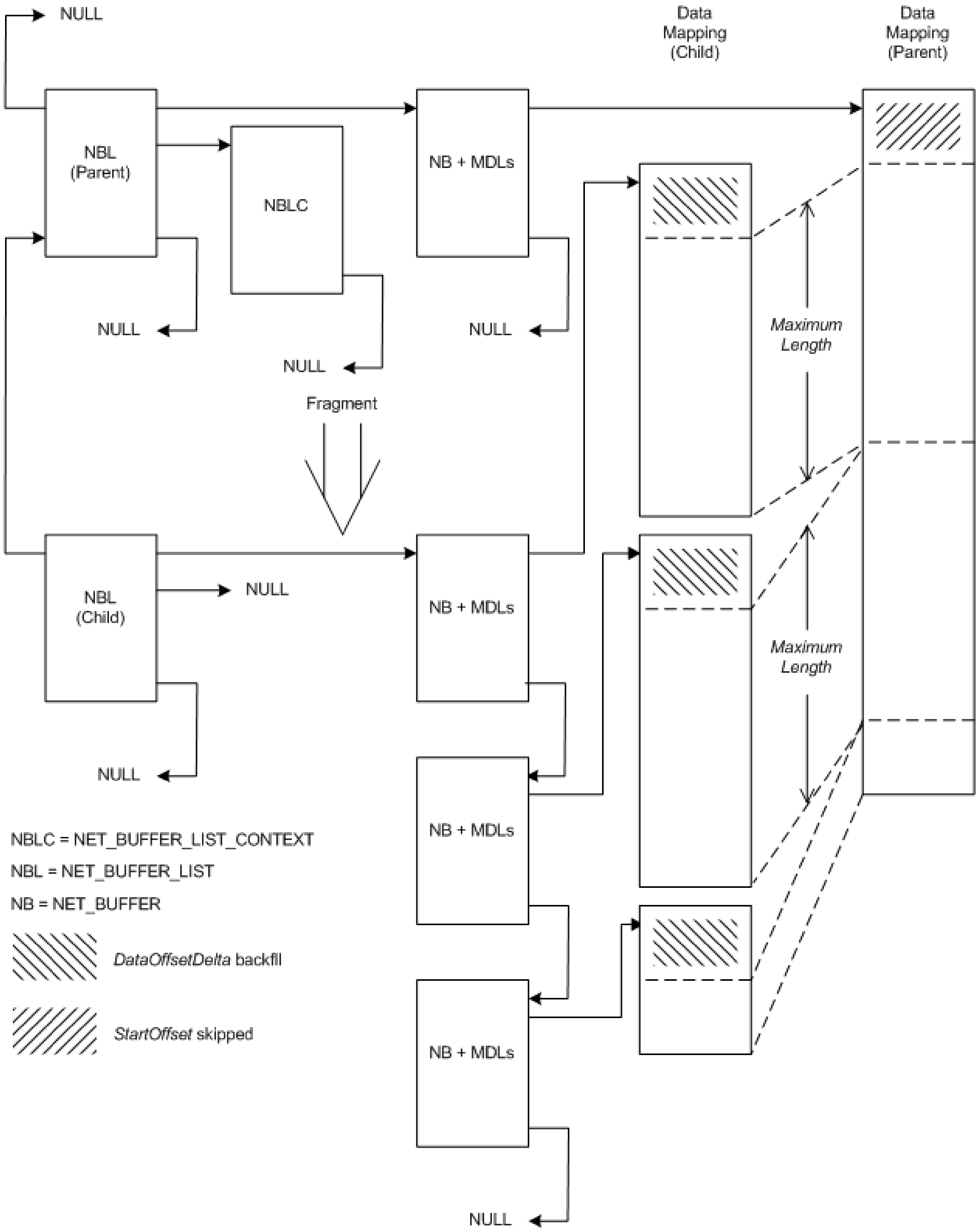
Derived NET_BUFFER_LIST Structures

Fragmented NET_BUFFER_LIST Structures

Article • 12/15/2021

An NDIS driver can create a fragmented **NET_BUFFER_LIST** structure from an existing NET_BUFFER_LIST structure. The fragmented structure references a set of **NET_BUFFER** structures that reference the original data; however, the data is divided into units that do not exceed a maximum size. Drivers can use this type of structure to efficiently break up large buffers into smaller buffers.

The following figure shows the relationship between a parent NET_BUFFER_LIST structure and a fragmented child.



The preceding figure contains a parent NET_BUFFER_LIST structure and a child structure that was derived from that parent. The parent structure has one NET_BUFFER_LIST_CONTEXT structure and one NET_BUFFER structure with MDLs attached. The parent structure's parent pointer is **NULL** indicating that it is not a derived structure.

The child NET_BUFFER_LIST structure has three NET_BUFFER structures with MDLs attached. The child NET_BUFFER_LIST structure has a pointer to the parent structure. The

NULL where a NET_BUFFER_LIST_CONTEXT structure pointer would be indicates that the child has no NET_BUFFER_LIST_CONTEXT structure.

NDIS drivers call the NdisAllocateFragmentNetBufferList function to create a new fragmented NET_BUFFER_LIST structure that is based on the data in an existing NET_BUFFER_LIST structure. NDIS allocates new NET_BUFFER structures and MDLs for the fragmented NET_BUFFER_LIST structure. NDIS does not allocate a NET_BUFFER_LIST_CONTEXT structure for the fragmented structure. The fragment NET_BUFFER structures and MDLs describe the same data as does the parent structure. The data is not copied.

NdisAllocateFragmentNetBufferList creates the fragments, starting from the beginning of the *used data space* in each parent NET_BUFFER structure and offset by the value specified in the *StartOffset* parameter.

NdisAllocateFragmentNetBufferList divides the *used data space* in each source NET_BUFFER structure into fragments. The length of the *used data space* of each fragment is less than or equal to the value specified in the *MaximumLength* parameter. The *used data space* of the last fragment can be less than *MaximumLength* . The data offset of the new NET_BUFFER structures is retreated by the number of bytes specified in the *DataOffsetDelta* parameter.

If there are multiple NET_BUFFER structures in the parent NET_BUFFER_LIST structure (not shown in the illustration) the fragmenting process for each NET_BUFFER structure is the same as for a single structure. For example, if the last piece of data in any parent NET_BUFFER structure is smaller than the maximum size, NDIS does not combine such data with the data at the start of the next NET_BUFFER structure.

NDIS drivers call the NdisFreeFragmentNetBufferList function to free a NET_BUFFER_LIST structure and all associated NET_BUFFER structures and MDL chains that were previously allocated by calling NdisAllocateFragmentNetBufferList.

Related topics

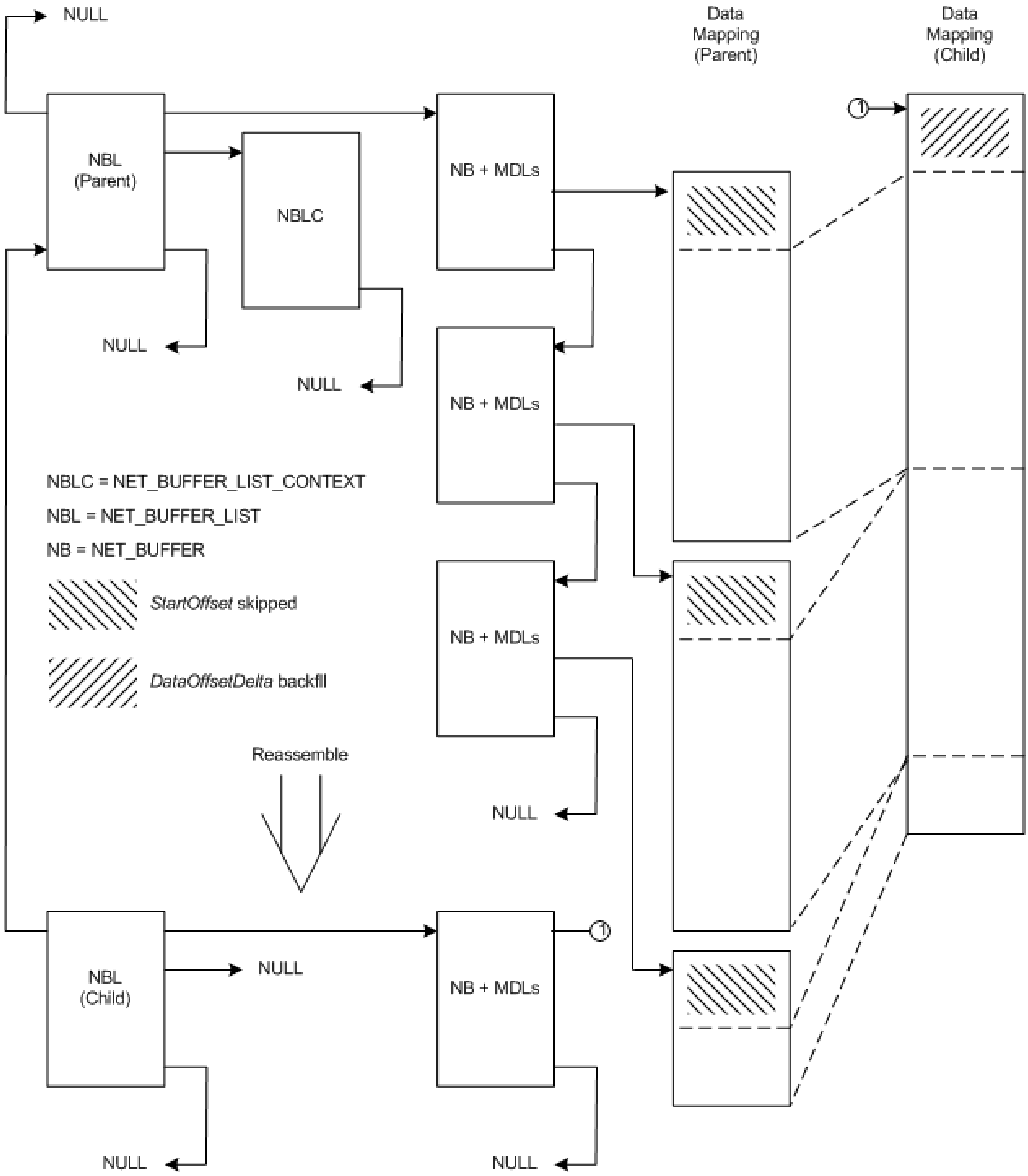
Derived NET_BUFFER_LIST Structures

Reassembled NET_BUFFER_LIST Structures

Article • 12/15/2021

An NDIS driver can create a reassembled **NET_BUFFER_LIST** structure from an existing NET_BUFFER_LIST structure. The reassembled structure references the original data from multiple source **NET_BUFFER** structures. Drivers can use this type of structure to efficiently combine many smaller buffers into a single large buffer.

The following figure shows the relationship between a parent NET_BUFFER_LIST structure and a reassembled child structure:



The preceding figure contains a parent NET_BUFFER_LIST structure and a child structure that was derived from that parent. The parent structure has one NET_BUFFER_LIST_CONTEXT structure and three NET_BUFFER structures with MDLs attached. The parent structure's parent pointer is **NULL** indicating that it is not a derived structure.

The child NET_BUFFER_LIST structure has one NET_BUFFER structure with MDLs attached. The child NET_BUFFER_LIST structure has a pointer to the parent structure. The **NULL** where a NET_BUFFER_LIST_CONTEXT structure pointer would be indicates that the child has no NET_BUFFER_LIST_CONTEXT structure.

NDIS drivers call the NdisAllocateReassembledNetBufferList function to reassemble a fragmented NET_BUFFER_LIST structure. NDIS allocates a new NET_BUFFER structure and MDLs with the reassembled NET_BUFFER_LIST structure. NDIS does not allocate a NET_BUFFER_LIST_CONTEXT structure for the reassembled structure. The reassembled NET_BUFFER structure and MDLs describe the same data as does the parent structure. The data is not copied.

To create the reassembled NET_BUFFER_LIST structure, **NdisAllocateReassembledNetBufferList** skips over the number of bytes specified in the *StartOffset* parameter in each of the parent NET_BUFFER structures. **NdisAllocateReassembledNetBufferList** concatenates the remaining data in each parent NET_BUFFER structure into the MDL chain of one reassembled NET_BUFFER structure. **NdisAllocateReassembledNetBufferList** retreats (increases the used data space in) the reassembled NET_BUFFER structure by the amount specified in *DataOffsetDelta* .

NDIS drivers call the NdisFreeReassembledNetBufferList function to free a reassembled NET_BUFFER_LIST structure and the associated NET_BUFFER structure and MDL chain.

Related topics

Derived NET_BUFFER_LIST Structures

Introduction to the NDIS PacketDirect Provider Interface

Article • 03/14/2023

The PacketDirect Provider Interface (PDPI) extends NDIS with an accelerated I/O model, for both physical and virtual environments, that can increase the number of packets processed per second by an order of magnitude and significantly decrease jitter when compared to the traditional NDIS I/O path.

Background

The traditional I/O model in Windows was implemented to be a multipurpose, general I/O platform that was intended to work with multiple media types with many different characteristics and where networking was only one aspect of the overall system. Today, as network virtualization has become a prevalent technology in datacenters, the traditional NDIS I/O model in the Windows Server OS is not only not sufficient to keep up with the network intensive workloads that we expect to become more and more common but also an inappropriate model to dedicate resources to network I/O processing. In datacenter environments, it is not uncommon to implement a single purpose machine dedicated to networking doing functions that were usually reserved for hardware appliances. Examples of these network appliances include software load balancers, DDoS appliances and forwarding gateways. To make matters worse, there are mechanisms on other OS' to accelerate I/O that make these alternative OS' the preferred platform to build network intensive applications such as virtual appliances.

PacketDirect (PD) extends the current NDIS model with an accelerated network I/O path that is optimized for packet per second (pps) counts an order of a magnitude higher than what has been seen with the traditional NDIS I/O model. This is accomplished through:

- Reduced latency
- Reduced cycles/packet
- Linear speed up with use of additional system resources

PacketDirect exists side-by-side with the traditional model. The new PD path can be used when an application prefers it and there are sufficient hardware resources to accommodate it. PD is not meant to replace the traditional I/O model and assumes that a client writing to the PD interface will have strict partitioning requirements for the underlying resources based on the system topology. PD is meant to be the new high speed data path that will help a Windows system replace high pps workloads that have been traditionally done in hardware, saving data center owners millions in infrastructure costs.

PacketDirect Concepts

PD works by allowing a PD client to explicitly manage networking traffic from a network adapter (NIC). PD gives the PD client control of the high performance send and receive functionality of the NIC through the PacketDirect client interface (PDCI). Internally, the PDCI send/receive functions are mapped directly to the PDPI. PD send/receive functions operate on PD queues created by the PD client on PD-capable NICs. PD provides the PD clients with the ability to set custom filters for very specific types of traffic or very generic traffic, based on the needs of the PD client. This allows the PD client to direct certain incoming packets to its PD queues. Packet processing in the PD model always takes place in an execution context that's owned (or controlled/coordinated) by the PD client. The PD-capable NIC driver is completely passive, meaning it does not actively forward incoming packets or completion indications for sent packets to the PD client in a driver-owned execution context such as a DPC or worker-thread.

If a PD client does not understand how to process a packet or receives a control packet in one of its queues, such as an ARP, LLDP, or other protocol packets, the PD client can reroute the packet back to the current I/O path for processing. This allows PD to continue to process the packets that it has context for and not waste cycles on control traffic.

Important There can be one PD provider and one PD client per net adapter. Therefore, there can be multiple PD clients and PD providers on a single system.

The PD client has control over the resources that are allocated to PD in the system. In cases of high network traffic, the PD client is responsible for minimizing its workload so that the OS can be responsive to other workloads.

The PacketDirect platform implemented by Windows maps the client interface to the provider interface. The platform controls buffer management and ability to re-inject packets received via PD to the current NDIS receive path. It also handles the interaction with PD clients for satisfying the NDIS control path requirements such as NIC disabling, going into low-power, system shutdown, and surprise removal in a fashion that does NOT hamper the PD data path performance.

PacketDirect Provider Interface (PDPI)

The PDPI allows NIC drivers to expose their high-performance send and receive functionality to the Windows OS. The functions implemented are a subset of the complete MiniPort functionality and are generic to all NICs that implement PD. For reference documentation for PDPI, see PacketDirect Provider Interface (PDPI) Reference.

PacketDirect Client Interface (PDCI)

The PDCI allows first-party Windows services/applications (e.g., Load-balancer, NAT, VM-switch, etc.) to speed up their data path by leveraging the PacketDirect I/O model through use of the PD clients. This interface is a layer 2 interface just like the current NDIS send/receive interface. The main functionality PDCI provides (in addition to PDPI access) is PD packet buffer allocation/management, a back-channel for injecting packets back to regular NDIS receive path, handling of NDIS power/PnP events.

Related topics

PacketDirect Provider Interface (PDPI) Reference

Overview of NDIS versions

Article • 09/27/2024

If you're writing an NDIS driver for more than one version of Microsoft Windows, be sure the features that you're using are supported on each Windows version. New features have been added to NDIS with each release. Other features became obsolete and were removed from later NDIS versions.

This set of design guide documentation is targeted at Windows Vista and later operating systems and NDIS 6.0 and later drivers. Documentation for earlier Windows and NDIS versions is contained in prior releases of the documentation. For the Windows XP and NDIS 5.1 documentation, see Windows 2000 and Windows XP Networking Design Guide.

> **Note**
>
> A driver can query the NDIS version by calling the NdisReadConfiguration function with the *Keyword* parameter set to **NdisVersion**.

The following table describes Windows operating system, Microsoft Windows Driver Kit (WDK), and Driver Development Kit (DDK) version support for NDIS versions. This table also describes support for major NDIS features across NDIS versions.

Expand table

Operating system	Development Kit	Supported NDIS version	CoNDIS	Deserialized driver	Intermediate driver
Windows 11, version 24H2	See Download kits for Windows hardware development.	6.89. For more information about NDIS 6.89 features, see Introduction to NDIS 6.89.	X	X	X
Windows Server 2022 23H2	See Download kits for Windows hardware development.	6.88. For more information about NDIS 6.88 features, see Introduction to NDIS 6.88.	X	X	X
Windows 11, version 22H2	See Download kits for Windows hardware development.	6.87. For more information about NDIS 6.87 features, see Introduction to NDIS 6.87.	X	X	X

Operating system	Development Kit	Supported NDIS version	CoNDIS	Deserialized driver	Intermediate driver
Windows 11, version 21H2	See Download kits for Windows hardware development .	6.86. For more information about NDIS 6.86 features, see Introduction to NDIS 6.86.	X	X	X
Windows Server 2022	See Download kits for Windows hardware development .	6.85. For more information about NDIS 6.85 features, see Introduction to NDIS 6.85.	X	X	X
Windows 10, version 2004	See Download kits for Windows hardware development .	6.84. For more information about NDIS 6.84 features, see Introduction to NDIS 6.84.	X	X	X
Windows 10, version 1903	See Download kits for Windows hardware development .	6.83. For more information about NDIS 6.83 features, see Introduction to NDIS 6.83.	X	X	X
Windows 10, version 1809	See Download kits for Windows hardware development .	6.82. For more information about NDIS 6.82 features, see Introduction to NDIS 6.82.	X	X	X
Windows 10, version 1803	See Download kits for Windows hardware development .	6.81. For more information about NDIS 6.81 features, see Introduction to NDIS 6.81.	X	X	X
Windows 10, version 1803	See Download kits for Windows hardware development .	6.81. For more information about NDIS 6.81 features, see Introduction to NDIS 6.81.	X	X	X
Windows 10, version 1709	See Download kits for Windows hardware development .	6.80. For more information about NDIS 6.80 features, see Introduction to NDIS 6.80.	X	X	X

Operating system	Development Kit	Supported NDIS version	CoNDIS	Deserialized driver	Intermediate driver
Windows 10, version 1703	See Download kits for Windows hardware development.	6.70. NDIS 6.70 coincided with a preview release of the Network Adapter WDF Class Extension, also known as NetAdapterCx. For more information about NDIS 6.70 features, see Introduction to NDIS 6.70	X	X	X
Windows 10, version 1607 and Windows Server 2016	See Download kits for Windows hardware development.	6.60. For more information about NDIS 6.60 features, see Introduction to NDIS 6.60.	X	X	X
Windows 10, version 1511	See Download kits for Windows hardware development.	6.51	X	X	X
Windows 10, version 1507	See Download kits for Windows hardware development.	6.50. For more information about NDIS 6.50 features, see Introduction to NDIS 6.50.	X	X	X
Windows 8.1 and Windows Server 2012 R2	See Download kits for Windows hardware development.	6.40. For information about NDIS 6.40 features, see Introduction to NDIS 6.40.	X	X	X
Windows 8 and Windows Server 2012	See Download kits for Windows hardware development.	6.30. For information about NDIS 6.30 features, see Introduction to NDIS 6.30.	X	X	X
Windows 7 and Windows Server 2008 R2	See Download kits for Windows hardware development.	6.20. For information about NDIS 6.20 features, see Introduction to NDIS 6.20. For information about backward compatibility and obsolete features that aren't supported in NDIS 6.20 drivers, see NDIS 6.20 Backward Compatibility.	X	X	X

Operating system	Development Kit	Supported NDIS version	CoNDIS	Deserialized driver	Intermediate driver
Windows Vista with Service Pack 1 (SP1) and Windows Server 2008	See Download kits for Windows hardware development ↗ .	6.1. For information about NDIS 6.1 features, see Introduction to NDIS 6.1.	X	X	X
Windows Vista	See Download kits for Windows hardware development ↗	6.0. Major improvements in the following provide significant performance gains for both clients and servers: • Network data packaging • Send and receive paths • Run-time reconfiguration capabilities • Scatter/gather DMA • Filter drivers • Multiprocessor scaling of received data handling • Offloading TCP tasks to NICs The following improvements simplify driver development: • Streamlined driver initialization • Versioning support for NDIS interfaces • Simplified reset handling • A standard interface for obtaining management information • A filter driver model to replace filter intermediate drivers For more information about NDIS 6.0 features, see Introduction to NDIS 6.0. For information about backward compatibility and obsolete features that aren't supported in NDIS 6.0 drivers,	X	X	X

Operating system	Development Kit	Supported NDIS version	CoNDIS	Deserialized driver	Intermediate driver
		see NDIS 6.0 Backward Compatibility.			
Windows XP	See Download kits for Windows hardware development	5.1. Added support for: New miniport driver attribute flags, 64-bit statistical counters, Remote NDIS, Scatter/gather support for both serialized and deserialized miniport drivers, Packet stacking for intermediate drivers, VLAN tagging, Offloading the Processing of UDP-Encapsulated ESP Packets (Windows Server 2003 only), Wi-Fi Protected Access (WPA) in Windows XP SP1. Dropped support for: Full Mac drivers, NDIS 3.0 protocols, **NdisQueryMapRegisterCount**, EISA bus	X	X	X
Windows 2000	Windows 2000 DDK	5.0	X	X	X
Windows NT 4.0 SP3	Windows NT DDK with updated NDIS header and library	4.1	X	X	X
Windows NT 4.0	Windows NT 4.0 DDK	4.0			
Windows NT 3.5	Windows NT 3.5 DDK	3.0			
Windows Me	Windows NT 4.0 DDK or Windows 98 DDK for Vxds	5.0	X	X	X
Windows 98 SE	Windows NT 4.0 DDK or Windows 98 DDK	5.0. Added support for new INF file format compatible with Windows 95/98/Me, Plug and Play and Power Management, WMI, LBFO, and scatter/gather DMA support	X	X	X

Operating system	Development Kit	Supported NDIS version	CoNDIS	Deserialized driver	Intermediate driver
		for deserialized miniport drivers.			
Windows 98	Windows NT 4.0 DDK or Windows 98 DDK	4.1. Protocol driver is a vxd-type driver.	X	X	X
Windows 95 OSR2	Windows NT 4.0 DDK or Windows 95 DDK	4.0. Protocol driver is a vxd-type driver. Added these features: MiniportSendPackets, ProtocolReceivePacket, MiniportAllocateComplete.			
Windows 95	Windows NT 4.0 DDK or Windows 95 DDK	3.1. Added support for miniport drivers and Plug and Play.			

Introduction to NDIS 6.89

Article • 05/22/2024

This topic introduces Network Driver Interface Specification (NDIS) 6.89 and describes its major design additions. NDIS 6.89 is included in Windows 11, version 24H2 and Windows Server 2022 and later.

NDIS 6.89 is a minor version update to NDIS 6.88. For more information about porting NDIS 6.x drivers to NDIS 6.89, see Porting NDIS 6.x drivers to NDIS 6.89.

Feature updates

NDIS 6.89 adds support for UDP Receive Segment Coalescing Offload (URO). This hardware offload enables NICs to coalesce UDP receive segments. NICs can combine UDP datagrams from the same flow that match a set of rules into a logically contiguous buffer. These combined datagrams are then indicated to the Windows networking stack as a single large packet. Coalescing UDP datagrams reduces the CPU cost to process packets in high-bandwidth flows, resulting in higher throughput and fewer cycles per byte.

Implementing an NDIS 6.89 driver

An NDIS 6.89 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.89 driver must be compliant with the following requirements:

- An NDIS 6.89 driver must report the correct NDIS version when it registers with NDIS.
    - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.89. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 89. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.89 driver).
    - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.

    - Filter drivers must set the **Header** member of NDIS_FILTER_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

    - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.89 miniport drivers for Windows 11, version 24H2 and Windows Server 2022 and later must use the NDIS 6.89 versions of data structures.

Compiling an NDIS 6.89 driver

The WDK for Windows Server 2022 supports header versioning. Header versioning makes sure that NDIS 6.89 drivers use the appropriate NDIS 6.89 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS689_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS689=1`.

For information on building a driver with the Windows Server 2022 release of the WDK, see Building a Driver.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Introduction to NDIS 6.88

Article • 05/22/2024

This topic introduces Network Driver Interface Specification (NDIS) 6.88 and describes its major design additions. NDIS 6.88 is included in Windows Server 2022 23H2 and later.

NDIS 6.88 is a minor version update to NDIS 6.87. For more information about porting NDIS 6.x drivers to NDIS 6.88, see Porting NDIS 6.x drivers to NDIS 6.88.

Feature updates

NDIS 6.88 is an incremental update to NDIS 6.87 and does not contain any major new features.

Implementing an NDIS 6.88 driver

An NDIS 6.88 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.88 driver must be compliant with the following requirements:

- An NDIS 6.88 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.88. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 88. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.88 driver).
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

  - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.88 miniport drivers for Windows Server 2022 23H2 and later must use the NDIS 6.88 versions of data structures.

Compiling an NDIS 6.88 driver

The WDK for Windows Server 2022 supports header versioning. Header versioning makes sure that NDIS 6.88 drivers use the appropriate NDIS 6.88 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS688_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS688=1`.

For information on building a driver with the Windows Server 2022 release of the WDK, see Building a Driver.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Introduction to NDIS 6.87

Article • 05/22/2024

This topic introduces Network Driver Interface Specification (NDIS) 6.87 and describes its major design additions. NDIS 6.87 is included in Windows 11, version 22H2 and Windows Server 2022 and later.

NDIS 6.87 is a minor version update to NDIS 6.86. For more information about porting NDIS 6.x drivers to NDIS 6.87, see Porting NDIS 6.x drivers to NDIS 6.87.

Feature updates

NDIS 6.87 is an incremental update to NDIS 6.86 and does not contain any major new features.

Implementing an NDIS 6.87 driver

An NDIS 6.87 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.87 driver must be compliant with the following requirements:

- An NDIS 6.87 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.87. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 87. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.87 driver).
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

  - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.87 miniport drivers for Windows 11, version 22H2 and Windows Server 2022 and later must use the NDIS 6.87 versions of data structures.

Compiling an NDIS 6.87 driver

The WDK for Windows Server 2022 supports header versioning. Header versioning makes sure that NDIS 6.87 drivers use the appropriate NDIS 6.87 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS687_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS687=1`.

For information on building a driver with the Windows Server 2022 release of the WDK, see Building a Driver.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Introduction to NDIS 6.86

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.86 and describes its major design additions. NDIS 6.86 is included in Windows 11, version 21H2 and Windows Server 2022 and later.

NDIS 6.86 is a minor version update to NDIS 6.85. For more information about porting NDIS 6.x drivers to NDIS 6.86, see Porting NDIS 6.x drivers to NDIS 6.86.

Feature updates

NDIS 6.86 is an incremental update to NDIS 6.85 and does not contain any major new features.

Implementing an NDIS 6.86 driver

An NDIS 6.86 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.86 driver must be compliant with the following requirements:

- An NDIS 6.86 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.86. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 86. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.86 driver).
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

    - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.86 miniport drivers for Windows 11, version 21H2 and Windows Server 2022 and later must use the NDIS 6.86 versions of data structures.

Compiling an NDIS 6.86 driver

The WDK for Windows Server 2022 supports header versioning. Header versioning makes sure that NDIS 6.86 drivers use the appropriate NDIS 6.86 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS686_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS686=1`.

For information on building a driver with the Windows Server 2022 release of the WDK, see Building a Driver.

Introduction to NDIS 6.85

Article • 06/15/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.85 and describes its major design additions. NDIS 6.85 is included in Windows 10, version 21H2 and Windows Server 2022 and later.

NDIS 6.85 is a minor version update to NDIS 6.84. For more information about porting NDIS 6.x drivers to NDIS 6.85, see Porting NDIS 6.x drivers to NDIS 6.85.

Feature updates

NDIS Poll mode

NDIS 6.85 introduces NDIS Poll Mode, an OS controlled polling execution model that drives the network interface datapath. Previously, NDIS drivers typically relied on Deferred Procedure Calls (DPCs) to implement their execution model. NDIS Poll Mode moves the complexity of scheduling decisions away from NIC drivers and into NDIS. For more information, see NDIS Poll Mode.

Network Virtualization using Generic Routing Encapsulation (NVGRE) with UDP segmentation offload (USO)

NDIS 6.85 introduces Supporting NVGRE in UDP Segmentation Offload (USO). NDIS miniport, protocol, and filter drivers, as well as NICs that perform USO, should support NVGRE and VXLAN encapsulations.

Implementing an NDIS 6.85 driver

An NDIS 6.85 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.85 driver must be compliant with the following requirements:

- An NDIS 6.85 driver must report the correct NDIS version when it registers with NDIS.

  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.85. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 85. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.85 driver).

  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.

  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

  - Protocol drivers must set the **Header** member of **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.85 miniport drivers for Windows 10, version 21H2 and Windows Server 2022 and later must use the NDIS 6.85 versions of data structures.

Compiling an NDIS 6.85 driver

The WDK for Windows 10, version 21H2 supports header versioning. Header versioning makes sure that NDIS 6.85 drivers use the appropriate NDIS 6.85 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS685_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS685=1`.

For information on building a driver with the Windows 10, version 21H2 release of the WDK, see Building a Driver.

Introduction to NDIS 6.84

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.84 and describes its major design additions. NDIS 6.84 is included in Windows 10, version 2004 and Windows Server 2022 and later.

NDIS 6.84 is a minor version update to NDIS 6.83. For more information about porting NDIS 6.x drivers to NDIS 6.84, see Porting NDIS 6.x drivers to NDIS 6.84.

Feature updates

NDIS 6.84 is an incremental update to NDIS 6.83 and does not contain any major new features.

Implementing an NDIS 6.84 driver

An NDIS 6.84 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.84 driver must be compliant with the following requirements:

- An NDIS 6.84 driver must report the correct NDIS version when it registers with NDIS.
    - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.84. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 84. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.84 driver).
    - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.
    - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

- Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.84 miniport drivers for Windows 10, version 2004 and Windows Server 2022 and later must use the NDIS 6.84 versions of data structures.

Compiling an NDIS 6.84 driver

The WDK for Windows 10, version 2004 supports header versioning. Header versioning makes sure that NDIS 6.84 drivers use the appropriate NDIS 6.84 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS684_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS684=1`.

For information on building a driver with the Windows 10, version 2004 release of the WDK, see Building a Driver.

Introduction to NDIS 6.83

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.83 and describes its major design additions. NDIS 6.83 is included in Windows 10, version 1903 and Windows Server 2022 and later.

NDIS 6.83 is a minor version update to NDIS 6.82. For more information about porting NDIS 6.x drivers to NDIS 6.83, see Porting NDIS 6.x drivers to NDIS 6.83.

Feature updates

NDIS 6.83 is an incremental update to NDIS 6.82 and does not contain any major new features.

Implementing an NDIS 6.83 driver

An NDIS 6.83 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.83 driver must be compliant with the following requirements:

- An NDIS 6.83 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.83. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 83. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.83 driver).
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

    - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.83 miniport drivers for Windows 10, version 1903 and Windows Server 2022 and later must use the NDIS 6.83 versions of data structures.

Compiling an NDIS 6.83 driver

The WDK for Windows 10, version 1903 supports header versioning. Header versioning makes sure that NDIS 6.83 drivers use the appropriate NDIS 6.83 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS683_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS683=1`.

For information on building a driver with the Windows 10, version 1903 release of the WDK, see Building a Driver.

Introduction to NDIS 6.82

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.82 and describes its major design additions. NDIS 6.82 is included in Windows 10, version 1809 and Windows Server 2019 and later.

NDIS 6.82 is a minor version update to NDIS 6.81. For more information about porting NDIS 6.x drivers to NDIS 6.82, see Porting NDIS 6.x drivers to NDIS 6.82.

Feature updates

NDIS 6.82 is an incremental update to NDIS 6.81 and does not contain any major new features.

Implementing an NDIS 6.82 driver

An NDIS 6.82 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.82 driver must be compliant with the following requirements:

- An NDIS 6.82 driver must report the correct NDIS version when it registers with NDIS.

  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.82. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 82. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.82 driver).

  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.

  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

- Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.82 miniport drivers for Windows 10, version 1809 and Windows Server 2019 and later must use the NDIS 6.82 versions of data structures.

Compiling an NDIS 6.82 driver

The WDK for Windows 10, version 1809 supports header versioning. Header versioning makes sure that NDIS 6.82 drivers use the appropriate NDIS 6.82 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS682_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS682=1`.

For information on building a driver with the Windows 10, version 1809 release of the WDK, see Building a Driver.

Introduction to NDIS 6.81

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.81 and describes its major design additions. NDIS 6.81 is included in Windows 10, version 1803 and Windows Server 2016 and later.

NDIS 6.81 is a minor version update to NDIS 6.80. For more information about porting NDIS 6.x drivers to NDIS 6.81, see Porting NDIS 6.x drivers to NDIS 6.81.

Feature updates

NDIS 6.81 is an incremental update to NDIS 6.80 and does not contain any major new features.

Implementing an NDIS 6.81 driver

An NDIS 6.81 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.81 driver must be compliant with the following requirements:

- An NDIS 6.81 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.81. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 81. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.81 driver).
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.

    - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.81 miniport drivers for Windows 10, version 1803 and Windows Server 2016 and later must use the NDIS 6.81 versions of data structures.

Compiling an NDIS 6.81 driver

The WDK for Windows 10, version 1803 supports header versioning. Header versioning makes sure that NDIS 6.81 drivers use the appropriate NDIS 6.81 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS681_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS681=1`.

For information on building a driver with the Windows 10, version 1803 release of the WDK, see Building a Driver.

Introduction to NDIS 6.80

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.80 and describes its major design additions. NDIS 6.80 is included in Windows 10, version 1709.

NDIS 6.80 is a minor version update to NDIS 6.70 for miniport, protocol, filter, and intermediate drivers. For more information about porting NDIS 6.x drivers to NDIS 6.80, see Porting NDIS 6.x drivers to NDIS 6.80.

For NIC drivers, the NetAdapter class extension (NetAdapterCx) has been updated from version 1.0 to version 1.1 in Windows 10, version 1709.

Feature updates

Synchronous OID requests

NDIS 6.80 introduces a new feature for OIDs, Synchronous OID requests. Synchronous OID calls are low-latency, non-blocking, scalable, and reliable compared to regular OID requests. For more info, see Synchronous OID Request Interface in NDIS 6.80.

RSSv2

In NDIS 6.80, Receive Side Scaling (RSS) has been upgraded to RSS version 2 (RSSv2). RSSv2 improves on RSSv2 by offering per-VPort spreading. For more info, see Receive Side Scaling Version 2 (RSSv2) in NDIS 6.80.

RSSv2 is preview only in Windows 10, version 1709.

Other new networking features

NDIS forms the core foundation for the network driver platform on Windows. For a list of other network driver features that were updated at the same time as NDIS 6.80, see the Windows 10, version 1709 section for Networking on What's new in driver development.

Implementing an NDIS 6.80 driver

An NDIS 6.80 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.80 driver must be compliant with the following requirements:

- An NDIS 6.80 driver must report the correct NDIS version when it registers with NDIS.
    - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.80. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 80. This requirement applies to miniport, protocol and filter drivers.
    - You must also update the version information for the compiler (see Compiling an NDIS 6.80 driver).
    - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_3.
    - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_3 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_3.
    - Protocol drivers must set the **Header** member of **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

Compiling an NDIS 6.80 driver

NIC drivers

For more information about compiling a NIC driver with the NetAdapterCx, see Porting NDIS miniport drivers to NetAdapterCx (Compilation settings).

Miniport, protocol, and filter drivers

The WDK for Windows 10, version 1709 supports header versioning. Header versioning makes sure that NDIS 6.80 drivers use the appropriate NDIS 6.80 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS680_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS680=1`.

For information on building a driver with the Windows 10, version 1709 release of the WDK, see Building a Driver.

API and data structure changes

New APIs and data structures

The following APIs and data structures are new in NDIS 6.80.

- MINIPORT_SYNCHRONOUS_OID_REQUEST
- FILTER_SYNCHRONOUS_OID_REQUEST
- FILTER_SYNCHRONOUS_OID_REQUEST_COMPLETE
- NdisFSynchronousOidRequest
- NdisSynchronousOidRequest
- OID_GEN_RECEIVE_SCALE_PARAMETERS_V2
- OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES
- NDIS_RECEIVE_SCALE_PARAMETERS_V2
- NDIS_RSS_SET_INDIRECTION_ENTRIES
- NDIS_RSS_SET_INDIRECTION_ENTRY

Updated APIs and data structures

The following APIs and data structures were updated in NDIS 6.80.

- NDIS_MINIPORT_DRIVER_CHARACTERISTICS
- NDIS_FILTER_DRIVER_CHARACTERISTICS
- NDIS_RECEIVE_SCALE_CAPABILITIES

Synchronous OID request interface in NDIS 6.80

Article • 03/14/2023

Windows network drivers use OID requests to send control messages down the NDIS binding stack. Protocol drivers, such as TCPIP or vSwitch, rely on dozens of OIDs to configure each feature of the underlying NIC driver. Before Windows 10, version 1709, OID requests were sent in two ways: Regular and Direct.

This topic introduces a third style of OID call: Synchronous. A Synchronous call is meant to be low-latency, non-blocking, scalable, and reliable. The Synchronous OID request interface is available starting in NDIS 6.80, which is included in Windows 10, version 1709 and later.

Comparison to Regular and Direct OID requests

With Synchronous OID requests, the payload of the call (the OID itself) is exactly the same as with Regular and Direct OID requests. The only difference is in the call itself. Therefore, the *what* is the same across all three types of OIDs; only the *how* is different.

The following table describes the differences between Regular OIDs, Direct OIDs, and Synchronous OIDs.

Attribute	Regular OID	Direct OID	Synchronous OID
Payload	NDIS_OID_REQUEST	NDIS_OID_REQUEST	NDIS_OID_REQUEST
OID types	Stats, Query, Set, Method	Stats, Query, Set, Method	Stats, Query, Set, Method
Can be issued by	Protocols, filters	Protocols, filters	Protocols, filters
Can be completed by	Miniports, filters	Miniports, filters	Miniports, filters
Filters can modify	Yes	Yes	Yes
NDIS allocates memory	For each filter (OID clone)	For each filter (OID clone)	Only if unusually large number of filters (call context)
Can pend	Yes	Yes	No

Attribute	Regular OID	Direct OID	Synchronous OID
Can block	Yes	No	No
IRQL	== PASSIVE	<= DISPATCH	<= DISPATCH
Serialized by NDIS	Yes	No	No
Filters are invoked	Recursively	Recursively	Iteratively
Filters clone the OID	Yes	Yes	No

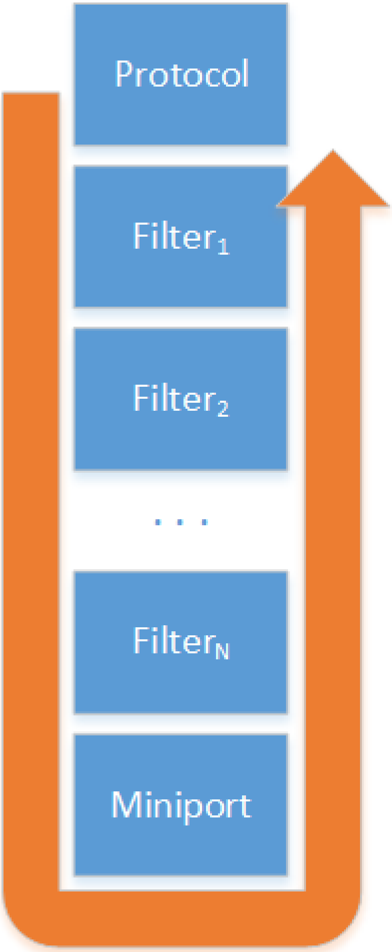
Filtering

Like the other two types of OID calls, filter drivers have full control over the OID request in a Synchronous call. Filter drivers can observe, intercept, modify, and issue Synchronous OIDs. However, for efficiency, the mechanics of a Synchronous OID are somewhat different.

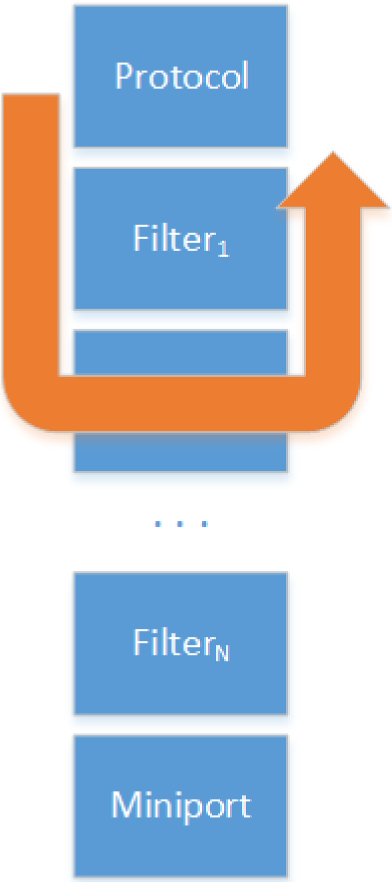
Passthrough, interception, and origination

Conceptually, all OID requests are issued from a higher driver and are completed by a lower driver. Along the way, the OID request may pass through any number of filter drivers.

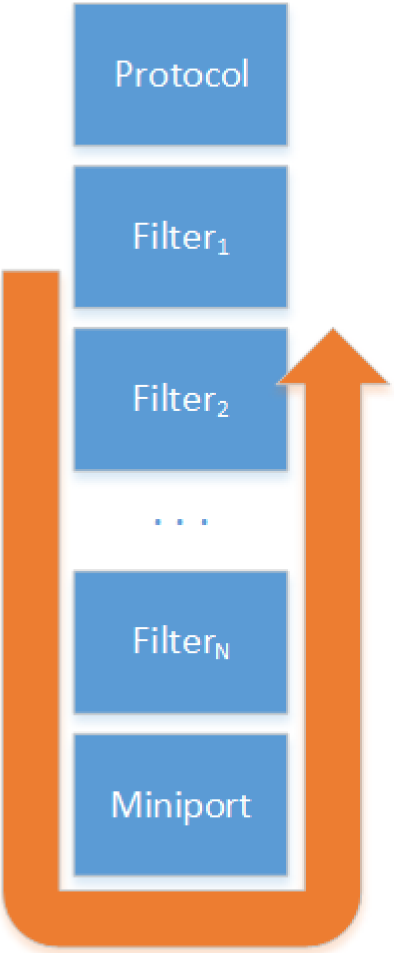
In the most common case, a protocol driver issues an OID request and all filters simply pass the OID request down, unmodified. The following figure illustrates this common scenario.



However, any filter module is allowed to intercept the OID request and complete it. In that case, the request does not pass through to lower drivers, as shown in the following diagram.



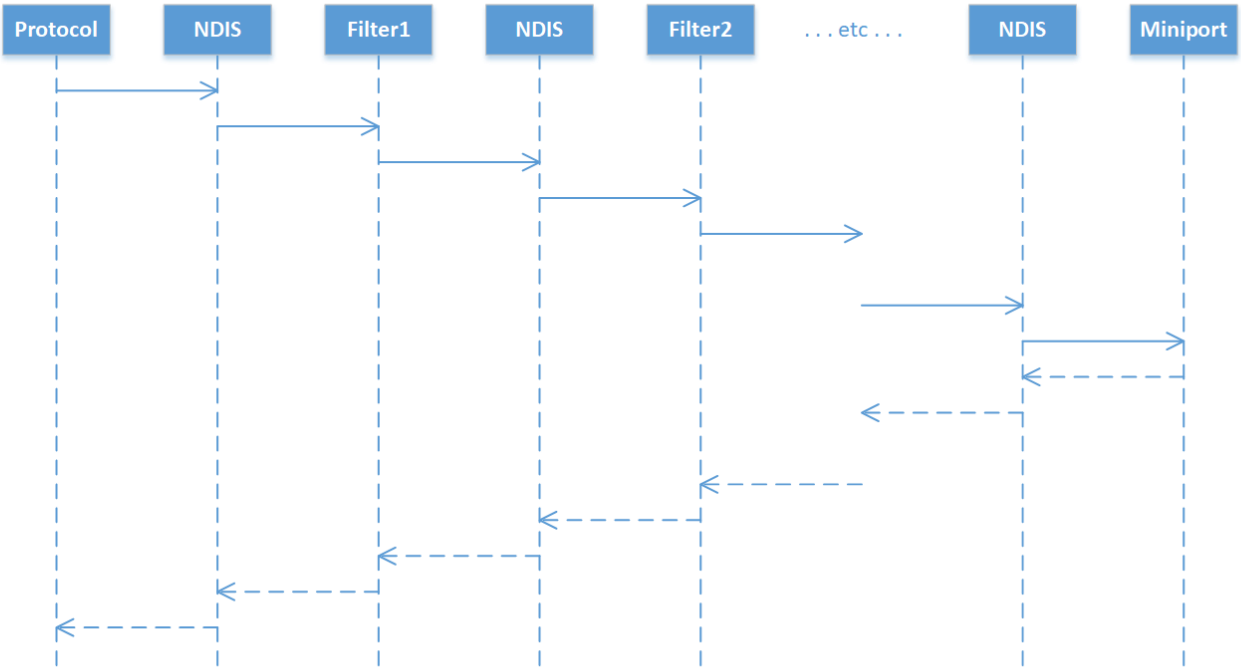
In some cases, a filter module may decide to originate its own OID request. This request starts at the filter module's level and only traverses lower drivers, as the following diagram shows.



All OID requests have this basic flow: a higher driver (either a protocol or filter driver) issues a request and a lower driver (either a miniport or filter driver) completes it.

How Regular and Direct OID requests work

Regular or Direct OID requests are dispatched recursively. The following diagram shows the function call sequence. Note that the sequence itself is much like the sequence described in the diagrams from the previous section, but is arranged to show the recursive nature of the requests.



If there are enough filters installed, NDIS will be forced to allocate a new thread stack to keep recursing deeper.

NDIS considers an NDIS_OID_REQUEST structure to be valid for only a single hop along the stack. If a filter driver wants to pass the request down to the next lower driver (which is the case for the vast majority of OIDs), the filter driver *must* insert several dozen lines of boilerplate code to clone the OID request. This boilerplate has several problems:

1. It forces a memory allocation to clone the OID. Hitting the memory pool is both slow and makes it impossible to guarantee forward progress of the OID request.
2. The OID structure design must remain the same over time because all filter drivers hard-code the mechanics of copying the contents of one NDIS_OID_REQUEST to another.
3. Requiring so much boilerplate obscures what the filter is really doing.

The filtering model for Synchronous OID requests

The filtering model for Synchronous OID requests takes advantage of the synchronous nature of the call, to solve the problems discussed in the previous section.

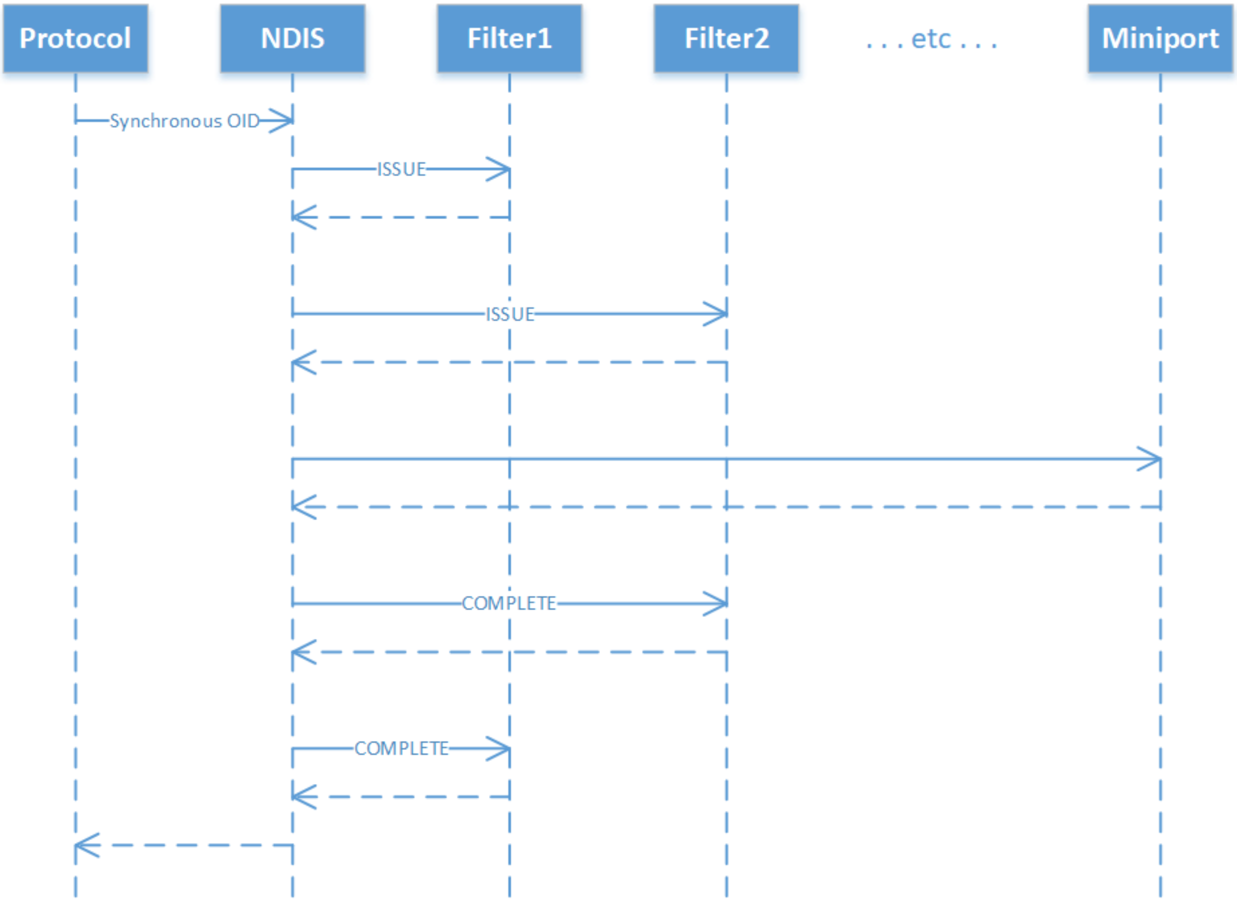
Issue and Complete handlers

Unlike Regular and Direct OID requests, there are two filter hooks for Synchronous OID requests: an Issue handler and a Complete handler. A filter driver can register neither, one, or both hooks.

Issue calls are invoked for each filter driver, starting from the top of the stack down to the bottom of the stack. Any filter's Issue call can stop the OID from continuing downward, and complete the OID with some status code. If no filter decides to intercept the OID, then the OID reaches the NIC driver, which must complete the OID synchronously.

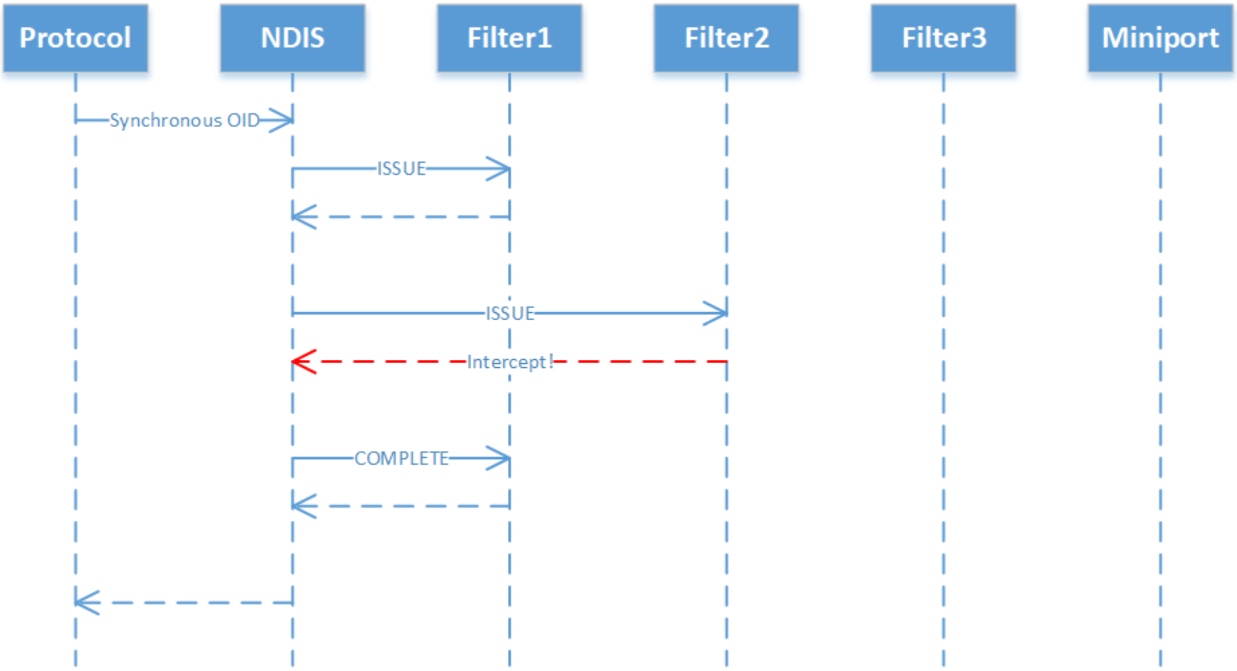
After an OID is completed, Complete calls are invoked for each filter driver, starting from wherever in the stack the OID was completed, up to the top of the stack. A Complete call can inspect or modify the OID request, and inspect or modify the OID's completion status code.

The following diagram illustrates the typical case, where a protocol issues a Synchronous OID request and the filters do not intercept the request.



Note that the call model for Synchronous OIDs is iterative. This keeps stack usage bounded by a constant, eliminating the need to ever expand the stack.

If a filter driver intercepts a Synchronous OID in its Issue handler, the OID is not given to lower filters or the NIC driver. However, Complete handlers for higher filters are still invoked, as shown in the following diagram:



Minimal memory allocations

Regular and Direct OID requests require a filter driver to clone an NDIS_OID_REQUEST. In contrast, Synchronous OID requests are not permitted to be cloned. The advantage of this design is that Synchronous OIDs have lower latency – the OID request isn't repeatedly cloned as it travels down the filter stack – and there are fewer opportunities for failure.

However, that does raise a new problem. If the OID cannot be cloned, where does a filter driver store its per-request state? For example, suppose a filter driver translates one OID to another. On the way down the stack, the filter needs to save off the old OID. On the way back up the stack, the filter needs to restore the old OID.

To solve this problem, NDIS allocates a pointer-sized slot for each filter driver, for each in-flight Synchronous OID request. NDIS preserves this slot across the call from a filter's Issue handler to its Complete handler. This allows the Issue handler to save off state that is later consumed by the Complete handler. The following code snippet shows an example.

C++

```
NDIS_STATUS
MyFilterSynchronousOidRequest(
  _In_ NDIS_HANDLE FilterModuleContext,
  _Inout_ NDIS_OID_REQUEST *OidRequest,
  _Outptr_result_maybenull_ PVOID *CallContext)
{
  if ( . . . should intercept this OID . . . )
  {
    // preserve the original buffer in the CallContext
    *CallContext = OidRequest->DATA.SET_INFORMATION.InformationBuffer;

    // replace the buffer with a new one
    OidRequest->DATA.SET_INFORMATION.InformationBuffer = . . . something . .
.;
  }

  return NDIS_STATUS_SUCCESS;
}

VOID
MyFilterSynchronousOidRequestComplete(
  _In_ NDIS_HANDLE FilterModuleContext,
  _Inout_ NDIS_OID_REQUEST *OidRequest,
  _Inout_ NDIS_STATUS *Status,
  _In_ PVOID CallContext)
{
  // if the context is not null, we must have replaced the buffer.
  if (CallContext != null)
  {
    // Copy the data from the miniport back into the protocol’s original
buffer.
    RtlCopyMemory(CallContext, OidRequest-
>DATA.SET_INFORMATION.InformationBuffer,...);

    // restore the original buffer into the OID request
    OidRequest->DATA.SET_INFORMATION.InformationBuffer = CallContext;
  }
}
```

NDIS saves one PVOID per filter per call. NDIS heuristically allocates a reasonable number of slots on the stack, so that there are zero pool allocations in the common case. This is usually no more than seven filters. If the user sets up a pathological case, NDIS does fall back to a pool allocation.

Reduced boilerplate

Consider the boilerplate on Example boilerplate for handling Regular or Direct OID requests. That code is the cost of entry just to register an OID handler. If you want to issue your own OIDs, you have to add another dozen lines of boilerplate. With Synchronous OIDs, there's no need for the additional complexity of handling asynchronous completion. Therefore, you can cut out much of that boilerplate.

Here's a minimal issue handler with Synchronous OIDs:

```C++
NDIS_STATUS
MyFilterSynchronousOidRequest(
  NDIS_HANDLE FilterModuleContext,
  NDIS_OID_REQUEST *OidRequest,
  PVOID *CallContext)
{
  return NDIS_STATUS_SUCCESS;
}
```

If you want to intercept or modify a particular OID, you can do it by adding just a couple lines of code. The minimal Complete handler is even simpler:

```C++
VOID
MyFilterSynchronousOidRequestComplete(
  NDIS_HANDLE FilterModuleContext,
  NDIS_OID_REQUEST *OidRequest,
  NDIS_STATUS *Status,
  PVOID CallContext)
{
  return;
}
```

Likewise, a filter driver can issue a new Synchronous OID request of its own using only one line of code:

```C++
status = NdisFSynchronousOidRequest(binding->NdisBindingHandle, &oid);
```

In contrast, a filter driver that needs to issue a Regular or Direct OID must set up an asynchronous completion handler and implement some code to distinguish its own OID completions from the completions of OIDs that it just cloned. An example of this boilerplate is shown on Example boilerplate for issuing a Regular OID request.

Interoperability

Although the Regular, Direct, and Synchronous calling styles all use the same data structures, the pipelines do not go to the same handler in the miniport. Furthermore, some OIDs cannot be used in some of the pipelines. For example, OID_PNP_SET_POWER requires careful synchronization and often forces the miniport to make blocking calls. This makes handling it difficult in a Direct OID callback and prevents its use in a Synchronous OID callback.

Therefore, just as with Direct OID requests, Synchronous OID calls can only be used with a subset of OIDs. In Windows 10, version 1709, only the

OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OID used in Receive Side Scaling Version 2 (RSSv2) is supported in the Synchronous OID path.

Implementing Synchronous OID requests

For more info about implementing the Synchronous OID request interface in drivers, see the following topics:

- Miniport Adapter OID Requests
- Filter Module OID Requests
- Protocol Driver OID Requests

Example boilerplate for handling Regular or Direct OID requests

Article • 12/15/2021

This topic provides example boilerplate code for handling Regular or Direct OID requests, to contrast against the examples in Synchronous OID Request Interface in NDIS 6.80. The Synchronous OID Request Interface is available on Windows 10, version 1709 and later.

C++

```
NDIS_STATUS
FilterOidRequest(
  NDIS_HANDLE     FilterModuleContext,
  PNDIS_OID_REQUEST  Request)
{
  PMS_FILTER       pFilter = (PMS_FILTER)FilterModuleContext;

  Status = NdisAllocateCloneOidRequest(pFilter->FilterHandle,
                     Request,
                     FILTER_TAG,
                    &ClonedRequest);
  if (Status != NDIS_STATUS_SUCCESS)
    return Status;

  Context = (PFILTER_REQUEST_CONTEXT)(&ClonedRequest->SourceReserved[0]);
  *Context = Request;

  ClonedRequest->RequestId = Request->RequestId;

  Status = NdisFOidRequest(pFilter->FilterHandle, ClonedRequest);

  if (Status != NDIS_STATUS_PENDING)
  {
    FilterOidRequestComplete(pFilter, ClonedRequest, Status);
    Status = NDIS_STATUS_PENDING;
  }
  return Status;
}

VOID
FilterOidRequestComplete(
  NDIS_HANDLE     FilterModuleContext,
  PNDIS_OID_REQUEST  Request,
  NDIS_STATUS     Status)
{
  PMS_FILTER             pFilter = (PMS_FILTER)FilterModuleContext;
  PNDIS_OID_REQUEST          OriginalRequest;
  Context = (PFILTER_REQUEST_CONTEXT)(&Request->SourceReserved[0]);
  OriginalRequest = (*Context);

  //
  // Copy the information from the returned request to the original request
  //
  switch(Request->RequestType)
  {
    case NdisRequestMethod:
      OriginalRequest->DATA.METHOD_INFORMATION.OutputBufferLength = Request-
>DATA.METHOD_INFORMATION.OutputBufferLength;
      OriginalRequest->DATA.METHOD_INFORMATION.BytesRead = Request-
>DATA.METHOD_INFORMATION.BytesRead;
      OriginalRequest->DATA.METHOD_INFORMATION.BytesNeeded = Request-
>DATA.METHOD_INFORMATION.BytesNeeded;
      OriginalRequest->DATA.METHOD_INFORMATION.BytesWritten = Request-
>DATA.METHOD_INFORMATION.BytesWritten;
      break;

    case NdisRequestSetInformation:
      OriginalRequest->DATA.SET_INFORMATION.BytesRead = Request-
>DATA.SET_INFORMATION.BytesRead;
      OriginalRequest->DATA.SET_INFORMATION.BytesNeeded = Request-
>DATA.SET_INFORMATION.BytesNeeded;
      break;

    case NdisRequestQueryInformation:
    case NdisRequestQueryStatistics:
    default:
      OriginalRequest->DATA.QUERY_INFORMATION.BytesWritten = Request-
>DATA.QUERY_INFORMATION.BytesWritten;
      OriginalRequest->DATA.QUERY_INFORMATION.BytesNeeded = Request-
>DATA.QUERY_INFORMATION.BytesNeeded;
      break;
  }

  NdisFreeCloneOidRequest(pFilter->FilterHandle, Request);

  NdisFOidRequestComplete(pFilter->FilterHandle, OriginalRequest, Status);
}
```

Example boilerplate for issuing a Regular OID request

Article • 05/30/2023

This topic provides example boilerplate code for issuing a Regular OID request, to contrast against the examples in Synchronous OID Request Interface in NDIS 6.80. The Synchronous OID Request Interface is available on Windows 10, version 1709 and later.

This example is taken from the sample NDIS filter driver.

```C++
Status = filterDoInternalRequest(pFilter,
                 NdisRequestQueryInformation,
                 OID_802_3_CURRENT_ADDRESS,
                 MacAddress,
                 sizeof(MacAddress),
                 0,
                 0,
                 &BytesProcessed);

_IRQL_requires_max_(DISPATCH_LEVEL)
NDIS_STATUS
filterDoInternalRequest(
  _In_ PMS_FILTER          FilterModuleContext,
  _In_ NDIS_REQUEST_TYPE      RequestType,
  _In_ NDIS_OID           Oid,
  _Inout_updates_bytes_to_(InformationBufferLength, *pBytesProcessed)
     PVOID            InformationBuffer,
  _In_ ULONG            InformationBufferLength,
  _In_opt_ ULONG          OutputBufferLength,
  _In_ ULONG            MethodId,
  _Out_ PULONG           pBytesProcessed
  )
{
  FILTER_REQUEST       FilterRequest;
  PNDIS_OID_REQUEST      NdisRequest = &FilterRequest.Request;
  NDIS_STATUS         Status;
  BOOLEAN           bFalse;

  bFalse = FALSE;
  *pBytesProcessed = 0;
  NdisZeroMemory(NdisRequest, sizeof(NDIS_OID_REQUEST));

  NdisInitializeEvent(&FilterRequest.ReqEvent);

  NdisRequest->Header.Type = NDIS_OBJECT_TYPE_OID_REQUEST;
  NdisRequest->Header.Revision = NDIS_OID_REQUEST_REVISION_1;
  NdisRequest->Header.Size = sizeof(NDIS_OID_REQUEST);
  NdisRequest->RequestType = RequestType;

  switch (RequestType)
  {
    case NdisRequestQueryInformation:
       NdisRequest->DATA.QUERY_INFORMATION.Oid = Oid;
       NdisRequest->DATA.QUERY_INFORMATION.InformationBuffer =
                  InformationBuffer;
       NdisRequest->DATA.QUERY_INFORMATION.InformationBufferLength =
                  InformationBufferLength;
      break;

    case NdisRequestSetInformation:
       NdisRequest->DATA.SET_INFORMATION.Oid = Oid;
       NdisRequest->DATA.SET_INFORMATION.InformationBuffer =
                  InformationBuffer;
       NdisRequest->DATA.SET_INFORMATION.InformationBufferLength =
                  InformationBufferLength;
      break;

    case NdisRequestMethod:
       NdisRequest->DATA.METHOD_INFORMATION.Oid = Oid;
       NdisRequest->DATA.METHOD_INFORMATION.MethodId = MethodId;
       NdisRequest->DATA.METHOD_INFORMATION.InformationBuffer =
                  InformationBuffer;
       NdisRequest->DATA.METHOD_INFORMATION.InputBufferLength =
                  InformationBufferLength;
       NdisRequest->DATA.METHOD_INFORMATION.OutputBufferLength =
OutputBufferLength;
       break;



    default:
      FILTER_ASSERT(bFalse);
      break;
  }

  NdisRequest->RequestId = (PVOID)FILTER_REQUEST_ID;

  Status = NdisFOidRequest(FilterModuleContext->FilterHandle,
              NdisRequest);


  if (Status == NDIS_STATUS_PENDING)
  {
    NdisWaitEvent(&FilterRequest.ReqEvent, 0);
    Status = FilterRequest.Status;
  }

  if (Status == NDIS_STATUS_SUCCESS)
  {
    if (RequestType == NdisRequestSetInformation)
    {
      *pBytesProcessed = NdisRequest->DATA.SET_INFORMATION.BytesRead;
    }

    if (RequestType == NdisRequestQueryInformation)
    {
      *pBytesProcessed = NdisRequest->DATA.QUERY_INFORMATION.BytesWritten;
    }

    if (RequestType == NdisRequestMethod)
    {
      *pBytesProcessed = NdisRequest->DATA.METHOD_INFORMATION.BytesWritten;
    }

    //
    // The driver below should set the correct value to BytesWritten
    // or BytesRead. But now, we just truncate the value to
InformationBufferLength
    //
    if (RequestType == NdisRequestMethod)
    {
      if (*pBytesProcessed > OutputBufferLength)
      {
        *pBytesProcessed = OutputBufferLength;
      }
    }
    else
    {

      if (*pBytesProcessed > InformationBufferLength)
      {
        *pBytesProcessed = InformationBufferLength;
      }
    }
  }

  return Status;
}

VOID
filterInternalRequestComplete(
  _In_ NDIS_HANDLE         FilterModuleContext,
  _In_ PNDIS_OID_REQUEST      NdisRequest,
  _In_ NDIS_STATUS         Status
  )
{
  PFILTER_REQUEST       FilterRequest;
  FilterRequest = CONTAINING_RECORD(NdisRequest, FILTER_REQUEST, Request);
  FilterRequest->Status = Status;
  NdisSetEvent(&FilterRequest->ReqEvent);
}

_Use_decl_annotations_
VOID
FilterOidRequestComplete(
  NDIS_HANDLE     FilterModuleContext,
  PNDIS_OID_REQUEST  Request,
  NDIS_STATUS     Status
  )
{
  PMS_FILTER             pFilter = (PMS_FILTER)FilterModuleContext;
  PNDIS_OID_REQUEST          OriginalRequest;
  PFILTER_REQUEST_CONTEXT       Context;

  Context = (PFILTER_REQUEST_CONTEXT)(&Request->SourceReserved[0]);
  OriginalRequest = (*Context);

  //
  // This is an internal request
  //
  if (OriginalRequest == NULL)
  {
    filterInternalRequestComplete(pFilter, Request, Status);
    return;
  }

  // . . . other code for handling completion of non-"internal" requests
}
```

Receive Side Scaling Version 2 (RSSv2) in NDIS 6.80

Article • 03/14/2023

⚠ Warning

Some information in this topic relates to prereleased product, which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

RSSv2 is preview only in Windows 10, version 1809.

Receive Side Scaling improves the system performance related to handling of network data on multiprocessor systems. NDIS 6.80 and later support RSS Version 2 (RSSv2), which extends RSS by offering per-VPort spreading of queues.

RSSv2 uses the NDIS 6.80 Synchronous OID request interface for one of its OIDs. For more info about Synchronous OID calls, see Synchronous OID request interface in NDIS 6.80.

For more info about RSSv2, see Receive Side Scaling Version 2 (RSSv2).

Introduction to NDIS 6.70

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.70 and describes its major design additions. NDIS 6.70 is included in Windows 10, version 1703.

NDIS 6.70 is a minor version update to NDIS 6.60 for miniport, protocol, filter, and intermediate drivers. For more information about porting NDIS 6.x drivers to NDIS 6.70, see Porting NDIS 6.x drivers to NDIS 6.70.

Feature updates

NetAdapterCx

Alongside NDIS 6.70, Windows 10, version 1703 includes a major new feature for NIC drivers called the Network Adapter WDF Class Extension, a.k.a. NetAdapterCx. NetAdapterCx is preview only in Windows 10, version 1703. The NetAdapterCx model enables NIC driver developers to harness the full functionality and simplified driver model of WDF, meaning NIC drivers are easier to write.

Other feature updates

NDIS forms the core foundation for the network driver platform on Windows. For a list of other network driver features that were updated at the same time as NDIS 6.70, see the Windows 10, version 1703 section for Networking on What's new in driver development.

Feature deprecations

The following network driver features have been deprecated along with the release of NDIS 6.70:

- TCP Chimney Offload
- IPsec Offload Version 2

Implementing an NDIS 6.70 driver

NIC drivers

For more information about implementing a NIC driver with the NetAdapterCx, see NetAdapterCx.

Miniport, protocol, filter, and intermediate drivers

An NDIS 6.70 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.70 driver must be compliant with the following requirements:

- An NDIS 6.70 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.70. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 70. This requirement applies to miniport, protocol and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.70 driver).
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_2.
  - Protocol drivers must set the **Header** member of **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

Compiling an NDIS 6.70 driver

NIC drivers

For more information about compiling a NIC driver with the NetAdapterCx, see Porting NDIS miniport drivers to NetAdapterCx (Compilation settings).

Miniport, protocol, and filter drivers

The WDK for Windows 10, version 1703 supports header versioning. Header versioning makes sure that NDIS 6.70 drivers use the appropriate NDIS 6.70 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS670_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS670=1`.

For information on building a driver with the Windows 10, version 1703 release of the WDK, see Building a Driver.

Using NDIS 6.70 driver data structures

NIC drivers

For more information about NetAdapterCx data structures, see NetAdapterCx.

Miniport, protocol, filter, and intermediate drivers

New data structures

The following data structures are new in NDIS 6.70.

- NDIS_STATUS_WWAN_DEVICE_CAPS_EX

Introduction to NDIS 6.60

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.60 and describes its major design additions. NDIS 6.60 is included in Windows 10, version 1607 and Windows Server 2016 and later.

NDIS 6.60 is a minor version update to NDIS 6.50. For more information about porting NDIS 6.x drivers to NDIS 6.60, see Porting NDIS 6.x drivers to NDIS 6.60.

Feature updates

NDIS 6.60 is an incremental update to NDIS 6.50 and does not contain any major new features.

Implementing an NDIS 6.60 driver

An NDIS 6.60 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.60 driver must be compliant with the following requirements:

- An NDIS 6.60 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.60. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 60. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.60 driver).
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_2.

  - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.60 miniport drivers for Windows 10, version 1607 and Windows Server 2016 and later must use the NDIS 6.60 versions of data structures. For more information, see Using NDIS 6.60 data structures.

Compiling an NDIS 6.60 driver

The WDK for Windows 10, version 1607 supports header versioning. Header versioning makes sure that NDIS 6.60 drivers use the appropriate NDIS 6.60 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS660_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS660=1`.

For information on building a driver with the Windows 10, version 1607 release of the WDK, see Building a Driver.

Using NDIS 6.60 data structures

Updated data structures

The following data structures were updated in NDIS 6.60.

- NDIS_NIC_SWITCH_CAPABILITIES
- NDIS_RECEIVE_SCALE_PARAMETERS
- NDIS_RECEIVE_SCALE_CAPABILITIES

Introduction to NDIS 6.50

Article • 03/14/2023

This topic introduces Network Driver Interface Specification (NDIS) 6.50 and describes its major design additions. NDIS 6.50 is included in Windows 10, version 1507 and later.

NDIS 6.50 is a minor version update to NDIS 6.40. For more information about porting NDIS 6.x drivers to NDIS 6.50, see Porting NDIS 6.x drivers to NDIS 6.50.

Feature updates

NDIS 6.50 is an incremental update to NDIS 6.40 and does not contain any major new features.

Implementing an NDIS 6.50 driver

An NDIS 6.50 driver must follow the requirements that are defined in Implementing an NDIS 6.30 driver.

In addition, an NDIS 6.50 driver must be compliant with the following requirements:

- An NDIS 6.50 driver must report the correct NDIS version when it registers with NDIS.
    - You must update the major and minor NDIS version number in the NDIS_Xxx_DRIVER_CHARACTERISTICS structure to support NDIS 6.50. The MajorNdisVersion member must contain 6 and the MinorNdisVersion member must contain 50. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler (see Compiling an NDIS 6.50 driver).
    - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2.
    - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_2.

  - Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

- NDIS 6.50 miniport drivers for Windows 10, version 1507 and later must use the NDIS 6.50 versions of data structures. For more information, see Using NDIS 6.50 data structures.

Compiling an NDIS 6.50 driver

The WDK for Windows 10, version 1507 supports header versioning. Header versioning makes sure that NDIS 6.50 drivers use the appropriate NDIS 6.50 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add `NDIS650_MINIPORT=1`.
- For a filter or protocol driver, add `NDIS650=1`.

For information on building a driver with the Windows 10, version 1507 release of the WDK, see Building a Driver.

Using NDIS 6.50 data structures

New data structures

The following data structures are new in NDIS 6.50.

- OID_WWAN_SYS_CAPS
- OID_WWAN_DEVICE_CAPS_EX
- OID_WWAN_SLOT_INFO_STATUS
- OID_WWAN_NETWORK_IDLE_HINT
- NDIS_STATUS_PD_CURRENT_CONFIG
- NDIS_PD_CAPABILITIES
- NDIS_PD_CLOSE_PROVIDER_PARAMETERS
- NDIS_PD_CONFIG
- NDIS_PD_COUNTER_PARAMETERS
- NDIS_PD_COUNTER_VALUE
- NDIS_PD_FILTER_COUNTER
- NDIS_PD_FILTER_PARAMETERS

- NDIS_PD_ON_RSS_QUEUE_PARAMETERS
- NDIS_PD_OPEN_PROVIDER_PARAMETERS
- NDIS_PD_PROVIDER_DISPATCH
- NDIS_PD_QUEUE
- NDIS_PD_QUEUE_DISPATCH
- NDIS_PD_QUEUE_PARAMETERS
- NDIS_PD_RECEIVE_QUEUE_COUNTER
- NDIS_PD_TRANSMIT_QUEUE_COUNTER
- PD_BUFFER
- PD_BUFFER_8021Q_INFO
- PD_BUFFER_VIRTUAL_SUBNET_INFO

Updated data structures

The following data structures were updated in NDIS 6.50.

- NET_PNP_EVENT_NOTIFICATION
- NDIS_OID_REQUEST
- NDIS_NET_BUFFER_LIST_INFO
- NdisMGetDeviceProperty
- NDIS_SWITCH_OPTIONAL_HANDLERS
- NDIS_SWITCH_NIC_SAVE_STATE
- NDIS_RECEIVE_FILTER_PARAMETERS

NDIS 6.51

NDIS 6.51 is a very minor version update to NDIS 6.50. NDIS 6.51 is included in Windows 10, version 1511 and later. All information for NDIS 6.50 also applies to NDIS 6.51 with the following exceptions:

- The MinorNdisVersion changes from 50 to 51 when registering your driver with NDIS.
- The compiler settings change from `NDIS650_MINIPORT=1` for miniport drivers and `NDIS650=1` for filter or protocol drivers, to `NDIS651_MINIPORT=1` and `NDIS651=1` respectively.

Introduction to NDIS 6.40

Article • 03/14/2023

This section introduces Network Driver Interface Specification (NDIS) 6.40 and describes its major design additions. NDIS 6.40 is included in the Windows 8.1 and Windows Server 2012 R2 and later operating systems.

Feature Updates

Windows 8.1 and Windows Server 2012 R2 introduce minor updates to the following features:

NDKPI

NDKPI 1.2 adds the following new elements to the NDKPI DDI:

- *NdkSendAndInvalidate* (*NDK_FN_SEND_AND_INVALIDATE*) function
- *NdkGetCqResultsEx* (*NDK_FN_GET_CQ_RESULTS_EX*) function
- **NDK_RESULT_EX** structure
- New request callback *Flags* value: **NDK_OP_FLAG_DEFER**
- New **NDK_ADAPTER_INFOAdapterFlags** value: **NDK_ADAPTER_FLAG_RDMA_READ_LOCAL_INVALIDATE_SUPPORTED**

Native 802.11 Wireless LAN

IEEE 802.11ac very-high throughput (VHT) PHY is now supported. The following DDI elements have been updated:

- **DOT11_PHY_TYPE** enumeration
- OID_DOT11_CURRENT_CHANNEL
- OID_DOT11_SUPPORTED_PHY_TYPES
- OID_DOT11_SUPPORTED_OFDM_FREQUENCY_LIST

Sample and Documentation Updates

The Hyper-V Extensible Switch forwarding extension sample has been updated to implement Hybrid Forwarding.

The following documentation sections have been added or significantly expanded:

- Porting NDIS 6.x Drivers to NDIS 6.30
- Network Direct Kernel Provider Interface (NDKPI) Design Guide
- Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload
- Receive Segment Coalescing (RSC) Design Guide
- Getting Started Writing a Hyper-V Extensible Switch Extension
- NVGRE Task Offload Reference

The NetDMA interface is not supported in Windows 8 and Windows Server 2012 and later. The documentation has now been updated to reflect this.

This section includes the following topics:

- Implementing an NDIS 6.40 Driver
- Using NDIS 6.40 Data Structures
- Compiling an NDIS 6.40 Driver

Implementing an NDIS 6.40 Driver

Article • 03/14/2023

An NDIS 6.40 driver must follow the requirements that are defined in Implementing an NDIS 6.30 Driver.

In addition, an NDIS 6.40 driver must be compliant with the following requirements:

- An NDIS 6.40 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_*Xxx*_DRIVER_CHARACTERISTICS structure to support NDIS 6.40. The **MajorNdisVersion** member must contain **6** and the **MinorNdisVersion** member must contain **40**. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler, see Compiling an NDIS 6.40 Driver.
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_2.
  - Protocol drivers must set the **Header** member of **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.
- NDIS 6.40 miniport drivers for the Windows 8.1 and Windows Server 2012 R2 operating systems must use the NDIS 6.40 versions of data structures. For more information, see Using NDIS 6.40 Data Structures.

Using NDIS 6.40 Data Structures

Article • 03/14/2023

The following structures and enumerations were updated for NDIS 6.40.

The following union was updated for NDIS 6.40:

- NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO

The following enumeration was updated for NDIS 6.40:

- NDIS_SWITCH_PORT_PROPERTY_TYPE

Compiling an NDIS 6.40 Driver

Article • 03/14/2023

The WDK for Windows 8.1 supports header versioning. Header versioning makes sure that NDIS 6.40 drivers use the appropriate NDIS 6.40 data structures at compile time.

Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add NDIS640_MINIPORT=1.
- For a filter or protocol driver, add NDIS640=1.

For information on building a driver with the Windows 8.1 release of the WDK, see Building a Driver.

For information on converting an driver's build files to a Visual Studio project , see Creating a Driver From Existing Source Files.

Introduction to NDIS 6.30

Article • 03/14/2023

This section introduces Network Driver Interface Specification (NDIS) 6.30 and describes its major design additions. NDIS 6.30 is included in the Windows 8 and Windows Server 2012 and later operating systems.

You should be familiar with earlier versions of NDIS 6.x before learning about NDIS 6.30. For more information about previous NDIS 6.x versions, see the following topics:

Introduction to NDIS 6.0

Introduction to NDIS 6.1

Introduction to NDIS 6.20

This section includes the following topics:

- Virtualized Networking Enhancements in NDIS 6.30
- Power Management Enhancements in NDIS 6.30
- Quality of Service (QoS) Support in NDIS 6.30
- Windows Filtering Platform Enhancements in NDIS 6.30
- Scalable Networking Enhancements in NDIS 6.30
- Implementing an NDIS 6.30 Driver
- Using NDIS 6.30 Data Structures
- Compiling an NDIS 6.30 Driver

Virtualized Networking Enhancements in NDIS 6.30

Article • 03/14/2023

NDIS supports virtualized networking interface that allow Hyper-V parent and child partitions to interface the underlying physical networking interface.

NDIS 6.20 included the virtual machine queue (VMQ) interface to support Microsoft Hyper-V network performance improvements. For more information about VMQ, see Virtual Machine Queue (VMQ).

NDIS 6.30 extends the support for virtualized networking interfaces with the following technologies, as described in Overview of Virtualized Networking:

Single Root I/O Virtualization (SR-IOV)

The SR-IOV interface allows for the partitioning of the hardware resources on a PCI Express (PCIe) network adapter into one or more virtual interfaces, known as *virtual functions (VFs)*. This allows the adapter resources to be shared in a virtual environment. SR-IOV enables network traffic to bypass the virtual software switch layer by assigning a VF to the Hyper-V child partition directly. By doing this, the I/O overhead in the software emulation layer is diminished and network throughput achieves almost the same performance as in non-virtualized environments.

For more information about the SR-IOV interface, see Single Root I/O Virtualization (SR-IOV).

Hyper-V Extensible Switch

The Hyper-V Extensible Switch is a virtualized Ethernet switch that runs in the management operating system of the Hyper-V parent partition. Each instance of the extensible switch routes packets between ports that are connected to the following types of network adapters:

- The external and internal network adapters that are exposed in the management operating system that runs in the Hyper-V parent partition.
- The synthetic or emulated network adapters that are exposed in the guest operating system that runs in a Hyper-V child partition.

Starting with NDIS 6.30, the Hyper-V Extensible Switch supports an extensibility interface. This interface allows instances of NDIS filter drivers (known as *extensions*) to bind within the Hyper-V Extensible Switch driver stack. Once bound and enabled within the driver stack, extensions are exposed to all packet traffic within the extensible switch data path. This allows extensions to monitor, modify, and forward packets to extensible switch ports. This also allows extensions to inspect and inject packets in the virtual network interfaces that are used by the various Hyper-V partitions.

For more information about the Hyper-V extensible switch interface, see Hyper-V Extensible Switch.

Power Management Enhancements in NDIS 6.30

Article • 03/14/2023

NDIS 6.20 included power management new features and improvements to reduce computer power consumption. NDIS 6.30 extends the NDIS 6.20 power management support with the following capabilities, as described in Power Management (NDIS 6.30):

NDIS Packet Coalescing

Starting with NDIS 6.30, network adapters can support NDIS packet coalescing. This feature reduces the processing overhead and power consumption on a host system due to the reception of random broadcast or multicast packets.

For more information, see NDIS Packet Coalescing.

NDIS Selective Suspend

Starting with NDIS 6.30, the NDIS selective suspend interface allows NDIS to suspend an idle network adapter by transitioning the adapter to a low-power state. This enables the system to reduce the power overhead on the CPU and network adapter.

For more information, see NDIS Selective Suspend.

NDIS Wake Reason Status Indications

Starting with NDIS 6.30, miniport drivers issue an NDIS wake reason status indication (**NDIS_STATUS_PM_WAKE_REASON**) to notify NDIS and overlying drivers about the reason for a system wake-up event. If the network adapter generates a wake-up event, the miniport driver immediately issues this NDIS status indication when the system resumes to a full-power state.

Note Support for NDIS wake reason status indications is optional for Mobile Broadband (MB) miniport drivers.

For more information, see NDIS Wake Reason Status Indications.

NDIS No Pause On Suspend

Starting with NDIS 6.30, miniport drivers can specify an attribute flag (**NDIS_MINIPORT_ATTRIBUTES_NO_PAUSE_ON_SUSPEND**) in the **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure. The driver passes a pointer to this structure in its call to the **NdisMSetMiniportAttributes** function.

If the miniport sets the **NDIS_MINIPORT_ATTRIBUTES_NO_PAUSE_ON_SUSPEND** attribute flag, NDIS does not call the miniport driver's *MiniportPause* function before the object identifier (OID) request of OID_PNP_SET_POWER is issued to the driver. When the miniport driver handles the OID request, it must not assume that it had been previously paused when preparing the miniport adapter for the transition to a low-power state.

For more information, see **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES**.

Quality of Service (QoS) Support in NDIS 6.30

Article • 03/14/2023

NDIS 6.30 and later provide support for quality of service (QoS). Miniport drivers use NDIS QoS for traffic prioritization of transmit, or *egress*, packets over a network adapter that is compliant with IEEE 802.1 Data Center Bridging (DCB).

IEEE 802.1 Data Center Bridging (DCB) is a collection of standards that defines a unified 802.3 Ethernet media interface, or fabric, for local area network (LAN) and storage area network (SAN) technologies. DCB extends the current 802.1 bridging specification to support the co-existence of LAN- and SAN-based applications over the same networking fabric within a data center. DCB also supports technologies, such as Fibre Channel over Ethernet (FCoE) and iSCSI, by defining link-level policies that prevents packet loss.

NDIS QoS support for DCB allows a miniport driver to be configured with traffic classes that specify a set of policies. Each policy determine how the network adapter handles egress packets for prioritized delivery.

Each traffic class specifies the following policies that are applied to egress packets:

Priority Level and Flow Control
This policy defines the IEEE 802.1p priority level and optional flow control algorithms for the egress traffic.

Traffic Selection Algorithms (TSAs)
This policy specifies how the network adapter selects egress traffic for delivery from its transmit queues. For example, the adapter could select egress packets based on IEEE 802.1p priority or the percentage of the egress bandwidth that is allocated to each traffic class.

For more information about NDIS QoS support for DCB, see NDIS QoS for Data Center Bridging.

Windows Filtering Platform Enhancements in NDIS 6.30

Article • 03/14/2023

Windows filtering platform (WFP) includes the following enhancements for NDIS 6.30 and later drivers:

Layer 2 Filtering Proxied Connections Tracking Virtual Switch Filtering For more information about new WFP features, see New Information for WFP.

Scalable Networking Enhancements in NDIS 6.30

Article • 03/14/2023

Scalable networking includes the following enhancements for NDIS 6.30 and later drivers:

Receive Segment Coalescing (RSC) Network Direct Kernel Provider Interface (NDKPI) Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload IPsec Offload Version 2 (IPsecOV2) Updates Receive Side Scaling (RSS) Updates

Note NetDMA is not supported in Windows 8 and later versions of the Windows operating system.

Implementing an NDIS 6.30 Driver

Article • 03/14/2023

An NDIS 6.30 driver must follow the requirements that are defined in Implementing an NDIS 6.20 Driver.

In addition, an NDIS 6.30 driver must be compliant with the following requirements:

- An NDIS 6.30 driver must report the correct NDIS version when it registers with NDIS.
  - You must update the major and minor NDIS version number in the NDIS_*Xxx*_DRIVER_CHARACTERISTICS structure to support NDIS 6.30. The **MajorNdisVersion** member must contain **6** and the **MinorNdisVersion** member must contain **30**. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler, see Compiling an NDIS 6.30 Driver.
  - Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2.
  - Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_2.
  - Protocol drivers must set the **Header** member of **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.
- To inform NDIS and overlying drivers about device and driver capabilities, NDIS 6.30 drivers must implement the NDIS 6.30 device capability interfaces for the following features:
  - Virtualized Networking
  - Power Management
  - NDIS Quality of Service (QoS)

- NDIS 6.30 miniport drivers for the Windows 8 and Windows Server 2012 operating systems must use the NDIS 6.30 versions of data structures. For more information, see Using NDIS 6.30 Data Structures.

Using NDIS 6.30 Data Structures

Article • 03/14/2023

NDIS can support multiple versions of the same data structure. For the Windows 8 and Windows Server 2012 operating systems, miniport drivers that use an NDIS 6.30 version of a structure must initialize the **Header** member of the structure with the correct version and size values. The **Header** member is an NDIS_OBJECT_HEADER structure, and the driver must initialize the **Revision** member and **Size** member value of the **Header** member to the NDIS 6.30 version and size values.

Note To determine the correct version and size information see the reference pages for each structure that includes a **Header** member. The reference pages for structures that contain a **Header** member and that were updated for NDIS 6.30 include new information for NDIS 6.30 drivers. If there is no update to the structure for NDIS 6.30, the information that is provided for earlier versions of NDIS also applies to NDIS 6.30 drivers.

The following structures were updated for NDIS 6.30:

- NDIS_BIND_PARAMETERS
- NDIS_MINIPORT_ADAPTER_ATTRIBUTES
- NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES
- NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES
- NDIS_MINIPORT_ADAPTER_NATIVE_802_11_ATTRIBUTES
- NDIS_NET_BUFFER_LIST_FILTERING_INFO
- NDIS_NIC_SWITCH_CAPABILITIES
- NDIS_OFFLOAD
- NDIS_OFFLOAD_PARAMETERS
- NDIS_PM_CAPABILITIES
- NDIS_PM_PARAMETERS
- NDIS_RECEIVE_FILTER_CAPABILITIES
- NDIS_RECEIVE_FILTER_INFO_ARRAY
- NDIS_RECEIVE_FILTER_PARAMETERS
- NDIS_RECEIVE_QUEUE_INFO
- NDIS_RECEIVE_QUEUE_PARAMETERS
- NDIS_RECEIVE_SCALE_CAPABILITIES
- NDIS_RSS_PROCESSOR_INFO
- NDIS_SHARED_MEMORY_PARAMETERS

Compiling an NDIS 6.30 Driver

Article • 03/14/2023

In the Windows 8 release of the Windows Driver Kit (WDK), the driver development environment is integrated into Visual Studio. Most of the tools you need for coding, building, testing, debugging, deploying, and releasing a driver are available in the Visual Studio user interface. This is a departure from previous releases of the WDK where the various stages of the driver life cycle were performed as separate tasks with stand-alone tools.

The WDK for Windows 8 supports header versioning. Header versioning makes sure that NDIS 6.30 drivers use the appropriate NDIS 6.30 data structures at compile time. Add the following compiler settings to the Visual Studio project for your driver:

- For a miniport driver, add NDIS630_MINIPORT=1.
- For a filter or protocol driver, add NDIS630=1.

For information on building a driver with the Windows 8 release of the WDK, see Building a Driver.

For information on converting an driver's build files to a Visual Studio project , see Creating a Driver From Existing Source Files.

Introduction to NDIS 6.20

Article • 03/14/2023

This section introduces Network Driver Interface Specification (NDIS) 6.20 and describes its major design additions. NDIS 6.20 is included in Windows 7 and Windows Server 2008 R2 and later.

You should be familiar with NDIS 6.0 and NDIS 6.1 before learning about NDIS 6.20:

- For more information about NDIS 6.0, see Introduction to NDIS 6.0.
- For more information about NDIS 6.1, see Introduction to NDIS 6.1.

For a description of the differences between NDIS 6.20 and earlier versions, and for detailed information about porting drivers to NDIS 6.20, see Porting NDIS 6.x Drivers to NDIS 6.20.

This section includes the following topics:

- Power Management Enhancements in NDIS 6.20
- Virtual Machine Queue (VMQ) in NDIS 6.20
- Support for More than 64 Processors in NDIS 6.20
- Receive Side Throttle in NDIS 6.20
- Media Extensibility in NDIS 6.20
- Implementing an NDIS 6.20 Driver
- Using NDIS 6.20 Data Structures
- Compiling an NDIS 6.20 Driver

Related topics

Introduction to NDIS 6.30

Power Management Enhancements in NDIS 6.20

Article • 03/14/2023

NDIS 6.20 introduces power management enhancements to reduce computer power consumption. NDIS 6.20 power management support is mandatory for NDIS 6.20 and later drivers.

The NDIS 6.20 power management interface is backward compatible with NICs and miniport drivers that do not support the latest power management features.

The power management interface in NDIS 6.20 and later supports:

- Wake on LAN (WOL) patterns that are based on the packet type in addition to the NDIS 6.1 and earlier methods that support an offset and pattern match. Therefore, NDIS 6.20 and later WOL patterns can be more specific to avoid unnecessary wake up events. For example, a NIC can identify TCP synchronize (SYN) packets.

- Protocol offloads to NICs for some of the most common protocols. Because the protocols are offloaded to the NIC, it can respond on behalf of the computer to avoid unwanted wake up events. For example, a NIC can handle IPv4 Address Resolution Protocol (ARP) and IPv6 Neighbor Solicitation (NS) protocol packets without waking the computer.

The power management interface in NDIS 6.20 and later also supports:

- WOL WLAN enhancements. If necessary, a NIC can handle IEEE 802.11 group temporal key (GTK) rekey requests in a low power state.

- NDIS 6.20 and later can wake the computer when media connects. The operating system puts the NIC in a low power state when the media is disconnected.

Some of the NDIS device driver interface elements are obsolete for NDIS 6.20 and later drivers. For more information about obsolete interfaces, see Obsolete Interfaces in NDIS 6.20.

For more information about power management for NDIS 6.20 and later versions of NDIS, see Power Management (NDIS 6.20).

Related topics

Power Management Enhancements in NDIS 6.30

Virtual Machine Queue (VMQ) in NDIS 6.20

Article • 03/14/2023

NDIS 6.20 introduces the virtual machine queue (VMQ) interface to support Microsoft Hyper-V network performance improvements. NDIS 6.20 and later drivers must provide information about VMQ capabilities during initialization. However, VMQ support is optional.

The VMQ interface in NDIS 6.20 and later supports:

- Classification of received packets by using the destination MAC address to route the packets to different receive queues.
- NIC ability to use DMA to transfer packets directly to a Hyper-V child partition's shared memory.
- Scaling to multiple processors by processing packets for different Hyper-V partitions on different processors.

Microsoft Hyper-V network performance enhancements also provide chimney support for Hyper-V child partitions with no driver changes.

For more information about VMQ, see Virtual Machine Queue (VMQ).

Support for More than 64 Processors in NDIS 6.20

Article • 03/14/2023

The NDIS 6.20 interface introduces support for more than 64 processors. Previous NDIS versions are limited to no more than 64 processors (32 in x86 versions of the operating system).

To remain backward compatible with older implementations, NDIS supports processor groups. Software that has not been updated to support more than 64 processors can default to processor group zero.

To support more than 64 processors, NDIS 6.20 and later provide updated versions of these interfaces:

- Receive Side Scaling (RSS)
- Processor information device driver interfaces (see NDIS System Information Functions)
- Resource allocation (see NDIS Memory Management Interface)
- Read and write locks (see NDIS Read Write Lock Reference)

Some of the NDIS device driver interface elements are obsolete for NDIS 6.20 and later drivers. For more information about obsolete interfaces, see Obsolete Interfaces in NDIS 6.20.

Receive Side Throttle in NDIS 6.20

Article • 03/14/2023

NDIS 6.20 introduces receive-side throttle (RST) enhancements to reduce the possibility of disruptions during media playback in multimedia applications. RST support is mandatory for NDIS 6.20 and later drivers.

If an NDIS driver spends too much time at dispatch IRQ level in a deferred procedure call (DPC), it increases the scheduling latency for multimedia application threads and might cause disruptions during media playback. To improve media playback with NDIS 6.20 and later drivers, NDIS can control the number of packets that a miniport driver indicates in a receive DPC.

Related topics

MiniportInterrupt

MiniportInterruptDPC

MiniportMessageInterrupt

MiniportMessageInterruptDPC

NDIS_RECEIVE_THROTTLE_PARAMETERS

NdisMQueueDpcEx

Media Extensibility in NDIS 6.20

Article • 03/14/2023

NDIS 6.20 introduces additional media extensibility features. That is, the network layer of the driver stack is more media independent.

These features in NDIS 6.20 and later reduce the complexity of the code that is required to implement drivers that do not implement IEEE 802.3. In addition, these non-IEEE 802.3 implementations do not have to implement ARP and DHCP.

NDIS 6.20 and later provide *raw IP* frame support with a new media type for raw IP (NdisMediumIP). For example, NDIS WWAN support uses raw IP.

NDIS 6.20 introduces enhanced support for media specific out of band (OOB) data. The media specific information has a tag that Microsoft assigns. NDIS 6.20 and later support multiple media specific information tags.

For more information about media specific information, for more information about media extensibility, see OID_GEN_PHYSICAL_MEDIUM_EX and **NDIS_NBL_MEDIA_SPECIFIC_INFORMATION_EX**.

Implementing an NDIS 6.20 Driver

Article • 03/14/2023

An NDIS 6.20 driver must report the correct NDIS version when it registers with NDIS:

- You must update the major and minor NDIS version number in the NDIS_*Xxx*_DRIVER_CHARACTERISTICS structure to support NDIS 6.20. The **MajorNdisVersion** member must contain 6 and the **MinorNdisVersion** member must contain 20. This requirement applies to miniport, protocol, and filter drivers. You must also update the version information for the compiler, see Compiling an NDIS 6.20 Driver.

- Miniport drivers must set the **Header** member of **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2.

- Filter drivers must set the **Header** member of **NDIS_FILTER_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_2.

- Protocol drivers must set the **Header** member of **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS**: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

The NDIS 6.20 power management services are mandatory for NDIS 6.20 and later miniport drivers. For more information about the NDIS 6.20 power management interface, see Power Management Enhancements in NDIS 6.20.

The NDIS direct OID request interface is mandatory for NDIS 6.20 and later miniport drivers. For more information about the direct OIDs interface, see Direct OID Request Interface in NDIS 6.1.

To inform NDIS and overlying drivers about device and driver capabilities, NDIS 6.20 and later drivers must implement the NDIS 6.20 device capability interfaces for the following features:

- Power Management

- Receive Side Scaling (RSS)

- Virtual Machine Queue (VMQ)

NDIS 6.20 and later drivers must support receive side throttle (RST) in receive interrupts. For more information about RST, see Receive Side Throttle in NDIS 6.20.

Replace code that uses obsolete interfaces with the NDIS 6.20 equivalents. For more information about obsolete functions, see Obsolete Interfaces in NDIS 6.20. For information about updating structures to support NDIS 6.20 versions, see Using NDIS 6.20 Data Structures.

Use NDIS interfaces that support more than 64 processors, for example, use the NDIS 6.20 read and write lock interface. For more information about support for more than 64 processors, see Support for More than 64 Processors in NDIS 6.20.

Using NDIS 6.20 Data Structures

Article • 03/14/2023

NDIS can support multiple versions of the same data structure. NDIS 6.20 and later drivers that use updated structures must report the correct **Revision** member and **Size** member value in the NDIS_OBJECT_HEADER structure that is in the **Header** member of a structure, if any, when the drivers initialize NDIS 6.20 data structures.

Note To determine the correct version and size information see the reference pages for each structure that includes a **Header** member. The reference pages for structures that contain a **Header** member and that have been updated for NDIS 6.20 include new information for NDIS 6.20 and later drivers. If there is no update to the structure for NDIS 6.20, the information that is provided for NDIS 6.0 or NDIS 6.1 drivers also applies to NDIS 6.20 and later drivers.

Compiling an NDIS 6.20 Driver

Article • 03/14/2023

The Build utility for Windows 7 and later supports header versioning. Header versioning makes sure that NDIS 6.20 drivers use the appropriate NDIS 6.20 data structures at compile time. You must update the SOURCES file to indicate NDIS 6.20.

For each type of driver, add information to the SOURCES file as follows:

- For a miniport driver, add NDIS620_MINIPORT=1.
- For a protocol driver, add NDIS620=1.
- For a filter driver, add NDIS620=1.

Introduction to NDIS 6.1

Article • 03/14/2023

This section introduces Network Driver Interface Specification (NDIS) 6.1 and describes its major design additions. NDIS 6.1 is included in the Windows Server 2008 and Windows Vista with Service Pack 1 (SP1) operating systems.

You should be familiar with NDIS 6.0 before learning about NDIS 6.1. For more information about NDIS 6.0, see Introduction to NDIS 6.0.

This section includes the following topics:

- Header-Data Split in NDIS 6.1
- Direct OID Request Interface in NDIS 6.1
- IPsec Task Offload Version 2 in NDIS 6.1
- NETDMA Updates in NDIS 6.1
- Implementing an NDIS 6.1 Driver
- Using NDIS 6.1 Data Structures
- Compiling an NDIS 6.1 Driver

Related topics

Introduction to Network Drivers

Introduction to NDIS 6.20

Introduction to NDIS 6.30

Header-Data Split in NDIS 6.1

Article • 03/14/2023

Header-data split services improve network performance by splitting the header and data in received Ethernet frames into separate buffers. By separating the headers and the data, these services enable the headers to be collected together into smaller regions of memory. Therefore, more headers fit into a single memory page and more headers fit into the system caches, so the overhead for memory accesses in the driver stack is reduced.

The header-data split interface is an optional service that is provided for header-data-split-capable network interface cards (NICs).

For more information about header-data split, see Header-Data Split.

Direct OID Request Interface in NDIS 6.1

Article • 03/14/2023

NDIS provides a direct OID request interface for NDIS 6.1 and later drivers. The *direct OID request path* supports OID requests that are queried or set frequently. For example, the IPsec offload version 2 (IPsecOV2) interface provides the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID for direct OID requests.

The direct OID request interface is optional for NDIS drivers. To support the direct OID path, drivers provide entry points and NDIS provides **Ndis*Xxx*** functions for protocol, filter, and miniport drivers.

Note NDIS supports specific OIDs for use with the direct OID request interface. To determine whether your driver can use an OID in the direct OIDs interface, see the notes in the OID reference page.

For NDIS 6.1, the only interface that uses the direct OID request interface is IPsecOV2. For more information about IPsecOV2, see IPsec Task Offload Version 2 in NDIS 6.1.

For NDIS 6.1 drivers in the Windows Server 2008 and Windows Vista with Service Pack 1 (SP1) operating systems, you can use only the following OIDs with the direct OID request interface:

- OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_UPDATE_SA

Miniport drivers and filter drivers must be able to handle direct OID requests that are not serialized. Unlike the standard OID request interface, NDIS does not serialize direct OID requests with other requests that are sent with the direct OID interface or with the standard OID request interface. Also, miniport drivers and filter drivers must be able to handle direct OID requests at IRQL <= DISPATCH_LEVEL.

For more information about how to implement the direct OID interface in drivers, see the following topics:

- Miniport Adapter OID Requests
- Protocol Driver OID Requests
- Filter Module OID Requests

IPsec Task Offload Version 2 in NDIS 6.1

Article • 03/14/2023

[The IPsec Task Offload feature is deprecated and should not be used.]

IPsec task offload provides offloading services for IPsec network data processing to IPsec offload-capable network interface cards (NICs). NDIS 6.1 and later support IPsec offload version 2 (IPsecOV2). IPsecOV2 extends support for additional crypto-algorithms, IPv6, and co-existence with large send offload (LSO).

IPsecOV2 uses the NDIS 6.1 direct OID request interface. For more information about the direct OID request interface, see Direct OID Request Interface in NDIS 6.1.

For more information about IPsecOV2, see IPsec Offload Version 2.

NetDMA Updates in NDIS 6.1

Article • 03/14/2023

> ⓘ **Important**
>
> The NetDMA interface is not supported in Windows 8 and later.

The NetDMA interface provides offloading of direct memory access (DMA) to network interface cards (NICs) that support a NetDMA DMA engine. The Windows Server 2008 and Windows Vista with Service Pack 1 (SP1) operating systems add NetDMA versions 1.1 and 2.0. NDIS 6.1 and later drivers can use NetDMA version 1.0, 1.1, and 2.0 interfaces. These interfaces manage interactions with the DMA engine and manage DMA transfers at run-time.

The NetDMA interface is an optional service that is provided for NICs and other drivers.

For more information about NetDMA, see NetDMA Drivers.

Implementing an NDIS 6.1 Driver

Article • 03/14/2023

An NDIS 6.1 driver must report the correct NDIS version when it registers with NDIS:

- You must update the major and minor NDIS version number in the NDIS_*Xxx*_DRIVER_CHARACTERISTICS structure to support NDIS 6.1. The **MajorNdisVersion** member must contain 0x06 and the **MinorNdisVersion** member must contain 0x01. This requirement applies to miniport, protocol, and filter drivers.

- Miniport drivers must set the **Header** member of NDIS_MINIPORT_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_MINIPORT_DRIVER_CHARACTERISTICS_REVISION_2.

- Filter drivers must set the **Header** member of NDIS_FILTER_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_FILTER_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_FILTER_DRIVER_CHARACTERISTICS_REVISION_2.

- Protocol drivers must set the **Header** member of NDIS_PROTOCOL_DRIVER_CHARACTERISTICS: Set **Revision** to NDIS_PROTOCOL_CHARACTERISTICS_REVISION_2 and **Size** to NDIS_SIZEOF_PROTOCOL _DRIVER_CHARACTERISTICS_REVISION_2.

Using NDIS 6.1 Data Structures

Article • 03/14/2023

NDIS can support multiple versions of the same data structure. NDIS 6.1 drivers that use updated structures in Windows Server 2008, Windows Vista with Service Pack 1 (SP1), or both operating systems must report the correct **Revision** member and **Size** member value in the **NDIS_OBJECT_HEADER** structure that is in the **Header** member of a structure, if any, when the drivers initialize NDIS 6.1 data structures.

Note To determine the correct version and size information, see the reference pages for each structure that includes a **Header** member. The reference pages for structures that contain a **Header** member and that have been updated for NDIS 6.1 include new information for NDIS 6.1 drivers. If there is no update to the structure for NDIS 6.1, the information that is provided for NDIS 6.0 drivers also applies to NDIS 6.1 drivers.

Compiling an NDIS 6.1 Driver

Article • 03/14/2023

The Build utility for Windows Server 2008 supports header versioning. Header versioning makes sure that NDIS 6.1 drivers use the appropriate NDIS 6.1 data structures at compile time. You must update the SOURCES file to indicate NDIS 6.1.

For each type of driver, add information to the SOURCES file as follows:

- For a miniport driver, add NDIS61_MINIPORT=1.
- For a protocol driver, add NDIS61=1.
- For a filter driver, add NDIS61=1.

Introduction to NDIS 6.0

Article • 03/14/2023

NDIS 6.0 is the next major version of the Network Driver Interface Specification (NDIS) library. NDIS specifies a standard interface between kernel-mode network drivers and the operating system. NDIS also specifies a standard interface between layered network drivers, thereby abstracting lower-level drivers that manage hardware from upper-level drivers, such as network transports. NDIS 6.0 is included in the Windows Vista operating system.

This section provides an introduction to NDIS 6.0 and describes the major design objectives, how these objectives have been met, and the advantages of NDIS 6.0.

This section includes the following topics:

NDIS 6.0 Design Objectives

Enhanced Performance and Scalability

Simplified Driver Model

For a description of the differences between NDIS 6.0 and earlier versions, and for detailed information about porting drivers to NDIS 6.0, see Porting NDIS 5.x Drivers to NDIS 6.0.

Related topics

Introduction to Network Drivers

Introduction to NDIS 6.1

Introduction to NDIS 6.20

Introduction to NDIS 6.30

NDIS 6.0 Design Objectives

Article • 03/14/2023

Two major objectives have guided the design and development of NDIS 6.0:

1. Enhancing driver performance and scalability. (See Enhanced Performance and Scalability for more information.)

   Major improvements in the following provide significant performance gains for both clients and servers:

   - Network data packaging
   - Send and receive paths
   - Run-time reconfiguration capabilities
   - Scatter/gather DMA
   - Filter drivers
   - Multiprocessor scaling of received data handling
   - Offloading TCP tasks to NICs

2. Simplifying the NDIS driver model. (See Simplified Driver Model for more information.)

   The following improvements simplify driver development:

   - Streamlined driver initialization
   - Versioning support for NDIS interfaces
   - Simplified reset handling
   - A standard interface for obtaining management information
   - A filter driver model to replace filter intermediate drivers

Enhanced Performance and Scalability

Article • 12/15/2021

NDIS 6.0 introduced the following features to improve performance and scalability:

- NET_BUFFER Data Packaging
- Improved Send and Receive Paths
- Enhanced Run-time Reconfiguration Abilities
- Receive Side Scaling Support
- New Scatter/Gather DMA Support
- Filter Drivers
- Full TCP Offload

NET_BUFFER Data Packaging

Article • 12/15/2021

Data packaging was redesigned in NDIS 6.0. The send and receive architecture that is based on the NDIS_PACKET structure has been replaced with an architecture that is based on NET_BUFFER and NET_BUFFER_LIST structures. A NET_BUFFER structure is the functional equivalent of an NDIS_PACKET structure. A NET_BUFFER structure specifies a buffer (MDL chain) for network data, as well as reserved space for NDIS, protocol drivers, and miniport drivers. NET_BUFFER structures can be linked together in a list that is described by a NET_BUFFER_LIST structure. A NET_BUFFER_LIST structure also provides storage for out-of-band (OOB) data that applies to all the NET_BUFFER structures in the list.

All components in the Microsoft next generation network driver stack, including the TCP/IP transport and Winsock, use NET_BUFFER data packaging. Uniform data packaging throughout the driver stack eliminates the need to repackage data, simplifies data handling, and reduces the number of function calls.

To accommodate older drivers that use NDIS_PACKET structures, NDIS 6.0 translates NDIS_PACKET structures to NET_BUFFER structures and vice versa. This translation is transparent to NDIS drivers.

NDIS propagates a driver's data backfill requirements to higher-level drivers. When allocating NET_BUFFER and NET_BUFFER_LIST structures for send data, a higher-level driver allocates enough data space to accommodate all lower-level drivers in the stack. As a result, lower-level drivers do not have to allocate additional buffer space to accommodate layer-specific headers. Instead, they can use the preallocated backfill space for this purpose.

For more information about the NET_BUFFER architecture, see NET_BUFFER Architecture.

Improved Send and Receive Paths

Article • 12/15/2021

The NDIS 6.0 send and receive paths have been improved as follows to enhance performance:

- All of the NDIS 6.0 and later driver send and receive functions can transfer a linked list of **NET_BUFFER_LIST** structures and their associated **NET_BUFFER** structures with a single function call. This support for true multipacket send and receive operations substantially reduces the number of function calls that drivers must make.

- When calling a send or receive function, a driver running at DISPATCH_LEVEL can indicate its IRQL to NDIS. When NDIS subsequently makes calls to other drivers in the stack, it is not necessary for these drivers to test the IRQL or set it to DISPATCH_LEVEL. This reduces the overhead that is associated with testing and setting the IRQL in critical code sections.

- When drivers pass packets up and down the driver stack, they can request NDIS to adjust the NET_BUFFER data offsets to accommodate header information. When sending a packet, a driver can expand the used data space to accommodate the driver's header information. When indicating a receive packet, a driver can decrease the used data space after the driver is done accessing its header information. This ability to dynamically adjust the used data space in a NET_BUFFER structure, without allocating and freeing memory or copying data, reduces the overhead that is required to process network data.

For more information about send and receive data handling in NDIS 6.0, see NET_BUFFER Architecture.

Enhanced Run-time Reconfiguration Abilities

Article • 12/15/2021

NDIS 6.0 introduced the ability to pause and restart a driver stack without having to tear down the stack and build a new one. All NDIS 6.0 and later drivers must support pause and restart services.

Pausing the stack eliminates synchronization problems by putting all drivers in a known state before reconfiguration occurs. The ability to pause also gives NDIS the opportunity to query driver characteristics, and reconfigure other characteristics of the stack.

NDIS can pause a driver stack, for example, to temporarily stop data flow before performing a Plug and Play operation, such as adding or removing a filter driver, or binding or unbinding a protocol driver. NDIS restarts the stack after the reconfiguration takes place.

Miniport and filter drivers handle pause and restart services through function interfaces. Protocol drivers handle pause and restart services through Plug and Play event notifications.

For more information about pause and restart operations, see Driver Stack Management.

Receive Side Scaling Support

Article • 12/15/2021

NDIS 6.0 introduced support for the scaling of receive-packet processing across multiple processors. The receive side scaling (RSS) interface accommodates several levels of NIC hardware support.

Based upon its current RSS configuration, a miniport driver or a NIC determines the target processor to associate with the received data. The RSS configuration can be adjusted to make the most efficient use of available target processors.

The miniport driver or NIC assigns the received data to a receive queue that is associated with a target processor. The miniport driver requests NDIS to schedule deferred procedure calls (DPCs) for target processors with non-empty receive queues.

NDIS schedules a DPC on each of the specified target processors. Each DPC processes a particular receive queue on the specified target processor.

For more information about NDIS 6.0 receive side scaling, see Receive Side Scaling.

New Scatter/Gather DMA Support

Article • 12/15/2021

Unlike previous versions of NDIS, NDIS 6.0 passes a send packet to a miniport driver before the packet is mapped for a DMA transfer. After it has obtained the packet, the miniport driver can request NDIS to supply a scatter/gather list for the packet.

This provides the following benefits:

- Because a miniport driver has access to the packet before it is mapped, any changes that the miniport driver makes to the packet are reflected in the associated scatter/gather list data.
- A miniport driver can optimize the transmission of small or highly fragmented packets by copying them to a preallocated buffer, thereby eliminating the need for mapping. This eliminates unnecessary processing.
- NDIS can safely pass multiple **NET_BUFFER** structures to the miniport driver in one function call. This results in fewer calls to the miniport driver and thus improves system performance.
- Because a miniport driver can preallocate memory for a scatter/gather list, NDIS does not have to allocate memory for the scatter/gather list at run time.

For more information about NDIS 6.0 scatter/gather DMA, see NDIS 6.0 Scatter/Gather DMA.

Faster filter drivers

Article • 12/15/2021

The NDIS 6.x filter driver model supersedes the NDIS 5.x filter intermediate driver model. The new filter driver model enhances system performance in several ways:

- Unlike an NDIS filter intermediate driver, an NDIS 6.0 or later filter driver is not implemented as a combination miniport driver and protocol driver. It has a unique interface that is similar to miniport and protocol drivers but is optimized for filtering information that is passed on a driver stack.
- NDIS 6.0 and later filter drivers support bypass capabilities so that the driver does not process data when such processing is not required.
- NDIS 6.0 and later filter drivers can be dynamically inserted into or removed from a driver stack during run time without tearing down bindings. Before such a dynamic operation occurs, NDIS 6.0 pauses all the NDIS drivers in the stack. NDIS restarts the stack when the reconfiguration is complete.

For more information about NDIS 6.0 filter drivers, see NDIS 6.0 Filter Drivers.

Full TCP Offload

Article • 12/15/2021

NDIS 6.0 introduced an architecture for full TCP offload. This architecture is called a "chimney offload" architecture because it provides a direct connection, called a "chimney," between applications and an offload-capable NIC. The chimney enables the NIC to perform TCP processing for offloaded connections, including maintaining the protocol state.

The chimney offload architecture reduces host network processing for network-intensive applications. This allows networked applications to scale more efficiently while also reducing end-to-end latency. Fewer servers are needed to host an application, and servers are able to use the full Ethernet bandwidth.

The TCP chimney offloads all TCP processing for one or more TCP connections. The primary performance gains are obtained from offloading segmentation and reassembly (SAR), offloading processing that ensures reliable connections (for example, ACK processing and TCP retransmission timers), and reducing interrupt loading.

Note The Windows Vista operating system continues to support the individual TCP task offloads available in earlier versions of the operating system. These tasks can be offloaded on connections that have not been offloaded through a chimney. An offload-capable NIC should support both chimney offloads and task offloads. Such a NIC provides the highest degree of offload optimization.

For information on TCP chimney offload in NDIS 6.0 and later, see NDIS TCP Chimney Offload.

Simplified Driver Model

Article • 12/15/2021

The following NDIS 6.0 features simplify driver development:

Easier Initialization

Versioned Interfaces

Simplified Reset Handling

NDIS Interface Information

Easier-to-Write Filter Drivers

Easier Initialization

Article • 12/15/2021

All NDIS 6.0 and later drivers have updated driver registration interfaces. These NDIS interfaces provide simplified driver registration and the ability to register optional services separately from required services.

Miniport drivers require fewer function calls to register. In general, NDIS 6.0 and later function interfaces are more consistent when compared to the NDIS 5.*x* and earlier interfaces. Resources that are allocated also have a reciprocal function to free them.

An NDIS 6.0 or later intermediate driver can register as a miniport-intermediate driver. Such a driver has both a virtual miniport for a virtual device and a miniport adapter for a physical device. Registering as a miniport-intermediate driver simplifies the creation of an intermediate driver that binds only to a vendor's own NIC. The driver can pass network data, OID requests, and status indications between its virtual miniport and physical miniport adapter with internal calls.

Protocol drivers receive most of the information about an underlying adapter in a binding request. Therefore, protocol drivers do not send OID requests for the parameters that NDIS already provided in the bind request.

To initialize a miniport adapter, miniport drivers can receive OID requests that combine the information from many separate OID requests into fewer requests containing the combined information.

Intermediate drivers have fewer specialized functions and make better use of miniport driver and protocol driver interfaces.

A miniport driver can read or write the registry at any time -- not just during initialization. For example, when an application requests through Windows Management Instrumentation (WMI) that a driver change one of its operating parameters, the driver can record this change in the registry so that the change persists across reboots.

NDIS provides a bus-independent function call for reading and writing bus-specific configuration parameters. A driver can call this function regardless of the bus type in the system. This ensures that NDIS will be able to support future buses without the addition of new bus-specific functions.

For more information about driver initialization, see the initialization topics in the following sections:

Writing NDIS Miniport Drivers

Writing NDIS Protocol Drivers

NDIS Filter Drivers

Writing NDIS Intermediate Drivers

Versioned Interfaces

Article • 12/15/2021

NDIS 6.0 supports versioning for key structures. Also, many former function parameters are moved to structures. Moving the function parameters to versioned structures allows the function parameters to be changed in later NDIS versions without changing the function interface.

Versioned structures contain a header that specifies the version of the structure. If the set of function parameters or other structure members are changed, the structure and its version are updated.

This versioning simplifies backward compatibility and extends the life of NDIS 6.0 and later drivers. Also, NDIS drivers can support more than one version of NDIS.

For more information, see **NDIS_OBJECT_HEADER**.

Simplified Reset Handling

Article • 12/15/2021

NDIS 6.0 and later drivers do not reset miniport adapters to cancel send or OID requests. Instead, NDIS provides send cancellation and OID request cancellation functions.

An NDIS miniport driver can complete or cancel a pending send operation or pending OID request at any time -- either before or after completing a reset. The miniport driver does not have to keep track of when it received a request with respect to a reset. Also, the driver does not have to synchronize a canceled request with the completion of a reset.

For more information, see Canceling a Send Operation, OID Requests for an Adapter, Protocol Driver OID Requests, and Filter Module OID Requests.

NDIS Interface Information

Article • 03/14/2023

A standardized interface for querying NDIS management information bases (MIBs) makes it easier for overlying drivers and user-mode applications to query information about network interfaces. A MIB client calls NDIS-supplied functions to request information from an underlying NDIS interface provider. This causes NDIS to issue OID requests to retrieve the information. To supply the information to the client, NDIS calls a callback function that the client registered with NDIS.

For more information about NDIS network interface services, see NDIS Network Interfaces.

NDIS provides enhanced support for Management Instrumentation (WMI). For more information about NDIS 6.0 support for WMI, see NDIS Support for WMI.

Related topics

NDIS_INTERFACE_INFORMATION

Easier-to-Write Filter Drivers

Article • 12/15/2021

NDIS 6.0 filter drivers are easier to write than the previous NDIS filter intermediate drivers.

When compared to filter intermediate drivers, filter drivers provide the following implementation advantages:

- Filter drivers do not include a complete miniport driver interface and a complete protocol driver interfaces.
- Filter drivers do not create and manage a virtual device. There is no virtual miniport in a filter driver.
- If a filter driver filters specific services, the driver can bypass other services. The driver does not require code for services that are bypassed. For example, if a filter driver filters OID requests but does not filter send and receive operations, the filter driver does not require send and receive entry points.

For more information about NDIS 6.0 filter drivers, see NDIS 6.0 Filter Drivers.

Specifying NDIS Version Information

Article • 03/14/2023

This section provides an overview of the support that NDIS and NDIS drivers provide for NDIS version information.

Many NDIS structures include structure version information in a **Header** member. NDIS or NDIS drivers set the version information in such structures. NDIS drivers should check the version information in structures before they access the structure members.

Also, NDIS drivers specify the NDIS version that they support during driver initialization.

This section includes the following topics:

- Overview of NDIS Support for Header Versions
- Version Information Requirements for NDIS Drivers
- Version Information Requirements for NDIS
- Obtaining the NDIS Version
- NDIS Object Version Issues for WMI

Related topics

Overview of NDIS versions

Overview of NDIS Support for Header Versions

Article • 03/14/2023

Many NDIS structures include structure version information. NDIS or NDIS drivers initialize the **Header** member in such structures as required for each structure. NDIS drivers should check the version information, if any, in each structure before they access the structure members.

The **Header** member is an NDIS_OBJECT_HEADER structure. This structure contains the revision number, type, and size of the structure that includes the **Header** member.

Structures that include the **Header** member meet the following requirements:

- The structure will have a new revision value if new information is added to the member list for a new NDIS version. For example, if the NDIS 6.1 version of the structure has new members at the end of the member list, in a union, or in a bitmask, it will have a different revision value from the NDIS 6.0 version.

- After a structure is changed, the size of the later revision of the structure can be equal to or larger than the size of the earlier revision of the structure, but it will not be smaller. If the new size is larger than the size of the earlier revision, the new members are added at the end of the member list.

- A structure will only have a new revision if the earlier revision information is still valid and complete. That is, the new version of the structure contains a superset of the older versions members. **Note** If any of the preceding conditions cannot be met, NDIS provides a new structure with a new name that replaces the existing structure instead of providing a revised version of the existing structure.

- NDIS drivers should always use the predefined revision values. NDIS provides such definitions in the form Xxx_REVISION_Nn, and NDIS_SIZEOF_Xxx_REVISION_Nn, for the **Revision** and **Size** members of NDIS_OBJECT_HEADER respectively. Also, Xxx represents the name of the structure and Nn is the revision number. For example, the revision and size for the first revision of the NDIS_FILTER_PARTIAL_CHARACTERISTICS structure are NDIS_FILTER_PARTIAL_CHARACTERISTICS_REVISION_1 and NDIS_SIZEOF_FILTER_PARTIAL_CHARACTERISTICS_REVISION_1 respectively.

- The **Header.Size** value must be consistent with the **Header.Revision** value. That is, if the **Revision** member contains Xxx_REVISION_1, the **Size** member value must be equal to or greater than NDIS_SIZEOF_Xxx_REVISION_1.

Related topics

Overview of NDIS versions

Specifying NDIS Version Information

Version Information Requirements for NDIS Drivers

Article • 03/14/2023

NDIS structures that provide version information have a **Header** member that is defined as an NDIS_OBJECT_HEADER structure and NDIS drivers must provide support for such version information.

NDIS can support drivers that support a higher or lower NDIS version than the *current version of NDIS* (that is, the version of NDIS that is supported on the version of the operating system that a computer is running). Also the *registered NDIS version* (that is, the version that the driver reported during initialization) of the driver can be lower than the highest version that the driver supports. For example, an NDIS 5.1 driver or an NDIS 6.1 driver can run on a version of the operating system that is running NDIS 6.0. The NDIS 5.1 driver simply registers as an NDIS 5.1 driver during initialization. However, the NDIS 6.1 driver must check the current version of NDIS and must register as a driver that supports the highest level of NDIS that is available (in this example, NDIS 6.0). For more information about how to obtain the current NDIS version, see Obtaining the NDIS Version.

Note A driver is not required to support all the features in a later revision of a structure. For example, a miniport driver can create a version 2 structure and supply values that are appropriate for a version 1 structure.

To access the members in structures that have version information, NDIS drivers must complete the following process:

- Check the **Header.Revision** and **Header.Size** members before accessing any members in the structure.

- For earlier version structures (that is, structures that have a lower revision number than the number that is associated with the NDIS version that the driver supports):
  - The driver must verify that the **Header.Size** value is correct for the **Header.Revision** value. For example, the value of NDIS_SIZEOF_Xxx_REVISION_1 is correct for Xxx_REVISION_1 but it is too small for Xxx_REVISION_2.
  - The **Header.Size** value must be equal to or greater than NDIS_SIZEOF_Xxx_REVISION_Nn (where *Nn* is the revision number of the structure that the driver is using) and the driver must correctly handle the information in the structure as is appropriate for that revision.

- For later version structures (that is, structures that have a higher revision number than the number that is associated with the NDIS version that the driver supports), the driver can use the structure as if it were an older revision of the structure. The higher version structure is always compatible with the older version.

- Drivers must use the correct revision of a structure for the registered NDIS version of the driver. For example, an NDIS 6.1 driver must report its offload capabilities in NDIS_OFFLOAD structures by setting the members in the NDIS_OBJECT_HEADER structure to indicate NDIS_OFFLOAD_REVISION_2. However, the driver does not have to support all the features that are included with NDIS_OFFLOAD_REVISION_2.

- A driver that successfully handles an OID set request must set the **SupportedRevision** member in the NDIS_OID_REQUEST structure upon return from the OID set request. The **SupportedRevision** member notifies the initiator of the request of the revision that the driver supported. For example, a miniport driver can create an Xxx_REVISION_2 structure, supply values that are appropriate for an Xxx_REVISION_1 structure, and fill the rest of the structure with zeros. The miniport driver would report Xxx_REVISION_1 in the **SupportedRevision** member. In this case, a protocol driver that can support an Xxx_REVISION_2 will use Xxx_REVISION_1 information that the miniport driver supported.

- To determine what information was successfully handled by an underlying driver, overlying drivers that issue OID requests must check the value in the **SupportedRevision** member in the NDIS_OID_REQUEST structure after the OID request returns.

Related topics

Overview of NDIS versions

Specifying NDIS Version Information

Version Information Requirements for NDIS

Article • 03/14/2023

NDIS supports various header version information requirements that guarantee consistent behavior between NDIS versions. To support header version information, NDIS has the following responsibilities:

- Handles structures with lower revisions. That is, NDIS checks the header information and interprets the structure based upon the revision information in the header.
- Fails a function call and returns an appropriate error code if a driver uses an incorrect structure revision. For example, NDIS fails the function call if an NDIS 6.30 driver uses Xxx_REVISION_1 structures when there is an NDIS 6.30 Xxx_REVISION_2 structure.

Related topics

Overview of NDIS versions

Specifying NDIS Version Information

Obtaining the NDIS Version

Article • 03/14/2023

NDIS versions might not be the same as the operating system versions. For example, if you use the **RtlGetVersion** and **RtlVerifyVersionInfo** routines to get the operating system version, you do not get a guaranteed association with a particular NDIS version. Therefore, NDIS drivers must get the NDIS version and the operating system version separately. NDIS drivers can get the NDIS version with the **NdisGetVersion** function.

Related topics

Overview of NDIS versions

Specifying NDIS Version Information

NDIS Object Version Issues for WMI

Article • 03/14/2023

This topic describes the NDIS object version issues that affect Windows Management Instrumentation (WMI) support.

There is no versioning inside a WMI managed object format (MOF) file. Therefore, if the corresponding NDIS structure has a new revision, more fields have been added to the MOF data objects.

The NDIS_WMI_Xxx_HEADER structures have a new revision when more members are added for a new NDIS version. For a list of the current NDIS_WMI_Xxx_HEADER structures, see NDIS WMI Structures.

When applications access the WMI information for a query operation, they must check the version in the returned buffer before they access any data. For a set operation, applications must check the **SupportedRevision** member in the NDIS_WMI_OUTPUT_INFO structure to determine which version the underlying driver has accepted.

Many WMI objects contain the **MSNdis_ObjectHeader** property, which is equivalent to the **NDIS_OBJECT_HEADER** structure. When populating the **MSNdis_ObjectHeader** property, set the **Type** and **Revision** fields as documented in the **NDIS_OBJECT_HEADER** topic. To ensure seamless portability to 64-bit systems, set the **Size** field to `0xFFFF`.

Related topics

Overview of NDIS versions

Specifying NDIS Version Information

Porting NDIS 6.x drivers to NDIS 6.89

Article • 05/22/2024

NDIS 6.89 is substantially the same as NDIS 6.88. For detailed information about new features for NDIS 6.89, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.89.

If you are porting an NDIS 6.x driver to NDIS 6.89, you should be familiar with the changes to each version between your driver's version and 6.89. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.88
- Introduction to NDIS 6.87
- Introduction to NDIS 6.86
- Introduction to NDIS 6.85
- Introduction to NDIS 6.84
- Introduction to NDIS 6.83
- Introduction to NDIS 6.82
- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Porting NDIS 6.x drivers to NDIS 6.88

Article • 05/22/2024

NDIS 6.88 is substantially the same as NDIS 6.87. For detailed information about new features for NDIS 6.88, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.88.

If you are porting an NDIS 6.x driver to NDIS 6.88, you should be familiar with the changes to each version between your driver's version and 6.88. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.87
- Introduction to NDIS 6.86
- Introduction to NDIS 6.85
- Introduction to NDIS 6.84
- Introduction to NDIS 6.83
- Introduction to NDIS 6.82
- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.87

Article • 05/22/2024

NDIS 6.87 is substantially the same as NDIS 6.86. For detailed information about new features for NDIS 6.87, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.87.

If you are porting an NDIS 6.x driver to NDIS 6.87, you should be familiar with the changes to each version between your driver's version and 6.87. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.86
- Introduction to NDIS 6.85
- Introduction to NDIS 6.84
- Introduction to NDIS 6.83
- Introduction to NDIS 6.82
- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.86

Article • 03/14/2023

NDIS 6.86 is substantially the same as NDIS 6.85. For detailed information about new features for NDIS 6.86, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.86.

If you are porting an NDIS 6.x driver to NDIS 6.86, you should be familiar with the changes to each version between your driver's version and 6.86. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.85
- Introduction to NDIS 6.84
- Introduction to NDIS 6.83
- Introduction to NDIS 6.82
- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.85

Article • 03/14/2023

NDIS 6.85 is substantially the same as NDIS 6.84. For detailed information about new features for NDIS 6.85, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.85.

If you are porting an NDIS 6.x driver to NDIS 6.85, you should be familiar with the changes to each version between your driver's version and 6.85. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.84
- Introduction to NDIS 6.83
- Introduction to NDIS 6.82
- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.84

Article • 03/14/2023

NDIS 6.84 is substantially the same as NDIS 6.83. For detailed information about new features for NDIS 6.84, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.84.

If you are porting an NDIS 6.x driver to NDIS 6.84, you should be familiar with the changes to each version between your driver's version and 6.84. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.83
- Introduction to NDIS 6.82
- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.83

Article • 03/14/2023

NDIS 6.83 is substantially the same as NDIS 6.82. For detailed information about new features for NDIS 6.83, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.83.

If you are porting an NDIS 6.x driver to NDIS 6.83, you should be familiar with the changes to each version between your driver's version and 6.83. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.82
- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.82

Article • 03/14/2023

NDIS 6.82 is substantially the same as NDIS 6.81. For detailed information about new features for NDIS 6.82, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.82.

If you are porting an NDIS 6.x driver to NDIS 6.82, you should be familiar with the changes to each version between your driver's version and 6.82. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.81
- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.81

Article • 03/14/2023

NDIS 6.81 is substantially the same as NDIS 6.80. For detailed information about new features for NDIS 6.81, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.81.

If you are porting an NDIS 6.x driver to NDIS 6.81, you should be familiar with the changes to each version between your driver's version and 6.81. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.80
- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.80

Article • 03/14/2023

For NDIS miniport, protocol, filter, and intermediate drivers, NDIS 6.80 is substantially the same as NDIS 6.70. For NIC drivers, the NetAdapter class extension (NetAdapterCx) has been updated to from version 1.0 in NDIS 6.70 to version 1.1 in NDIS 6.80. To port an NDIS 6.x miniport driver to NetAdapterCx, see Porting NDIS miniport drivers to NetAdapterCx.

For detailed information about new features for NDIS 6.80, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.80.

If you are porting an NDIS 6.x miniport, protocol, filter, or intermediate driver to NDIS 6.80, you should be familiar with the changes to each version between your driver's version and 6.80. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.70
- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.70

Article • 03/14/2023

For NDIS miniport, protocol, filter, and intermediate drivers, NDIS 6.70 is substantially the same as NDIS 6.60. Starting in NDIS 6.70, however, NDIS driver developers can also write a NIC driver using the new Network Adapter WDF Class Extension, a.k.a. NetAdapterCx. To port an NDIS 6.x miniport driver to NetAdapterCx, see Porting NDIS miniport drivers to NetAdapterCx.

For detailed information about new features for NDIS 6.70, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.70.

If you are porting an NDIS 6.x miniport, protocol, filter, or intermediate driver to NDIS 6.70, you should be familiar with the changes to each version between your driver's version and 6.70. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.60
- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.60

Article • 03/14/2023

NDIS 6.60 is substantially the same as NDIS 6.50. For detailed information about new features for NDIS 6.60, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.60.

If you are porting an NDIS 6.x driver to NDIS 6.60, you should be familiar with the changes to each version between your driver's version and 6.60. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.50
- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x drivers to NDIS 6.50

Article • 03/14/2023

NDIS 6.50 is substantially the same as NDIS 6.40. For detailed information about new features for NDIS 6.50, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.50.

If you are porting an NDIS 6.x driver to NDIS 6.50, you should be familiar with the changes to each version between your driver's version and 6.50. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.40
- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x Drivers to NDIS 6.40

Article • 03/14/2023

NDIS 6.40 is substantially the same as NDIS 6.30. For detailed information about new features for NDIS 6.40, including implementation and compilation details specific to this version of NDIS, see Introduction to NDIS 6.40.

If you are porting an NDIS 6.x driver to NDIS 6.40, you should be familiar with the changes to each version between your driver's version and 6.40. For more information about previous NDIS 6.x versions, see the following topics:

- Introduction to NDIS 6.30
- Introduction to NDIS 6.20
- Introduction to NDIS 6.1
- Introduction to NDIS 6.0

Porting NDIS 6.x Drivers to NDIS 6.30

Article • 03/14/2023

This section describes the requirements to port NDIS 6.x drivers to NDIS 6.30.

- For information about porting NDIS 6.x drivers to NDIS 6.20, see Porting NDIS 6.x Drivers to NDIS 6.20.
- For information about porting NDIS 5.x and earlier drivers to NDIS 6.x, see Porting NDIS 5.x Drivers to NDIS 6.0.

For more information about NDIS 6.30 features, see Introduction to NDIS 6.30.

The following topics discuss how to port miniport, protocol, and intermediate drivers to NDIS 6.30:

- NDIS 6.30 Backward Compatibility
- Summary of Changes Required to Port a Miniport Driver to NDIS 6.30
- Summary of Changes Required to Port a Protocol or Filter Driver to NDIS 6.30
- Summary of Changes Required to Port an Intermediate Driver to NDIS 6.30

NDIS 6.30 Backward Compatibility

Article • 03/14/2023

NDIS 6.30 adds backward compatibility features to those that apply to NDIS 6.20 and NDIS 6.0 drivers. For information about NDIS 6.20 compatibility issues, see NDIS 6.20 Backward Compatibility. For information about NDIS 6.0 compatibility issues, see NDIS 6.0 Backward Compatibility.

For more information about NDIS 6.30 features, see Introduction to NDIS 6.30.

Features that are no longer supported

The following features are not supported in Windows 8 and later:

- TCP chimney offload is no longer supported for virtual machines. However, it is still supported for native use.
- IPsec task offload version 1. All drivers that support IPsec task offload should be updated to support IPsec task offload version 2.
- Filter intermediate drivers. Instead, use the NDIS 6.*x* filter driver interface. For more information about filter drivers, see NDIS Filter Drivers.
- 802.11 drivers that emulate 802.3. NDIS 802.11 drivers must support the native 802.11 interface. For more information about native 802.11, see Native 802.11 Wireless LAN.
- NDIS WAN drivers. NDIS WAN drivers must be ported to the NDIS 6.0 CoNDIS WAN driver model. For more information about CoNDIS WAN, see WAN Miniport Drivers.

Features that have been removed

The following features have been removed from Windows 8 and later:

- NDIS 5.x and earlier
- IrDA miniport drivers
- NetDMA
- Token Ring (802.5)

Other changes

- The TCP/IP protocol driver that ships with Windows 8 has been updated to NDIS 6.30. However, this change was relatively minor, so it's not worth porting your driver just for this feature. The TCP/IP protocol driver is backward compatible with NDIS 6.20 and earlier drivers in the driver stack.

Summary of Changes Required to Port a Miniport Driver to NDIS 6.30

Article • 03/14/2023

To update an NDIS 6.x miniport driver to support NDIS 6.30, you must modify it as outlined in the following sections.

- Build Environment and Testing
- General Porting Requirements
- Wi-Fi Direct Miniport Drivers
- USB-Based WWAN (Mobile Broadband) Miniport Drivers

For more information about NDIS 6.30 features, see Introduction to NDIS 6.30.

Build Environment and Testing

- Replace the preprocessor definition NDIS60_MINIPORT or NDIS61_MINIPORT or NDIS620_MINIPORT with NDIS630_MINIPORT. For more information, see Compiling an NDIS 6.30 Driver

- Replace the preprocessor definition NDIS60 or NDIS61 or NDIS620, if present, with NDIS630. **Note** This item applies only to NDIS intermediate, protocol, and filter drivers. Most NDIS miniport drivers don't need this preprocessor definition.

- In NDIS 6.30, NDIS can call *MiniportInitializeEx* twice in parallel if there are two adapters plugged into the system at the same time or during system startup. Be sure to test your miniport driver under this "parallel startup" condition.

General Porting Requirements

- Update the major and minor NDIS version number in the NDIS_*Xxx*_DRIVER_CHARACTERISTICS structure as described in Implementing an NDIS 6.30 Driver.
- For all structures that were updated for NDIS 6.30, miniport drivers need to update the **Header** member of the structure with the correct **Revision** and **Size** values. For more information, see Using NDIS 6.30 Data Structures.
- All miniport drivers should implement the no-pause-on-suspend feature. For more information, see:
  - Power Management Enhancements in NDIS 6.30
  - **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES**

    - **NET_PNP_EVENT**
    - OID_PNP_SET_POWER

Wi-Fi Direct Miniport Drivers

During *MiniportInitializeEx*, a Wi-Fi Direct-capable miniport driver must initialize the default 802.11 MAC entity. It must also report its Wi-Fi Direct and Virtual Wi-Fi capabilities using the **NdisMSetMiniportAttributes** function.

Note The driver is not required to register with NDIS the NDIS port corresponding to the default MAC entity.

USB-Based WWAN (Mobile Broadband) Miniport Drivers

For USB-based Mobile Broadband devices, Windows 8 provides a class driver that works with devices conforming to the MBIM specification. This model is referred to as the Mobile Broadband (MB) Class Driver. However, a class driver cannot support all of the functionality exposed by an MB device. For this reason, the MB feature provides a well-defined mechanism that you can use to extend the class driver functionality. For more information, see MB Device Services.

If your USB-based WWAN miniport driver cannot implement the MB class driver, it must at least implement the NDIS Selective Suspend feature.

Summary of Changes Required to Port a Protocol or Filter Driver to NDIS 6.30

Article • 03/14/2023

To update an NDIS 6.x protocol or filter driver to support NDIS 6.30, you must modify it as outlined in the following sections.

- Build Environment
- General Porting Requirements for Protocol Drivers
- General Porting Requirements for Filter Drivers

Build Environment

- Replace the preprocessor definition NDIS60 or NDIS61 or NDIS620, if present, with NDIS630.
- Update the major and minor NDIS version number in the NDIS_*Xxx*_DRIVER_CHARACTERISTICS structure as described in Implementing an NDIS 6.30 Driver.

General Porting Requirements for Protocol Drivers

A protocol driver should always complete **NetEventSetPower** without waiting for packets. For more information, see:

- Power Management Enhancements in NDIS 6.30
- Handling PnP Events and Power Management Events in a Protocol Driver

For more information about NDIS 6.30 features, see Introduction to NDIS 6.30.

General Porting Requirements for Filter Drivers

A filter driver should always complete **NetEventSetPower** without waiting for packets. For more information, see:

- Power Management Enhancements in NDIS 6.30
- **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES**
- **NET_PNP_EVENT**
- OID_PNP_SET_POWER

For more information about NDIS 6.30 features, see Introduction to NDIS 6.30.

Summary of Changes Required to Port an Intermediate Driver to NDIS 6.30

Article • 03/14/2023

To update an NDIS 6.x intermediate (IM) driver to support NDIS 6.30, you must modify it as outlined in the following sections.

Build Environment

- Replace the preprocessor definition NDIS60 or NDIS61 or NDIS620, if present, with NDIS630.
- Update the major and minor NDIS version number in the NDIS_*Xxx*_DRIVER_CHARACTERISTICS structure as described in Implementing an NDIS 6.30 Driver.

General Porting Requirements

- Except where noted otherwise, protocol driver and miniport driver changes also apply to intermediate drivers. For more information about porting these drivers, see the Summary of Changes Required to Port a Protocol or Filter Driver to NDIS 6.30 and Summary of Changes Required to Port a Miniport Driver to NDIS 6.30.

Porting NDIS 6.x Drivers to NDIS 6.20

Article • 03/14/2023

This section describes the requirements to port NDIS 6.0 and 6.1 drivers to NDIS 6.20.

For information about porting NDIS 5.x or earlier drivers to NDIS 6.x, see Porting NDIS 5.x Drivers to NDIS 6.0.

For more information about NDIS 6.20 features, see Introduction to NDIS 6.20.

For more information about NDIS 6.1 features, see Introduction to NDIS 6.1.

The following topics describe how to port miniport, protocol, and intermediate drivers to NDIS 6.20 in more detail:

- NDIS 6.20 Backward Compatibility
- NDIS 6.20 Updates to NDIS 6.1 Features
- Obsolete Interfaces in NDIS 6.20
- Summary of Changes Required to Port a Miniport Driver to NDIS 6.20
- Summary of Changes Required to Port a Protocol Driver to NDIS 6.20
- Summary of Changes Required to Port a Filter Driver to NDIS 6.20
- Summary of Changes Required to Port an Intermediate Driver to NDIS 6.20

NDIS 6.20 Backward Compatibility

Article • 03/14/2023

NDIS 6.20 adds backward compatibility features to those that apply to NDIS 6.0 drivers. For information about NDIS 6.0 compatibility issues, see NDIS 6.0 Backward Compatibility. In addition to the translation features that NDIS 6.0 provides for NDIS 5.x and earlier drivers, NDIS 6.20 also provides translation for the power management interface. NDIS 6.20 drivers must support the NDIS 6.20 power management interface.

NDIS 6.20 supports updated versions of the features that were added for NDIS 6.1. For more information about updates to NDIS 6.1 features, see NDIS 6.20 Support for NDIS 6.1 Features.

Note The NDIS 6.20 interface supports more than 64 processors. Previous NDIS versions are limited to no more than 64 processors (32 in x86 versions of the operating system).

To remain backward compatible with older NDIS versions, drivers that have not been updated to support more than 64 processors default to processor group zero. For more information about processor groups, see the Kernel-Mode Driver Architecture Design documentation for processor groups.

Some NDIS 6.1 and earlier functions are obsolete and cannot be used with NDIS 6.20 drivers. See the requirements section of the reference page for a particular function to determine its NDIS version compatibility. For a list of obsolete interfaces, see Obsolete Interfaces in NDIS 6.20.

The TCP/IP protocol driver that ships with Windows 7 has been updated to NDIS 6.20. The TCP/IP protocol driver is backward compatible with NDIS 6.1 and earlier drivers in the driver stack. However, to obtain the best performance on the Windows 7 driver stack, all drivers should be updated to NDIS 6.20.

NDIS 5.x and earlier NDIS drivers are deprecated in Microsoft Windows versions after Windows 7. No new NDIS 5.x drivers will be WHQL certified. All new drivers should be NDIS 6.0 or later drivers.

IrDA miniport drivers will not be supported in Microsoft Windows versions after Windows 7.

IPsec task offload version 1 will not be supported in Microsoft Windows versions after Windows 7. All drivers that support IPsec task offload should be updated to support IPsec task offload version 2.

Filter intermediate drivers will not be supported in Microsoft Windows versions after Windows 7. You should use the NDIS 6.0 filter drivers interface. For more information about filter drivers, see NDIS Filter Drivers.

802.11 drivers that emulate 802.3 will not be supported in Microsoft Windows versions after Windows 7. NDIS 802.11 drivers must support the native 802.11 interface. For more information about native 802.11, see Native 802.11 Wireless LAN.

NDIS WAN drivers will not be supported in Microsoft Windows versions after Windows 7. NDIS WAN drivers must be ported to the NDIS 6.0 CoNDIS WAN driver model. For more information about CoNDIS WAN, see WAN Miniport Drivers.

ATM and Token Ring drivers will not be supported in Microsoft Windows versions after Windows 7.

NDIS 6.20 Updates to NDIS 6.1 Features

Article • 03/14/2023

NDIS 6.1 added the following interfaces to NDIS 6.0:

Header-Data Split

Direct OID Requests

IPsec Task Offload Version 2

NetDMA 1.1 and 2.0

For more information about NDIS 6.1, see Introduction to NDIS 6.1.

NDIS 6.1 also includes updates to support MSI-X dynamic configuration for receive side scaling (RSS). For more information about NDIS 6.1 changes in RSS, see NDIS MSI-X. RSS is updated in NDIS 6.20 to provide support for more than 64 processors.

The direct OID request interface is optional for NDIS 6.1 drivers but it is mandatory for NDIS 6.20 miniport drivers.

After NDIS 6.20 IPsec task offload version 1 will not be supported. All drivers that support IPsec task offload should be updated to support IPsec task offload version 2.

NetDMA 1.1 and 2.0 were introduced with NDIS 6.1. NetDMA 2.1 is introduced with NDIS 6.20 to provide support for more than 64 processors.

Obsolete Interfaces in NDIS 6.20

Article • 03/14/2023

Some NDIS 6.1 application interface elements are obsolete for NDIS 6.20 drivers.

The following table lists NDIS 6.1 interface elements and their NDIS 6.20 replacements.

Obsolete interface	Use instead
LOCK_STATE	LOCK_STATE_EX
NDIS_CURRENT_PROCESSOR_NUMBER	NdisCurrentProcessorIndex
NDIS_MAX_PROCESSOR_COUNT(constant)	NdisGroupMaxProcessorCount
NDIS_RW_LOCK	NDIS_RW_LOCK_EX
MAXIMUM_PROCESSORS(constant)	NdisGroupMaxProcessorCount
NdisSystemProcessorCount	NdisGroupMaxProcessorCount
NdisSystemActiveProcessorCount	NdisGroupActiveProcessorCount
NdisGetProcessorInformation	NdisGetProcessorInformationEx
NdisMQueueDpc	NdisMQueueDpcEx
Do not use the *TargetProcessors* parameter of MINIPORT_ISR_HANDLER (*MiniportInterrupt*)	NdisMQueueDpcEx
Do not use the *TargetProcessors* parameter of MINIPORT_MSI_ISR_HANDLER (*MiniportMessageInterrupt*)	NdisMQueueDpcEx
NDIS_NBL_MEDIA_SPECIFIC_INFORMATION	NDIS_NBL_MEDIA_SPECIFIC_INFORMATION_EX
OID_GEN_PHYSICAL_MEDIUM	OID_GEN_PHYSICAL_MEDIUM_EX
OID_PNP_ADD_WAKE_UP_PATTERN	OID_PM_ADD_WOL_PATTERN
OID_PNP_CAPABILITIES	OID_PM_CURRENT_CAPABILITIES
OID_PNP_ENABLE_WAKE_UP	OID_PM_PARAMETERS
OID_PNP_REMOVE_WAKE_UP_PATTERN	OID_PM_REMOVE_WOL_PATTERN
OID_PNP_WAKE_UP_PATTERN_LIST	OID_PM_WOL_PATTERN_LIST

Summary of Changes Required to Port a Miniport Driver to NDIS 6.20

Article • 03/14/2023

This topic summarizes the changes that are required to port an NDIS 6.x miniport driver to NDIS 6.20.

NDIS 6.20 retains backward compatibility with earlier NDIS versions. For more information about backward compatibility, see NDIS 6.20 Backward Compatibility.

To update a miniport driver to support the NDIS 6.20 environment, you must modify the NDIS 6.x miniport driver as follows:

Build Environment
Replace the preprocessor definition NDIS60_MINIPORT_DRIVER or NDIS61_MINIPORT_DRIVER with NDIS620_MINIPORT_DRIVER.

General Porting Requirements

- Replace obsolete interfaces with NDIS 6.20 versions. For more information about obsolete interfaces, see Obsolete Interfaces in NDIS 6.20.

- Update the following interfaces to support more than 64 processors:
  - Receive side scaling (RSS)
  - Processor information device driver interfaces
  - Resource allocation
  - Read and write locks

  For more information about supporting more than 64 processors, see Support for More than 64 Processors in NDIS 6.20.

Driver Initialization

- Set the NDIS version to 6.20 in the **MajorNdisVersion** and **MinorNdisVersion** members of the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure, which is passed to the **NdisMRegisterMiniportDriver** function.

- Set the miniport driver version in the **MajorDriverVersion** and **MinorDriverVersion** members of the NDIS_MINIPORT_DRIVER_CHARACTERISTICS structure to an appropriate driver-specific value.

- Define direct OID request entry points in the NDIS_MINIPORT_DRIVER_CHARACTERISTICS structure. Support for the direct OID request interface was optional for NDIS 6.1 drivers but it is mandatory for NDIS 6.20 drivers. For more information about the miniport driver direct OID request interface, see Miniport Adapter OID Requests.

Miniport Adapter Initialization

- Use the latest version of the miniport adapter capabilities advertisement interfaces. The following interfaces have updated capabilities:
  - Power Management
  - Receive side scaling (RSS)
  - Hardware assist (VMQ)

- Use the updated versions of these structures:
  - **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES**
  - **NDIS_RESTART_GENERAL_ATTRIBUTES**
  - **NDIS_RECEIVE_SCALE_PARAMETERS**
  - **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES**

  For information about NDIS structure version information, see Specifying NDIS Version Information.

Send and Receive Code Paths

- NDIS 6.20 drivers must support receive-side throttle (RST) in processing receive interrupts. The *ReceiveThrottleParameters* parameters of the *MiniportInterruptDPC* and *MiniportMessageInterruptDPC* DPC handler functions point to an **NDIS_RECEIVE_THROTTLE_PARAMETERS** structure. On entry to the deferred procedure call (DPC) handler, the **MaxNblsToIndicate** member of the NDIS_RECEIVE_THROTTLE_PARAMETERS structure specifies the maximum number of **NET_BUFFER_LIST** structures that the miniport driver should indicate in the DPC. For more information about RST, see Receive Side Throttle in NDIS 6.20.

- Use the updated version of the **NET_BUFFER** structure.

- Optionally support the virtual machine queue (VMQ) interface. For more information about VMQ, see Virtual Machine Queue (VMQ) in NDIS 6.20.

Summary of Changes Required to Port a Protocol Driver to NDIS 6.20

Article • 03/14/2023

This topic summarizes the changes that are required to port an NDIS 6.*x* protocol driver to NDIS 6.20.

NDIS 6.20 retains backward compatibility with earlier NDIS versions. For more information about backward compatibility, see NDIS 6.20 Backward Compatibility.

To update a protocol driver to support the NDIS 6.20 environment, you must modify the NDIS 6.x protocol driver as follows:

Build Environment
Replace the preprocessor definition NDIS61 or NDIS60 with NDIS620.

General Porting Requirements

- Replace obsolete interfaces with NDIS 6.20 versions. For more information about obsolete interfaces, see Obsolete Interfaces in NDIS 6.20.

- Update the following interfaces to support more than 64 processors:
  - Receive side scaling (RSS)
  - Processor information device driver interfaces
  - Resource allocation
  - Read and write locks

  For more information about supporting more than 64 processors, see Support for More than 64 Processors in NDIS 6.20.

Driver Initialization

- Set the NDIS version to 6.20 in the **MajorNdisVersion** and **MinorNdisVersion** members of the **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS** structure that is passed to the **NdisRegisterProtocolDriver** function.

- Set the protocol driver version in the **MajorDriverVersion** and **MinorDriverVersion** members of the NDIS_PROTOCOL_DRIVER_CHARACTERISTICS structure to an appropriate driver-specific value.

Protocol Bind and Unbind Operations

- Use the latest version of the miniport adapter capabilities advertisement interfaces. The following interfaces have updated capabilities:
  - Power Management
  - Power Management
  - Receive side scaling (RSS)
  - Hardware assist (VMQ)

- Use the updated versions of these structures:
  - **NDIS_BIND_PARAMETERS**
  - **NDIS_OFFLOAD_PARAMETERS**

  For information about NDIS structure version information, see Specifying NDIS Version Information.

Send and Receive Data Paths

- Use the updated version of the **NET_BUFFER** structure.

- Optionally support the virtual machine queue (VMQ) interface. For more information about VMQ, see Virtual Machine Queue (VMQ) in NDIS 6.20.

Summary of Changes Required to Port a Filter Driver to NDIS 6.20

Article • 03/14/2023

This topic summarizes the changes that are required to port an NDIS 6.*x* filter driver to NDIS 6.20.

NDIS 6.20 retains backward compatibility with earlier NDIS versions. For more information about backward compatibility, see NDIS 6.20 Backward Compatibility.

To update a filter driver to support the NDIS 6.20 environment, you must modify the NDIS 6.x filter driver as follows:

Build Environment
Replace the preprocessor definition NDIS61 or NDIS60 with NDIS620.

General Porting Requirements

- Replace obsolete interfaces with NDIS 6.20 versions. For more information about obsolete interfaces, see Obsolete Interfaces in NDIS 6.20.

- Update the following interfaces to support more than 64 processors:
  - Receive side scaling (RSS)
  - Processor information device driver interfaces
  - Resource allocation
  - Read and write locks

  For more information about supporting more than 64 processors, see Support for More than 64 Processors in NDIS 6.20.

Driver Initialization

- Set the NDIS version to 6.20 in the **MajorNdisVersion** and **MinorNdisVersion** members of the **NDIS_FILTER_DRIVER_CHARACTERISTICS** structure that is passed to the **NdisFRegisterFilterDriver** function.

- Set the filter driver version in the **MajorDriverVersion** and **MinorDriverVersion** members of the NDIS_FILTER_DRIVER_CHARACTERISTICS structure to an appropriate driver-specific value.

Filter Module Attach and Detach Operations

- Use the latest version of the miniport adapter capabilities advertisement interfaces. The following interfaces have updated capabilities:
  - Power Management
  - Receive-side scaling (RSS)
  - Hardware assist (VMQ)

- Use the updated versions of these structures:
  - **NDIS_FILTER_ATTACH_PARAMETERS**
  - **NDIS_OFFLOAD_PARAMETERS**

  For information about NDIS structure version information, see Specifying NDIS Version Information.

Send and Receive Data Paths

- Use the updated version of the **NET_BUFFER** structure.

- Optionally support the virtual machine queue (VMQ) interface. For more information about VMQ, see Virtual Machine Queue (VMQ) in NDIS 6.20.

Summary of Changes Required to Port an Intermediate Driver to NDIS 6.20

Article • 03/14/2023

This topic summarizes the changes that are required to port an NDIS 6.*x* intermediate driver to NDIS 6.20.

To update an intermediate driver to support the NDIS 6.20 environment, you must modify the NDIS 6.*x* intermediate driver as follows:

Build Environment

- Replace the preprocessor definition NDIS60_MINIPORT_DRIVER or NDIS61_MINIPORT_DRIVER with NDIS620_MINIPORT_DRIVER.

- Replace the preprocessor definition NDIS61 or NDIS60 with NDIS620.

General Porting Requirements

- Except where noted otherwise, protocol driver and miniport driver changes also apply to intermediate drivers. For more information about porting these drivers, see the protocol driver porting summary at Summary of Changes Required to Port a Protocol Driver to NDIS 6.20 and the miniport driver porting summary at Summary of Changes Required to Port a Miniport Driver to NDIS 6.20.

- NDIS 5.x filter intermediate drivers will not be supported in Microsoft Windows versions after Windows 7. You should use the NDIS filter drivers interface for all filter drivers. For more information about NDIS filter drivers, see Summary of Changes Required to Port a Filter Driver to NDIS 6.20.

Roadmap for Developing NDIS Miniport Drivers

Article • 03/14/2023

To create a Network Driver Interface Specification (NDIS) miniport driver package, follow these steps:

- Step 1: Learn about Windows architecture and drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- Step 2: Learn about NDIS.

  For general information about NDIS and NDIS drivers, see the following topics:

  Windows Network Architecture and the OSI Model

  Network Driver Programming Considerations

  Driver Stack Management

  NET_BUFFER Architecture

- Step 3: Determine additional Windows driver design decisions.

  For more information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- Step 4: Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user-mode application. For more information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Lab Kit (HLK) testing, see Building, Debugging, and Testing Drivers. For more information about building, testing, verifying, and debugging tools, see Driver Development Tools.

- Step 5: Read the miniport driver introduction topics:

  Types of NDIS Miniport Drivers

Network Interface Card Support

Sample NDIS Miniport Drivers

- Step 6: Read the writing miniport drivers section.

  This section provides an overview of the primary miniport driver interfaces. These interfaces included functions that miniport drivers provide (*MiniportXxx* functions) and NDIS calls to initiate operations. NDIS provides **Ndis*Xxx*** functions that miniport drivers call to perform NDIS operations.

- Step 7: Review the NDIS miniport driver sample in the Windows driver samples repository on GitHub.

- Step 8: (optional reading) Additional considerations for Miniport Drivers.

  Additional considerations include topics that expand on the primary interfaces that are described in the writing miniport drivers section.

  Obtaining and Setting Miniport Driver Information and NDIS Support for WMI

  NDIS MSI-X

  NDIS Scatter/Gather DMA

  NDIS Power Management

  Plug and Play for NDIS Miniport Drivers

  Reset, Halt, and Shutdown Functions

  Miniport Driver with a WDM Lower Interface

  WAN Miniport Drivers

  Scalable Networking

- Step 9: Develop (or port), build, test, and debug your NDIS driver.

  See the porting guides if you are porting an existing driver:
  - Porting NDIS 5.x Drivers to NDIS 6.0
  - Porting NDIS 6.x Drivers to NDIS 6.20
  - Porting NDIS 6.x Drivers to NDIS 6.30

  For more information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process will help ensure that you build a driver that works.

- Step 10: Create a driver package for your driver.

  For more information about how to install drivers, see Providing a Driver Package. For more information about how to install an NDIS driver, see Components and Files Used for Network Component Installation and Notify Objects for Network Components.

- Step 11: Sign and distribute your driver.

  The final step is to sign (optional) and distribute the driver. If your driver meets the quality standards that are defined for the Windows Hardware Lab Kit (HLK), you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual driver.

Deserialized NDIS Miniport Drivers

Article • 03/14/2023

All NDIS 6.0 and later drivers are *deserialized*.

A *deserialized NDIS miniport driver* serializes the operation of its own *MiniportXxx* functions and queues internally all send requests rather than relying on NDIS to perform these functions. As a result, a deserialized miniport driver can achieve significantly better full-duplex performance than a serialized miniport driver.

The deserialized driver model is the default model for NDIS miniport drivers. Connection-oriented miniport drivers, as well as miniport drivers with a WDM lower edge, must be deserialized drivers. When writing a new NDIS miniport driver, you should write a deserialized driver. If possible, you should also port older drivers to NDIS 6.0 or later. For more information about porting drivers, see:

- Porting NDIS 5.x Drivers to NDIS 6.0
- Porting NDIS 6.x Drivers to NDIS 6.20
- Porting NDIS 6.x Drivers to NDIS 6.30

A deserialized miniport driver must meet the following requirements when it interfaces with NDIS:

- A deserialized miniport driver must identify itself as such to NDIS during initialization.

- A deserialized miniport driver must complete all send requests asynchronously. To complete a send request, connectionless NDIS 6.0 and later miniport drivers call the **NdisMSendNetBufferListsComplete** function. Connection-oriented NDIS 6.0 and later miniport drivers call the **NdisMCoSendNetBufferListsComplete** function.

- A deserialized miniport driver that supports NDIS 6.0 or later sets the **Status** member of the NET_BUFFER_LIST structure that it will pass to **NdisMSendNetBufferListsComplete**.

- If a deserialized miniport driver cannot immediately complete send requests, it cannot return the requests to NDIS for requeuing. Instead, the miniport driver must queue send requests internally until sufficient resources are available to transmit the data.

- A deserialized miniport driver must not examine the structures that it passes to NDIS in receive indications until after NDIS returns them. NDIS returns

NET_BUFFER_LIST structures to a miniport driver's *MiniportReturnNetBufferLists* function.

A deserialized miniport driver must meet the following driver-internal requirements:

- A deserialized miniport driver must protect its network buffer queues with spin locks. A deserialized miniport driver must also protect its shared state from simultaneous access by its own *MiniportXxx* functions.

- A deserialized miniport driver's *MiniportXxx* functions can run at IRQL <= DISPATCH_LEVEL. Consequently, the driver writer cannot assume that *MiniportXxx* functions will be called in the sequence in which they process requests. One *MiniportXxx* function can preempt another *MiniportXxx* function that is running at a lower IRQL.

- A deserialized miniport driver is responsible for network buffer-queue management. When the miniport driver experiences a resource problem, it cannot return send requests to NDIS for requeuing. Instead, the miniport driver must queue internally all send requests until sufficient resources are available to send the data.

- A deserialized miniport driver should complete send requests in the protocol-determined order.

For more information about send and receive requirements for NDIS drivers, see Send and Receive Operations.

Note that a deserialized miniport driver usually completes send requests in protocol-determined order. However, a miniport driver that supports packet priority (for example, IEEE 802.1p) can reorder send requests based on priority information.

Serialized NDIS Miniport Drivers

Article • 03/14/2023

Serialized NDIS miniport drivers are obsolete for Windows Vista and later versions. Serialized miniport drivers are not supported for NDIS 6.0 drivers. Windows Vista supports serialized miniport drivers only for NDIS 5.1 and earlier drivers. Unlike deserialized miniport drivers, a serialized miniport driver relies on NDIS to serialize the operation of its own *MiniportXxx* functions and to manage the queue for sending network data packets.

If you are writing a new miniport driver, you should write a deserialized driver. If possible, you should also port older drivers to NDIS 6.0 or later. For more information about porting drivers to NDIS 6.0, see Porting NDIS 5.x Drivers to NDIS 6.0.

Connection-Oriented NDIS Miniport Drivers

Article • 03/14/2023

A *connection-oriented miniport driver* controls one or more miniport adapters for connection-oriented media. Connection-oriented miniport drivers must be deserialized. For more information about deserialized drivers, see Deserialized NDIS Miniport Drivers.

A connection-oriented miniport driver provides an interface between connection-oriented protocol drivers (connection-oriented clients and call managers) and NIC hardware (for example, physical miniport adapters). For a summary of connection-oriented operations performed by a connection-oriented miniport driver, see Connection-Oriented Operations Performed by Miniport Drivers.

A connection-oriented miniport driver must register the following *MiniportXxx* functions that are specific to connection-oriented operations:

- **MiniportCoCreateVc**
- **MiniportCoDeleteVc**
- **MiniportCoActivateVc**
- **MiniportCoDeactivateVc**
- **MiniportCoSendNetBufferLists**
- **MiniportCoOidRequest**

For more information about registering these functions, see **NdisMRegisterMiniportDriver**.

NDIS Miniport Drivers with a WDM Lower Edge

Article • 03/14/2023

You can write an NDIS miniport driver that controls a device on a bus — for example, the Universal Serial Bus (USB) or the IEEE 1394 (firewire) bus. Such a miniport driver must expose a standard NDIS miniport driver interface at its upper edge and use the class interface for the particular bus at its lower edge. The miniport driver communicates with devices that are attached to the bus by sending I/O request packets (IRPs) to the bus through its Microsoft Windows Driver Model (WDM) lower interface.

A miniport driver with a WDM lower edge must be deserialized. For more information about deserialized drivers, see Deserialized NDIS Miniport Drivers.

For more information about miniport drivers with a WDM lower edge, see Miniport Drivers with a WDM Lower Interface.

Network Interface Card Support

Article • 12/15/2021

This topic describes the types of Network Interface Cards (NICs) that an NDIS miniport driver can manage, as well as how the different kinds of NICs affect the way a driver transfers network data.

Reporting a NIC's medium type to NDIS

To report a medium type for a NIC, a miniport driver passes a pointer to an NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure in the *MiniportAttributes* parameter of the NdisMSetMiniportAttributes function. A miniport driver calls **NdisMSetMiniportAttributes** from its *MiniportInitializeEx* function during initialization. Miniport drivers should set the *MiniportAttributes* attributes after setting the registration attributes in the NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES structure and before setting any other attributes. Setting the *MiniportAttributes* attributes is mandatory. The driver sets the **MediaType** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure to the appropriate media type when setting the *MiniportAttributes* attributes.

When an overlying NDIS protocol driver calls NdisOpenAdapterEx to bind to a specified miniport adapter, it provides a list of medium types on which it can operate. NDIS uses the information from the miniport driver and from the protocol driver to set up a binding. This binding provides the path for transferring network data up and down the driver stack.

Physical NICs

The steps that a miniport driver completes to initialize a miniport adapter and to send and receive network data can depend on the features of a physical device, as follows.

- NDIS-WDM NICs

  With NDIS-WDM NICs, such as USB-based NICs, the way the miniport driver manages memory with DMA does not matter to NDIS and is not visible to it.

- Bus-master DMA NICs

  These NICs can directly access host memory through an on-board DMA controller that manages the transfer of data between the network and host memory without using the host CPU.

To send, the miniport driver sets up the NIC to map the outgoing buffers. The miniport driver then causes the device to start its transfer from this memory. The NIC DMA controller transfers the data from shared system memory onto the network and interrupts the CPU when the send is complete. To receive, the DMA controller transfers incoming data to host memory before notifying the host with an interrupt.

A bus-master DMA NIC typically has an onboard ring buffer that the miniport driver maps to a set of buffers in system memory. Typically, the NIC can be programmed to efficiently handle several packets. A miniport driver that manages such a NIC typically supports multipacket sends and receives because the NIC can efficiently handle several packets and thereby improve its I/O throughput.

- Nonbusmaster DMA NICs

  Currently, nonbusmaster DMA NICs include the following:

  - System DMA NICs

    A miniport driver that manages such a NIC uses the system DMA controller to manage the transfer of packet data to and from the network. Transfer of the data requires the cooperation of the host CPU.

Virtual NICs and miniports

In a virtual machine, NDIS miniport drivers can manage either software-only resources as a virtual miniport, or they can manage a virtual NIC that represents hardware resources. The following table explains the differences between a virtual miniport and a virtual NIC.

Attribute	Virtual miniport	Virtual NIC
Definition	An NDIS miniport driver that maps to a software-enumerated PnP device.	A NIC managed by the host OS hypervisor. The hypervisor makes the virtual machine think that it has some hardware, but no such hardware actually exists in the physical world.
Has interrupts	No	Yes
Can use DMA	No	Yes

Attribute	Virtual miniport	Virtual NIC
Is created or destroyed by...	The guest OS	The host OS
Can reach outside of a guest VM	No	Yes

MiniportXxx Functions

Article • 12/15/2021

The typical miniport driver uses a small number of functions to communicate through NDIS with the upper layers and hardware. Not all of these functions are required. For more information about which functions are optional, which are required, and why, see Initializing a Miniport Driver.

NDIS miniport drivers and upper-layer drivers use the NDIS Library (Ndis.sys) to communicate with each other through calls to **Ndis*Xxx*** functions.

Many miniport driver functions can operate either synchronously or asynchronously. The asynchronous functions have **Ndis*Xxx*Complete** functions that must be called when an operation is finished. For example, if a protocol driver calls **NdisOidRequest** to query miniport driver information, the miniport driver's *MiniportOidRequest* function can pend the reset operation by returning NDIS_STATUS_PENDING. Eventually, the miniport driver must call **NdisMOidRequestComplete** to indicate the final status of the query request.

Linking to the NDIS Library

Article • 03/14/2023

The NDIS Library is packaged in Ndis.sys, a kernel-mode export library, as a set of functions, with emphasis on macros for maximum performance. (An export library is a .sys file that functions similarly to a dynamic-link library.) All NDIS drivers link themselves to the NDIS Library. The NDIS Library functions are described in the Network Reference sections of the Microsoft Windows Driver Kit (WDK) documentation.

The WDK provides Ndis.h as the main header file for miniport drivers. This file defines the entry points for the miniport driver, the NDIS Library functions, and common data structures. The Network Reference section describes the miniport driver, protocol driver, and **Ndis*Xxx*** functions and the common data structures and OIDs.

Miniport Adapter Context

Article • 12/15/2021

NDIS uses a software object called a *miniport adapter* to represent each virtual or physical network device in the system. This object is maintained by NDIS and is opaque to the miniport driver and to protocol drivers. NDIS passes a handle to this structure to the miniport driver's *MiniportInitializeEx* function. The miniport driver subsequently supplies this handle in all calls to **Ndis*Xxx*** functions that pertain to the miniport adapter that the handle specifies.

When a miniport driver is called to initialize a miniport adapter that it manages, it creates its own internal data structure to represent the miniport adapter. The driver uses this structure, referred to as the *miniport adapter context*, to maintain device-specific state information that the driver needs to manage the miniport adapter. The driver passes a handle to this structure to NDIS. For more information about specifying the miniport adapter context, see Initializing an Adapter.

When NDIS calls one of the miniport driver's *MiniportXxx* functions that pertains to a miniport adapter, NDIS passes the miniport adapter context to identify the correct miniport adapter to the driver. The miniport adapter context is owned and maintained by the miniport driver and is opaque to NDIS and to protocol drivers.

Virtual Connection Context

Article • 12/15/2021

Before making a call, a connection-oriented client requests a connection-oriented miniport driver to set up a virtual connection (VC) over which packets can be transmitted or received. Similarly, before indicating an incoming call to a connection-oriented client, a call manager or integrated miniport call manager (MCM) driver requests the miniport driver to set up a VC for the incoming call.

A VC is a logical connection between two connection-oriented entities. Connection-oriented transmissions and receptions always occur on a specific VC.

A connection-oriented miniport driver maintains state information in a miniport driver-allocated context area about each VC that it sets up. This per-VC context is maintained by the miniport driver and is opaque to NDIS and to protocol drivers. In its **MiniportCoCreateVc** function, a connection-oriented miniport driver passes a handle to the VC context area to NDIS, and NDIS passes an *NdisVcHandle* that uniquely identifies the created VC back to the miniport driver, to the appropriate connection-oriented client, and to the call manager or integrated miniport call manager (MCM) driver.

Before data can be sent or received on a VC, the VC must be activated. The call manager initiates activation of the VC by calling **Ndis(M)CmActivateVc** and passing call parameters that include the characteristics of the VC to be activated. In response to this call, NDIS calls the miniport driver's **MiniportCoActivateVc** function, which activates the VC.

After a call is complete or a VC is otherwise not needed, the call manager can deactivate the VC by calling **Ndis(M)CmDeactivateVc**, which causes NDIS to call the miniport driver's **MiniportCoDeactivateVc** function. Either the connection-oriented client or the call manager can initiate the deletion of the VC by calling **NdisCoDeleteVc**, which causes NDIS to call the miniport driver's **MiniportCoDeleteVc** function.

For more information about miniport driver operations on VCs, see Operations on VCs.

Debugger 2PF KDNET Support

Article • 08/23/2023

This topic describes how to enable your miniport NDIS driver for 2PF debugger support to allow increased performance for high speed adapters, often used in data centers. This feature is available in Windows 11 and later.

When enabling kernel debugging on a NIC, the kernel debugging support takes over the physical device to provide both a kernel debugging and network connection on the box. This works fine on consumer low bandwidth NICs (1-10 Gbps), but on high throughput devices that support 10-40+ Gbps the kernel debugging extensibility modules that talk to the hardware generally cannot keep up with the amount of traffic that comes from Windows networking stack, so this degrades overall system performance.

Using the PCI multiple Physical Function (PF) feature for KDNET allows for debugging to be enabled with almost no performance impact.

The Physical Function (PF) is a PCI Express (PCIe) function of a network adapter that supports the single root I/O virtualization (SR-IOV) interface. The PF includes the SR-IOV Extended Capability in the PCIe Configuration space. The capability is used to configure and manage the SR-IOV functionality of the network adapter, such as enabling virtualization and exposing PCIe Virtual Functions (VFs).

The PF supports the SR-IOV Extended Capability structure in its PCIe configuration space. This structure is defined in the PCI-SIG Single Root I/O Virtualization and Sharing 1.1 specification .

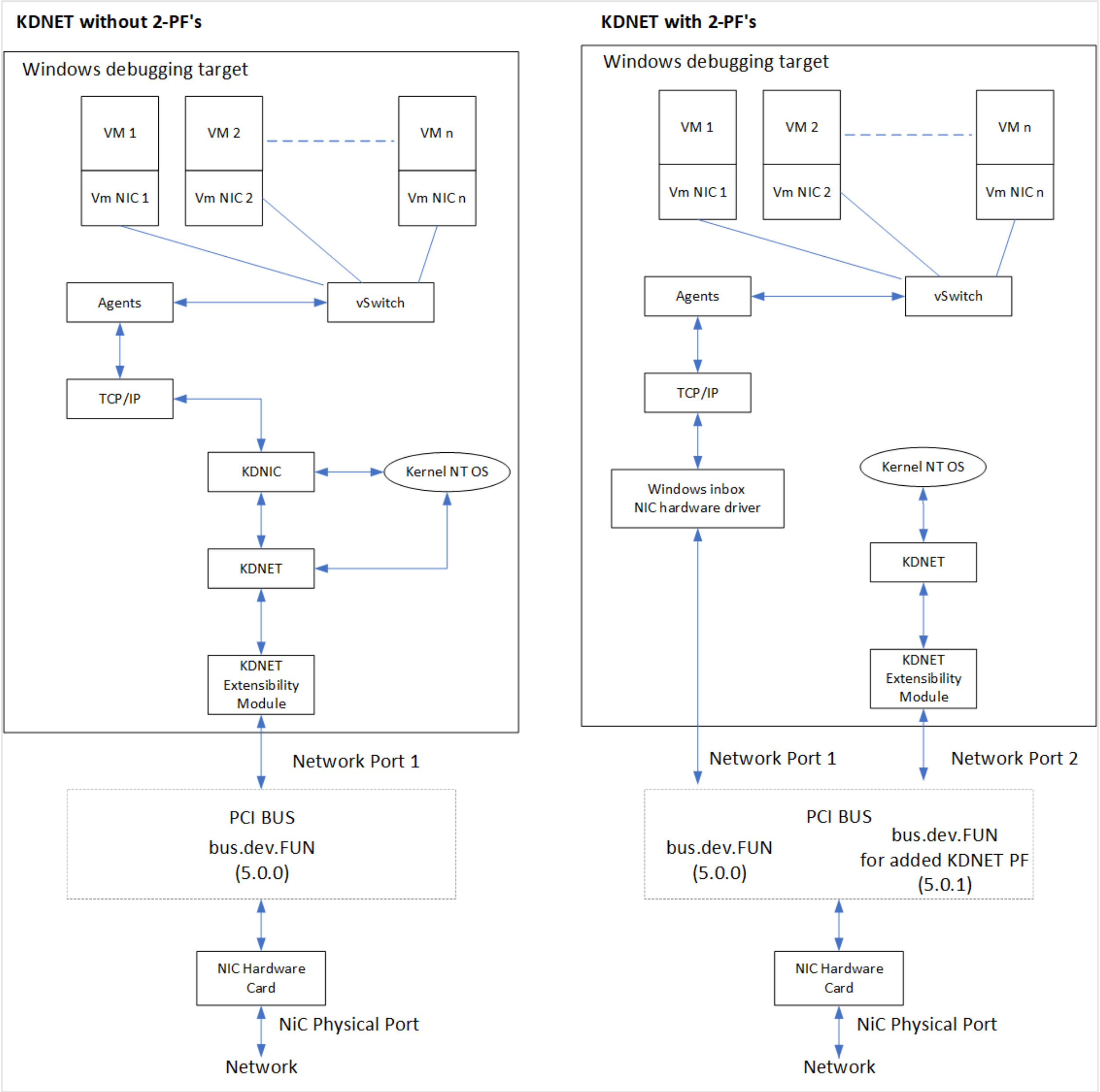
The debugger transport will take advantage of multiple or 2PF enabled miniport drivers. To allow debugging of systems of high speed servers, it is recommended that NIC vendors enable 2PF in all NICs that support multiple PF in the the network card firmware.

For information on configuring 2PF support to test a connection, see Setting Up 2PF Kernel-Mode Debugging using KDNET.

Multiple PF KDNET architecture overview

- The Multiple PF (2PF) functionality is to add/assign a new PF to the original PCI network port (e.g. Bus.dev.fun0.0).

- The new added PF (e.g. bus.dev.fun0.1) is used only by KDNET to route Debugger packets to/from the target.
- The original PF will be used by the Windows inbox NIC driver to route the Windows networking packets (TCP/IP) .
- Using this approach both drivers can work in parallel w/o interfering with each other work.
- Both drivers will run over the partitioned PCI configuration space
    - Windows Inbox driver will run out of the original network port at bus.dev.**fun0.0**
    - KDNET-KDNET-Ext. module will run out of the added PF at bus.dev.**fun0.1**, This way ensures that the Windows inbox NIC driver does not get impacted by sharing the NIC with KDNET.
- The kdnet.exe user mode tool configures the 2PF feature using the Windows inbox driver by adding specific IOCTL codes to add/remove KDNET PF.



Multiple PFs feature design requirements

1. The KDNET 2PF feature needs to work for all current KD scenarios whether it is the pre-NT OS (e.g. Boot Manager, OS loader, WinResume, Hyper-V, SK, etc.), NT OS, or Windows Desktop.

2. Rebooting the system will be required when adding a new PF for a device results in a change needed to the BCD configuration for debugging settings. This means that the configuration for an additional PF must be persistent across boots.

3. The KDNET 2PF should be used only by the debugger to ensure that there is not any other Windows/UEFI ethernet driver accessing/running from the PCI 2PF location when the debugger owns the debug device (the 2PF location is configured using dbgsettings::busparams).

4. Windows or UEFI Ethernet drivers cannot run out of the added KDNET 2PF even when KDNET is not enabled in the system.

5. The 2PF feature should support a dynamic mechanism for adding/enabling and removing/disabling the functionality on the current NIC.

6. The Windows miniport drivers will implement the 2PF feature via servicing the following NDIS OIDs.

OID Name	Description
OID_KDNET_ENUMERATE_PFS	Enumerates PFs on the current bus.dev.fun (BDF), where the miniport driver is running.
OID_KDNET_ADD_PF	Adds a PF to the current BDF, where the miniport driver is running.
OID_KDNET_REMOVE_PF	Removes the added PF, from the passed in BDF.
OID_KDNET_QUERY_PF_INFORMATION	Queries PF information data from the passed in BDF.

The OIDs and their structures are defined in ntddndis.h and kdnetpf.h files that are released with the public WDK.

See the details below on Input/Output parameters for each OID and the information provided in the kdnetpf.h header file.

7. KDNET should be configured via the KDNET 2PF feature on NICS where multiple PF feature is available, and the NIC enables 2PF functionality by following all of the requirements described above.

KDNET Multiple PF Interface for Windows NIC Drivers

To support the KDNET Multiple PF Interface Miniport drivers will need to implement the handling of the following four NDIS OIDs.

- OID_KDNET_ENUMERATE_PFS
- OID_KDNET_ADD_PF
- OID_KDNET_REMOVE_PF
- OID_KDNET_QUERY_PF_INFORMATION

These OIDs and structures are populated in the ntddndis.h and kdnetpf.h files in the public WDK release on this path:

`<WDK root directory>\ddk\inc\ndis`

These files also are available in the Windows SDK, and can be found in this directory.

`\Program Files (x86)\Windows Kits\10\Include\<Version for example 10.0.21301.0>\shared`

The client tool (kdnet.exe) uses a private NDIS IOCTL to route the KDNET 2PF NDIS OIDs to the miniport drivers.

The Multiple PF feature NDIS OIDs

The Multiple PF feature is operated by using these four NDIS OIDs.

1. Enumerate PFs on the miniport BDF primary port using OID: *OID_KDNET_ENUMERATE_PFS*, see definition below.

- *OID_KDNET_ENUMERATE_PFS* returns a list of all BDFs associated to the given primary port from where the miniport driver is running from. The port is represented by the bus.dev.fun (BDF). The operation will list/enumerate the list of PFs that are **associated only** to the bus.dev.fun (BDF port) from where the miniport driver is running on the system, since every miniport driver can determine its BDF location.

- The list of PFs will be returned to the client via a NDIS Query operation.

- The ***OID_KDNET_ENUMERATE_PFS*** OID is associated with the NDIS_KDNET_ENUMERATE_PFS structure.

- The ***OID_KDNET_ENUMERATE_PFS*** driver handler will return a buffer containing the PFs list with each PF element described by the type NDIS_KDNET_PF_ENUM_ELEMENT.

  The PfNumber field contains the PF Function Number, (e.g. bus.dev.**fun**)

  The PfState field contains the PF state possible values- each element type described by NDIS_KDNET_PF_STATE enum.

  NDIS_KDNET_PF_STATE::NdisKdNetPfStatePrimary - This is a primary PF and it's usually used only by the miniport driver.

NDIS_KDNET_PF_STATE::NdisKdnetPfStateEnabled - This is an added secondary PF, that is used by KDNET.

NDIS_KDNET_PF_STATE::NdisKdnetPfStateConfigured - This is an added PF, but it is only added/configured and is not used.

- If the PF list output buffer size is not large enough to allocate the actual PFs list, then the OID handler needs to return `E_NOT_SUFFICIENT_BUFFER` error return value, together with the required buffer size, so the client tool can allocate the required size buffer, and then the client can make another call with the correct buffer size allocated. In addition, the that the OID request status field (described by NDIS_IOCTL_OID_REQUEST_INFO.status) should be set to equal to `NDIS_STATUS_BUFFER_TOO_SHORT`.

2. Add PCI PF to the miniport BDF primary port (OID: *OID_KDNET_ADD_PF,* see definition below)

- Add a PF to the miniport primary port. The port is represented by the BDF.
- The newly added PF will be returned to the client via a NDIS Query operation.
- The *OID_KDNET_ADD_PF* OID is associated with the NDIS_KDNET_ADD_PF structure.
- The *OID_KDNET_ADD_PF* driver handler will return an ULONG containing the *added* PF function number.
- This OID request will have only one Output parameter: `AddedFunctionNumber`. The `AddedFunctionNumber` indicates the added Function number value at the miniport PCI location (the BDF miniport). The kdnet.exe utility will receive this value and setup dbgsettings::busparams to points to the added PF.

> ⓘ Note
>
> The added PF can be used exclusively by KDNET, so Windows NIC drivers are rigged to expressly *NOT* run on an added PF, so this also applies when KDNET is *NOT* enabled on the system and the PF has been added to the port.

3. Remove PCI PF (OID: *OID_KDNET_REMOVE_PF*, see definition below)

- Remove a PF from the given port. The port is represented by the BDF.

- The ***OID_KDNET_REMOVE_PF*** OID is associated with the NDIS_KDNET_REMOVE_PF structure.
- The ***OID_KDNET_REMOVE_PF*** OID has an input BDF port and returns an ULONG containing the *removed* PF function number via a NDIS Method operation.
- This function will succeed only on the PFs that has been added via using the ***OID_KDNET_ADD_PF*** OID.
- This OID request will have the input BDF port from where needs to be removed the BDF. This function has an Output parameter of `FunctionNumber`. The output `FunctionNumber` will contain the removed Function number value.

4. Query PCI PF information (OID: *OID_KDNET_QUERY_PF_INFORMATION*, see definition below)

- This OID code allows querying specific PF data on a given port. The port is represented by the BDF.
- The requested PF information will be returned to the client via a NDIS Method operation.
- The ***OID_KDNET_QUERY_PF_INFORMATION*** OID is associated with the NDIS_KDNET_QUERY_PF_INFORMATION structure.
- The ***OID_KDNET_QUERY_PF_INFORMATION*** OID has an input BDF port and returns a buffer containing the following data:
    - MAC Address: Network address of the assigned new KDNET PF if there is any.
    - Usage Tag: Describes the entity that owns the PF port. It contains a constant value described by NDIS_KDNET_PF_USAGE_TAG enum.
    - Maximum Number of PFs: Contains an ULONG with the maximum number of PFs that can be added to the given BDF.
    - Device ID: Contains the device ID associated to the given BDF port. This is required for cases where the NIC FW assigns a new device ID to the new added KDNET PF port.
- This OID requests the information for any passed in BDF port (BDF is an input parameter for this operation), so it’s *not* necessarily related to the current BDF from where the driver is running from.

NDIS OIDs for KDNET on 2PF

Ntddndis.h file defines the OIDs.

C++

```
#if (NDIS_SUPPORT_NDIS686)

 //

 // Optional OIDs to handle network multiple PF feature.

 //
#define OID_KDNET_ENUMERATE_PFS 0x00020222
#define OID_KDNET_ADD_PF 0x00020223
#define OID_KDNET_REMOVE_PF 0x00020224
#define OID_KDNET_QUERY_PF_INFORMATION 0x00020225
#endif // (NDIS_SUPPORT_NDIS686)
```

Kdnetpf.h file describes the type and structures associated with the NDIS OIDs.

C++

```
#if (NDIS_SUPPORT_NDIS686)

 //
 // Used to query/add/remove Physical function on a network port.
 // These structures are used by these OIDs:
 // OID_KDNET_ENUMERATE_PFS
 // OID_KDNET_ADD_PF
 // OID_KDNET_REMOVE_PF
 // OID_KDNET_QUERY_PF_INFORMATION
 // These OIDs handle PFs that are primary intended to be used by  KDNET.
 //
 //
 // PCI location of the port to query
 //
 typedef struct _NDIS_KDNET_BDF
 {
 ULONG SegmentNumber;
 ULONG BusNumber;
 ULONG DeviceNumber;
 ULONG FunctionNumber;
 ULONG Reserved;
 } NDIS_KDNET_BDF, *PNDIS_KDNET_PCI_BDF;

 //
 // PF supported states.
 //
 typedef enum _NDIS_KDNET_PF_STATE
 {
 NdisKdNetPfStatePrimary = 0x0,
 NdisKdnetPfStateEnabled = 0x1,
 NdisKdnetPfStateConfigured = 0x2,
 } NDIS_KDNET_PF_STATE,*PNDIS_KDNET_PF_STATE;

 //
 // PF Usage Tag
 // Used to indicate the entity that owns the PF.
 // Used by the query NdisKdnetQueryUsageTag.
 //
 typedef enum _NDIS_KDNET_PF_USAGE_TAG
 {
 NdisKdnetPfUsageUnknown = 0x0,
 NdisKdnetPfUsageKdModule = 0x1,
 } NDIS_KDNET_PF_USAGE_TAG,*PNDIS_KDNET_PF_USAGE_TAG;

 //
 // PF element array structure
 //
 typedef struct _NDIS_KDNET_PF_ENUM_ELEMENT
 {
 NDIS_OBJECT_HEADER Header;

 //
 // PF value (e.g. if <bus.dev.fun>, then PF value = fun)
 //
 ULONG PfNumber;

 //
 // The PF state value (defined by NDIS_KDNET_PF_STATE)
 //
 NDIS_KDNET_PF_STATE PfState;

 } NDIS_KDNET_PF_ENUM_ELEMENT, *PNDIS_KDNET_PF_ENUM_ELEMENT;
#define NDIS_KDNET_PF_ENUM_ELEMENT_REVISION_1 1
#define NDIS_SIZEOF_KDNET_PF_ENUM_ELEMENT_REVISION_1 \
 RTL_SIZEOF_THROUGH_FIELD(NDIS_KDNET_PF_ENUM_ELEMENT, PfState)

 //
 // This structure describes the data required to enumerate the list of PF
 // Used by OID_KDNET_ENUMERATE_PFS.
 //
 typedef struct _NDIS_KDNET_ENUMERATE_PFS
 {
 NDIS_OBJECT_HEADER Header;

 //
 // The size of each element is the sizeof(NDIS_KDNET_PF_ENUM_ELEMENT)
 //
 ULONG ElementSize;

 //
 // The number of elements in the returned array
 //
 ULONG NumberOfElements;

 //
 // Offset value to the first element of the returned array.
 // Each array element is defined by NDIS_KDNET_PF_ENUM_ELEMENT.
 //
 ULONG OffsetToFirstElement;
 } NDIS_KDNET_ENUMERATE_PFS, *PNDIS_KDNET_ENUMERATE_PFS;

#define NDIS_KDNET_ENUMERATE_PFS_REVISION_1 1
#define NDIS_SIZEOF_KDNET_ENUMERATE_PFS_REVISION_1 \
 RTL_SIZEOF_THROUGH_FIELD(NDIS_KDNET_ENUMERATE_PFS,
 OffsetToFirstElement)

 //
 // This structure indicates the data required to add a PF to the BDF port.
 // Used by OID_KDNET_ADD_PF.
 //
 typedef struct _NDIS_KDNET_ADD_PF
 {
 NDIS_OBJECT_HEADER Header;

 //
 // One element containing the added PF port number
 //
 ULONG AddedFunctionNumber;
 } NDIS_KDNET_ADD_PF, *PNDIS_KDNET_ADD_PF;

#define NDIS_KDNET_ADD_PF_REVISION_1 1
#define NDIS_SIZEOF_KDNET_ADD_PF_REVISION_1 \
 RTL_SIZEOF_THROUGH_FIELD(NDIS_KDNET_ADD_PF, AddedFunctionNumber)

 //
 // This structure indicates the data required to remove a PF from the BDF
port.
 // Used by OID_KDNET_REMOVE_PF.
 //

 typedef struct _NDIS_KDNET_REMOVE_PF
 {
 NDIS_OBJECT_HEADER Header;

 //
 // PCI location that points to the PF that needs to be removed
 //
 NDIS_KDNET_BDF Bdf;

 //
 // One element containing the removed PF port
 //
 ULONG FunctionNumber;
 } NDIS_KDNET_REMOVE_PF, *PNDIS_KDNET_REMOVE_PF;
#define NDIS_KDNET_REMOVE_PF_REVISION_1 1
#define NDIS_SIZEOF_KDNET_REMOVE_PF_REVISION_1 \
 RTL_SIZEOF_THROUGH_FIELD(NDIS_KDNET_REMOVE_PF, FunctionNumber)

 //
 // This structure describes the data required to query the PF management
data
 // Used by OID_KDNET_QUERY_PF_INFORMATION
 //
 typedef struct _NDIS_KDNET_QUERY_PF_INFORMATION
 {
 NDIS_OBJECT_HEADER Header;

 //
 // PF PCI location to query for
 //
 NDIS_KDNET_BDF Bdf;

 //
 // PF assigned MAC address
 //
 UCHAR NetworkAdddress[6];

 //
 // PF Usage tag described by NDIS_KDNET_PF_USAGE_TAG
 //
 ULONG UsageTag;

 //
 // Maximum number of Pfs that can be associated to the Primary BDF.
 //
 ULONG MaximumNumberOfSupportedPfs;

 //
 // KDNET PF device ID (Used if there is a new added PF and
 // the FW assigns a new DeviceID to the added KDNET PF)
 //
 ULONG DeviceId;

 } NDIS_KDNET_QUERY_PF_INFORMATION, *PNDIS_KDNET_QUERY_PF_INFORMATION;
#define NDIS_KDNET_QUERY_PF_INFORMATION_REVISION_1 1
#define NDIS_SIZEOF_KDNET_QUERY_PF_INFORMATION_REVISION_1 \
 RTL_SIZEOF_THROUGH_FIELD(NDIS_KDNET_QUERY_PF_INFORMATION, DeviceId)

#endif // (NDIS_SUPPORT_NDIS686)
```

See also

Setting Up 2PF Kernel-Mode Debugging using KDNET

Network OIDs

kdnetpf.h header

Network OIDs

Article • 12/15/2021

A miniport driver maintains information about its capabilities and current status, as well as information about each miniport adapter that it manages. Each information type is identified by an object identifier (OID). OIDs are system-defined. NDIS handles many of the OID requests for miniport drivers and NDIS does not pass such requests on to the miniport driver. The miniport driver reports many of its capabilities, which were formerly reported in response to OID queries, in its attributes during initialization. For more information about reporting attributes, see Initializing an Adapter.

NDIS and higher level drivers can query and, in some cases, set information by using OIDs.

- Higher level drivers for connectionless media call **NdisOidRequest** to query or set information in a connectionless miniport driver. To perform a query or a set operation, NDIS calls the miniport driver's *MiniportOidRequest* function.

- Higher level drivers for connection-oriented media call **NdisCoOidRequest** to query or set information in a connection-oriented miniport driver. To perform both query and set operations, NDIS calls the miniport driver's **MiniportCoOidRequest** function.

NDIS maps many of the system-defined OIDs for miniport drivers to globally unique identifiers (GUIDs). NDIS registers these GUIDs with the kernel-mode Microsoft Windows Management Instrumentation (WMI) that supports user-mode Web-Based Enterprise Management (WBEM) applications. When a WMI client queries or sets one of these GUIDs, NDIS translates the request to a query OID operation or a set OID operation, as appropriate, and then passes any returned information and the status back to WMI. You can map custom GUIDs to custom OIDs or miniport driver status. A miniport driver must register custom GUID-to-OID or GUID-to-status mappings with NDIS during initialization.

For more information about querying and setting OIDs, creating custom OIDs, and NDIS support for WMI, see Obtaining and Setting Miniport Driver Information and NDIS Support for WMI.

Sample NDIS Miniport Drivers

Article • 03/14/2023

The Network driver samples in the Windows driver samples repository on GitHub includes sample code for miniport drivers that manage several types of network cards. You can modify these sample drivers to your needs. The sample drivers contain functions that can be adapted to a new but similar driver. There are always hardware-dependent functions that you must write. However, many functions are fairly standard. For example, functions that communicate with the NDIS Library instead of a network interface card are typically standard. For these driver functions, the code in a sample driver might be usable with little or no modification.

Initializing a Miniport Driver

Article • 12/15/2021

When a networking device becomes available, the system loads the NDIS miniport driver to manage the device (if the driver is not already loaded). Every miniport driver must provide a DriverEntry function. The system calls **DriverEntry** after it loads the driver. **DriverEntry** registers the miniport driver's characteristics with NDIS (including the supported NDIS version and the driver entry points).

The system passes two arguments to DriverEntry:

- A pointer to the driver object, which was created by the I/O system.
- A pointer to the registry path, which specifies where driver-specific parameters are stored.

In DriverEntry, miniport drivers pass both of these pointers in a call to the NdisMRegisterMiniportDriver function. Miniport drivers export a set of standard *MiniportXxx* functions by storing their entry points in an **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure and passing that structure to **NdisMRegisterMiniportDriver**.

DriverEntry for miniport drivers returns the value that is returned by the call to **NdisMRegisterMiniportDriver**.

A miniport driver also performs any other driver-specific initialization that it requires in DriverEntry. The driver performs adapter-specific initialization in the *MiniportInitializeEx* function. For more information about adapter initialization, see Initializing an Adapter.

DriverEntry can allocate the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure on the stack because the NDIS library copies the relevant information to its own storage. **DriverEntry** should clear the memory for this structure with NdisZeroMemory before setting any driver-supplied values in its members. The **MajorNdisVersion** and **MinorNdisVersion** members must contain the major and minor versions of NDIS that the driver supports. In each Xxx**Handler** member of the characteristics structure, **DriverEntry** must set the entry point of a driver-supplied *MiniportXxx* function, or the member must be **NULL**.

To enable a miniport driver to configure optional services, NDIS calls the MiniportSetOptions function within the context of the miniport driver's call to NdisMRegisterMiniportDriver. For more information about optional services, see Configuring Optional Miniport Driver Services.

Drivers that call NdisMRegisterMiniportDriver must be prepared for NDIS to call their *MiniportInitializeEx* functions any time after **DriverEntry** returns. Such a driver must have sufficient installation and configuration information stored in the registry or available from calls to an **NdisXxx** bus-type-specific configuration function to set up any NIC-specific resources the driver will need to carry out network I/O operations.

The miniport driver must eventually call **NdisMDeregisterMiniportDriver** to release resources that it allocated by calling **NdisMRegisterMiniportDriver**. If the driver initialization fails after the call to **NdisMRegisterMiniportDriver** succeeded, the driver can call **NdisMDeregisterMiniportDriver** from within DriverEntry. Otherwise, the miniport driver must release the driver-specific resources that it allocates in its *MiniportDriverUnload* function. In other words, if NdisMRegisterMiniportDriver does not return NDIS_STATUS_SUCCESS, **DriverEntry** must release any resources that it allocated before it returns control. The driver will not be loaded if this occurs. For more information, see Unloading a Miniport Driver.

Unloading a Miniport Driver

Article • 12/15/2021

The driver object that is associated with an NDIS miniport driver specifies an **Unload** routine. The system calls the *Unload* routine when all the devices that the driver services have been removed. NDIS provides the *Unload* routine for miniport drivers. NDIS calls a miniport driver's *MiniportDriverUnload* function from the *Unload* routine.

A miniport driver must call **NdisMDeregisterMiniportDriver** from *MiniportDriverUnload*.

A miniport driver's *MiniportDriverUnload* function should also release any driver-specific resources. The system will complete a driver unload operation after *MiniportDriverUnload* returns.

The functionality of the *MiniportDriverUnload* function is driver-specific. As a general rule, *MiniportDriverUnload* should undo the operations that were performed during driver initialization. For more information about driver initialization, see Initializing a Miniport Driver.

Miniport Adapter States and Operations

Article • 12/15/2021

For each adapter that it manages, an NDIS 6.0 or later miniport driver must support the following set of operational states:

Halted
The Halted state is the initial state of all adapters. When an adapter is in the Halted state, NDIS can call the driver's *MiniportInitializeEx* function to initialize the adapter.

Shutdown
In the Shutdown state, a system shutdown and restart must occur before the system can use the adapter again.

Initializing
In the Initializing state, a miniport driver completes any operations that are required to initialize an adapter.

Paused
In the Paused state, the adapter does not indicate received network data or accept send requests.

Restarting
In the Restarting state, a miniport driver completes any operations that are required to restart send and receive operations for an adapter.

Running
In the Running state, a miniport driver performs send and receive processing for an adapter.

Pausing
In the Pausing state, a miniport driver completes any operations that are required to stop send and receive operations for an adapter.

In the following table, the headings are the adapter states. Major events are listed in the first column. The rest of the entries in the table specify the next state that the adapter enters after an event occurs within a state. The blank entries represent invalid event/state combinations.

Event \ State	**Halted**	**Shutdown**	**Initializing**	**Paused**	**Restarting**	**Running**	**Pausing**
MiniportInitializeEx	Initializing						
Initialize is complete			Paused				
MiniportShutdownEx				Shutdown	Shutdown	Shutdown	Shutdown
MiniportHaltEx				Halted			

Event \ State	Halted	Shutdown	Initializing	Paused	Restarting	Running	Pausing
MiniportRestart				Restarting			
Restart is complete					Running		
MiniportPause						Pausing	
Pause is complete							Paused
Initialize failed			Halted				
Restart failed					Paused		
Send and receive operations						Running	Pausing
OID requests				Paused	Restarting	Running	Pausing

Note The events listed in the preceding table are the primary events for an NDIS 6.0 or later adapter.

Note The reset operation does not affect miniport adapter operational states. The state of the adapter might change while a reset operation is in progress. For example, NDIS might call a driver's pause handler when there is a reset operation in progress. In this case, the driver can complete either the reset or the pause operation in any order while following the normal requirements for each operation. For a reset operation, the driver can fail transmit request packets or it can keep them queued and complete them later. However, you should note that an overlying driver cannot complete a pause operation while its transmit packets are pending.

The primary miniport driver events are defined as follows:

MiniportInitializeEx
NDIS called the driver's *MiniportInitializeEx* function to initialize an adapter. For more information about adapter initialization, see Initializing a Miniport Adapter.

Initialize is complete
After *MiniportInitializeEx* returns successfully, the initialize operation is complete and the adapter is in the Paused state.

MiniportShutdownEx
NDIS called the driver's *MiniportShutdownEx* function to shutdown an adapter. For more information, see Miniport Adapter Shutdown.

MiniportHaltEx
NDIS called the driver's *MiniportHaltEx* function to halt an adapter. For more information, see Halting a Miniport Adapter.

MiniportRestart
NDIS called the driver's **MiniportRestart** function to restart a paused adapter. Because an

adapter is in the Paused state after initialization, this event is also required to start the adapter after adapter initialization is complete. For more information, see Starting an Adapter.

Restart is complete
After the driver is ready to handle send and receive operations, the restart operation is complete and the adapter is in the Running state.

MiniportPause
NDIS called the driver's *MiniportPause* function to pause an adapter. For more information, see Pausing an Adapter.

Pause is complete
After the driver has completed all operations that are necessary to stop send and receive operations, the pause operation is complete and the adapter is in the Paused state.

Note The driver must wait for NDIS to return all its outstanding receive indications before the pause operation is complete.

Initialize failed
If NDIS calls a driver's *MiniportInitializeEx* function and the initialization attempt fails, the adapter returns to the Halted state.

Restart failed
If NDIS calls a driver's **MiniportRestart** function and the restart attempt fails, the adapter remains in the Paused state.

Send and Receive Operations
A driver must handle send and receive operations in the Running and Pausing states. For more information about send and receive operations, see Miniport Driver Send and Receive Operations.

OID Requests
A driver must handle OID Requests in the Running, Restarting, Paused, and Pausing states. For more information about OID requests, see OID Requests for an Adapter.

Related topics

Halting a Miniport Adapter

Initializing a Miniport Adapter

Miniport Adapter Shutdown

Miniport Driver Send and Receive Operations

Pausing an Adapter

Starting an Adapter

Initializing a Miniport Adapter

Article • 08/29/2023

When a networking device becomes available, the system loads the required NDIS miniport driver, if it is not already loaded. Subsequently, the Plug and Play (PnP) manager sends NDIS a Plug and Play IRP to start the device. NDIS calls the miniport driver's *MiniportInitializeEx* function to initialize an adapter for network I/O operations. NDIS can call *MiniportInitializeEx* at any time after the driver is initialized. For more information about miniport driver initialization, see Initializing a Miniport Driver.

Until *MiniportInitializeEx* returns, NDIS submits no requests for the adapter being initialized. The adapter is in the Initializing state.

A miniport driver typically performs the following tasks in *MiniportInitializeEx*:

1. Obtains configuration information for the adapter.

2. Obtains information about the hardware resources for the adapter.

3. Calls the **NdisMSetMiniportAttributes** and provides the following attributes that are associated with the adapter:

    - A pointer to a driver-allocated context area.
    - Appropriate attributes flags.
    - The time-out interval for calling its *MiniportCheckForHangEx* function.
    - The interface type.

4. Initializes adapter-specific resources.

The miniport driver specifies the adapter attributes in the **NDIS_MINIPORT_ADAPTER_ATTRIBUTES** structure that *MiniportInitializeEx* passes to **NdisMSetMiniportAttributes**.

Typically, *MiniportInitializeEx* allocates adapter-specific resources in the following order:

1. Nonpaged pool memory.

2. **NET_BUFFER** and **NET_BUFFER_LIST** pools (see Miniport Driver Send and Receive Operations).

3. Spin locks.

4. Timers.

5. IO ports.

6. DMA (see Scatter/Gather DMA).

7. Shared memory.

8. Interrupts (see Managing Interrupts).

After *MiniportInitializeEx* returns successfully, the adapter is in the Paused state. NDIS can call the **MiniportRestart** function to transition the adapter to the Running state. For more information, see Starting a Miniport Adapter.

If *MiniportInitializeEx* returns NDIS_STATUS_SUCCESS, the driver should release all the resources for the adapter in the *MiniportHaltEx* function. For more information, see Halting a Miniport Adapter.

The driver must call **NdisMSetMiniportAttributes** and set the **GeneralAttributes** in the **NDIS_MINIPORT_ADAPTER_ATTRIBUTES** structure if it returns NDIS_STATUS_SUCCESS.

If *MiniportInitializeEx* failed, *MiniportInitializeEx* must release all resources that it allocated before it returns and the adapter returns to the Halted state.

Related topics

Halting a Miniport Adapter

Miniport Adapter States and Operations

Miniport Driver Send and Receive Operations

Scatter/Gather DMA

Starting a Miniport Adapter

Halting a Miniport Adapter

Article • 12/15/2021

NDIS calls an NDIS miniport driver's *MiniportHaltEx* function to deallocate resources when an adapter is removed from the system, and to stop the hardware. NDIS can call *MiniportHaltEx* after the driver's *MiniportInitializeEx* function returns successfully. For more information about *MiniportInitializeEx*, see Initializing a Miniport Adapter.

MiniportHaltEx must free any resources that the driver allocated for a device. The driver must call the reciprocals of the **Ndis*Xxx*** functions with which it originally allocated the resources. As a general rule, a *MiniportHaltEx* function should call the reciprocal **Ndis*Xxx*** functions in the reverse order used during initialization.

If an adapter generates interrupts, a miniport driver's *MiniportHaltEx* function can be preempted by the driver's *MiniportInterrupt* function until *MiniportHaltEx* disables interrupts.

NDIS does not call *MiniportHaltEx* if there are outstanding OID requests or send requests. NDIS submits no further requests for the affected device after NDIS calls *MiniportHaltEx*.

After *MiniportHaltEx* returns, the miniport driver is in the Halted state.

Related topics

Adapter States of a Miniport Driver

Miniport Adapter States and Operations

Miniport Driver Halt Handler

Writing NDIS Miniport Drivers

Starting and Pausing a Miniport Adapter Overview

Article • 12/15/2021

NDIS pauses an adapter to stop data flow that could interfere with Plug and Play operations, such as adding or removing a filter driver, or requests, such as setting a packet filter or multicast address list on the NIC. For more information about how to modify a running driver stack, see Modifying a Running Driver Stack.

NDIS restarts an adapter from the Paused state. The adapter enters the Paused start after adapter initialization is complete or after a pause operation is complete.

The following topics provide more information about starting and pausing and adapter:

- Starting an Adapter
- Pausing an Adapter

Starting an Adapter

Article • 12/15/2021

NDIS calls a miniport driver's MiniportRestart function to initiate a restart request for an adapter that is in the Paused state. The driver can resume indicating received data immediately after NDIS calls *MiniportRestart* and before the miniport driver completes the restart operation, either synchronously or asynchronously.

When it calls a miniport driver's MiniportRestart function, NDIS passes a pointer to an NDIS_RESTART_ATTRIBUTES structure to the miniport driver in the **RestartAttributes** member of the NDIS_MINIPORT_RESTART_PARAMETERS structure.

To complete the restart operation asynchronously, *MiniportRestart* returns NDIS_STATUS_PENDING and the driver must call the NdisMRestartComplete function after the operation is complete.

The miniport driver should be ready to accept send requests after it completes the restart operation. NDIS does not initiate any other Plug and Play operations, such as halt, initialize, or a pause request, until the restart operation is complete.

After the driver is ready to handle send and receive operations, the adapter is in the Running state.

Pausing an Adapter

Article • 12/15/2021

NDIS calls a miniport driver's *MiniportPause* function to initiate a pause operation. The adapter remains in the Pausing state until the pause operation is complete.

In the Pausing state, the miniport driver must complete outstanding receive operations. The driver must also complete any outstanding send operations and it should reject any new send requests.

To complete receive operations, the driver waits for all calls to the **NdisMIndicateReceiveNetBufferLists** function to return and NDIS must return all outstanding **NET_BUFFER_LIST** structures to the miniport driver's *MiniportReturnNetBufferLists* function.

To complete outstanding send operations, the miniport driver should call the **NdisMSendNetBufferListsComplete** function for all of the outstanding NET_BUFFER_LIST structures. The driver should reject any new send requests made to its *MiniportSendNetBufferLists* function immediately.

After a miniport driver completes all outstanding send and receive operations, the driver must complete the pause request either synchronously or asynchronously. If the pause operation is completed asynchronously, the driver calls **NdisMPauseComplete** to complete the pause request. After completing the pause request, the miniport driver is in the Paused state.

NDIS does not initiate other Plug and Play operations, such as halt, initialize, power change, or restart operations, while the miniport driver is in the Pausing state. NDIS can initiate these Plug and Play operations after a miniport driver is in the Paused state.

Configuring Optional Miniport Driver Services

Article • 12/15/2021

NDIS calls a miniport driver's *MiniportSetOptions* function to allow the driver to configure optional services. NDIS calls *MiniportSetOptions* within the context of the miniport driver's call to the **NdisMRegisterMiniportDriver** function.

MiniportSetOptions registers the default entry points for optional *MiniportXxx* functions and can allocate other driver resources. To register optional *MiniportXxx* functions, the miniport driver calls the **NdisSetOptionalHandlers** function and passes a characteristics structure at the *OptionalHandlers* parameter.

Starting with NDIS 6.0, the valid characteristics structures include the following:

NDIS_MINIPORT_CO_CHARACTERISTICS

NDIS_MINIPORT_PNP_CHARACTERISTICS

NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS

NDIS_PROVIDER_CHIMNEY_OFFLOAD_GENERIC_CHARACTERISTICS (see NDIS 6.0 TCP chimney offload documentation)

NDIS_PROVIDER_CHIMNEY_OFFLOAD_TCP_CHARACTERISTICS (see NDIS 6.0 TCP chimney offload documentation)

Starting with NDIS 6.30, the valid characteristics structures also include the following:

NDIS_MINIPORT_SS_CHARACTERISTICS

NDIS_NDK_PROVIDER_CHARACTERISTICS

Miniport Driver Send and Receive Operations

Article • 12/15/2021

Miniport drivers handle send requests from overlying drivers and originate receive indications. In a single function call, NDIS miniport drivers can indicate a linked list with multiple received **NET_BUFFER_LIST** structures. Miniport drivers can handle send requests for lists of multiple NET_BUFFER_LIST structures with multiple **NET_BUFFER** structures on each NET_BUFFER_LIST structure.

Miniport drivers must manage receive buffer pools. Most miniport drivers create pools that preallocate a single NET_BUFFER structure with each NET_BUFFER_LIST structure.

The following topics provide more information about miniport driver buffer management, send operations, and receive operations:

Miniport Driver Buffer Management

Sending Data from a Miniport Driver

Canceling a Send Request in a Miniport Driver

Indicating Received Data from a Miniport Driver

Miniport Driver Buffer Management

Article • 12/15/2021

Miniport drivers typically call NdisAllocateNetBufferListPool from *MiniportInitializeEx* to create a pool of NET_BUFFER_LIST structures. Miniport drivers use these structures to indicate received data.

Typically, a miniport driver that allocates a NET_BUFFER_LIST structure will allocate and queue one NET_BUFFER structure on that NET_BUFFER_LIST structure. It is more efficient to preallocate NET_BUFFER structures when you allocate a pool of NET_BUFFER_LIST structures than to allocate NET_BUFFER_LIST structures and NET_BUFFER structures separately.

Miniport drivers can call **NdisAllocateNetBufferListPool** and set the *AllocateNetBuffer* parameter to **TRUE** to indicate that NET_BUFFER structures are preallocated. In this case, a NET_BUFFER structure is preallocated with each NET_BUFFER_LIST structure that the driver allocates from the pool. Such drivers must call NdisAllocateNetBufferAndNetBufferList to allocate structures from this pool.

Typically, a miniport driver calls **NdisAllocateNetBufferAndNetBufferList** from *MiniportInitializeEx* to allocate as many buffers as it will require for subsequent receive operations. In this case, the driver manages an internal list of free buffers.

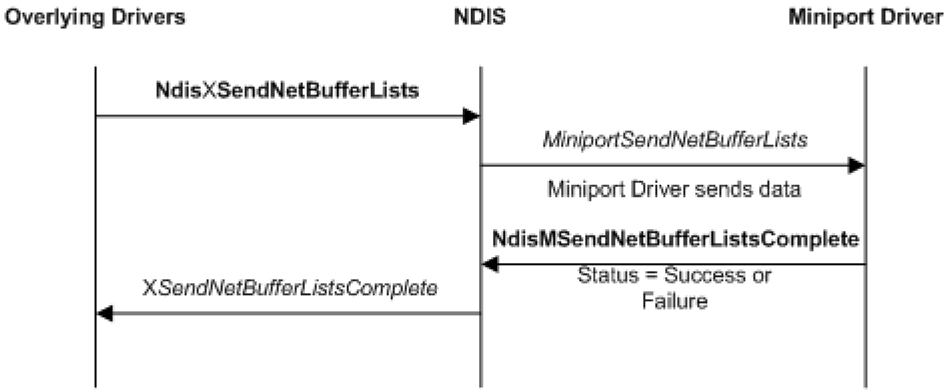
The *MiniportReturnNetBufferLists* function can prepare a returned NET_BUFFER_LIST structure for reuse in a subsequent receive indication. Although *MiniportReturnNetBufferLists* could return the NET_BUFFER_LIST structures to a pool (for example, it could call NdisFreeNetBufferList), it can be more efficient to reuse the structures without returning them to the pool.

A miniport driver should free all the NET_BUFFER_LIST structures and associated data when NDIS halts the adapter. A driver can call **NdisFreeNetBufferList** to free the structures and the NdisFreeNetBufferListPool function to free the NET_BUFFER_LIST pool.

Sending Data from a Miniport Driver

Article • 12/15/2021

The following figure illustrates a miniport driver send operation.


Overlying Drivers
NDIS
Miniport Driver
NdisXSendNetBufferLists
MiniportSendNetBufferLists
Miniport Driver sends data
NdisMSendNetBufferListsComplete
Status = Success or Failure
XSendNetBufferListsComplete

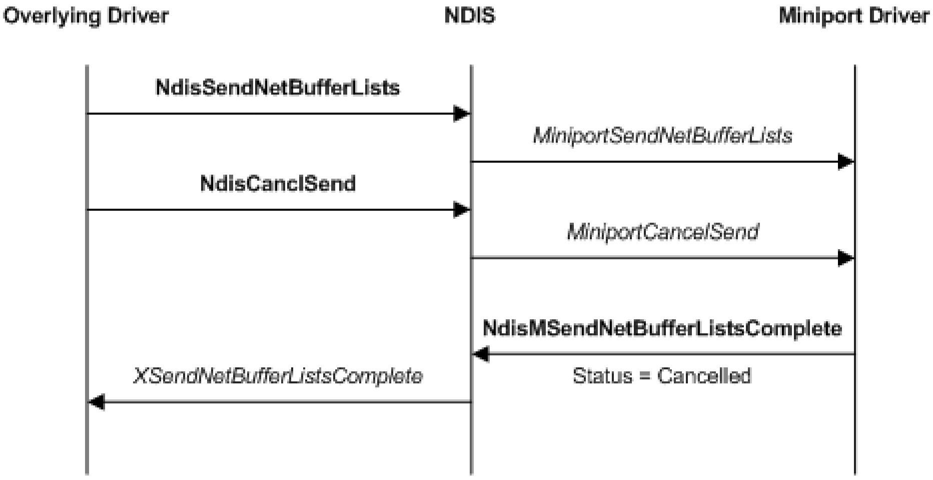
NDIS calls a miniport driver's *MiniportSendNetBufferLists* function to transmit the network data that is described by a linked list of **NET_BUFFER_LIST** structures.

Miniport drivers call the **NdisMSendNetBufferListsComplete** function to return a linked list of NET_BUFFER_LIST structures to an overlying driver and to return the final status of a send request.

Canceling a Send Request in a Miniport Driver

Article • 12/15/2021

The following figure illustrates a miniport driver cancel send operation.



Protocol, filter, and intermediate drivers can call **NdisCancelSendNetBufferLists** to cancel outstanding send requests. These overlying drivers must mark the send data with a cancellation ID before making a send request.

NDIS calls a miniport driver's *MiniportCancelSend* function to cancel the transmission of all **NET_BUFFER_LIST** structures that are marked with a specified cancellation identifier.

A miniport driver's *MiniportCancelSend* function performs the following operations:

1. Traverses its list of outstanding send requests for the specified adapter and calls **NDIS_GET_NET_BUFFER_LIST_CANCEL_ID** to obtain the cancellation identifier for each NET_BUFFER_LIST structure. The miniport driver compares the cancellation ID that NDIS_GET_NET_BUFFER_LIST_CANCEL_ID returns with the cancellation ID that NDIS passed to *MiniportCancelSend*.

2. Removes from all NET_BUFFER_LIST structures whose cancellation identifiers match the specified cancellation identifier from its list of outstanding send requests.

3. Calls the **NdisMSendNetBufferListsComplete** function for all canceled NET_BUFFER_LIST structures to return the structures.The miniport driver sets the status field of the NET_BUFFER_LIST structures to NDIS_STATUS_SEND_ABORTED.

NDIS Poll Mode

Article • 01/17/2024

Overview of NDIS Poll Mode

NDIS Poll Mode is an OS controlled polling execution model that drives the network interface datapath.

Previously, NDIS had no formal definition of a datapath execution context. NDIS drivers typically relied on Deferred Procedure Calls (DPCs) to implement their execution model. However using DPCs can overwhelm the system when long indication chains are made and avoiding this problem requires a lot of code that's tricky to get right. NDIS Poll Mode offers an alternative to DPCs and similar execution tools.

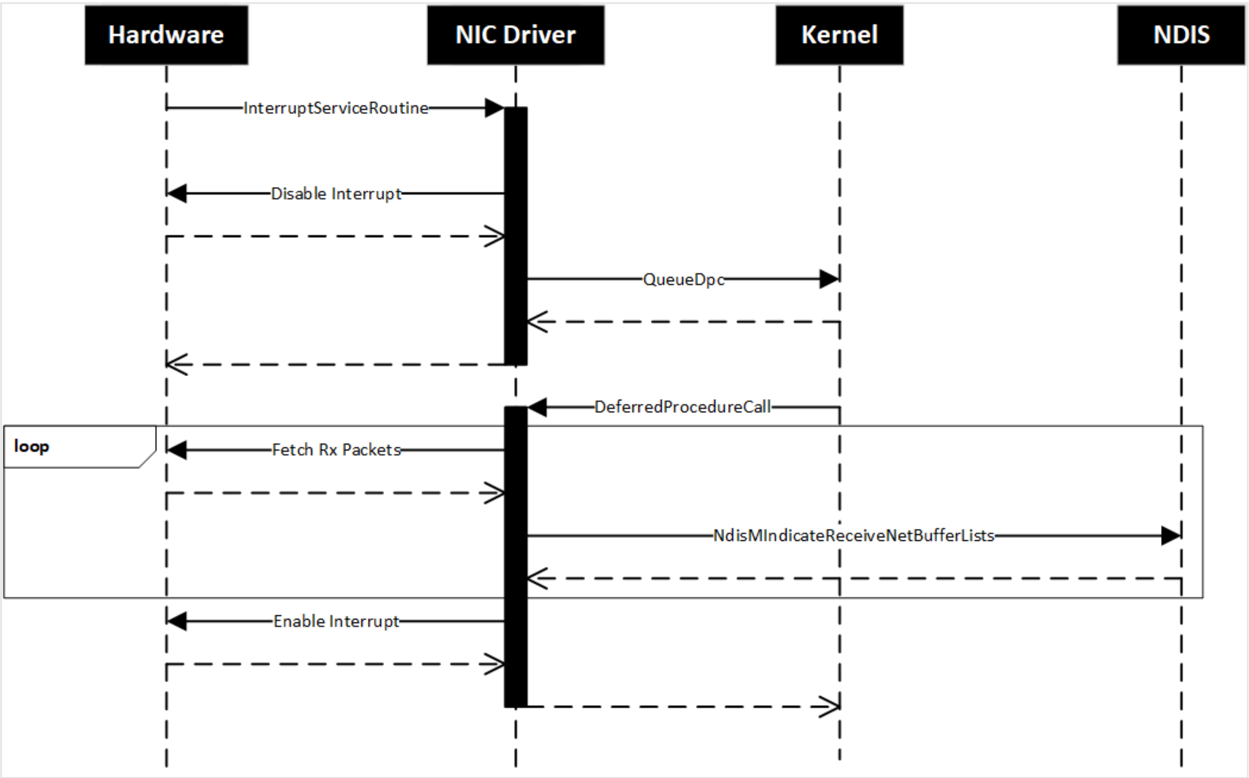
NDIS Poll Mode moves the complexity of scheduling decisions away from NIC drivers and into NDIS, where NDIS sets work limits per iteration. To achieve this Poll Mode provides:

1. A mechanism for the OS to exert back pressure on the NIC.
2. A mechanism for the OS to finely control interrupts.

NDIS Poll Mode is available to NDIS 6.85 and later miniport drivers.

Problems with the DPC model

The following sequence diagram illustrates a typical example of how an NDIS miniport driver handles a burst of Rx packets using a DPC. In this example the hardware is standard in terms of PCIe NICs. It has a receive hardware queue and an interrupt mask for that queue.



When there's no network activity the hardware has the Rx interrupt enabled. When an Rx packet arrives:

1. The hardware generates an interrupt and NDIS calls the driver's *MiniportInterrupt* function (ISR).

2. The driver does very little work in the ISR because they run at a very high IRQL. The driver disables the interrupt from the ISR and defers the hardware processing to a *MiniportInterruptDPC* function (DPC).

3. NDIS eventually calls the driver's DPC and the driver drains any completions from the hardware queue and indicates them to the OS.

Two pain points can affect the network stack when the driver defers I/O operations to a DPC:

1. The driver doesn't know if the system is capable of processing all of the data that is being indicated, so the driver has no choice but to drain as many elements as possible from its hardware queue and indicate them up the stack.

2. Since the driver is using a DPC to defer work from its ISR, all the indications are made at DISPATCH_LEVEL. This can overwhelm the system when long indication chains are made and cause Bug Check 0x133 DPC_WATCHDOG_VIOLATION.

Avoiding these pain points requires a lot of tricky code in your driver. While you can check if the DPC watchdog is close to the limit with the **KeQueryDpcWatchdogInformation** function and break out of the DPC, you still need to build an infrastructure around this in your driver: You need some way to pause for a bit, then continue to indicate the packets, and at the same time you need to synchronize all this with the lifetime of the datapath.

Introduction to Poll objects

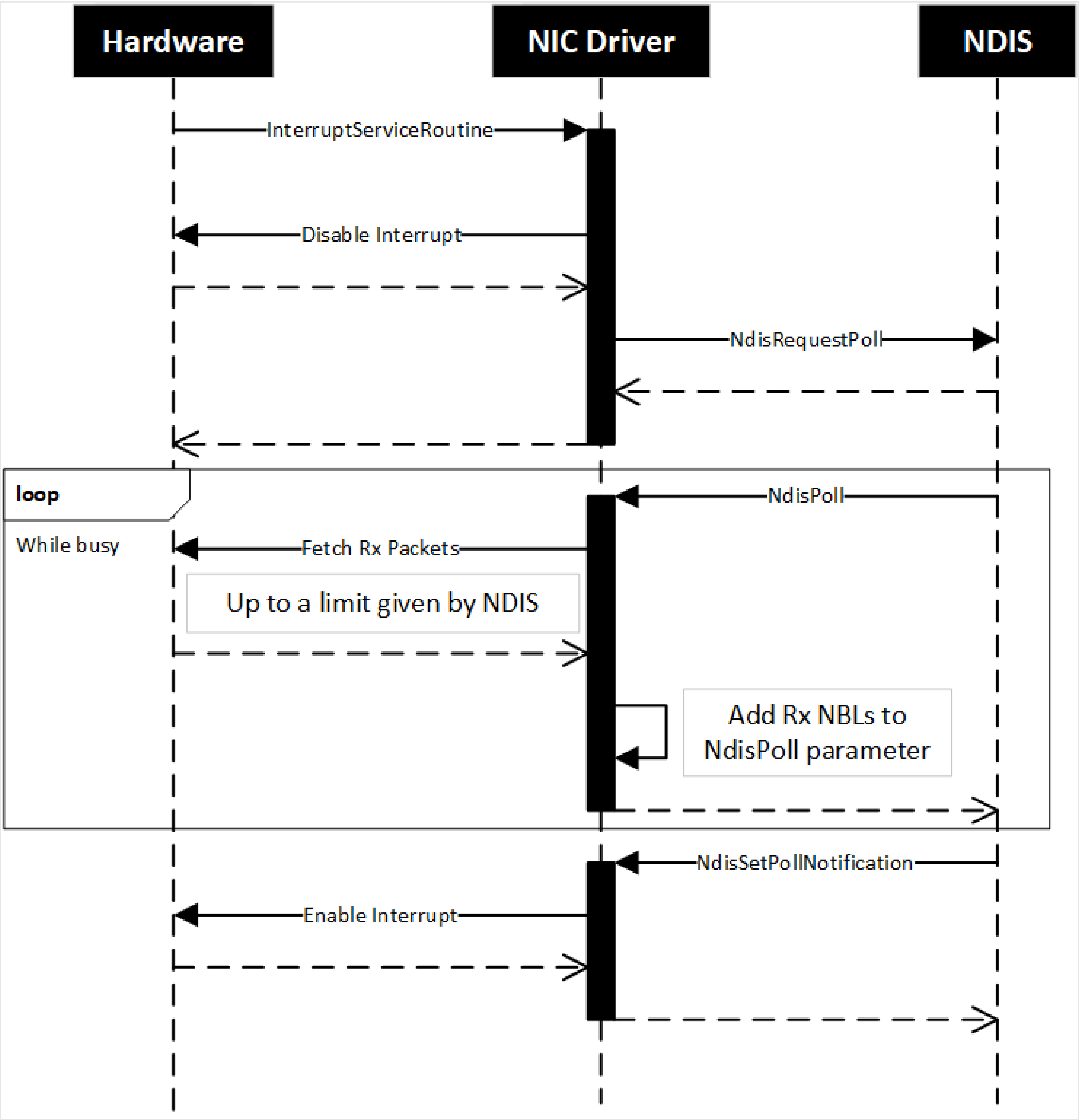
NDIS Poll Mode introduces the Poll object to resolve the pain points associated with DPCs. A Poll object is an execution context construct. Miniport drivers can use a Poll object in place of a DPC when dealing with datapath operations.

A Poll object offers the following:

- It provides a way for NDIS to set work limits per iteration.
- It is closely tied to a notification mechanism. This keeps the OS and the NIC in sync regarding when work needs to be processed.
- It has a concept of iteration and interrupts built in. When using DPCs, drivers are forced to re-enable the interrupt every time they finish a DPC. When using Poll objects, drivers don't need to re-enable the interrupt each polling iteration because Poll Mode will let your driver know when it's done polling and it's time to re-enable the interrupt again.
- When making scheduling decisions, the system can be smart about whether to run at DISPATCH_LEVEL or PASSIVE_LEVEL. This can allow fine-tuned prioritization of traffic from different NICs and lead to a fairer workload distribution on the machine.
- It has serialization guarantees. Once you are running code from within a Poll object's execution context you are guaranteed that no other code related to the same execution context will run. This allows a NIC driver to have a lock free implementation of its datapath.

The NDIS Poll Mode model

The following sequence diagram illustrates how the same hypothetical PCIe NIC driver handles a burst of Rx packets using a Poll object instead of a DPC.



Like the DPC model, when an Rx packet arrives the hardware generates an interrupt, NDIS calls the driver's ISR, and the driver disables the interrupt from the ISR. At this point the Poll Mode model diverges:

1. Instead of queueing a DPC, the driver queues a Poll object (that it previously created) from the ISR to notify NDIS that new work is ready to be processed.

2. At some point in the future NDIS calls the driver's poll iteration handler to process the work. Unlike a DPC, the driver is not allowed to indicate as many Rx NBLs as there are elements ready in its hardware queue. The driver should instead check the handler's poll data parameter to get the maximum number of NBLs it can indicate.

   Once the driver fetches up to the maximum number of Rx packets it should initialize NBLs, add them to the NBL queue provided by the poll handler, and exit the callback. The driver shouldn't enable the interrupt before exiting.

3. NDIS continues to poll the driver until it assesses that the driver is no longer making forward progress. At this point NDIS will stop polling and ask the driver to re-enable the interrupt.

Standardized INF keyword for NDIS Poll Mode

The following keyword must be used to enable or disable support for NDIS Poll Mode:

***NdisPoll** Enumeration standardized INF keywords have the following attributes:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with SubkeyName.

Value
The enumeration integer value that is associated with each option in the list. This value is stored in NDI\params\ *SubkeyName**Value*.

EnumDesc
The display text that is associated with each value that appears in the menu.

Default
The default value for the menu.

⛶ Expand table

SubkeyName	ParamDesc	Value	EnumDesc
***NdisPoll**	Ndis Poll Mode	0	Disabled
		1 (Default)	Enabled

For more information about using enumeration keywords, see Enumeration Keywords.

Creating a Poll object

To create a Poll object, the miniport driver does the following in its *MiniportInitializeEx* callback function:

1. Allocates a private miniport context.
2. Allocates an **NDIS_POLL_CHARACTERISTICS** structure to specify entry points for the *NdisPoll* and *NdisSetPollNotification* callback functions.
3. Calls **NdisRegisterPoll** to create the Poll object and store it in the miniport context.

The following example shows how a miniport driver might create a Poll object for a receive queue flow. Error handling is omitted for simplicity.

```C++
NDIS_SET_POLL_NOTIFICATION NdisSetPollNotification;
NDIS_POLL NdisPoll;

NDIS_STATUS
MiniportInitialize(
    _In_ NDIS_HANDLE NdisAdapterHandle,
    _In_ NDIS_HANDLE MiniportDriverContext,
    _In_ NDIS_MINIPORT_INIT_PARAMETERS * MiniportInitParameters
)
{
    // Allocate a private miniport context
    MINIPORT_CONTEXT * miniportContext = ...;

    NDIS_POLL_CHARACTERISTICS pollCharacteristics;
    pollCharacteristics.Header.Type = NDIS_OBJECT_TYPE_DEFAULT;
    pollCharacteristics.Header.Revision = NDIS_POLL_CHARACTERISTICS_REVISION
_1;
    pollCharacteristics.Header.Size = NDIS_SIZEOF_NDIS_POLL_CHARACTERISTICS_
REVISION_1;
    pollCharacteristics.SetPollNotificationHandler =
NdisSetPollNotification;
    pollCharacteristics.PollHandler = NdisPoll;

    // Create a Poll object and store it in the miniport context
    NdisRegisterPoll(
        NdisAdapterHandle,
        miniportContext,
        &pollCharacteristics,
        &miniportContext->RxPoll);

    return NDIS_STATUS_SUCCESS;
}
```

Queuing a Poll object for execution

From an ISR, miniport drivers call **NdisRequestPoll** to queue a Poll object for execution. The following example shows receive handling but ignores the sharing of interrupt lines for simplicity.

```C++
BOOLEAN
MiniportIsr(
  KINTERRUPT * Interrupt,
  void * Context
)
{
    auto miniportContext = static_cast<MINIPORT_CONTEXT *>(Context);
    auto hardwareContext = miniportContext->HardwareContext;

    // Check if this interrupt is due to a received packet
    if (hardwareContext->ISR & RX_OK)
    {
        // Disable the receive interrupt and queue the Poll
        hardwareContext->IMR &= ~RX_OK;
        NdisRequestPoll(miniportContext->RxPoll, nullptr);
    }

    return TRUE;
}
```

Implementing the Poll iteration handler

NDIS invokes the miniport driver's *NdisPoll* callback to poll for receive indications and send completions. NDIS first invokes *NdisPoll* when the driver calls NdisRequestPoll to queue a Poll object. NDIS will keep invoking *NdisPoll* while the driver is making forward progress on receive indications or transmit completions.

For receive indications, the driver should do the following in *NdisPoll*:

1. Check the **receive** parameter of the NDIS_POLL_DATA structure to get the maximum number of NBLs it can indicate.
2. Fetch up to the maximum number of Rx packets.
3. Initialize the NBLs.
4. Add them to the NBL queue provided by the NDIS_POLL_RECEIVE_DATA structure (located in the NDIS_POLL_DATA structure of the *NdisPoll* **PollData** parameter).
5. Exit the callback.

For transmit completions, the driver should do the following in *NdisPoll*:

1. Check the **transmit** parameter of the NDIS_POLL_DATA structure to get the maximum number of NBLs it can complete.
2. Fetch up to the maximum number of Tx packets.
3. Complete the NBLs.

4. Add them to the NBL queue provided by the NDIS_POLL_TRANSMIT_DATA structure (located in the NDIS_POLL_DATA structure of the *NdisPoll* **PollData** parameter).
5. Exit the callback.

The driver shouldn't enable the Poll object's interrupt before exiting the *NdisPoll* function. NDIS will keep polling the driver until it assesses that no forward progress is being made. At this point NDIS will stop polling and ask the driver to re-enable the interrupt.

Here's how a driver might implement *NdisPoll* for a receive queue flow.

C++

```
_Use_decl_annotations_
void
NdisPoll(
    void * Context,
    NDIS_POLL_DATA * PollData
)
{
    auto miniportContext = static_cast<MINIPORT_CONTEXT *>(Context);
    auto hardwareContext = miniportContext->HardwareContext;

    // Drain received frames
    auto & receive = PollData->Receive;
    receive.NumberOfRemainingNbls = NDIS_ANY_NUMBER_OF_NBLS;
    receive.Flags = NDIS_RECEIVE_FLAGS_SHARED_MEMORY_VALID;

    while (receive.NumberOfIndicatedNbls < receive.MaxNblsToIndicate)
    {
        auto rxDescriptor = HardwareQueueGetNextDescriptorToCheck(hardwareCo
ntext->RxQueue);

        //
If this descriptor is still owned by hardware stop draining packets
        if ((rxDescriptor->Status & HW_OWN) != 0)
            break;

        auto nbl = MakeNblFromRxDescriptor(miniportContext-
>NblPool, rxDescriptor);

        AppendNbl(&receive.IndicatedNblChain, nbl);
        receive.NumberOfIndicatedNbls++;

        // Move to next descriptor
        HardwareQueueAdvanceNextDescriptorToCheck(hardwareContext->RxQueue);
    }
}
```

Managing interrupts

Miniport drivers implement the *NdisSetPollNotification* callback to enable or disable the interrupt associated with a Poll object. NDIS typically invokes the *NdisSetPollNotification* callback when it detects that the miniport driver is not making forward progress in *NdisPoll*. NDIS uses *NdisSetPollNotification* to tell the driver that it will stop invoking *NdisPoll*. The driver should invoke **NdisRequestPoll** when new work is ready to be processed.

Here's how a driver might implement *NdisSetPollNotification* for a receive queue flow.

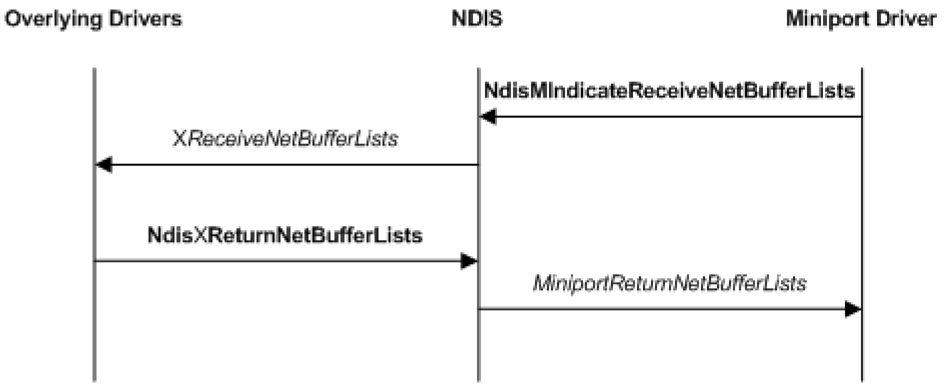
```C++
_Use_decl_annotations_
void
NdisSetPollNotification(
    void * Context,
    NDIS_POLL_NOTIFICATION * Notification
)
{
    auto miniportContext = static_cast<MINIPORT_CONTEXT *>(Context);
    auto hardwareContext = miniportContext->HardwareContext;

    if (Notification->Enabled)
    {
        hardwareContext->IMR |= RX_OK;
    }
    else
    {
        hardwareContext->IMR &= ~RX_OK;
    }
}
```

Indicating Received Data from a Miniport Driver

Article • 12/15/2021

The following figure illustrates a miniport driver receive indication.


Overlying Drivers
NDIS
Miniport Driver
NdisMIndicateReceiveNetBufferLists
XReceiveNetBufferLists
NdisXReturnNetBufferLists
MiniportReturnNetBufferLists

Miniport drivers call the NdisMIndicateReceiveNetBufferLists function to indicate the receipt of data from the network. The **NdisMIndicateReceiveNetBufferLists** function passes the indicated list of NET_BUFFER_LIST structures up the stack to overlying drivers.

If a miniport driver sets the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of NdisMIndicateReceiveNetBufferLists, this indicates that the miniport driver must regain ownership of the NET_BUFFER_LIST structures immediately. In this case, NDIS does not call the miniport driver's *MiniportReturnNetBufferLists* function to return the **NET_BUFFER_LIST** structures. The miniport driver regains ownership immediately after **NdisMIndicateReceiveNetBufferLists** returns.

If a miniport driver does not set the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of NdisMIndicateReceiveNetBufferLists, NDIS returns the indicated NET_BUFFER_LIST structures to the miniport driver's *MiniportReturnNetBufferLists* function. In this case, the miniport driver relinquishes ownership of the indicated structures until NDIS returns them to *MiniportReturnNetBufferLists*.

Registering and Deregistering Interrupts

Article • 12/15/2021

A miniport driver calls NdisMRegisterInterruptEx to register an interrupt. The driver allocates and initializes an NDIS_MINIPORT_INTERRUPT_CHARACTERISTICS structure to specify the interrupt characteristics and function entry points. The driver passes the structure to **NdisMRegisterInterruptEx**.

Drivers call the NdisMDeRegisterInterruptEx function to release resources that were previously allocated with **NdisMRegisterInterruptEx**.

Handling interrupts for NDIS miniport drivers

Article • 03/14/2023

NDIS calls the *MiniportInterrupt* function when a NIC, or another device that shares the interrupt with the NIC, generates an interrupt.

MiniportInterrupt should return **FALSE** immediately if the underlying NIC did not generate the interrupt. Otherwise, it returns **TRUE** after processing the interrupt.

A miniport driver should do as little work as possible in its *MiniportInterrupt* function. It should defer I/O operations to the *MiniportInterruptDPC* function. NDIS calls *MiniportInterruptDPC* to complete the deferred processing of an interrupt.

To queue additional DPCs after *MiniportInterrupt* returns, the miniport driver sets the bits of the **TargetProcessors** parameter of the *MiniportInterrupt* function. To request additional DPCs from *MiniportInterrupt* or *MiniportInterruptDPC*, the miniport driver calls the **NdisMQueueDpc** function.

The miniport driver can call **NdisMQueueDpc** to request additional DPC calls for other processors.

Synchronizing with Interrupts

Article • 12/15/2021

If a miniport driver's *MiniportInterrupt* function shares resources, such as NIC registers or state variables, with another *MiniportXxx* function that runs at a lower IRQL, that *MiniportXxx* function must call **NdisMSynchronizeWithInterruptEx**. This call ensures that the miniport driver's *MiniportSynchronizeInterrupt* function accesses the shared resources in a synchronized and multiprocessor-safe manner.

Interrupt Moderation

Article • 12/15/2021

To reduce the number of interrupts, many NICs use *interrupt moderation*. With interrupt moderation, the NIC hardware will not generate an interrupt immediately after it receives a packet. Instead, the hardware waits for more packets to arrive, or for a time-out to expire, before generating an interrupt. The hardware vendor specifies the maximum number of packets, time-out interval, or other interrupt moderation algorithm.

The measured round-trip time for a packet is one of the most commonly used techniques to determine the network bandwidth between two endpoints. However, when interrupt moderation is enabled, receiving a packet does not generate an immediate interrupt and therefore the perceived round-trip time for a particular packet becomes larger than the average time. To allow accurate measurement of round trip time for a packet, NDIS provides the ability to disable and enable interrupt moderation on demand.

All NDIS 6.0 and later miniport drivers must support the OID_GEN_INTERRUPT_MODERATION OID. If a miniport driver does not support interrupt moderation, the driver must specify **NdisInterruptModerationNotSupported** in the **InterruptModeration** member of the **NDIS_INTERRUPT_MODERATION_PARAMETERS** structure.

NDIS 6.0 and later miniport drivers must support both the OID_GEN_INTERRUPT_MODERATION OID set and query requests. The set request directs the miniport driver to enable or disable interrupt moderation and the query request reports the current state of interrupt moderation.

A miniport driver that supports interrupt moderation should turn this capability on by default unless the **InterruptModeration** standard keyword in the registry disables it. For more information about the standard keywords, see Standardized INF Keywords for Network Devices.

Miniport Adapter OID Requests

Article • 12/15/2021

NDIS defines object identifier (OID) values to identify miniport adapter parameters, which include operating parameters such as device characteristics, configurable settings and statistics. For more information about OIDs, see NDIS OIDs.

For NDIS 6.1 and later miniport drivers, NDIS provides a Direct OID Request Interface. The *Direct OID request path* supports OID requests that are queried or set frequently. The Direct OID Request Interface is optional for NDIS drivers.

For NDIS 6.80 and later miniport drivers, NDIS provides a Synchronous OID Request Interface. The *Synchronous OID request path* supports OIDs that require synchronization or OIDs that should not be queued by filter drivers, such as RSSv2 OIDs. The Synchronous OID Request Interface is optional for NDIS drivers but is required if the miniport driver advertises support for RSSv2.

The following topics provide more information about miniport driver OID requests:

Handling OID Requests In a Miniport Adapter

Miniport Adapter OID Request Serialization

Miniport Adapter Direct OID Requests

Miniport Adapter Synchronous OID Requests

Miniport adapter OID request serialization

Article • 12/15/2021

All OID requests to a miniport adapter are serialized by NDIS except for direct OID requests, which were designed not be serialized. A miniport adapter will not receive a new OID request until any pending request is completed. Therefore, miniport adapters must complete OIDs promptly.

> ⓘ **Note**
>
> We recommend completing an OID request in less than 1000ms, or 1 second, so the user will not notice any delay in performance. For specific information about timing OID requests, see the **NdisTimedOidComplete** Driver Verifier rule.

One exception to this OID serialization rule is for Wi-Fi miniport adapters that use WDI, which may see a second OID request if they take too long to complete the previous OID. The following example explains what happens in this situation:

1. The first OID request is passed to the WDI miniport adapter.
2. The NIC does not respond to the OID within the time limit specified by the driver.
3. WDI calls the driver's MINIPORT_WDI_ADAPTER_HANG_DIAGNOSE callback function to collect diagnostic data about the NIC.
4. The first OID is no longer considered to block serialization. This means the WDI miniport adapter can now receive other OID requests, even though the first OID is serialized. However, these other OIDS are also serialized, which means the WDI miniport adapter will not pend more than 2 OIDs simultaneously (the first OID that is still hung and a second OID).

Related topics

For more information about WDI UE hang detection, see UE hang detection: Steps 1-14.

For more information about OID requests in NDIS, see Simplifying your OID request handler on the NDIS blog.

Handling OID Requests In a Miniport Adapter

Article • 12/15/2021

NDIS calls a miniport driver's *MiniportOidRequest* function to submit an OID request to query or set information in the driver. NDIS calls the *MiniportOidRequest* function either on its own behalf or on behalf of an overlying driver that called the **NdisOidRequest** or **NdisFOidRequest** function.

NDIS passes *MiniportOidRequest* a pointer to an **NDIS_OID_REQUEST** structure that contains the request information. The request structure contains an OID_Xxx identifier that indicates the type of request and other members to define the request data.

The **Timeout** member specifies a time-out, in seconds, for the request. NDIS can reset the driver or cancel the request if the time-out expires before the driver completes the request.

The **RequestId** member specifies an optional identifier for the request. Miniport drivers can set the **RequestId** member of a status indication to the value obtained from the **RequestId** member of an associated OID request. Typically, miniport drivers can ignore this member. If a driver must set this member, the reference page for the particular OID provides the required values. For more information about status indications, see Adapter Status Indications.

A miniport driver that successfully handles an OID set request must set the **SupportedRevision** member in the **NDIS_OID_REQUEST** structure upon return from the OID set request. The **SupportedRevision** member notifies the initiator of the request of the revision that the driver supported. For example, a miniport driver can create an Xxx_REVISION_2 structure, supply values that are appropriate for an Xxx_REVISION_1 structure, and fill the rest of the structure with zeros. The miniport driver would report Xxx_REVISION_1 in the **SupportedRevision** member. In this case, a protocol driver that can support an Xxx_REVISION_2 will use Xxx_REVISION_1 information that the miniport driver supported. For more information about version information in NDIS structures, see Specifying NDIS Version Information.

A miniport driver can complete an OID request synchronously by returning a success or failure status.

A miniport driver can complete an OID request asynchronously by returning NDIS_STATUS_PENDING. In this case, the miniport driver must call the **NdisMOidRequestComplete** function to complete the operation.

If *MiniportOidRequest* returns NDIS_STATUS_PENDING, NDIS will not call *MiniportOidRequest* with another request for the adapter until the pending request is completed.

NDIS can call a miniport driver's *MiniportCancelOidRequest* function to cancel an OID request.

Miniport Adapter Direct OID Requests

Article • 12/15/2021

To support the direct OID request path, miniport drivers provide *MiniportXxx* function entry points in the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure and NDIS provides **NdisM*Xxx*** functions for miniport drivers.

The *direct OID request interface* is similar to the standard OID request interface. For example, the **NdisMDirectOidRequestComplete** and *MiniportDirectOidRequest* functions are similar to the **NdisMOidRequestComplete** and *MiniportOidRequest* functions.

Note NDIS 6.1 supports specific OIDs for use with the direct OID request interface. OIDs that existed before NDIS 6.1 and some NDIS 6.1 OIDs are not supported. To determine if an OID can be used in the direct OIDs interface, see the OID reference page.

Miniport drivers must be able to handle direct OID requests that are not serialized. Unlike the standard OID request interface, NDIS does not serialize direct OID requests with other requests that are sent with the direct OID interface or with the standard OID request interface. Also, miniport drivers must be able to handle direct OID requests at IRQL <= DISPATCH_LEVEL.

To support the direct OIDs request interface, use the documentation for the standard OID request interface. The following table shows the relationship between the functions in the direct OID request interface and the standard OID request interface.

Direct OID function	Standard OID function
MiniportDirectOidRequest	*MiniportOidRequest*
MiniportCancelDirectOidRequest	*MiniportCancelOidRequest*
NdisMDirectOidRequestComplete	**NdisMOidRequestComplete**

Miniport Adapter Synchronous OID Requests

Article • 12/15/2021

To support the Synchronous OID request path, miniport drivers provide a *MiniportSynchronousOidRequest* function entry point in the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure when they call the **NdisMRegisterMiniportDriver** function.

For miniport drivers, the *Synchronous OID request interface* differs from the Regular and Direct OID request interfaces in that miniport drivers do not have to register an asynchronous *complete* callback function. This is because of the synchronous nature of the path. For more info about the differences between Regular, Direct, and Synchronous OIDs in general, see Synchronous OID Request Interface in NDIS 6.80.

ⓘ **Note**

NDIS 6.80 supports specific OIDs for use with the Synchronous OID request interface. OIDs that existed before NDIS 6.80 and some NDIS 6.80 OIDs are not supported. To determine if an OID can be used in the Synchronous OID request interface, see the OID reference page.

To support the Synchronous OID request interface, use the documentation for the standard OID request interface. The following table shows the relationship between the functions in the Synchronous OID request interface and the standard OID request interface.

Synchronous OID function	Standard OID function
MiniportSynchronousOidRequest	*MiniportOidRequest*

Miniport Adapter Status Indications

Article • 12/15/2021

Miniport drivers call the NdisMIndicateStatusEx function to report a change in the status of a miniport adapter. The miniport driver passes **NdisMIndicateStatusEx** a pointer to an NDIS_STATUS_INDICATION structure that contains the status information.

The status indication includes information to identify the type of status and a reason for the status change.

The miniport driver should set the **SourceHandle** member to the handle that NDIS passed to the *MiniportAdapterHandle* parameter of the *MiniportInitializeEx* function. If the status indication is associated with an OID request, the miniport driver can set the **DestinationHandle** and **RequestId** members so that NDIS can provide the status indication to a specific protocol binding.

Miniport Adapter Device PnP Event Notifications

Article • 12/15/2021

NDIS calls a miniport driver's *MiniportDevicePnPEventNotify* function to notify the driver of Plug and Play (PnP) events.

NDIS provides an event code that describes the PnP event. The code can indicate that the adapter has been unexpectedly removed from the system or that the power profile of the host system has changed.

If the event code indicates that the power profile has changed, NDIS also indicates the type of change. Either the system is running on battery power or the system is running on AC power.

The miniport driver should adjust the adapter settings accordingly.

Miniport Adapter Check-for-Hang and Reset Operations

Article • 12/15/2021

Overview

> ⚠ **Warning**
>
> Check-for-Hang (CFH) and Reset operations are discouraged for all NDIS 6.83 and later drivers. For more information, see **Check-for-Hang and Reset operations in NDIS 6.83 and later**.

NDIS calls an NDIS miniport driver's *MiniportCheckForHangEx* function to check the operational state of an NDIS adapter that represents a network interface card (NIC). *MiniportCheckForHangEx* checks the internal state of the adapter and returns **TRUE** if it detects that the adapter is not operating correctly.

By default, NDIS calls *MiniportCheckForHangEx* approximately every 2 seconds. If *MiniportCheckForHangEx* returns **TRUE**, NDIS calls the NDIS miniport driver's *MiniportResetEx* function. If the default time-out value of 2 seconds is too small, your miniport driver can set a different value at initialization time as follows:

1. Set the **CheckForHangTimeInSeconds** member of the **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure to a nonzero value.
2. Pass the **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure in the *MiniportAttributes* parameter of the **NdisMSetMiniportAttributes** function.

For more information about setting driver attributes, see Initializing an Adapter. The value of **CheckForHangTimeInSeconds** should be greater than the initialize time of your miniport driver. However, if your driver takes longer than **CheckForHangTimeInSeconds** seconds to initialize, this time-out expires, causing NDIS to call your driver's *MiniportCheckForHangEx* function. If *MiniportCheckForHangEx* returns **TRUE**, NDIS will then call your driver's *MiniportResetEx* function. For this reason, you should synchronize your driver's *MiniportCheckForHangEx* function with driver initialization so that *MiniportCheckForHangEx* will not return **TRUE** if the driver has not finished initializing.

If your miniport driver does not complete an OID request within two successive calls to *MiniportCheckForHangEx*, NDIS can call the driver's *MiniportResetEx* function. For some

OID requests, NDIS calls *MiniportResetEx* if the driver does not complete the request within four successive calls to *MiniportCheckForHangEx*.

The reset operation does not affect miniport adapter operational states. Also, the state of the adapter might change while a reset operation is in progress. For example, NDIS might call a driver's *MiniportPause* function when there is a reset operation in progress. In this case, the driver can complete either the reset or the pause operation in any order while following the normal requirements for each operation.

For a reset operation, the driver can fail transmit request packets or it can keep them queued and complete them later. However, you should note that an overlying driver cannot complete a pause operation while its transmit packets are pending.

A miniport driver can complete a reset request synchronously by returning a success or failure status. The driver can complete a reset request asynchronously by returning **NDIS_STATUS_PENDING**. In this case, the driver must call **NdisMResetComplete** to complete the operation.

Check-for-Hang and Reset operations in NDIS 6.83 and later

In versions of NDIS before 6.83, Check-for-Hang (CFH) and Reset operations were discouraged for Always On, Always Connected (AOAC) systems due to battery life issues. However, drivers could still use CFH on other non-AOAC Windows systems by implementing the optional *MiniportCheckForHangEx* and *MiniportResetEx* callback functions.

Starting in NDIS 6.83, these callback functions are discouraged on **all** Windows systems regardless of power capabilities. Although it is not a logo test violation to use CFH in NDIS 6.83 and later, NDIS drivers should use the following table for guidance about its usage.

Caller	Recommendation	Notes
Drivers targeting AOAC systems	Must not implement	Causes battery life issues due to periodic check-for-hang activity
Drivers targeting Windows Server systems	Must not implement	Causes issues when the CPU is stressed

Caller	Recommendation	Notes
Virtual (software-only) miniport drivers	Must not implement	Reset not possible without hardware
Other new NDIS 6.83 and later drivers	Should not implement	
Other existing NDIS 6.82 and earlier code	Not required to change, but should consider removing Check-for-Hang and Reset in future rework	

Related topics

Miniport Driver Hardware Reset

Miniport Driver Reset and Halt Functions

Miniport Adapter Shutdown

Article • 12/15/2021

An NDIS miniport driver must register a *MiniportShutdownEx* function during miniport driver initialization.

NDIS calls an NDIS miniport driver's *MiniportShutdownEx* function when the system is shutting down. *MiniportShutdownEx* restores the hardware to a known state.

The *ShutdownAction* parameter that NDIS passed to *MiniportShutdownEx* informs the miniport driver of the reason for the shutdown.

The shutdown handler can be called as a result of a user operation, in which case it runs at IRQL = PASSIVE_LEVEL. It can also be called as a result of an unrecoverable system error, in which case it can be running at any IRQL.

MiniportShutdownEx should call no **Ndis*Xxx*** functions. The miniport driver can call functions for reading and writing I/O ports or disabling the DMA engine to return the hardware to a known state.

Unlike *MiniportHaltEx*, *MiniportShutdownEx* should not release any allocated resources. *MiniportShutdownEx* should just stop the NIC.

Related topics

Adapter States of a Miniport Driver

Halting a Miniport Adapter

Miniport Adapter States and Operations

Writing NDIS Miniport Drivers

NDIS Management Information and OIDs

Article • 03/14/2023

Each miniport driver contains its own *management information base (MIB)*, which is an information block in which the driver stores dynamic configuration information and statistical information that a management entity can query or set. An Ethernet multicast address list is an example of configuration information. The number of broadcast packets received is an example of statistical information. Each information element within the MIB is referred to as an *object*. To refer to each such managed object, NDIS defines an *object identifier (OID)*. Therefore, if a management entity wants to query or set a particular managed object, it must provide the specific OID for that object.

The MIB tracks three classes of objects:

- Objects that are general to all NDIS miniport drivers.
- Objects that are specific to all NDIS miniport drivers for a given medium type, such as Ethernet.
- Objects that are specific to a particular vendor implementation.

The *general* and mandatory *media-specific* OIDs are documented in the Network Reference section of the WDK documentation. The implementation-specific OIDs for a particular network interface card (NIC) driver should be listed and described in the documentation that accompanies a given miniport driver.

Objects are classified as *operational characteristics* (for example, multicast address list) or *statistics* (for example, broadcast packets received), and they are also classified as *mandatory* or *optional*. All operational characteristics objects for general or media-specific classes are mandatory, but only some statistics objects are mandatory. All implementation-specific objects are classified as mandatory.

For more information about OID classifications, see NDIS OIDs.

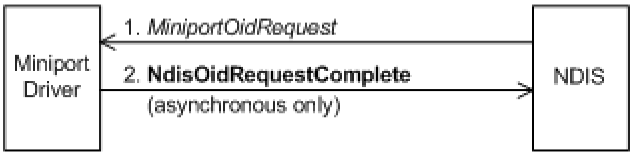
Querying a Connectionless Miniport Driver

Article • 12/15/2021

To query OIDs that a connectionless miniport driver maintains, a bound protocol calls NdisOidRequest and passes an NDIS_OID_REQUEST structure that specifies the object (OID) that is being queried and that points to a buffer into which NDIS eventually writes the requested information.

If NDIS does not respond for the miniport driver, the call to **NdisOidRequest** causes NDIS to call the miniport driver's *MiniportOidRequest* function, which returns the requested information to NDIS. *MiniportOidRequest* can complete synchronously or asynchronously with a call to NdisMOidRequestComplete.

NDIS can also call a miniport driver's *MiniportOidRequest* function on its own behalf--for example, after the miniport driver's *MiniportInitializeEx* function has returned NDIS_STATUS_SUCCESS--to query the miniport driver's capabilities, status, or statistics. The following diagram illustrates querying a connectionless miniport driver.



NDIS responds to many OID requests on behalf of the miniport driver. The miniport driver reports many of its OID values during initialization and in status indications. For more information about OID values that are reported in attributes, see NDIS_MINIPORT_ADAPTER_ATTRIBUTES and the related attributes structures.

When *MiniportOidRequest* is called with OID_GEN_MAC_OPTIONS, it must return a bitmask that specifies the optional operations that the miniport driver performs. The set of flags includes:

- NDIS_MAC_OPTION_COPY_LOOKAHEAD_DATA. This flag indicates to a protocol driver that it can access indicated data by any means. If a miniport driver indicates data out of on-board shared memory, it must not set this flag.

- NDIS_MAC_OPTION_NO_LOOPBACK. If this flag is set, the miniport driver does not loopback a packet that is passed to *MiniportSendNetBufferLists(Packets)* that is directed to a receiver on the same computer and that the miniport driver expects NDIS to perform the loopback. If NDIS performs the loopback of a packet, the packet is not passed down to the miniport driver. A miniport driver always sets this flag unless a NIC performs hardware loopbacks.

- NDIS_MAC_OPTION_RECEIVE_SERIALIZED. If this flag is set, the miniport driver does not indicate any newly received packet up until the previously received packet has been fully processed, including transferring the data. Most miniport drivers, except those that indicate up packets by calling **NdisMIndicateReceiveNetBufferLists**, set this flag.

A miniport driver must never use the flag NDIS_MAC_OPTION_RESERVED, which is reserved for NDIS internal use.

MiniportOidRequest is also queried with a media-specific OID to determine the NIC's current address. For instance, the miniport driver for a NIC of type 802.3 will be queried with OID_802_3_CURRENT_ADDRESS.

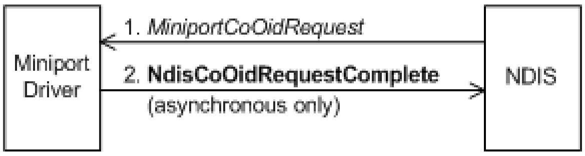
The miniport drivers for certain media types will receive additional OIDs that are media-specific. For example, a miniport driver whose NIC is of type 802.3 is queried with OID_802.3_MAXIMUM_LIST_SIZE. For more information, see General Objects.

Querying a Connection-Oriented Miniport Driver

Article • 12/15/2021

To query information objects that a connection-oriented miniport driver maintains, a bound protocol calls **NdisCoOidRequest** and passes an **NDIS_OID_REQUEST** structure that specifies the object (OID) that is being queried and that provides a buffer into which NDIS eventually writes the requested information. The call to **NdisCoOidRequest** causes NDIS to call the miniport driver's *MiniportCoOidRequest* function, which returns the requested information to NDIS. *MiniportCoOidRequest* can complete synchronously or asynchronously with a call to **NdisCoOidRequestComplete**.

NDIS can also call a miniport driver's *MiniportCoOidRequest* function on its own behalf—for example, after the miniport driver's *MiniportInitializeEx* function has returned NDIS_STATUS_SUCCESS—to query the miniport driver's capabilities, status, or statistics. The following diagram illustrates querying a connection-oriented miniport driver.



A connection-oriented miniport driver must be able to provide information about a global basis for all virtual connections (VCs) for a particular NIC and also on a per-VC basis. For example, if a non-**NULL** *NdisVcHandle* is supplied to *MiniportCoOidRequest* for a query of OID_GEN_CO_RCV_CRC_ERROR, the miniport driver returns the number of CRC errors that were encountered in all receives on the specified VC. For the same request with a **NULL** *NdisVcHandle*, the miniport driver returns the total number of CRC errors that are encountered for all incoming receives through a NIC.

The following list contains the set of mandatory general operational OIDs for connection-oriented miniport drivers:

OID_GEN_CO_SUPPORTED_LIST

OID_GEN_CO_HARDWARE_STATUS

OID_GEN_CO_MEDIA_SUPPORTED

OID_GEN_CO_MEDIA_IN_USE

OID_GEN_CO_LINK_SPEED

OID_GEN_CO_VENDOR_ID

OID_GEN_CO_VENDOR_DESCRIPTION

OID_GEN_CO_VENDOR_DRIVER_VERSION

OID_GEN_CO_DRIVER_VERSION

OID_GEN_CO_MAC_OPTIONS

OID_GEN_CO_MEDIA_CONNECT_STATUS

OID_GEN_CO_MINIMUM_LINK_SPEED

The miniport driver's *MiniportCoOidRequest* function must be prepared to respond to queries or sets, as appropriate, to any of the preceding OIDs.

When *MiniportCoOidRequest* is called with OID_GEN_CO_MAC_OPTIONS, it must return a bitmask that specifies the optional operations that the miniport driver performs. The set of flags includes:

- NDIS_MAC_OPTION_NO_LOOPBACK. If this flag is set, the miniport driver does not loopback a packet that is passed to **MiniportCoSendNetBufferLists** that is directed to a receiver on the same computer and that the miniport driver expects NDIS to perform the loopback. If NDIS performs the loopback of a packet, the packet is not passed down to the miniport driver. A miniport driver always sets this flag unless a NIC performs hardware loopbacks.

- NDIS_MAC_ETOX_INDICATION. If this flag is set, the miniport driver indicates that a send is complete only after the NIC transmits the packet.

A miniport driver must never use the NDIS_MAC_OPTION_RESERVED flag, which is reserved for NDIS internal use.

MiniportCoOidRequest will also be queried with a media-specific OID to determine the NIC's current address.

For more information, see OIDs for Connection-Oriented Call Managers and Clients and General Objects.

Querying a Miniport Driver Directly From User Mode

Article • 12/15/2021

An application can use **IOCTL_NDIS_QUERY_GLOBAL_STATS** to directly query information from a miniport driver's NIC. In this operation, the application can use any query OID that the miniport driver supports.

Querying 64-Bit Statistics OIDs

Article • 12/15/2021

All miniport drivers that are 1 Gigabyte per second (Gbps) and faster must support 64-bit counters for certain statistics OIDs. All 100 Megabytes per second (Mbps) and faster miniport drivers should support 64-bit counters for such OIDs. For more information about statistics OIDs for connectionless miniport drivers, see General Statistics. For more information about such OIDs for connection-oriented miniport drivers, see General Statistics for Connection-Oriented Miniport Drivers.

A requester that queries a statistics OID sets NDIS_OID_REQUEST **InformationBufferLength** to 4 (bytes) to indicate a 32-bit statistics request or to 8 (bytes) to indicate a 64-bit statistics request. In its response, the miniport driver sets NDIS_OID_REQUEST **BytesNeeded** to the size of the statistics value that the miniport driver supports (4 for 32-bits or 8 for 64-bits). The miniport driver sets NDIS_OID_REQUEST **BytesWritten** to the smaller of the **InformationBufferLength** value and the size of statistics that the miniport driver supports.

The following sections describe how a miniport driver that supports 64-bit statistics OIDs responds to queries of such OIDs.

64-bit query of a 64-bit value

NDIS_OID_REQUEST **InformationBufferLength** is greater than or equal to 8.

The miniport driver:

- Returns the 64-bit value in the information buffer.
- Sets NDIS_OID_REQUEST **BytesWritten** to 8.
- Returns NDIS_STATUS_SUCCESS from its *MiniportOidRequest* or **MiniportCoOidRequest** function.

32-bit query of a 64-bit value

NDIS_OID_REQUEST **InformationBufferLength** is greater to or equal to 4 and less than 8.

The miniport driver:

- Returns, in the information buffer, the lower 32 bits of the 64-bit value.

- Sets NDIS_OID_REQUEST **BytesWritten** to 4.
- Sets NDIS_OID_REQUEST **BytesNeeded** to 8.
- Returns NDIS_STATUS_SUCCESS from its *MiniportOidRequest* or **MiniportCoOidRequest** function.

Invalid-length query of a 64-bit value

NDIS_OID_REQUEST **InformationBufferLength** is less than 4.

The miniport driver:

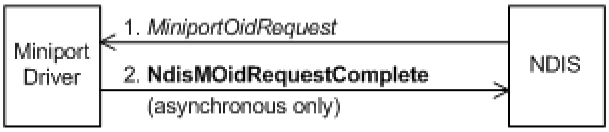
- Does not return any value in the information buffer.
- Sets NDIS_OID_REQUEST **BytesWritten** to 0.
- Sets NDIS_OID_REQUEST **BytesNeeded** to 8.
- Returns NDIS_STATUS_INVALID_LENGTH from its *MiniportOidRequest* or **MiniportCoOidRequest** function.

Setting Information for a Connectionless Miniport Driver

Article • 12/15/2021

To set an OID that a connectionless miniport driver maintains, a bound protocol calls NdisOidRequest and passes an NDIS_OID_REQUEST structure that specifies the object (OID) that is being queried and that points to a buffer that contains the value to which the object should be set. The call to **NdisOidRequest** causes NDIS to call the miniport driver's *MiniportOidRequest* function, which sets the object with the supplied value.

The call to *MiniportOidRequest* can complete synchronously or asynchronously. To complete the call asynchronously, the miniport driver calls NdisMOidRequestComplete. The following diagram illustrates setting information in a connectionless miniport driver (per binding).

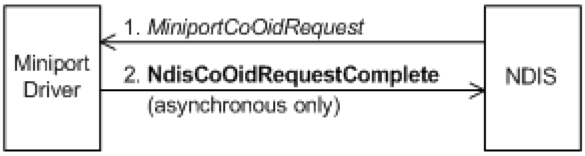


Setting Information for a Connection-Oriented Miniport Driver

Article • 12/15/2021

To set an OID that a connection-oriented miniport driver maintains, a bound protocol calls NdisCoOidRequest and passes an NDIS_OID_REQUEST structure that specifies the object (OID) that is being queried and that points to a buffer that contains the value to which the object should be set. The call to **NdisCoOidRequest** causes NDIS to call the miniport driver's MiniportCoOidRequest function, which sets the object with the supplied value.

The call to **NdisCoOidRequest** can complete synchronously or asynchronously. To complete the call asynchronously, a miniport driver calls NdisCoOidRequestComplete. The following diagram illustrates setting information in a connection-oriented miniport driver.



Occasions for Setting Miniport Driver Information

Article • 12/15/2021

The *MiniportOidRequest* function in a connectionless miniport driver and the **MiniportCoOidRequest** function in a connection-oriented miniport driver are called during initialization. These functions can also be called:

- During a hardware reset,
- If a protocol calls **NdisCloseAdapterEx**.

MiniportOidRequest or *MiniportCoOidRequest* is called during hardware reset operation. In this case, *MiniportOidRequest* or *MiniportCoOidRequest* is called to reset the miniport driver to its initial state with respect to its addresses.

NDIS calls *MiniportOidRequest* or *MiniportCoOidRequest* when a miniport driver's NIC is closed by a protocol's **NdisCloseAdapterEx** call. Such a miniport driver will be requested to update its addressing information.

Reporting Hardware Status

Article • 12/15/2021

A connectionless miniport driver indicates changes in hardware status to upper layers by calling NdisMIndicateStatusEx. A connection-oriented miniport driver indicates changes by calling NdisMCoIndicateStatusEx.

NdisM(Co)IndicateStatusEx takes both a general status code and a buffer that contains media-specific information that further defines the reason for the status change. NDIS reports this status change to bound protocol drivers. NDIS does not interpret or otherwise intercept the status code.

The miniport driver can make one or more such calls. However, unlike earlier versions of NDIS, the miniport driver does not indicate that it has finished sending status. The protocol driver or configuration manager can log the status or take corrective action, as appropriate.

NdisMCoIndicateStatusEx takes any valid NDIS_STATUS_*Xxx* value.

The miniport driver is responsible for indicating status codes that make sense to a protocol or higher level driver. A protocol driver ignores any status values in which it is not interested or that do not make sense in the context of its operations.

A miniport driver cannot indicate status in the context of its *MiniportInitializeEx*, *MiniportInterrupt*, *MiniportHaltEx*, or *MiniportShutdownEx* function.

A miniport driver can also be interrogated by an upper layer driver or by NDIS about the miniport driver's hardware status. When the *MiniportOidRequest* function of a connectionless miniport driver or the MiniportCoOidRequest function of a connection-oriented miniport driver receives OID_GEN_HARDWARE_STATUS, it responds with any of the applicable status values that are defined in NDIS_HARDWARE_STATUS. These status values include:

- NdisHardwareStatusReady
- NdisHardwareStatusInitializing
- NdisHardwareStatusReset
- NdisHardwareStatusClosing
- NdisHardwareStatusNotReady

The miniport driver can be queried so that NDIS can synchronize operations between layers of NDIS drivers--for example, by determining whether a NIC is ready to accept packets.

Indicating Connection Status

Article • 12/15/2021

A miniport driver calls NdisMIndicateStatusEx or NdisMCoIndicateStatusEx to indicate a change in the media connection status. The miniport driver passes one of the following status indications to **NdisM(Co)IndicateStatus**:

NDIS_STATUS_MEDIA_CONNECT
Indicates a media connection status change from disconnected to connected. A media connect status change occurs when a disconnected adapter makes a network connection. For example, the adapter connects when it comes within range (for a wireless adapter) or the user connects the network cable.

NDIS_STATUS_MEDIA_DISCONNECT
Indicates a media connection status change from connected to disconnected. A media disconnect status change occurs when a connected adapter loses a network connection. For example, the adapter loses the connection because it is out of range (for a wireless adapter) or the user unplugs the network cable.

Unless specified otherwise, miniport drivers should indicate media connection status changes within two seconds after detecting the status change.

A miniport driver can check the media connection status while performing certain operations (see the following list). If the status is the same after the operation is complete as it was before the operation started,the miniport driver does not have to report any status changes that might have occurred during the operation.

The following list describes additional requirements for indicating media connection status changes for miniport drivers:

Resetting
NDIS calls *MiniportResetEx* to reset a miniport driver. The miniport driver can complete the reset either synchronously or asynchronously.

If the media connection status is different after resetting, the driver should indicate the status within two seconds after completing the reset.

A miniport driver should not complete the reset operation until it has determined the media connection status.

Halting
A miniport driver must not indicate any media connection status changes when NDIS calls *MiniportHaltEx*.

Initializing
NDIS calls a miniport driver's *MiniportInitializeEx* function to initialize an adapter. During the adapter initialization, the miniport driver must follow these guidelines:

- If the miniport driver does not indicate the media connection status after returning from *MiniportInitializeEx*, NDIS uses the value of the **MediaConnectState** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure to determine the media connection status. The miniport driver provides NDIS with this structure when the driver calls **NdisMSetMiniportAttributes** from its *MiniportInitializeEx* function.

  Note If the **MediaConnectState** member is set to MediaConnectStateUnknown, NDIS will proceed as if the adapter is disconnected.

- If an adapter is connected after NDIS calls *MiniportInitializeEx*, the miniport driver can indicate NDIS_STATUS_MEDIA_CONNECT within 5 seconds after it returns from *MiniportInitializeEx*.

- If an adapter is disconnected after NDIS calls *MiniportInitializeEx*, the miniport driver should indicate NDIS_STATUS_MEDIA_DISCONNECT within 2 seconds after it returns from *MiniportInitializeEx*.

- While initializing, the miniport driver should process OID_GEN_MEDIA_CONNECT_STATUS or OID_GEN_CO_MEDIA_CONNECT_STATUS requests asynchronously. The miniport driver should not complete such requests until after it has determined the connection status.

- Determination of the media connection status should not delay initialization. If necessary, the miniport driver should initiate the process to determine the connection status within *MiniportInitializeEx*, and complete the process at a later time. For example, the miniport driver could set a timer to poll the adapter for the connection status.

- A deserialized miniport driver can indicate a media disconnect during initialization, but a serialized miniport driver should not.

Sleeping
A miniport driver enters a network sleep state when it receives an OID_PNP_SET_POWER request to set a device power state of D1, D2, or D3.

A miniport driver must not indicate any media connection status changes when it enters a sleep state or while it is in a sleeping state.

Waking
A miniport driver wakes from a sleep state when it receives an OID_PNP_SET_POWER request to set the device power state to D0.

If the adapter's media connection status after waking is the same as the status was prior to sleeping, the miniport driver should not indicate a media connection status change. If the connection status changed, the miniport driver should indicate the new connection status within two seconds after waking.

While waking, the miniport driver should process OID_GEN_MEDIA_CONNECT_STATUS or OID_GEN_CO_MEDIA_CONNECT_STATUS requests asynchronously. The miniport driver should not complete such requests until after it has determined the connection status.

NDIS Support for WMI

Article • 03/14/2023

Through NDIS, clients of Windows Management Instrumentation (WMI) can obtain and set information that NDIS and NDIS drivers service. WMI clients can also register to receive status updates.

NDIS automatically registers miniport adapters, named virtual connections (VCs), and a set of globally unique identifiers (GUIDs) for each miniport adapter with WMI. For more information about these GUIDs, see Standard Miniport Driver OIDs Registered with WMI. Miniport drivers can also provide support for custom object identifiers (OIDs) and custom status indications, as the Customized OIDs and Status Indications topic describes.

NDIS does not provide WMI support for protocol drivers. A protocol driver, or an intermediate driver, can create a device object for itself and register directly with WMI. For more information about registering directly with WMI, see Registering as a WMI Data Provider.

For more information about the WMI architecture, see Windows Management Instrumentation.

This section includes:

Registration and Deregistration of NDIS Miniport Drivers with WMI

Mapping of GUIDs to OIDs and Miniport Driver Status

Support for Named VCs

NDIS-Supported WMI Operations

Standard WMI OIDs and Status Indications

Customized OIDs and Status Indications

NDIS WMI GUIDs

Registration and Deregistration of NDIS Miniport Drivers with WMI

Article • 03/14/2023

NDIS automatically registers each miniport adapter with WMI. A miniport driver does not have to explicitly register with WMI, because NDIS automatically registers for the associated miniport adapter after the miniport driver returns from the *MiniportInitializeEx* function.

After NDIS registers a miniport adapter as a data provider with WMI, WMI clients can send it query and set requests and register to receive status indications.

Before NDIS calls the miniport driver's *MiniportHaltEx* function, NDIS automatically deregisters the miniport adapter with WMI so that WMI will no longer send WMI requests to the miniport driver.

For each miniport adapter that NDIS registers with WMI, NDIS registers GUIDs that correspond to particular OIDs or status indications. NDIS registers GUIDs for a miniport adapter's supported set of standard OIDs and status indications. For more information about these standard GUIDs, see Standard Miniport Driver OIDs Registered with WMI and Standard Miniport Driver Status Registered with WMI.

NDIS can also register custom GUIDs for custom OIDs and status indications. If the miniport driver supports custom OIDs, it must provide the associated custom GUIDs. For more information about customized OIDs and status indications, see Customized OIDs and Status Indications.

For connection-oriented miniport drivers, NDIS also registers any named virtual connections (VCs). WMI clients can work only with VCs that a stand-alone call manager, or connection-oriented client, has named with the **NdisCoAssignInstanceName** function. For more information about NDIS WMI support for named VCs, see Support for Named VCs.

Mapping of GUIDs to OIDs and Miniport Driver Status

Article • 12/15/2021

When WMI sends a WMI request to a miniport adapter (that is, when WMI sends an I/O request packet [IRP] to a functional device object that NDIS created), NDIS intercepts the request. NDIS does not forward the request to the miniport driver if NDIS already has the information that it requires to service the request. Otherwise, NDIS maps the WMI GUID to an OID and then queries or sets the OID.

If the miniport driver is a connectionless miniport driver, NDIS can call the miniport driver's *MiniportOidRequest* function to handle the OID request. If the miniport driver is a connection-oriented miniport driver, NDIS can call the miniport driver's **MiniportCoOidRequest** function to handle the OID request. NDIS returns the results of the query or set request to WMI.

Miniport drivers generate status indications with the **NdisMIndicateStatusEx** or **NdisMCoIndicateStatusEx** function. If a WMI client registers for a WMI event and a miniport driver generates an associated status indication, NDIS maps that status indication to a WMI GUID and passes a WMI event indication to WMI. WMI then passes the WMI event indication to all of the WMI clients that have registered for the WMI event.

Support for Named VCs

Article • 12/15/2021

NDIS allows WMI clients to query and set information on a per-virtual connection (VC) basis for connection-oriented miniport adapters. WMI clients can also enumerate VCs. Before a WMI client can query or set information that is associated with a particular VC, a stand-alone call manager or connection-oriented client must name the VC by calling the **NdisCoAssignInstanceName** function.

After a stand-alone call manager or connection-oriented client initiates the setup of a VC by calling the **NdisCoCreateVC** function, the stand-alone call manager or connection-oriented client can name the VC with **NdisCoAssignInstanceName**. NDIS assigns the VC an instance name and registers the instance name with WMI. WMI clients can then enumerate the VC and query or set OIDs that are relative to the VC.

A miniport call manager (MCM) cannot use **NdisCoAssignInstanceName** to name its VCs. Instead, an MCM should create a custom GUID and OID for the VC and register the GUID-to-OID mapping with NDIS. For more information about registering custom OIDs, see Customized OIDs and Status Indications.

NDIS-Supported WMI Operations

Article • 03/14/2023

NDIS supports the following WMI operations:

- Enumerate adapter and enumerate virtual connection (VC).

  NDIS registers global GUIDs (GUID_NDIS_ENUMERATE_ADAPTER_EX and GUID_NDIS_ENUMERATE_VC) with WMI that enable WMI clients to enumerate all miniport adapters (that is, miniport driver instances) and all named VCs. Because NDIS track all of the loaded miniport drivers and all of the named VCs, NDIS does not query miniport drivers for such information.

- QUERY SINGLE INSTANCE and SET SINGLE INSTANCE

  Through NDIS, a WMI client can query or set a single instance of a data block, which corresponds to a single OID. For a query, NDIS returns all of the information that is associated with an adapter or VC. A WMI client cannot query or set a data item that is within an OID. For example, a query of the GUID_NDIS_GEN_CO_LINK_SPEED GUID returns both the outbound and inbound speed. A WMI client cannot query only the outbound speed or only the inbound speed.

- QUERY ALL DATA

  NDIS satisfies a QUERY ALL DATA request on a particular GUID by obtaining the appropriate data and returning the combined data for all of the instances of the GUID to WMI. For example, in response to a QUERY ALL DATA request on GUID_NDIS_ENUMERATE_ADAPTER_EX, NDIS returns a list of all of the loaded miniport drivers to WMI. For a QUERY ALL DATA on the GUID that maps to OID_GEN_CO_XMIT_PDUS_OK, NDIS queries that OID for each VC on each connection-oriented miniport driver and returns the combined data to WMI. Because the overhead for a QUERY ALL DATA request might be very high, WMI clients should use a QUERY ALL DATA request only to enumerate adapters and VCs. After determining the adapter or VC interest, the client can then query individual GUID instances.

- EVENT NOTIFICATION

  WMI clients can register with NDIS to be notified for a particular status indication. When such a status indication occurs, NDIS passes the status information with the appropriate GUID to WMI for delivery to the clients as a WMI event.

- EXECUTE METHOD

  Through NDIS, a WMI client can run a method that is associated with a data block, which corresponds to a single OID. WMI clients provide the information that NDIS requires to run the method. Method requests can be associated with miniport adapters, NDIS ports, or VCs. NDIS returns the resulting information after the method is successfully run.

Standard Miniport Driver OIDs Registered with WMI

Article • 12/15/2021

NDIS registers WMI GUIDs with WMI for miniport adapters. To obtain the list of OIDs that a miniport adapter supports, NDIS issues an OID_GEN_SUPPORTED_LIST query to the associated miniport driver. The miniport driver must provide the list of all of the OIDs that the miniport adapter supports. This list must contain all of the mandatory OIDs and should contain optional and custom OIDs, if any.

NDIS maps the supported OIDs to WMI GUIDs and registers the GUIDs with WMI. NDIS translates WMI GUID requests to OID requests, if necessary, for the registered OIDs.

NDIS drivers can also register custom GUIDs with WMI. For more information about custom GUIDs, see Customized OIDs and Status Indications.

NDIS also translates status indications to WMI events. For more information about translating status indications to WMI events, see Standard Miniport Driver Status Registered with WMI.

Standard Miniport Driver Status Indications Registered with WMI

Article • 12/15/2021

NDIS automatically registers GUIDs with WMI for the NDIS status indications that miniport drivers indicate with the **NdisMIndicateStatusEx** or **NdisMCoIndicateStatusEx** function. For a list of general status indications, see Status Indications.

If a WMI client registers with WMI to receive an NDIS WMI event, NDIS translates the corresponding NDIS status indication to the WMI event and reports the event to all of the WMI clients that registered for the event.

NDIS drivers can also generate custom status indications. For more information about custom status indications and WMI, see Customized OIDs and Status Indications.

Customized OIDs and Status Indications

Article • 12/15/2021

You can create a custom OID that NDIS maps to a custom GUID that you create. NDIS registers the custom GUID with WMI for the miniport driver so that WMI clients can query or set the associated information.

To provide a custom status indication, NDIS miniport drivers must use the NDIS_STATUS_MEDIA_SPECIFIC_INDICATION_EX status indication. The WMI clients must use the data that is included with the WMI event to identify the custom event. NDIS does not register custom GUIDs for status indications.

To obtain a miniport adapter's custom OIDs and the associated WMI GUIDs, NDIS issues OID requests to the miniport driver after the miniport driver has completed initialization. NDIS issues an OID_GEN_SUPPORTED_LIST query to obtain the list of the OIDs that the miniport driver supports. The miniport driver includes both custom OIDs and standard OIDs in its response. To obtain the GUIDs that are associated with the custom OIDs, NDIS issues an OID_GEN_SUPPORTED_GUIDS query to connectionless miniport drivers or an OID_GEN_CO_SUPPORTED_GUIDS query to connection-oriented miniport drivers.

The query to OID_GEN_SUPPORTED_GUIDS or OID_GEN_CO_SUPPORTED_GUIDS returns an array of NDIS_GUID structures to NDIS. Each NDIS_GUID structure maps a custom GUID to a custom OID.

To support custom OIDs and status indications, you must fill in NDIS_GUID structures. You must also create a managed object format (MOF) file that describes the GUID and build this file with the miniport driver.

This section includes:

Filling in an NDIS_GUID Structure

Including a MOF File

Filling in an NDIS_GUID Structure

Article • 03/14/2023

An NDIS_GUID structure is defined as follows:

```C++
typedef struct _NDIS_GUID {
  GUID  Guid;
  union {
    NDIS_OID  Oid;
    NDIS_STATUS  Status;
  };
  ULONG  Size;
  ULONG  Flags;
} NDIS_GUID, *PNDIS_GUID;
```

To obtain a GUID for the **Guid** member of the structure, you can run the Uuidgen.exe application. For more information about this application, see Generating Interface UUIDs.

The **Oid** or **Status** member is a ULONG that is an OID code. NDIS 6.0 does not map custom status indications to WMI GUIDs.

If the NDIS_GUID structure maps an OID that returns an array of data items, the **Size** member specifies the size, in bytes, of each data item in the array. If the data is not an array, the **Size** member specifies the size of the data. If the size of the data items is variable, or if the OID does not return data, the **Size** member must be -1.

A bitwise OR of the following values for the **Flags** member indicates the type of data that is associated with the GUID:

fNDIS_GUID_TO_OID
When this flag is set, the NDIS_GUID structure maps a GUID to an OID.

fNDIS_GUID_TO_STATUS
Reserved for NDIS. Miniport drivers should not use this flag.

fNDIS_GUID_ANSI_STRING
When this flag is set, a null-terminated ANSI string is supplied for the GUID.

fNDIS_GUID_UNICODE_STRING
When this flag is set, a Unicode string is supplied for the GUID.

fNDIS_GUID_ARRAY
When this flag is set, an array of data items is supplied for the GUID. The specified **Size** value indicates the length of each data item in the array.

fNDIS_GUID_ALLOW_READ
When this flag is set, all users are allowed to use this GUID to obtain information.

fNDIS_GUID_ALLOW_WRITE
When this flag is set, all users are allowed to use this GUID to set information.

Note By default, custom WMI GUIDs that a miniport driver supplies are accessible only to users with administrator privileges. A user with administrator privileges can always read or write to a custom GUID if the miniport driver supports the read or write operation for that GUID. You can set the fNDIS_GUID_ALLOW_READ and fNDIS_GUID_ALLOW_WRITE flags to allow all users to access a custom GUID.

Note that for all custom GUIDs that a driver registers, the driver must set fNDIS_GUID_TO_OID. Miniport drivers should never set fNDIS_GUID_TO_STATUS. All of the other flags can be combined by using a bitwise OR operation.

Including a MOF File

Article • 12/15/2021

You must include a description of all of the custom GUIDs that map to a miniport driver's custom OIDs in a managed object format (MOF) file that must be compiled and included in the miniport driver's resource (*.rc) file.

The MOF source file must be of type MOFDATA and must have an extension of .mof. You must compile the MOF source file into a binary file with Mofcomp.exe and must check this file with Wmimofck.exe.

You must insert the following line in the miniport driver's resource file (*.rc) to include the MOF binary:

Text

```
NdisMofResource MOFDATA filename.bmf
```

FileName represents the file name of the MOF source file.

GUID_NDIS_STATUS_LINK_STATE

Article • 03/14/2023

The GUID_NDIS_STATUS_LINK_STATE event GUID indicates that there has been a change in the link state of a miniport adapter. This WMI GUID is supported in NDIS 6.0 and later versions.

Miniport drivers use the NDIS_STATUS_LINK_STATE status indication to notify NDIS and overlying drivers that there has been a change in link state.

When a miniport driver indicates a link state change, NDIS translates the status indication to a WMI GUID_NDIS_STATUS_LINK_STATE event for WMI clients.

The data buffer that NDIS provides with the GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by an NDIS_LINK_STATE structure. The **NDIS_LINK_STATE** structure specifies the physical state of the medium.

For more information about link status, see OID_GEN_LINK_STATE and NDIS_STATUS_LINK_STATE.

GUID_NDIS_STATUS_OPER_STATUS

Article • 03/14/2023

The GUID_NDIS_STATUS_OPER_STATUS event GUID indicates the current operational state of an NDIS network interface. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS generates an NDIS_STATUS_OPER_STATUS status indication to report the current operational state of an NDIS network interface.

When NDIS indicates a change in the current operational state of an NDIS network interface, NDIS also translates the status indication to a WMI GUID_NDIS_STATUS_OPER_STATUS event for WMI clients.

The data buffer that NDIS provides with the GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by an NDIS_OPER_STATE structure. For a list of the possible values, see NDIS_STATUS_OPER_STATUS.

GUID_NDIS_STATUS_NETWORK_CHANGE

Article • 03/14/2023

The GUID_NDIS_STATUS_NETWORK_CHANGE event GUID indicates that the layer-three addresses must be renegotiated. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS or a miniport driver can generate an NDIS_STATUS_NETWORK_CHANGE status indication to report a change in network status.

When miniport drivers or NDIS indicate a change in the network status, NDIS translates the status indication to a WMI GUID_NDIS_STATUS_NETWORK_CHANGE event for WMI clients.

The data buffer that NDIS provides with this GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by an NDIS_NETWORK_CHANGE_TYPE-typed value. For a list of the possible values, see NDIS_STATUS_NETWORK_CHANGE.

GUID_NDIS_STATUS_PACKET_FILTER

Article • 03/14/2023

The GUID_NDIS_STATUS_PACKET_FILTER event GUID indicates that there has been a change in the packet filter for miniport adapter. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS generates the NDIS_STATUS_PACKET_FILTER status indication to notify overlying drivers that there might be a change in the packet filter configuration.

NDIS translates the status indication to a WMI GUID_NDIS_STATUS_PACKET_FILTER event for WMI clients.

The data buffer that NDIS provides with the GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by a ULONG value. For more information about packet filter status and the possible values, see OID_GEN_CURRENT_PACKET_FILTER.

GUID_NDIS_STATUS_MEDIA_SPECIFIC_INDICATION_EX

Article • 03/14/2023

The GUID_NDIS_STATUS_MEDIA_SPECIFIC_INDICATION_EX event GUID indicates a media-specific status. This WMI GUID is supported in NDIS 6.0 and later versions.

When a miniport driver indicates media-specific status, NDIS translates the status indication to a WMI GUID_NDIS_STATUS_MEDIA_SPECIFIC_INDICATION_EX indication for WMI clients.

Miniport drivers make media-specific status indications by calling the NdisMIndicateStatusEx function with the **StatusCode** member of the NDIS_STATUS_INDICATION structure that the *StatusIndication* parameter points to set to NDIS_STATUS_MEDIA_SPECIFIC_INDICATION_EX. The **StatusBuffer** member of this structure points to a driver-allocated buffer that contains data in a format that is specific to the status indication that is identified in **StatusCode**.

Depending on the type of media-specific indication, the GUID header could be followed by data that is specific to the media-specific indication. The data buffer that NDIS provides with this GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by the media-specific data, if any.

GUID_NDIS_GEN_LINK_STATE

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_LINK_STATE method GUID to determine the current link state. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS handles this GUID and miniport drivers do not receive an OID query.

When a WMI client issues a GUID_NDIS_GEN_LINK_STATE WMI method request, NDIS returns the current link state for the miniport adapter or NDIS port.

The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID, and the WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure.

The data buffer that NDIS returns with this GUID contains an NDIS_LINK_STATE structure.

Miniport drivers supply the link state during initialization and provide updates with status indications. WMI clients can use the GUID_NDIS_GEN_LINK_STATE GUID to receive updates when the link state changes.

For more information about link status, see OID_GEN_LINK_STATE and NDIS_STATUS_LINK_STATE.

GUID_NDIS_GEN_STATISTICS

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_STATISTICS method GUID to obtain miniport adapter statistics. This WMI GUID is supported in NDIS 6.0 and later versions.

When a WMI client issues a GUID_NDIS_GEN_STATISTICS WMI method request, NDIS returns the current statistics for the miniport adapter or NDIS port. The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID, and the WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure.

NDIS uses the OID_GEN_STATISTICS OID to obtain the statistics for a miniport adapter. This OID is mandatory for miniport drivers that support NDIS 6.0 and later versions. The statistics counters are unsigned 64-bit values. The miniport driver returns the statistics in an NDIS_STATISTICS_INFO structure.

The data buffer that NDIS returns with the GUID contains an NDIS_STATISTICS_INFO structure.

GUID_NDIS_STATUS_PORT_STATE

Article • 03/14/2023

The GUID_NDIS_STATUS_PORT_STATE event GUID indicates a change in the state of an NDIS port. This WMI GUID is supported in NDIS 6.0 and later versions.

Miniport drivers that support NDIS ports use the NDIS_STATUS_PORT_STATE status indication to indicate changes in the state of an NDIS port.

When a miniport driver indicates a port state change, NDIS translates the status indication to a WMI GUID_NDIS_STATUS_PORT_STATE event for WMI clients.

The data buffer that NDIS provides with this GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by an NDIS_PORT_STATE structure.

For more information about the port state, see OID_GEN_PORT_STATE.

GUID_NDIS_GEN_PORT_STATE

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_PORT_STATE method GUID to obtain the state of an NDIS port. This WMI GUID is supported in NDIS 6.0 and later versions.

GUID_NDIS_GEN_PORT_STATE requires a WMI method request to return the state of an NDIS port. The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID, and the WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure.

NDIS handles this GUID, and miniport drivers do not receive an OID query.

The data buffer that NDIS returns with the GUID contains an NDIS_PORT_STATE structure.

For more information about the port state, see OID_GEN_PORT_STATE.

GUID_NDIS_GEN_ENUMERATE_PORTS

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_ENUMERATE_PORTS method GUID to obtain list of the ports that are associated with a miniport adapter. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS handles this request, and miniport drivers do not receive an OID query.

GUID_NDIS_GEN_ENUMERATE_PORTS requires a WMI method request to enumerate the ports. The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID. The WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure that identifies the NDIS network interface name for the miniport adapter in the **NetLuid** member and that specifies zero for the **PortNumber** member. WMI clients can obtain the **NetLuid** value of the adapter with the GUID_NDIS_ENUMERATE_ADAPTERS_EX method GUID.

The data buffer that NDIS returns with the GUID contains an NDIS_PORT_ARRAY structure. The **NumberOfPorts** member of NDIS_PORT_ARRAY contains the number of active ports that are associated with the miniport adapter. The **Ports** member of NDIS_PORT_ARRAY contains a list of pointers to NDIS_PORT_CHARACTERISTICS structures. Each NDIS_PORT_CHARACTERISTICS structure defines the characteristics of a single NDIS port.

For more information about enumerating ports, see OID_GEN_ENUMERATE_PORTS.

GUID_NDIS_GEN_PORT_AUTHENTICATION_PARAMETERS

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_PORT_AUTHENTICATION_PARAMETERS set GUID to set the port authentication parameters for an NDIS port. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS translates this GUID to the OID_GEN_PORT_AUTHENTICATION_PARAMETERS OID to set the current authentication configuration of an NDIS port. Miniport drivers that support NDIS ports must support this OID.

The WMI input buffer specifies an NDIS_WMI_SET_HEADER structure that is followed by an NDIS_PORT_AUTHENTICATION_PARAMETERS structure.

For more information about port parameters, see OID_GEN_PORT_AUTHENTICATION_PARAMETERS.

GUID_NDIS_GEN_LINK_PARAMETERS

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_LINK_PARAMETERS set GUID to set the link parameters for a miniport adapter. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS translates this GUID to the OID_GEN_LINK_PARAMETERS OID to set the current link parameters of a miniport adapter. This OID is mandatory for miniport drivers that support NDIS 6.0 and later versions.

The WMI input buffer specifies the data that NDIS should set. The input buffer contains an NDIS_WMI_SET_HEADER structure that is followed by an NDIS_LINK_PARAMETERS structure.

GUID_NDIS_ENUMERATE_ADAPTERS_EX

Article • 03/14/2023

WMI clients can use the GUID_NDIS_ENUMERATE_ADAPTERS_EX GUID to obtain an enumeration of all of the miniport adapters on the computer. This WMI GUID is supported in NDIS 6.0 and later versions. Because NDIS tracks all of the loaded miniport adapters, NDIS does not query miniport drivers for this information.

WMI clients can use this GUID to find a device name and the associated value in the **NetLuid** member of the NDIS_WMI_ENUM_ADAPTER structure. WMI clients can use the **NetLuid** value of the adapter in subsequent GUID requests.

The data buffer that NDIS returns with the GUID contains array of NDIS_WMI_ENUM_ADAPTER structures.

GUID_NDIS_GEN_INTERRUPT_MODERATION

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_INTERRUPT_MODERATION method GUID to obtain the interrupt moderation parameters that are associated with the specified port of a miniport adapter.

This GUID requires a WMI method request to return the interrupt moderation parameters. The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID, and the WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure.

NDIS translates this GUID to an OID_GEN_INTERRUPT_MODERATION query request for the associated miniport adapter. This OID is mandatory for miniport drivers that support NDIS 6.0 and later versions.

The data buffer that NDIS returns with the GUID contains an NDIS_INTERRUPT_MODERATION_PARAMETERS structure.

For more information about the port state, see OID_GEN_INTERRUPT_MODERATION.

GUID_NDIS_GEN_INTERRUPT_MODERATION_PARAMETERS

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_PORT_PARAMETERS set GUID to set the interrupt moderation configuration for a miniport adapter. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS translates this GUID to the OID_GEN_INTERRUPT_MODERATION OID to set the current configuration. All NDIS miniport drivers must support this OID.

The WMI input buffer contains an NDIS_WMI_SET_HEADER structure that is followed by an NDIS_INTERRUPT_MODERATION_PARAMETERS structure.

For more information about port parameters, see OID_GEN_INTERRUPT_MODERATION.

GUID_NDIS_TCP_OFFLOAD_CAPABILITIES

Article • 03/14/2023

WMI clients can use the GUID_NDIS_TCP_OFFLOAD_CAPABILITIES method GUID to obtain the task offload capabilities that are associated with the specified port of a miniport adapter.

This GUID requires a WMI method request to return the offload capabilities of an NDIS port. The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID, and the WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure.

NDIS handles this GUID, and miniport drivers do not receive an OID query.

The data buffer that NDIS returns with the GUID contains an NDIS_OFFLOAD structure.

For more information about the port state, see OID_TCP_OFFLOAD_CURRENT_CONFIG.

GUID_NDIS_TCP_OFFLOAD_HW_CAPABILITIES

Article • 03/14/2023

WMI clients can use the GUID_NDIS_TCP_OFFLOAD_HW_CAPABILITIES method GUID to obtain the TCP offload capabilities that are supported by the hardware that is associated with the specified port of a miniport adapter.

This GUID requires a WMI method request to return the hardware capabilities of an NDIS port. The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID, and the WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure.

NDIS handles this GUID, and miniport drivers do not receive an OID query.

The data buffer that NDIS returns with the GUID contains an NDIS_OFFLOAD structure.

For more information about the port state, see OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES.

GUID_NDIS_TCP_OFFLOAD_ADMIN_SETTINGS

Article • 03/14/2023

WMI clients can use the GUID_NDIS_TCP_OFFLOAD_ADMIN_SETTINGS set GUID to set the offload configuration parameters for an NDIS port. This WMI GUID is supported in NDIS 6.0 and later versions.

NDIS translates this GUID to the OID_TCP_OFFLOAD_PARAMETERS OID to set the current configuration of an NDIS port. NDIS miniport drivers that provide any kind of support for task offload must support this OID.

The WMI input buffer contains an NDIS_WMI_SET_HEADER structure that is followed by an NDIS_OFFLOAD_PARAMETERS structure.

For more information about port parameters, see OID_TCP_OFFLOAD_PARAMETERS.

GUID_NDIS_STATUS_OFFLOAD_CAPABILITIES_CHANGE

Article • 03/14/2023

The GUID_NDIS_STATUS_OFFLOAD_CAPABILITIES_CHANGE event GUID indicates that there has been a change in the offload characteristics of an NDIS port or miniport adapter. This WMI GUID is supported in NDIS 6.0 and later versions.

Miniport drivers use the NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG status indication to notify NDIS and overlying drivers that there has been a change in task offload capabilities.

When a miniport driver indicates a task offload change, NDIS translates the status indication to a WMI GUID_NDIS_STATUS_OFFLOAD_CAPABILITIES_CHANGE event for WMI clients.

The data buffer that NDIS provides with the GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by an NDIS_OFFLOAD structure.

For more information about task offload capabilities, see NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG and OID_TCP_OFFLOAD_CURRENT_CONFIG.

GUID_NDIS_STATUS_OFFLOAD_HW_CAPABILITIES

Article • 03/14/2023

The GUID_NDIS_STATUS_OFFLOAD_HW_CAPABILITIES event GUID indicates that there has been a change in the offload characteristics of the hardware that is associated with an NDIS port or miniport adapter. The hardware change typically reflects adding or deleting hardware that is associated with an MUX intermediate driver. This WMI GUID is supported in NDIS 6.0 and later versions.

MUX intermediate drivers use the NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES status indication to notify NDIS and overlying drivers that there has been a change in task offload capabilities.

When the driver indicates a task offload hardware change, NDIS translates the status indication to a WMI GUID_NDIS_STATUS_OFFLOAD_HW_CAPABILITIES event for WMI clients.

The data buffer that NDIS provides with the GUID contains an NDIS_WMI_EVENT_HEADER structure that is followed by an NDIS_OFFLOAD structure.

For more information about task offload capabilities, see NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES and OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES.

Overview of NDIS MSI-X

Article • 03/14/2023

Message-signaled interrupts (MSIs) provide an alternative to traditional line-based interrupts that network devices and their miniport drivers can use. Starting with Windows Vista, the operating system supports two types of MSIs: PCI V2.2 MSI and PCI V3.0 MSI-X.

Miniport drivers that support MSI-X can specify an *interrupt affinity*, which is a subset of central processing units (CPUs) that the drivers' message interrupt service routines run on. You can specify the interrupt affinity for each MSI-X message--for example, you can specify interrupt affinities on computers with Non-Uniform Memory Access (NUMA) architecture in terms of the "nearness" of their device to certain CPUs.

MSI-X support can provide significant performance benefits, especially for network interface cards (NICs) that support receive side scaling (RSS). For more information about receive side scaling, see Receive Side Scaling.

For more information about line-based interrupts, see Managing Interrupts.

This section includes:

MSI-X Initialization

Handling an MSI Interrupt

Synchronizing with an MSI Interrupt

Changing the CPU Affinity of MSI-X Table Entries

MSI-X Initialization

Article • 12/15/2021

To support MSI-X, MSI initialization requires a pre-registration phase in which the miniport driver establishes a function that filters resource-requirements. This function can change the interrupt affinity for each MSI-X message or remove message interrupt resources if the driver will register for line-based interrupts in the *MiniportInitializeEx* function.

The pre-registration phase occurs before NDIS calls the *MiniportInitializeEx* function. As with line-based interrupts, miniport drivers also register MSI interrupts while initializing miniport adapters in *MiniportInitializeEx*.

This section includes:

MSI-X Pre-Registration

MSI-X Resource Filtering

Registering and Deregistering an MSI Interrupt

MSI-X Pre-Registration

Article • 12/15/2021

To support changing interrupt affinities for MSI-X or to remove message interrupt resources, a miniport driver must establish a resource-requirements filter function. This pre-registration step occurs before NDIS calls the *MiniportInitializeEx* function.

To establish a resource-requirements filter function, the miniport driver must provide a *MiniportSetOptions* function. When the miniport driver calls the **NdisMRegisterMiniportDriver** function from the **DriverEntry** routine, the driver passes the entry point for *MiniportSetOptions* in the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure. NDIS calls the *MiniportSetOptions* function in the context of **NdisMRegisterMiniportDriver**.

From *MiniportSetOptions*, the miniport driver calls the **NdisSetOptionalHandlers** function and specifies an **NDIS_MINIPORT_PNP_CHARACTERISTICS** structure. This structure defines the entry points for the *MiniportAddDevice*, *MiniportRemoveDevice*, *MiniportStartDevice*, and *MiniportFilterResourceRequirements* functions.

When NDIS receives an add-device request from the Plug and Play (PnP) manager, NDIS calls the miniport driver's *MiniportAddDevice* function. The handle that NDIS passes to *MiniportAddDevice* in the *MiniportAdapterHandle* parameter is the handle that NDIS later passes to the *MiniportInitializeEx* function.

In *MiniportAddDevice*, the driver initializes an **NDIS_MINIPORT_ADD_DEVICE_REGISTRATION_ATTRIBUTES** structure and passes this structure to the **NdisMSetMiniportAttributes** function. The NDIS_MINIPORT_ADD_DEVICE_REGISTRATION_ATTRIBUTES structure contains the **MiniportAddDeviceContext** member that is a handle to a miniport driver-allocated context area for the device. NDIS later provides this context handle to the *MiniportRemoveDevice*, *MiniportFilterResourceRequirements*, *MiniportStartDevice*, and *MiniportInitializeEx* functions. For *MiniportInitializeEx*, the context handle is passed in the **MiniportAddDeviceContext** member of the **NDIS_MINIPORT_INIT_PARAMETERS** structure that the *MiniportInitParameters* parameter points to.

After NDIS calls *MiniportAddDevice* and *MiniportAddDevice* returns NDIS_STATUS_SUCCESS, NDIS calls the *MiniportFilterResourceRequirements* function every time that it receives the **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** I/O request packet (IRP). *MiniportFilterResourceRequirements* can change the interrupt affinity for each MSI-X message, add message interrupt resources, or remove message interrupt resources if the driver will register for line-based interrupts in the *MiniportInitializeEx* function. For more information about establishing an interrupt affinity policy, see MSI-X Resource Filtering.

When NDIS receives a remove-device request from the PnP manager, NDIS calls the miniport driver's *MiniportRemoveDevice* function. The *MiniportRemoveDevice* function should undo the operations that the *MiniportAddDevice* function performed.

MSI-X Resource Filtering

Article • 12/15/2021

Miniport drivers must register a resource-requirements filter function if they support MSI-X and will change the interrupt affinity for each MSI-X message or will remove message interrupt resources.

NDIS calls the *MiniportFilterResourceRequirements* function after NDIS receives the **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** I/O request packet (IRP) for a network interface card (NIC). NDIS calls *MiniportFilterResourceRequirements* after the underlying function drivers in the device stack have completed the IRP.

NDIS will call *MiniportFilterResourceRequirements* after the *MiniportAddDevice* function returns NDIS_STATUS_SUCCESS. NDIS may call *MiniportFilterResourceRequirements* again at any time before calling *MiniportRemoveDevice*. NDIS may call *MiniportFilterResourceRequirements* while the miniport is running. While the miniport may modify the resource list as described below, the miniport should not immediately attempt to use the new resources. NDIS will eventually halt and re-initialize the miniport with the new resources; only then should the miniport attempt to use the new resources.

IRP_MN_FILTER_RESOURCE_REQUIREMENTS provides a resource list as an **IO_RESOURCE_REQUIREMENTS_LIST** structure at **Irp->IoStatus.Information**. The resources in the list are described by **IO_RESOURCE_DESCRIPTOR** structures.

A miniport driver can modify the interrupt affinity policy for each resource of type **CmResourceTypeInterrupt** that describes an MSI-X message. If an affinity policy requests targeting for a specific set of processors, the miniport driver also sets a **KAFFINITY** mask at **Interrupt.TargetedProcessors** in the IO_RESOURCE_DESCRIPTOR structure.

A miniport driver can remove all resources of type **CmResourceTypeInterrupt** that are message interrupt resources. The driver can then register for line-based interrupts in the *MiniportInitializeEx* function. If the miniport driver does not remove these message interrupt resources, the operating system will fail if the driver tries to register line-based interrupts in *MiniportInitializeEx*.

An NDIS 6.1 or later miniport driver can add message interrupt resources to the resources list. For example on a computer with eight CPUs, if the NIC can generate four MSI-X messages and if the operating system enables the four message interrupts, the operating system initializes four message table entries in the device's MSI-X configuration space and puts four message interrupt resources in the resources list. In this case, because the miniport driver requires more message interrupt resources, it can allocate four more message interrupt resources to the resources list and set the affinity of each MSI-X message to a CPU. If the operating system can provide more message interrupt resources, the miniport adapter receives eight message interrupt resources when it is started. In this case, the messages have numbers from 0 through 7.

Each message interrupt resource in the list is assigned a message number later that corresponds to the order it shows in the list. For example, the first message interrupt resources in the list is assigned to message 0, the second one is assigned to message 1, and so on.

To assign an MSI-X table entry to a CPU at run time, the miniport driver can call the **NdisMConfigMSIXTableEntry** function, which maps a table entry to an MSI-X message that already has the affinity set to the CPU. For more information about configuration operations for MSI-X table entries, see Changing the CPU Affinity of MSI-X Table Entries.

To allocate memory for a new resource-requirements list, use the **NdisAllocateMemoryWithTagPriority** function. You can free the memory for the old resources-requirement list with the **NdisFreeMemory** function.

Miniport drivers should not modify other resources, such as **CmResourceTypeMemory** and **CmResourceTypePort** resources. Miniport drivers should avoid adding a new resource to the resource list. However, NDIS 6.1 and later miniport drivers can add more message interrupt resources. If the miniport driver adds more message interrupt resources, it must not remove them from the *MiniportStartDevice* function.

For more information about adding and removing resources, see **IRP_MN_FILTER_RESOURCE_REQUIREMENTS**.

Registering and Deregistering an MSI Interrupt

Article • 12/15/2021

To register for MSI support, a miniport driver calls the NdisMRegisterInterruptEx function to register an MSI interrupt. The driver allocates and initializes an NDIS_MINIPORT_INTERRUPT_CHARACTERISTICS structure to specify the interrupt characteristics and function entry points. The driver must set the **MsiSupported** member of the NDIS_MINIPORT_INTERRUPT_CHARACTERISTICS structure to **TRUE**. The driver then passes the structure to **NdisMRegisterInterruptEx**.

You must define the following functions to support MSI interrupts:

- *MiniportMessageInterrupt*
- *MiniportMessageInterruptDpc*
- *MiniportDisableMessageInterrupt*
- *MiniportEnableMessageInterrupt*

The miniport driver should provide entry points for the line-based interrupt functions (which are shown in the following list), even if the driver supports the MSI entry points. If NDIS does not grant an MSI interrupt, it can grant a normal interrupt as a fallback condition.

The line-interrupt functions include the following:

- *MiniportInterrupt*
- *MiniportInterruptDPC*
- *MiniportDisableInterruptEx*
- *MiniportEnableInterruptEx*

Drivers should call the NdisMDeregisterInterruptEx function to release resources that were previously allocated with **NdisMRegisterInterruptEx**.

Handling an MSI Interrupt

Article • 12/15/2021

NDIS calls the *MiniportMessageInterrupt* function when a network interface card (NIC) generates an interrupt. The *MessageId* parameter in this function identifies the MSI-X message.

MiniportMessageInterrupt should always return **TRUE** after processing the interrupt because message interrupts are not shared.

A miniport driver should do as little work as possible in its *MiniportMessageInterrupt* function. The driver should defer I/O operations to the *MiniportMessageInterruptDpc* function, which NDIS calls to complete the deferred processing of an interrupt.

To queue additional deferred procedure calls (DPCs) after *MiniportMessageInterrupt* returns, the miniport driver sets the bits of the *TargetProcessors* parameter of the *MiniportMessageInterrupt* function. To request additional DPCs from *MiniportMessageInterrupt* or *MiniportMessageInterruptDPC*, the miniport driver can call the **NdisMQueueDpc** function.

The miniport driver can call **NdisMQueueDpc** to request additional DPCs for other processors.

NDIS 6.1 and later versions guarantees that DPCs for different messages that are scheduled for the same CPU are queued separately. For example, if a miniport driver schedules two DPCs at the same time on CPU 1 (one DPC for message 0 and the other DPC for message 1), two DPCs are queued for CPU 1 (one DPC with message 0 and the other DPC with message 1).

NDIS also guarantees that DPCs for the same message that are scheduled on different CPUs are queued separately. For example, if a miniport driver schedules two DPCs (one DPC on CPU 0 for message 0 and one DPC on CPU 1 for message 0), two separate DPCs are queued on CPU 0 and CPU 1, both for message 0.

Synchronizing with an MSI Interrupt

Article • 12/15/2021

If a miniport driver's *MiniportMessageInterrupt* function shares resources, such as network interface card (NIC) registers or state variables, with another *MiniportXxx* function that runs at a lower IRQL, the other *MiniportXxx* function must call the **NdisMSynchronizeWithInterruptEx** function. This call ensures that the miniport driver's **MiniportSynchronizeMessageInterrupt** function accesses the shared resources in a synchronized and multiprocessor-safe manner.

Changing the CPU Affinity of MSI-X Table Entries

Article • 12/15/2021

NDIS 6.1 and later miniport drivers that support MSI-X can call the **NdisMConfigMSIXTableEntry** function to mask, unmask, or map MSI-X table entries to device-assigned MSI-X messages. Miniport drivers that support RSS use **NdisMConfigMSIXTableEntry** to change the CPU affinity of MSI-X table entries at run time.

NdisMConfigMSIXTableEntry is a wrapper around the GUID_MSIX_TABLE_CONFIG_INTERFACE query. Miniport drivers can call **NdisMConfigMSIXTableEntry** after NDIS calls the *MiniportInitializeEx* function and before the drivers return from the *MiniportHaltEx* function.

A miniport driver that assigns an MSI-X table entry for each RSS queue and has fewer queues than the number of RSS processors can add additional MSI-X message resources in the *MiniportFilterResourceRequirements* function. For more information about how to modify assigned resources for a device, see MSI-X Resource Filtering.

The miniport driver can set the CPU affinity of MSI-X interrupt resources so that the device has at least one MSI-X message for each RSS processor. Note that the PCI bus driver initially maps the *n* MSI-X table entries (where *n* is the number of MSI-X table entries that the NIC hardware reported to the bus) to the first *n* MSI-X messages in modified resources. After NDIS calls *MiniportInitializeEx*, when the miniport driver changes the target processor of a particular MSI-X table entry, the driver calls **NdisMConfigMSIXTableEntry** to map that table entry to an MSI-X message that already has the affinity set to the desired processor.

NDIS Scatter/Gather DMA

Article • 03/14/2023

> ⊗ **Caution**
>
> For Arm and Arm64 processors, we strongly recommend that NDIS driver writers use WDF DMA or WDM DMA instead of NDIS Scatter/Gather DMA.
>
> For more information about WDF DMA, see **Handling DMA Operations in KMDF Drivers**.
>
> For more information about WDM DMA, see the DMA-related child topics of **Managing Input/Output for Drivers**.

NDIS miniport drivers can use the Scatter/Gather DMA (SGDMA) method to transfer data between a NIC and system memory. A successful DMA transfer requires the physical address of the data to be in an address range that the NIC supports. HAL provides a mechanism for drivers to obtain the physical address list for an MDL chain and, if necessary, will double-buffer the data to a physical address range.

In NDIS versions prior to NDIS 6.0, SGDMA support in miniport drivers and NDIS is limited in some respects, and in particular does not work well in a multipacket send scenario. The NDIS 6.0 SGDMA support overcomes these limitations while providing a simple interface for miniport drivers.

History of NDIS SGDMA

In NDIS versions prior to NDIS 6.0, NDIS obtains a scatter gather (SG) list for each packet before sending the packet to the miniport driver. NDIS also handles the case where the original attempt to get the SG list fails due to excessive fragmentation. In this case, NDIS double-buffers the packet to a contiguous buffer and tries again. HAL can also double-buffer the data to a physical address that the NIC supports if, for example, the physical address of the data is above the 32-bit maximum and the NIC does not support 64-bit DMA.

To avoid a deadlock situation, NDIS obtains a SG list for a packet, and sends one packet at a time. If NDIS attempts to map all the packets before sending them to the miniport driver, the system could run out resources. In this case, NDIS would be waiting for map registers to become available while some map registers are locked down for the packets that have not been sent. Locked down packets cannot be reused.

This approach to SGDMA support has the following limitations:

- Because the packet is mapped before it gets to the miniport driver, the driver cannot optimize for small packets or packets that are too fragmented. The miniport driver cannot double buffer the packet to a known physical address.
- There is no guarantee that the physical address array that NDIS passed to the miniport driver maps to the virtual address of the original data. Therefore, if the driver changes the data at the virtual address in the MDL chain before sending it, the modifications made to the data are not reflected in the data in the physical addresses. In this case, the NIC sends the unmodified data.
- NDIS is limited to sending one packet at a time to avoid a deadlock due to resource issues. This is not as efficient as sending multiple packets.
- Because NDIS cannot determine the transmit capabilities of miniport drivers, it cannot preallocate the storage for an SG list buffer. Therefore, NDIS must allocate the necessary storage at run time. This is not as efficient as preallocating the storage.
- HAL functions that allocate an SG list should be called at IRQL = DISPATCH_LEVEL. NDIS does not have the current IRQL information, so it has to set the IRQL to DISPATCH_LEVEL even if it is already at DISPATCH_LEVEL. This is not efficient if the IRQL is already at DISPATCH_LEVEL.

Benefits of NDIS SGDMA Support

In the NDIS 6.0 and later SGDMA interface, NDIS does not map the data buffer before sending it down to the miniport driver. Instead, NDIS provides an interface for the driver to map the network data.

This approach yields the following benefits:

- Since NDIS provides the interface to HAL for mapping the network data, NDIS shields miniport drivers from the complexity and details of the mapping process.
- Miniport drivers have access to the data before it is mapped. Therefore, any changes made to the original data are reflected in the data represented by the SG list even if NDIS or HAL double-buffers the data.
- Miniport drivers can optimize the transmission of small or highly fragmented packets by copying them to a preallocated buffer with a known physical address. This approach avoids mapping that is not required and therefore improves system performance.

- NDIS can send multiple buffers to the miniport driver safely. This results in fewer calls to miniport drivers and therefore improves system performance.
- Miniport drivers can preallocate the memory for an SG list as part of the transmit descriptor blocks. Therefore, NDIS or miniport drivers are not required to allocate memory for SG lists at run time.
- Because miniport drivers can be running at IRQL = DISPATCH_LEVEL, miniport drivers can avoid unnecessary calls to raise the IRQL to DISPATCH_LEVEL. For example, because completing a send happens in the context of an interrupt DPC, miniport drivers can free the SG list without raising the IRQL.

Registering and Deregistering DMA Channels

An NDIS miniport driver calls the **NdisMRegisterScatterGatherDma** function from its *MiniportInitializeEx* function to register a DMA channel with NDIS.

The miniport driver passes a DMA description to **NdisMRegisterScatterGatherDma** in the *DmaDescription* parameter. **NdisMRegisterScatterGatherDma** returns a size for the buffer that should be large enough to hold the scatter/gather list. Miniport drivers should use this size to preallocate the storage for scatter/gather lists.

The miniport driver also passes **NdisMRegisterScatterGatherDma** the entry points for the *MiniportXxx* functions that NDIS calls to process the scatter/gather list. NDIS calls the miniport driver's *MiniportProcessSGList* function after HAL has built the scatter/gather list for a buffer. **NdisMRegisterScatterGatherDma** supplies a handle in the *pNdisMiniportDmaHandle* parameter, which the miniport driver must use in subsequent calls to NDIS scatter/gather DMA functions.

An NDIS miniport driver calls the **NdisMDeregisterScatterGatherDma** function from its *MiniportHaltEx* function to release scatter/gather DMA resources.

Allocating and Freeing Scatter/Gather Lists

An NDIS miniport driver calls the **NdisMAllocateNetBufferSGList** function in its *MiniportSendNetBufferLists* function. The miniport driver calls **NdisMAllocateNetBufferSGList** once for each **NET_BUFFER** structure that it must map. After the resources become available and HAL has the SG list ready, NDIS calls the driver's *MiniportProcessSGList* function. NDIS can call *MiniportProcessSGList* before or after the miniport driver's call to **NdisMAllocateNetBufferSGList** returns.

To improve system performance, the scatter/gather list is generated from the network data starting at the beginning of the MDL that is specified at the **CurrentMdl** member of the associated NET_BUFFER_DATA structure. The start of the network data in the SG list is offset from the beginning of the SG list by the value specified in the **CurrentMdlOffset** member of the associated **NET_BUFFER_DATA** structure.

While handling a DPC for a send-complete interrupt, and after the miniport driver does not need the SG list any more, the miniport driver should call the NdisMFreeNetBufferSGList function to free the SG list.

Note Do not call NdisMFreeNetBufferSGList while the driver or hardware is still accessing the memory that is described by the NET_BUFFER structure that is associated with the scatter/gather list.

Before accessing received data, miniport drivers must call NdisMFreeNetBufferSGList to flush the memory cache.

Exporting a MiniportDevicePnPEventNotify Function

Article • 12/15/2021

NDIS calls a miniport driver's *MiniportDevicePnPEventNotify* function to notify the miniport driver of the following Plug and Play (PnP) events:

- The surprise removal of a NIC that the miniport driver controls.

- A change in the system's power source.

If a miniport driver does not export a *MiniportDevicePnPEventNotify* function, NDIS cannot notify the driver of these PnP events.

All NDIS 6.0 and later miniport drivers *must* export a *MiniportDevicePnPEventNotify* function. In addition, all miniport drivers that have a WDM lower edge *should* export a *MiniportDevicePnPEventNotify* function.

Handling the Surprise Removal of a NIC

Article • 12/15/2021

A surprise removal occurs when a user removes a network interface card (NIC) from a running system without notifying the system beforehand through the user interface (UI).

Miniport drivers for Windows Vista and later versions of the operating system should be able to handle surprise removals. In particular, NDIS miniport drivers with a Windows Driver Model (WDM) lower edge should be able to handle such events. If an NDIS-WDM miniport driver does not handle a surprise removal, any pending IRPs that the miniport driver sent to the underlying bus driver before the surprise removal cannot be completed.

For Windows Vista and later versions, a miniport driver (such as a miniport driver with a WDM lower edge) that does not control hardware directly should set the NDIS_MINIPORT_ATTRIBUTES_SURPRISE_REMOVE_OK attribute flag when calling **NdisMSetMiniportAttributes**. Setting this flag prevents a warning from being displayed when a user performs a surprise removal of a NIC. A miniport driver that cannot handle a surprise removal should not set this flag.

A miniport driver that supports surprise removal should itself attempt to detect a surprise removal during normal operations--outside of the context of *MiniportDevicePnPEventNotify*. After a NIC is removed, an attempt to read a NIC's I/O ports typically results in return values that have all bits set to one. If a miniport driver reads such a value, it should check for the presence of the hardware with a more conclusive test. For example, the miniport driver could write a value to an I/O port and then try to read the value from that port. The miniport driver could also check for valid values in the NIC's I/O registers. Detecting a surprise removal in such a way prevents the miniport driver from hanging in an infinite loop when it attempts to read a removed NIC's registers in an interrupt DPC.. A miniport driver that stops responding in this way stops NDIS from calling the driver's *MiniportDevicePnPEventNotify* function.

Handling a Change in the System's Power Source

Article • 12/15/2021

The system can change from battery power to AC power or vice versa.

After initializing a miniport driver, NDIS calls a miniport driver's *MiniportDevicePnPEventNotify* function to notify the miniport driver of the system's power source. The miniport driver can use this information to adjust the power consumption of a NIC. For example, the miniport driver for a wireless LAN (WLAN) device could reduce power consumption if the system is running on battery power or increase power consumption if the system is running on AC power.

Overview of NDIS Processing of Plug and Play Events

Article • 03/14/2023

The function drivers for a network interface card (NIC) are implemented as an NDIS and miniport driver pair. When a NIC is added to the system, NDIS creates the functional device object (FDO) for the NIC. NDIS then subsequently handles all IRPs, including Plug and Play (PnP) and power management IRPs, that are passed to this FDO. The miniport driver for the NIC provides the operational interface for the NIC.`

The following sections describe how NDIS processes PnP IRPs on behalf of a NIC:

Adding a NIC

Starting a NIC

Stopping a NIC

Removing a NIC

Processing the Surprise Removal of a NIC

Adding a NIC

Article • 12/15/2021

The following description starts with the loading of the miniport driver and describes how a NIC is added. For the initial processing that the PnP manager performs when a NIC is added to a running system, see steps 1-11 of Adding a PnP Device to a Running System.

1. If the miniport driver for the NIC is not already loaded, the PnP manager loads the driver and then calls the miniport driver's **DriverEntry** function. If the driver is already loaded, processing continues with step 4.

2. From its **DriverEntry** function, the miniport driver registers as a miniport drivers and performs other drivers initialization. For more information about registering as a miniport driver, see Initializing a Miniport Driver.

3. NDIS fills in the following entries in the driver object for the miniport driver:

   - The entry point for the *AddDevice* routine.
   - The *DispatchXxx* entry points for handling IRPs.
   - The entry point for the *Unload* routine.

4. The PnP manager calls NDIS's *AddDevice* routine. NDIS's *AddDevice* routine creates a functional device object (FDO) for the newly added NIC and attaches this FDO to the device stack for the NIC.

5. NDIS reads information from the registry to obtain configuration information for the NIC. This information includes binding information and the hardware attributes of the NIC.

6. The PnP manager assigns resources to the NIC, if necessary.

Starting a NIC

Article • 12/15/2021

The following steps describe how NDIS participates in the starting of a NIC:

1. The PnP manager issues an **IRP_MN_START_DEVICE** request. This IRP contains information that informs NDIS about the resources that the PnP manager has allocated for the NIC.

2. NDIS sets an **IoCompletion** routine and passes the IRP_MN_START_DEVICE request down the device stack to the next lowest driver, which is typically the bus driver. When the bus driver receives the IRP_MN_START_DEVICE request, the bus driver performs its start operations on the device and passes the completed IRP_MN_START_DEVICE request back up the device stack.

3. When NDIS receives the completed IRP_MN_START_DEVICE request (that is, when NDIS's **DispatchPnP** routine gains control after all lower drivers have finished with the IRP), NDIS calls the miniport driver's *MiniportInitializeEx* function.

4. If the *MiniportInitializeEx* function returns NDIS_STATUS_SUCCESS, NDIS schedules an event to call the *ProtocolBindAdapterEx* function of all protocol drivers that are supposed to bind to the adapter, as indicated by the binding information in the registry. Note that the miniport driver has no information about bindings.

5. NDIS completes the IRP_MN_START_DEVICE request.

Stopping a NIC

Article • 12/15/2021

The PnP manager stops a NIC so that it can reconfigure or rebalance the hardware resources that it assigned to the NIC. The following steps describe how NDIS participates in the stopping of a NIC:

1. The PnP manager issues an **IRP_MN_QUERY_STOP_DEVICE** request.

2. When NDIS receives this IRP, it calls the *FilterNetPnPEvent* function of the lowest filter driver that is attached to the NIC in the driver stack. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice**.

   Note NDIS performs this step only for filter drivers that advertise an entry point for the *FilterNetPnPEvent* function. A filter driver advertises this entry point when it calls the **NdisFRegisterFilterDriver** function.

3. Within the context of the call to its *FilterNetPnPEvent* function, the filter driver must call **NdisFNetPnPEvent** to forward the **NetEventQueryRemoveDevice** event up to the next filter driver in the driver stack. This causes NDIS to call that filter driver's *FilterNetPnPEvent* function with an event code of **NetEventQueryRemoveDevice**.

   Note NDIS performs this step only for the next filter driver in the driver stack that advertises an entry point for the *FilterNetPnPEvent* function.

4. Each filter driver in the driver stack repeats the previous step until the highest filter driver in the stack has forwarded the **NetEventQueryRemoveDevice** event.

   When this happens, NDIS calls the *ProtocolNetPnPEvent* function of all protocol drivers that are bound to the NIC. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice**.

5. If a protocol driver fails the **NetEventQueryRemoveDevice** event by returning a failure code from *ProtocolNetPnPEvent*, NDIS or the PnP manager might ignore the failure and subsequently succeed the IRP_MN_QUERY_STOP_DEVICE request. A protocol driver must, therefore, be prepared to handle the removal of the NIC even though the protocol driver failed the **NetEventQueryRemoveDevice** event.

6. The PnP manager issues an **IRP_MN_STOP_DEVICE** request to stop the device or an **IRP_MN_CANCEL_STOP_DEVICE** request to cancel the pending stop.

7. If the PnP manager issues an IRP_MN_CANCEL_STOP_DEVICE request, NDIS calls the *FilterNetPnPEvent* function of the lowest filter driver that is attached to the NIC in the driver stack. In this call, NDIS specifies an event code of **NetEventCancelRemoveDevice**.

   Note NDIS performs this step only for filter drivers that advertise an entry point for the *FilterNetPnPEvent* function.

8. Within the context of the call to its *FilterNetPnPEvent* function, the filter driver must call **NdisFNetPnPEvent** to forward the **NetEventCancelRemoveDevice** event up to the next filter driver in the driver stack. This causes NDIS to call that filter driver's *FilterNetPnPEvent* function with an event code of **NetEventCancelRemoveDevice**.

   Note NDIS performs this step only for the next filter driver in the driver stack that advertises an entry point for the *FilterNetPnPEvent* function.

9. Each filter driver in the driver stack repeats the previous step until the highest filter driver in the stack has forwarded the **NetEventCancelRemoveDevice** event.

   When this happens, NDIS calls the *ProtocolNetPnPEvent* function of all protocol drivers that are bound to the NIC. In this call, NDIS specifies an event code of **NetEventCancelRemoveDevice**.

10. If the PnP manager issues an IRP_MN_STOP_DEVICE request, NDIS performs these steps:

    a. It pauses all protocol drivers that are bound to the NIC.

    b. It pauses all filter drivers that are attached to the NIC.

    c. It pauses the miniport driver for the NIC.

    d. It calls the *ProtocolUnbindAdapterEx* function of all protocol drivers that are bound to the NIC.

    e. It calls the *FilterDetach* function of all filter modules that are attached to the NIC.

11. After all protocol and filter drivers are unbound and detached from the NIC, NDIS calls the miniport driver's *MiniportHaltEx* function. NDIS sets the *HaltAction* parameter of *MiniportHaltEx* to **NdisHaltDeviceStopped**.

12. When processing an IRP_MN_STOP_DEVICE request, NDIS does not destroy the functional device object (FDO) that it created for the NIC when the *AddDevice*

routine was called. NDIS destroys the device object only after receiving an **IRP_MN_REMOVE_DEVICE** request for the NIC.

If the PnP manager issues an IRP_MN_START_DEVICE to restart the NIC, NDIS will reuse the FDO that was previously created for the NIC. NDIS will then restart the NIC. For more information on this procedure, see Starting a NIC.

Removing a NIC

Article • 12/15/2021

The following steps describe how NDIS participates in the removal of a NIC:

1. The PnP manager issues an IRP_MN_QUERY_REMOVE_DEVICE request to query whether the NIC can be removed without disrupting the computer.

2. When NDIS receives this IRP, it calls the *FilterNetPnPEvent* function of the lowest filter driver that is attached to the NIC in the driver stack. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice**.

   Note NDIS performs this step only for filter drivers that advertise an entry point for the *FilterNetPnPEvent* function. A filter driver advertises this entry point when it calls the **NdisFRegisterFilterDriver** function.

3. Within the context of the call to its *FilterNetPnPEvent* function, the filter driver must call **NdisFNetPnPEvent** to forward the **NetEventQueryRemoveDevice** event up to the next filter driver in the driver stack. This causes NDIS to call that filter driver's *FilterNetPnPEvent* function with an event code of **NetEventQueryRemoveDevice**.

   Note NDIS performs this step only for the next filter driver in the driver stack that advertises an entry point for the *FilterNetPnPEvent* function.

4. Each filter driver in the driver stack repeats the previous step until the highest filter driver in the stack has forwarded the **NetEventQueryRemoveDevice** event.

   When this happens, NDIS calls the *ProtocolNetPnPEvent* function of all protocol drivers that are bound to the NIC. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice**.

5. If a protocol driver fails the **NetEventQueryRemoveDevice** event by returning a failure code NDIS_STATUS_FAILURE from *ProtocolNetPnPEvent*, NDIS or the PnP manager might ignore the failure and subsequently succeed the IRP_MN_QUERY_REMOVE_DEVICE request. A protocol driver must, therefore, be prepared to handle the removal of the NIC even though the protocol driver failed the **NetEventQueryRemoveDevice** event.

6. The PnP manager issues an IRP_MN_REMOVE_DEVICE request to remove the software representation (device objects, and so on) for the NIC or an IRP_MN_CANCEL_REMOVE_DEVICE request to cancel the pending removal. Note

that an IRP_MN_REMOVE_DEVICE request is not always preceded by an IRP_MN_QUERY_REMOVE_DEVICE request.

7. If the PnP manager issues an IRP_MN_CANCEL_REMOVE_DEVICE request, NDIS calls the *FilterNetPnPEvent* function of the lowest filter driver that is attached to the NIC in the driver stack. In this call, NDIS specifies an event code of **NetEventCancelRemoveDevice**.

   Note NDIS performs this step only for filter drivers that advertise an entry point for the *FilterNetPnPEvent* function.

8. Within the context of the call to its *FilterNetPnPEvent* function, the filter driver must call **NdisFNetPnPEvent** to forward the **NetEventCancelRemoveDevice** event up to the next filter driver in the driver stack. This causes NDIS to call that filter driver's *FilterNetPnPEvent* function with an event code of **NetEventCancelRemoveDevice**.

   Note NDIS performs this step only for the next filter driver in the driver stack that advertises an entry point for the *FilterNetPnPEvent* function.

9. Each filter driver in the driver stack repeats the previous step until the highest filter driver in the stack has forwarded the **NetEventCancelRemoveDevice** event.

   When this happens, NDIS calls the *ProtocolNetPnPEvent* function of all protocol drivers that are bound to the NIC. In this call, NDIS specifies an event code of **NetEventCancelRemoveDevice**. This event code ends the removal sequence.

10. If the PnP manager issues an IRP_MN_REMOVE_DEVICE request, NDIS performs these steps:

    a. It pauses all protocol drivers that are bound to the NIC.

    b. It pauses all filter drivers that are attached to the NIC.

    c. It pauses the miniport driver for the NIC.

    d. It calls the *ProtocolUnbindAdapterEx* function of all protocol drivers that are bound to the NIC.

    e. It calls the *FilterDetach* function of all filter modules that are attached to the NIC.

11. If the miniport driver was successfully initialized, NDIS calls the miniport driver's *MiniportHaltEx* function. NDIS sets the *HaltAction* parameter of *MiniportHaltEx* to **NdisHaltDeviceDisabled**.

12. NDIS sends the IRP_MN_REMOVE_DEVICE request to the next lower device object in the stack.

13. When NDIS receives the completed IRP_MN_REMOVE_DEVICE request from the next lower device object in the stack, NDIS destroys the functional device object (FDO) that it created for the NIC.

Processing the Surprise Removal of a NIC (Windows Vista)

Article • 12/15/2021

The following steps describe how NDIS participates in the surprise removal of a NIC in Windows Vista and Windows Server 2008:

1. The PnP manager issues an **IRP_MN_SURPRISE_REMOVAL** request to the device stack for the NIC.

2. When NDIS receives this IRP, it calls the *FilterNetPnPEvent* function of the lowest filter driver that is attached to the NIC in the driver stack. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice**.

   Note NDIS performs this step only for filter drivers that advertise an entry point for the *FilterNetPnPEvent* function. A filter driver advertise this entry point when it calls the **NdisFRegisterFilterDriver** function.

3. Within the context of the call to its *FilterNetPnPEvent* function, the filter driver must call **NdisFNetPnPEvent** to forward the **NetEventQueryRemoveDevice** event up to the next filter driver in the driver stack. This causes NDIS to call that filter driver's *FilterNetPnPEvent* function with an event code of **NetEventQueryRemoveDevice.**.

   Note NDIS performs this step only for the next filter driver in the driver stack that advertises an entry point for the *FilterNetPnPEvent* function.

4. Each filter driver in the driver stack repeats the previous step until the highest filter driver in the stack has forwarded the **NetEventQueryRemoveDevice.** event.

   When this happens, NDIS calls the *ProtocolNetPnPEvent* function of all protocol drivers that are bound to the NIC. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice.**.

5. If the miniport driver was successfully initialized, NDIS calls the *MiniportDevicePnPEventNotify* function with an event code of **NdisDevicePnPEventSurpriseRemoved**. The miniport driver should note that the device has been physically removed. If the miniport driver is an NDIS-WDM driver, it should cancel any pending IRPs that it sent down to the underlying bus driver. If the miniport driver was not successfully initialized, processing continues.

6. NDIS sends the IRP_MN_SURPRISE_REMOVAL request to the next-lower device object in the stack. After receiving the returned IRP_MN_SURPRISE_REMOVAL request from the next-lower device object in the stack, NDIS completes the IRP_MN_SURPRISE_REMOVAL request.

7. The PnP manager issues an **IRP_MN_REMOVE_DEVICE** request to remove the software representation (device objects, and so forth) for the NIC.

8. NDIS performs the following steps:

   a. It pauses all protocol drivers that are bound to the NIC.

   b. It pauses all filter drivers that are attached to the NIC.

   c. It pauses the miniport driver for the NIC.

   d. It calls the *ProtocolUnbindAdapterEx* function of all protocol drivers that are bound to the NIC.

   e. It calls the *FilterDetach* function of all filter modules that are attached to the NIC.

9. After all protocol and filter drivers are unbound and detached from the NIC, NDIS calls the miniport driver's *MiniportHaltEx* function. NDIS sets the *HaltAction* parameter of *MiniportHaltEx* to **NdisHaltDeviceSurpriseRemoved**.

10. NDIS sends the IRP_MN_REMOVE_DEVICE request to the next lower device object in the stack.

11. When NDIS receives the completed IRP_MN_REMOVE_DEVICE request from the next lower device object in the stack, NDIS destroys the functional device object (FDO) that it created for the NIC.

Processing the Surprise Removal of a NIC (Windows 7 and Later Versions)

Article • 12/15/2021

In Windows 7 and Windows Server 2008 R2 and later, NDIS may participate in the surprise removal of a network interface card (NIC) differently than it had in previous versions of Windows. NDIS performs a revised surprise removal procedure if *any* of the following conditions are true:

- The operating system is Windows 8 / Windows Server 2012 or later.
- The operating system is Windows 7, and the hotfix for KB2471472 has been installed.
- The operating system is Windows 7, and the network adapter is a mobile broadband (MBB) device.

If none of these conditions are met, NDIS participates in the surprise removal process as it did in previous versions of Windows. For more information about this procedure, see Processing the Surprise Removal of a NIC (Windows Vista).

The following steps describe the revised way in which NDIS participates in the surprise removal of a NIC:

1. The PnP manager issues an **IRP_MN_SURPRISE_REMOVAL** request to the device stack for the NIC.

2. When NDIS receives this IRP, it calls the *FilterNetPnPEvent* function of the lowest filter driver that is attached to the NIC in the driver stack. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice**.

   Note NDIS performs this step only for filter drivers that advertise an entry point for the *FilterNetPnPEvent* function. A filter driver advertise this entry point when it calls the **NdisFRegisterFilterDriver** function.

3. Within the context of the call to its *FilterNetPnPEvent* function, the filter driver must call **NdisFNetPnPEvent** to forward the **NetEventQueryRemoveDevice** event up to the next filter driver in the driver stack. This causes NDIS to call that filter driver's *FilterNetPnPEvent* function with an event code of **NetEventQueryRemoveDevice**.

   Note NDIS performs this step only for the next filter driver in the driver stack that advertises an entry point for the *FilterNetPnPEvent* function.

4. Each filter driver in the driver stack repeats the previous step until the highest filter driver in the stack has forwarded the **NetEventQueryRemoveDevice** event.

   When this happens, NDIS calls the *ProtocolNetPnPEvent* function of all protocol drivers that are bound to the NIC. In this call, NDIS specifies an event code of **NetEventQueryRemoveDevice**.

5. NDIS calls the *MiniportDevicePnPEventNotify* function with an event code of **NdisDevicePnPEventSurpriseRemoved**. The miniport driver should note that the device has been physically removed. If the miniport driver is an NDIS-WDM driver, it should cancel any pending IRPs that it sent down to the underlying bus driver.

6. NDIS performs the following steps:

   a. It pauses all protocol drivers that are bound to the NIC.

   b. It pauses all filter drivers that are attached to the NIC.

   c. It pauses the miniport driver for the NIC.

   d. It calls the *ProtocolUnbindAdapterEx* function of all protocol drivers that are bound to the NIC.

   e. It calls the *FilterDetach* function of all filter modules that are attached to the NIC.

7. After all protocol and filter drivers are unbound and detached from the NIC, NDIS calls the miniport driver's *MiniportHaltEx* function. NDIS sets the *HaltAction* parameter of *MiniportHaltEx* to **NdisHaltDeviceSurpriseRemoved**.

8. NDIS sends the IRP_MN_SURPRISE_REMOVAL request to the next-lower device object in the stack. After receiving the returned IRP_MN_SURPRISE_REMOVAL request from the next-lower device object in the stack, NDIS completes the IRP_MN_SURPRISE_REMOVAL request.

9. The PnP manager issues an **IRP_MN_REMOVE_DEVICE** request to remove the software representation (device objects, and so forth) for the NIC.

10. NDIS sends the IRP_MN_REMOVE_DEVICE request to the next lower device object in the stack.

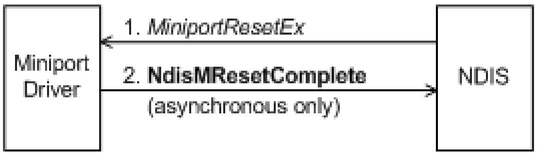
11. When NDIS receives the completed IRP_MN_REMOVE_DEVICE request from the next lower device object in the stack, NDIS destroys the functional device object (FDO) that it created for the NIC.

Miniport Driver Hardware Reset

Article • 12/15/2021

A miniport driver must register a *MiniportResetEx* function with **NdisMRegisterMiniportDriver**.

MiniportResetEx can complete synchronously or asynchronously with a call to **NdisMResetComplete**(see the following figure).



MiniportResetEx should:

- Disable further interrupts.
- Clear out the data that is associated with any sends in progress. For example, on a ring buffer for a bus-master direct memory access (DMA) device, the pointers to send buffers should be cleared. Deserialized and connection-oriented miniport drivers must return NDIS_STATUS_REQUEST_ABORTED for any queued send requests.
- Restore the hardware state and the miniport driver's internal state to the state that existed before the reset operation.

The miniport driver is responsible for restoring the hardware state of the device except for multicast addresses, packet filters, task offload settings, and wake up patterns. These setting are restored by either the miniport driver or NDIS. The miniport driver determines who is responsible for restoring these settings by returning a Boolean value in the *AddressingReset* parameter.

If the miniport driver returns **FALSE** in the *AddressingReset* parameter, the miniport driver restores its multicast addresses, packet filters, task offload settings, and wake up patterns to their initial state. If the miniport driver returns **TRUE** in *AddressingReset*, NDIS calls a connectionless miniport driver's *MiniportOidRequest* function or a connection-oriented miniport driver's **MiniportCoOidRequest** function to set the following configuration settings:

- The network packet filter through a set request of OID_GEN_CURRENT_PACKET_FILTER.
- The multicast address list through a set request of OID_802_3_MULTICAST_LIST.

- Task offload encapsulation settings through a set request of OID_OFFLOAD_ENCAPSULATION.
- Power management wake-up patterns through a set request of OID_PNP_ADD_WAKE_UP_PATTERN. **Note** Starting with NDIS 6.20, wake-up patterns set through OID_PM_ADD_WOL_PATTERN must be restored by the miniport driver.

Related topics

Adapter States of a Miniport Driver

Miniport Adapter States and Operations

Miniport Driver Reset and Halt Functions

Miniport Driver Halt Handler

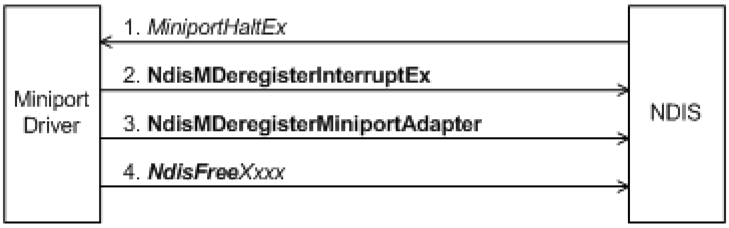
Article • 12/15/2021

An NDIS miniport driver must supply a *MiniportHaltEx* function to **NdisMRegisterMiniportDriver**.

MiniportHaltEx should undo everything that *MiniportInitializeEx* did. For example, the NDIS miniport driver might:

- Free ports. (For more information, see Freeing an NDIS Port.)
- Release all of the hardware resources that *MiniportInitializeEx* claimed.
- Free interrupt resources by calling **NdisMDeregisterInterruptEx**.
- Free any memory that *MiniportInitializeEx* allocated.
- Stop the NIC, unless the *MiniportShutdownEx* function has already restored the NIC to its initial state.

The following diagram illustrates unloading a miniport driver.


Miniport Driver
1. MiniportHaltEx
2. NdisMDeregisterInterruptEx
3. NdisMDeregisterMiniportAdapter
4. NdisFreeXxxx
NDIS

MiniportHaltEx should complete the operations that are necessary to unload the driver before returning. If the miniport driver has any outstanding receive indications (that is, received network data that it has indicated up to NDIS but which NDIS has not yet returned), *MiniportHaltEx* must not return until such data is returned to the miniport driver's *MiniportReturnNetBufferLists* function.

The preceding figure shows a set of calls that could be made by a *MiniportHaltEx* function. These calls are only a subset of the calls that could be made. The actual set of calls depends on previous actions of the miniport driver. The miniport driver can make these same calls in *MiniportInitializeEx* if it cannot successfully initialize the network adapter because of hardware problems or because it cannot acquire a resource that it needs. In such a case, *MiniportInitializeEx* should unload the driver by undoing its previous actions. Otherwise, *MiniportHaltEx* will undo the actions of *MiniportInitializeEx*.

The following list describes the calls that are required to reverse certain actions that the miniport driver might take:

- If the miniport driver registered an interrupt, it should call NdisMDeregisterInterruptEx.

- If the miniport driver set up a timer or timers, it should call NdisCancelTimerObject for each timer that it created. If a call to **NdisCancelTimerObject** fails, the timer might have already fired. In this case, the miniport driver should wait for the timer handler to complete before returning from *MiniportHaltEx*.

- If the miniport driver allocated any memory with NdisAllocateMemoryWithTagPriority, it should call NdisFreeMemory to free that memory.

- If the miniport driver allocated any memory with NdisMAllocateSharedMemory, or NdisMAllocateSharedMemoryAsyncEx, it should call NdisMFreeSharedMemory to free that memory.

- If the miniport driver allocated and initialized storage for a pool of packet descriptors with NdisAllocateNetBufferPool, it should call NdisFreeNetBufferPool to free that storage.

- If the miniport driver allocated or reserved any hardware resources, these should be returned. For example, if the miniport driver mapped an I/O port range on a NIC, it should release the ports by calling NdisMDeregisterIoPortRange.

Related topics

Adapter States of a Miniport Driver

Freeing an NDIS Port

Halting a Miniport Adapter

Miniport Adapter States and Operations

Miniport Driver Reset and Halt Functions

Overview of Miniport Drivers with a WDM Lower Interface

Article • 12/15/2021

A miniport driver with a Microsoft Windows Driver Model (WDM) lower interface is also known as an *NDIS-WDM miniport driver*. This type of miniport driver:

- Uses a WDM lower edge.
- Can call both NDIS and non-NDIS functions. However, whenever possible, the miniport driver should call NDIS functions.
- Can initialize a miniport instance that is used to control devices that are attached to a particular bus and that communicates with those devices over that bus.

For example, a miniport driver that controls devices on either Universal Serial Bus (USB) or IEEE 1394 (Firewire) buses must expose a standard NDIS miniport driver interface at its upper edge and use the class interface for the particular bus at its lower edge. Such a miniport driver communicates with devices that are attached to the bus by sending I/O request packets (IRPs) to the bus.

The following topics describe how to implement a miniport driver that uses a WDM lower edge:

Miniport Driver with a WDM Lower Edge

Registering Miniport Driver Functions for WDM Lower Edge

Initializing a Miniport Driver with a WDM Lower Edge

Issuing Commands to Communicate with Devices

Implementation Tips and Requirements for WDM Lower Edge

Compile Flags for WDM Lower Edge

Power Management for WDM Lower Edge

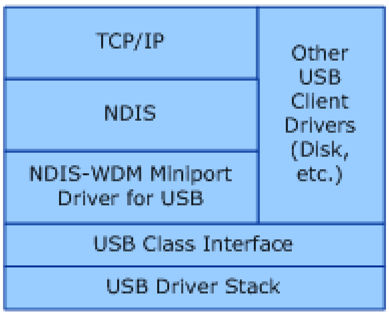
Installing NDIS-WDM Miniport Drivers

Miniport Driver with a WDM Lower Edge

Article • 12/15/2021

A miniport driver with a WDM lower edge (an NDIS-WDM miniport driver) follows the WDM rule that specifies that a WDM header file must be included in the driver's source files. An NDIS-WDM miniport driver requires a WDM header file to call kernel-mode routines on its lower edge. Typically, NDIS miniport drivers should just call functions that NDIS provides. This restriction is shown by the way NDIS wraps around NDIS miniport drivers in the figure in the NDIS Drivers section. Although typical NDIS miniport drivers are not called WDM drivers, they indirectly follow WDM rules because NDIS itself follows WDM rules.

The following diagram shows an NDIS-WDM miniport driver that interfaces with the USB driver stack by using a WDM lower edge.


TCP/IP
NDIS
NDIS-WDM Miniport Driver for USB
Other USB Client Drivers (Disk, etc.)
USB Class Interface
USB Driver Stack

The following list describes the components that the preceding diagram shows:

IPX/SPX Compatible and TCP/IP
NDIS protocol drivers that transmit packets by using underlying miniport drivers.

NDIS
The Ndis.sys driver that provides a standard interface between layered network drivers.

NDIS-WDM Miniport Driver for USB
An NDIS-WDM miniport driver that interfaces with the USB driver stack.

USB Client Drivers
Other vendor-supplied USB client drivers.

USB Class Interface
USB Routines and I/O requests that USB client drivers can use to interface with the USB driver stack.

USB Driver Stack

Driver stack for USB devices. For more information, see USB Driver Stack Architecture.

Registering Miniport Driver Functions for WDM Lower Edge

Article • 12/15/2021

A miniport driver that has a WDM lower edge must call the NdisMRegisterMiniportDriver function in its **DriverEntry** routine to register certain entry-point functions with the NDIS library. These entry-point functions compose the miniport driver's upper edge and are described in Initializing a Miniport Driver. However, a miniport driver that has a WDM lower edge is not required to set up certain entry-point functions. For example, the following entry-point functions are not set up for the following reasons:

- *MiniportInterrupt*, *MiniportInterruptDPC*, *MiniportEnableInterruptEx*, and *MiniportDisableInterruptEx*

  Because the miniport driver does not receive interrupts from a physical network interface card (NIC), it does not require these entry-point routines. The driver for the particular bus receives interrupts when packets arrive on the bus that are intended for the miniport driver. The bus driver then notifies the miniport driver.

- *MiniportSharedMemoryAllocateComplete*

  Because the miniport driver does not allocate shared memory, a completion entry-point routine is not specified.

- *MiniportCheckForHangEx*

  The miniport driver can rely on NDIS to determine if its miniport instance has stopped responding, based on sends and requests that time out, so this routine is not typically required.

Initializing a Miniport Driver with a WDM Lower Edge

Article • 12/15/2021

After a miniport driver has been loaded by the operating system, NDIS calls the miniport driver's *MiniportInitializeEx* function to initialize a miniport instance that the miniport driver manages. To communicate through a miniport instance that has a WDM lower edge, the miniport driver must retrieve specific information to set up its communications.

During initialization of this miniport instance, the miniport driver must call the **NdisMGetDeviceProperty** function to retrieve device objects that are required to set up communication with the miniport instance through a WDM interface. In this call, the miniport driver passes the handle to the miniport instance in the *MiniportAdapterHandle* parameter and buffers that receive pointers to **DEVICE_OBJECT** structures. The miniport driver uses the retrieved pointer to the next-device object (*NextDeviceObject* parameter) to create and submit IRPs. For more information, see Handling IRPs.

A miniport driver with a WDM lower edge must be a deserialized miniport driver. A deserialized miniport driver manages its own queue of send and receive requests internally whenever it has insufficient resources to handle these requests immediately; if a miniport driver is not deserialized, NDIS manages this queue. An NDIS-WDM miniport driver must be deserialized because it sends and receives packets outside of the context of NDIS calls. During initialization of a miniport instance, an NDIS-WDM miniport driver must specify the deserialized feature. All NDIS 6.0 and later miniport drivers are deserialized.

Note that an NDIS-WDM miniport driver cannot be an intermediate driver (a driver that exposes a miniport driver interface at the top and a protocol driver interface at the bottom).

Issuing Commands to Communicate with Devices

Article • 12/15/2021

A miniport driver with a WDM lower edge that controls devices on a bus issues commands to the bus interface to communicate with those devices. For more information, see Handling IRPs and the documentation for the particular bus-driver interface.

Implementation Tips and Requirements for WDM Lower Edge

Article • 12/15/2021

This topic describes tips and requirements for implementing an NDIS-WDM miniport driver. An NDIS-WDM miniport driver can call both NDIS and non-NDIS functions. These non-NDIS functions include, for example, WDM-kernel-mode support routines and functions for a particular bus-driver interface.

When implementing an NDIS-WDM miniport driver, keep the following in mind:

- Building an NDIS-WDM miniport driver requires that the NDIS_WDM flag is defined before the Ndis.h header file is included. Defining the NDIS_WDM flag ensures that Ndis.h automatically includes the appropriate WDM header file. The NDIS_WDM flag should be either embedded at the start of the miniport driver's source code or set in the miniport driver's Sources file. An NDIS-WDM miniport driver requires a WDM header file to call kernel-mode routines such as **IoCallDriver** and **IoAllocateIrp**.

- Function calls for a particular bus-driver interface require the header files for that bus driver.

- Including NDIS and non-NDIS headers in the same source file is not recommended because they might not be compatible. That is, separate source files should be created for code that calls NDIS functions and for code that calls non-NDIS functions.

- An NDIS-WDM miniport driver should call appropriate NDIS functions to allocate and release resources unless the NDIS-WDM miniport driver allocates and releases resources in one of the following scenarios:
  - A resource, typically a memory resource, is allocated by the NDIS-WDM miniport driver and is later released by a non-NDIS entity such as a bus-driver interface,
  - A resource, typically a memory resource, is allocated by a non-NDIS entity and is later released by the NDIS-WDM miniport driver.

  For the preceding scenarios, the NDIS-WDM miniport driver should call the appropriate WDM routines to allocate or release resources for the non-NDIS entity.

Compile Flags for WDM Lower Edge

Article • 12/15/2021

You must include the following compile flags in the Sources file for an NDIS-WDM miniport driver to build the NDIS-WDM miniport driver:

- -DNDIS_WDM=1

  Directs NDIS to include the appropriate WDM header file.

Alternatively, you can embed compile flags at the start of the miniport driver's source code before the Ndis.h header file is included.

Power Management for WDM Lower Edge

Article • 12/15/2021

NDIS handles all Plug and Play (PnP) and power management IRPs for NDIS-WDM miniport drivers. Therefore, NDIS-WDM miniport drivers should respond to PnP and power management OIDs, based on device capabilities, as described in Power Management for NDIS Miniport Drivers.

Installing NDIS-WDM Miniport Drivers

Article • 03/14/2023

When you implement the installation mechanism for an NDIS-WDM miniport driver, you should keep the following items in mind:

- Create an information (INF) file for a **Net** class of network component as described in Creating Network INF Files.

- Include Plug and Play (PnP) identifiers (ID) of devices as is typically done for any **Net** class of network component; however, make these IDs specific to the bus type to which the devices are attached.

Overview of WAN Miniport Drivers

Article • 09/27/2024

This section describes CoNDIS WAN miniport drivers, CoNDIS WAN call managers, CoNDIS WAN miniport call managers (MCMs), and the environments in which they operate. These WAN drivers support the Remote Access Service (RAS) and the Point-to-Point Protocol (PPP) over such media as ISDN, Frame Relay, or Switched 56.

This section includes the following topics:

Choosing a WAN Driver Model

Overview of the WAN Architecture

Implementing CoNDIS WAN Miniport Drivers

CoNDIS WAN Operations That Support Telephonic Services

WAN Packet Framing

WAN Miniport Driver Build Parameters

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

WAN Driver Models

Article • 12/15/2021

Microsoft Windows 2000 and later operating systems support two WAN driver models: NDIS WAN and CoNDIS WAN.

NDIS WAN miniport drivers are built on the NDIS model for connectionless miniport drivers. NDIS WAN miniport drivers are not supported for NDIS version 5.0 and later drivers. New drivers should be based on the CoNDIS WAN driver architecture.

CoNDIS WAN drivers are built on the connection-oriented NDIS (CoNDIS) driver model.

CoNDIS WAN miniport drivers and miniport call managers (MCMs) can:

- Call the same NDIS functions that non-WAN connection-oriented miniport drivers call.
- Export the same set of *MiniportXxx* functions that non-WAN connection-oriented miniport drivers export.
- Provide additional WAN-specific capabilities.

For more information about CoNDIS drivers, see Connection-Oriented NDIS.

If you are writing a new WAN driver, we recommend that you use the CoNDIS WAN model.

Microsoft will continue to support existing NDIS WAN miniport drivers. You do not have to write CoNDIS drivers for old hardware.

The following topics describe the primary advantages of using the CoNDIS WAN model:

CoNDIS WAN Is More Flexible

CoNDIS WAN Is Less Complex

Other Benefits of CoNDIS WAN

Other NDIS Features Available to CoNDIS WAN Drivers

CoNDIS WAN Is More Flexible

Article • 03/14/2023

In the CoNDIS model, major functions (such as call management and data transfer) are compartmentalized into discrete components or subcomponents. This organization enables you to use system-supplied and third-party components and update functionality more easily.

The CoNDIS model provides four types of drivers:

- Connection-oriented client drivers
- Call managers
- Connection-oriented miniport drivers
- Integrated miniport call managers (MCMs)

For more information about CoNDIS drivers, see Connection-Oriented NDIS.

The separation of call manager and miniport driver components enables you to update the miniport driver to support new hardware while the call manager remains unchanged. In many cases, the call manager might require upgrades only to correct defects.

The separation of architectural components remains clearly defined in an MCM. The call manager subcomponent of the MCM handles the signaling aspects of connections, and the CoNDIS WAN miniport driver subcomponent handles the NIC hardware.

You can use a system-supplied call manager. If the system does not provide a call manager for your media type (as with, for example, ISDN), you can write one or possibly obtain one from a third party.

The Microsoft Windows operating system includes a PPP CoNDIS client, and CoNDIS WAN miniport drivers are available for many devices. You can write CoNDIS WAN clients to extend the system to support other protocol drivers in addition to PPP.

The CoNDIS WAN model is not restricted to PPP data. You can implement a custom WAN client driver and miniport driver to handle, for example, raw data streaming or proprietary encryption.

CoNDIS WAN Is Less Complex

Article • 03/14/2023

CoNDIS defines objects that correspond to each of the logical entities that are involved in a connection. These entities include address families (AFs), virtual connections (VCs), service access points (SAPs), and parties.

In the CoNDIS environment, the system handles many of the complex TAPI requirements. Therefore, a CoNDIS WAN miniport driver or MCM does not have to handle as many TAPI OIDs as an NDIS WAN miniport driver. In addition, the CoNDIS WAN miniport driver or MCM is not required to handle the following status indications:

- NDIS_STATUS_TAPI_INDICATION
- NDIS_STATUS_WAN_LINE_UP
- NDIS_STATUS_WAN_LINE_DOWN

The separation of the call manager and miniport driver functions enables you to implement two simple drivers. The simplified drivers should be easier to maintain and debug than one large and complex driver.

Other Benefits of CoNDIS WAN

Article • 03/14/2023

In addition to flexibility and simplicity, the CoNDIS WAN model provides the following benefits:

- CoNDIS WAN miniport drivers support multipacket send and receive operations.
- When a CoNDIS miniport driver indicates a receive packet, a bound protocol can defer processing of the packet. When an NDIS WAN miniport driver indicates a receive packet, a bound protocol must copy the data immediately.
- CoNDIS WAN supports multipoint calls. For more information about making multipoint calls, see Making a Call.
- CoNDIS WAN supports quality of service (QoS). CoNDIS WAN drivers use the **NET_BUFFER** structure. For more information about CoNDIS QoS, see Client-Initiated Request to Change Call Parameters.
- Only CoNDIS WAN will support future NDIS enhancements that apply to WAN drivers.

Other NDIS Features Available to CoNDIS WAN Drivers

Article • 03/14/2023

CoNDIS WAN miniport drivers or MCMs can take advantage of the following functionality:

- Plug and Play (PnP) event notification for miniport drivers
- 64-bit statistical counters
- Canceling send packets
- Registering a *MiniportShutdownEx* function
- New miniport driver attributes
- Safe functions

Overview of the WAN Architecture

Article • 12/15/2021

The WAN architecture consists primarily of the components that interface directly to WAN miniport drivers. However, the WAN architecture is best introduced within the broader context of the RAS architecture. The RAS architecture includes some components that are essential for a high-level understanding of the WAN architecture but are otherwise out of the scope of the Microsoft Windows Driver Kit (WDK) and the Windows Driver Development Kit (DDK).

The following topics describe an overview of the RAS architecture, the major WAN system components, and the primary implementation differences between the CoNDIS WAN and NDIS WAN models:

RAS Architecture Overview

NDISWAN Overview

WAN Driver Bindings and Connections

NDISTAPI Overview

NDPROXY Overview

RAS Architecture Overview

Article • 12/06/2022

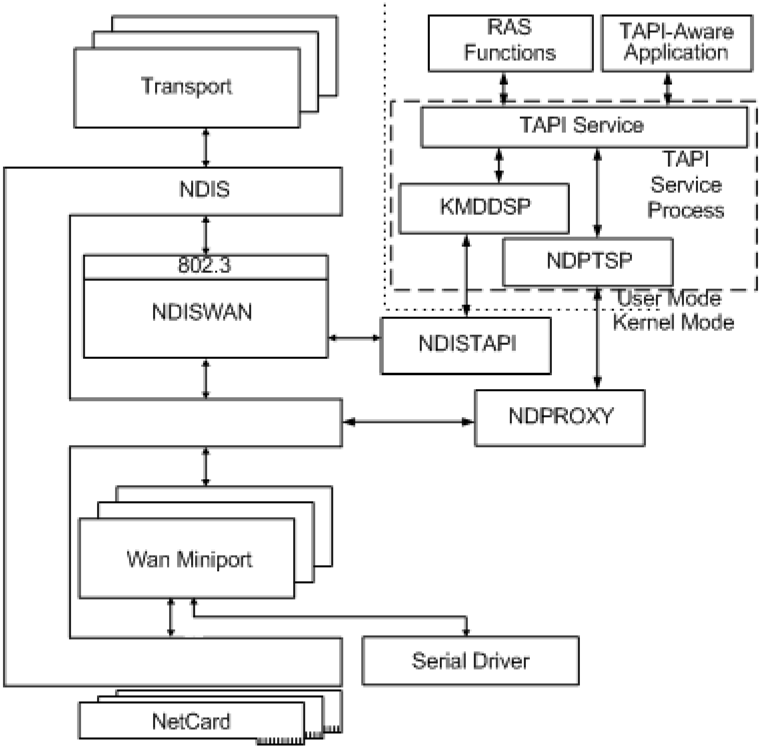
The Remote Access Service (RAS) enables remote workstations to establish a dial-up connection to a LAN and access resources on the LAN as if the remote workstation were on the LAN. WAN miniport drivers provide the interface between RAS and wide area network (WAN) cards such as ISDN, X.25, and Switched 56 adapters.

The primary system-supplied components of the RAS architecture include the following:

- NDISWAN
- TAPI service
- NDPROXY
- NDISTAPI

Developers provide TAPI-aware applications and WAN miniport drivers. CoNDIS WAN developers can also provide WAN client protocol drivers, a miniport call manager (MCM), or a separate call manager.

The following figure shows the RAS architecture.



The following sections briefly describe the components in the RAS architecture.

RAS and TAPI Components

The components on the right side of the preceding figure implement TAPI-related call management operations, such as setting up and tearing down calls and connections. The details of these operations depend on the WAN model (NDIS WAN or CoNDIS WAN).

RAS Functions

User-mode applications call RAS functions to make RAS connections with remote computers. After a RAS connection is established, such applications can connect to network services by using standard network interfaces such as Microsoft Windows Sockets, NetBIOS, Named Pipes, or RPC.

TAPI-Aware Applications

TAPI-aware applications, which are capable of telephony communication, run in both application and service processes. Service providers communicate with specific devices. TAPI-aware applications communicate through the TAPI interface (Tapi32.dll) with their service providers. These service providers run in the TAPI service process.

TAPI Service

The TAPI service (Tapisrv.exe) process presents the Telephony Service Provider Interface (TSPI) of service providers to TAPI-aware applications. These service providers are DLLs that run in the context of the TAPI service process.

The operating system supplies service providers that NDIS WAN or CoNDIS WAN miniport drivers use to communicate with user-mode applications. The service provider for NDIS WAN miniport drivers is KMDDSP. The service provider for CoNDIS WAN miniport drivers (and MCMs) is NDPTSP.

KMDDSP

KMDDSP (Kmddsp.tsp) is a service provider DLL that runs in the context of the TAPI service process. KMDDSP provides a TSPI interface that the TAPI service presents to TAPI-aware applications so that NDISTAPI can communicate with user-mode applications.

KMDDSP works with NDISTAPI to convert user-mode requests to corresponding TAPI OIDs (OID_TAPI_*Xxx*). For more information about TAPI OIDs, see TAPI Objects.

NDPTSP

NDPTSP (Ndptsp.tsp) is a service provider DLL that runs in the context of the TAPI service process. NDPTSP provides a TSPI interface that the TAPI service presents to TAPI-aware applications so that NDPROXY can communicate with user-mode applications.

NDPTSP works with NDPROXY to convert user-mode requests to TAPI connection-oriented OIDs (OID_CO_TAPI_*Xxx*). For more information about TAPI connection-oriented OIDs, see TAPI Extensions for Connection-Oriented NDIS.

NDISTAPI

NDISTAPI (Ndistapi.sys) receives TAPI requests from KMDDSP and then calls **NdisOidRequest** to route the corresponding TAPI OIDs to NDIS WAN miniport drivers. For more information about NDISTAPI, see NDISTAPI Overview.

NDPROXY

NDPROXY (Ndproxy.sys) communicates with TAPI through the TSPI interface that NDPTSP provides. NDPROXY communicates through NDIS with NDISWAN and CoNDIS WAN miniport drivers, MCMs, and call managers.

For more information about NDPROXY, see NDPROXY Overview.

Driver Stack

WAN Transports

The RAS system component provides transports such as PPP Authentication (PAP, CHAP) and network configuration protocol drivers (IPCP, IPXCP, NBFCP, LCP, and so on). A WAN miniport driver (or MCM) implements only PPP media-specific framing.

NDISWAN

NDISWAN (Ndiswan.sys) is an NDIS intermediate driver. NDISWAN binds to NDIS protocol drivers at its upper edge and WAN miniport drivers at its lower edge.

NDISWAN provides PPP protocol/link framing, compression/decompression, and encryption/decryption. NDISWAN interfaces with both NDIS WAN and CoNDIS WAN miniport drivers.

For more information about NDISWAN, see NDISWAN Overview.

Serial Driver

The serial driver component is a standard device driver for internal serial ports or multiport serial cards. The asynchronous WAN miniport driver included with Microsoft Windows 2000 and later uses the internal serial driver for modem communications. Any driver that exports the same functions as the serial driver can interface with the built-in asynchronous WAN miniport driver.

Note X.25 vendors can implement serial driver emulators for an X.25 interface card. In this case, each virtual circuit on the X.25 card appears as a serial port with an X.25 packet assembler/disassembler (PAD) attached to it. The connection interface must correctly emulate serial signals such as DTR, DCD, CTS, RTS, and DSR. X.25 vendors who implement a serial driver emulator for their X.25 card must also make an entry for their PAD in the Pad.inf file. This file contains the command/response script required to make a connection through the X.25 PAD.

WAN Miniport Driver

A WAN miniport driver provides the interface between NDISWAN and WAN NICs.

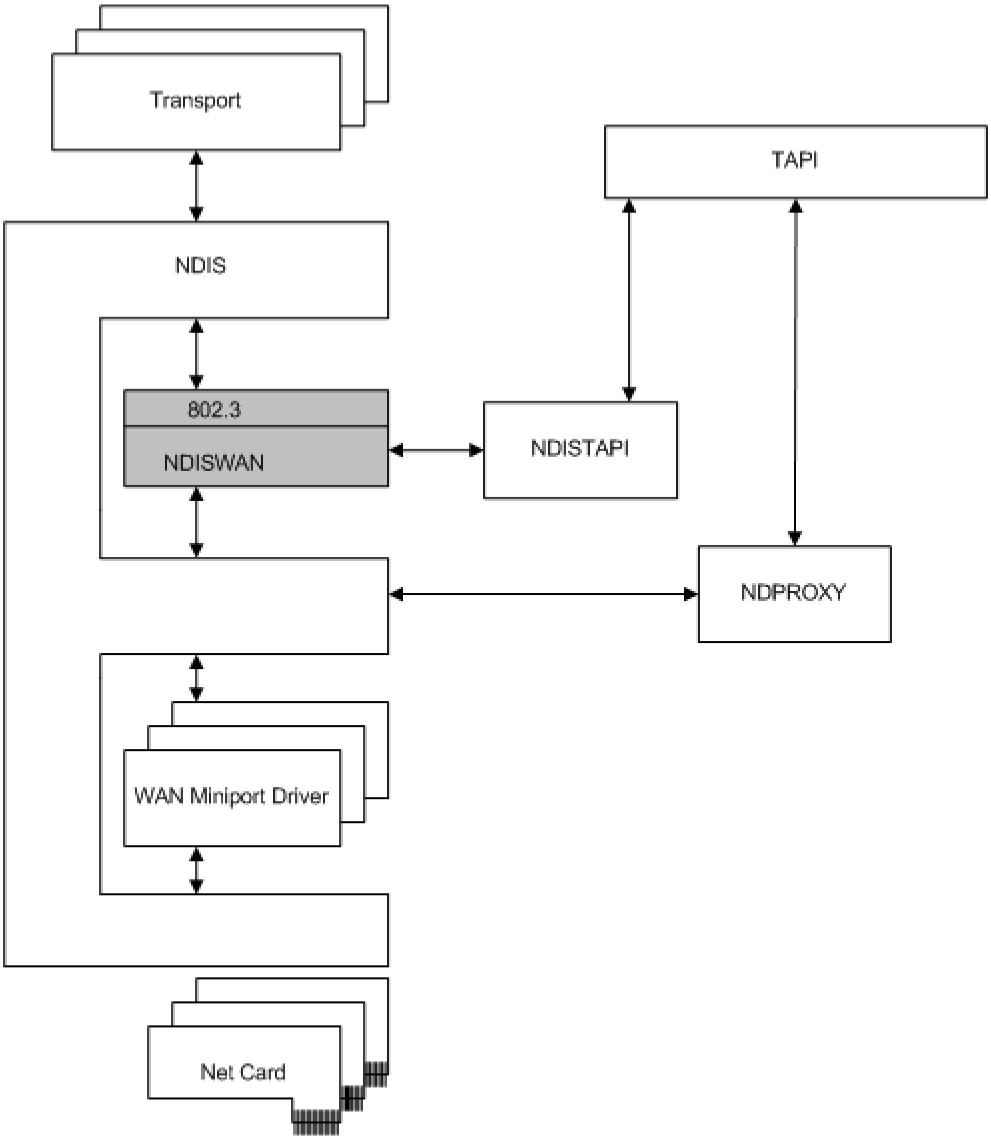
A WAN miniport driver can be implemented as an NDIS WAN miniport driver or a CoNDIS WAN miniport driver. For more information about choosing the miniport driver model that is most appropriate for your application, see Choosing a WAN Driver Model.

NDISWAN Overview

Article • 03/14/2023

NDISWAN is a system-supplied NDIS intermediate driver that provides functionality such as data compression, encryption, loopback, and simple PPP framing that is used by WAN miniport drivers. WAN miniport drivers are therefore required to implement only those features that are specific to the medium (for example, Q931 signaling is required for ISDN).

The following figure shows how NDISWAN interfaces with other components in the RAS architecture.



To overlying protocol drivers, NDISWAN presents both NDIS and CoNDIS miniport driver interfaces. To underlying WAN miniport drivers, NDISWAN presents both NDIS and CoNDIS protocol interfaces that include some WAN-specific elements.

In a CoNDIS environment, the WAN miniport driver can be a connection-oriented miniport driver or an integrated miniport call manager (MCM).

NDISWAN provides the following functionality:

- **Packet conversion**

  NDISWAN converts send packets that are passed to it by protocol drivers from LAN to PPP format. NDISWAN performs the reverse conversion for receive packets passed to it by WAN miniport drivers. NDISWAN uses simple HDLC framing. Most of the media-specific framing must be done by the miniport driver. For more information about WAN packet framing, see WAN Packet Framing.

- **Packet processing**

  Send packets include configuration options for header compression, data compression, and encryption. NDISWAN applies these operations in that order on send packets. NDISWAN applies these options in the reverse order on receive packets. If NDISWAN determines that a configuration option such as compression or encryption is enabled, NDISWAN sends an OID to inform the underlying WAN miniport driver.

- **Simplified binding for drivers**

  NDISWAN simplifies the bindings between protocol drivers and WAN miniport drivers. For more information about WAN driver bindings, see WAN Driver Bindings and Connections.

- **Data forwarding**

  In an NDIS WAN environment, NDISWAN examines the header of the descriptor of a send packet and determines over which link the packet will be sent. NDISWAN copies the packet into a contiguous buffer and forwards it to the underlying miniport driver. In a CoNDIS WAN environment, NDISWAN forwards packets based on the packet's associated virtual connection (VC). For more information about WAN driver links and connections, see WAN Driver Bindings and Connections.

WAN Driver Bindings and Connections

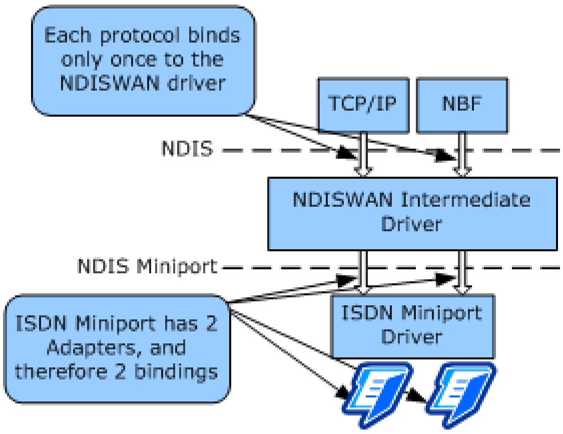
Article • 12/15/2021

This topic provides an overview of bindings and connections between NDISWAN, overlying protocol drivers, and the underlying WAN miniport drivers.

Bindings

NDISWAN binds to one or more WAN miniport drivers and one or more protocol drivers bind to NDISWAN.

The following figure illustrates the binding relationships between WAN client protocol drivers, NDISWAN, and WAN miniport drivers.



Each protocol binds only once to the NDISWAN driver
TCP/IP
NBF
NDIS
NDISWAN Intermediate Driver
NDIS Miniport
ISDN Miniport has 2 Adapters, and therefore 2 bindings
ISDN Miniport Driver

Protocol drivers bind once to NDISWAN and do not bind to WAN miniport drivers. This type of binding saves memory and simplifies WAN miniport drivers. Because there are typically several protocol drivers in a given system and there could be more than one WAN miniport driver, the reduction in the number of bindings saves memory. That is, each protocol does not have to bind to each WAN miniport driver. Also, because protocol drivers can rely on only having a single WAN binding, these protocol drivers can be simplified.

Connections

NDIS WAN and CoNDIS WAN miniport drivers implement different models for connections:

- An NDIS WAN miniport driver uses links to send and receive data. Links are logical, point-to-point bidirectional communication channels. There can be many links per NIC. Links are dynamically established and torn down. The link speed and quality of the link can vary for each connection. However, the padding and link parameters must be the same for all links that a NIC supports. For example, if an NDIS WAN miniport driver specifies a 20-byte header padding and 4-byte tail padding, this padding must remain constant for all links that the miniport driver's NIC supports.

- A CoNDIS WAN miniport driver sends and receives data over virtual connections (VCs). There can be many VCs per NIC. While the data transmission speed can vary from VC to VC, the other VC parameters are the same for all VCs that the NIC supports. A CoNDIS WAN miniport driver can specify a maximum frame size for any net packet that the miniport driver can send and receive. If the miniport driver specifies a maximum frame size, that maximum frame size must remain constant for all VCs on that NIC.

Like other miniport drivers, every WAN miniport driver must have at least one NIC for which it allocates and maintains a NIC-specific context area. The NIC-specific context area is simply a way to store, retrieve, and use information about the hardware specifics of the NIC (such as interrupt, bus type, I/O range, and memory) and to maintain the run-time state for connections. A miniport driver should specify one NIC-specific context area for each network card in the system that it supports.

If a particular WAN miniport driver specifies that it does not require PPP address and control-field compression, it is assumed true for all connections on the miniport driver's NIC.

Before a WAN miniport driver can send or receive packets on a wide area network, a connection must be created:

- In an NDIS environment, an application must set up a connection that originates on the sending node or accept a connection that originates on a remote node by making or accepting a call. The setup, supervision, and tear-down of a connection is done through TAPI. TAPI requests and status indications to TAPI all go through NDISTAPI. For more information about TAPI and NDISTAPI, see NDISTAPI Overview.

- In a CoNDIS environment, a VC must be created. The NDPROXY driver creates a VC for an outgoing call that an application originated. Similarly, a call manager (or MCM) initiates the creation of a VC for an incoming call that the call manager indicates to NDISWAN and NDPROXY. The call manager must communicate and sometimes negotiate the parameters for the VC with the remote party. The setup, supervision, and tear-down of a connection is done through TAPI. TAPI requests and status indications to TAPI all go through NDPROXY. For more information about TAPI and NDPROXY, see NDPROXY Overview.

NDISTAPI Overview

Article • 03/14/2023

NDISTAPI is a system-provided driver that interfaces NDISWAN and NDIS WAN miniport drivers to the TAPI services. NDIS WAN miniport drivers are not supported for NDIS 5.0 and later drivers. New drivers should be based on the CoNDIS WAN driver architecture.

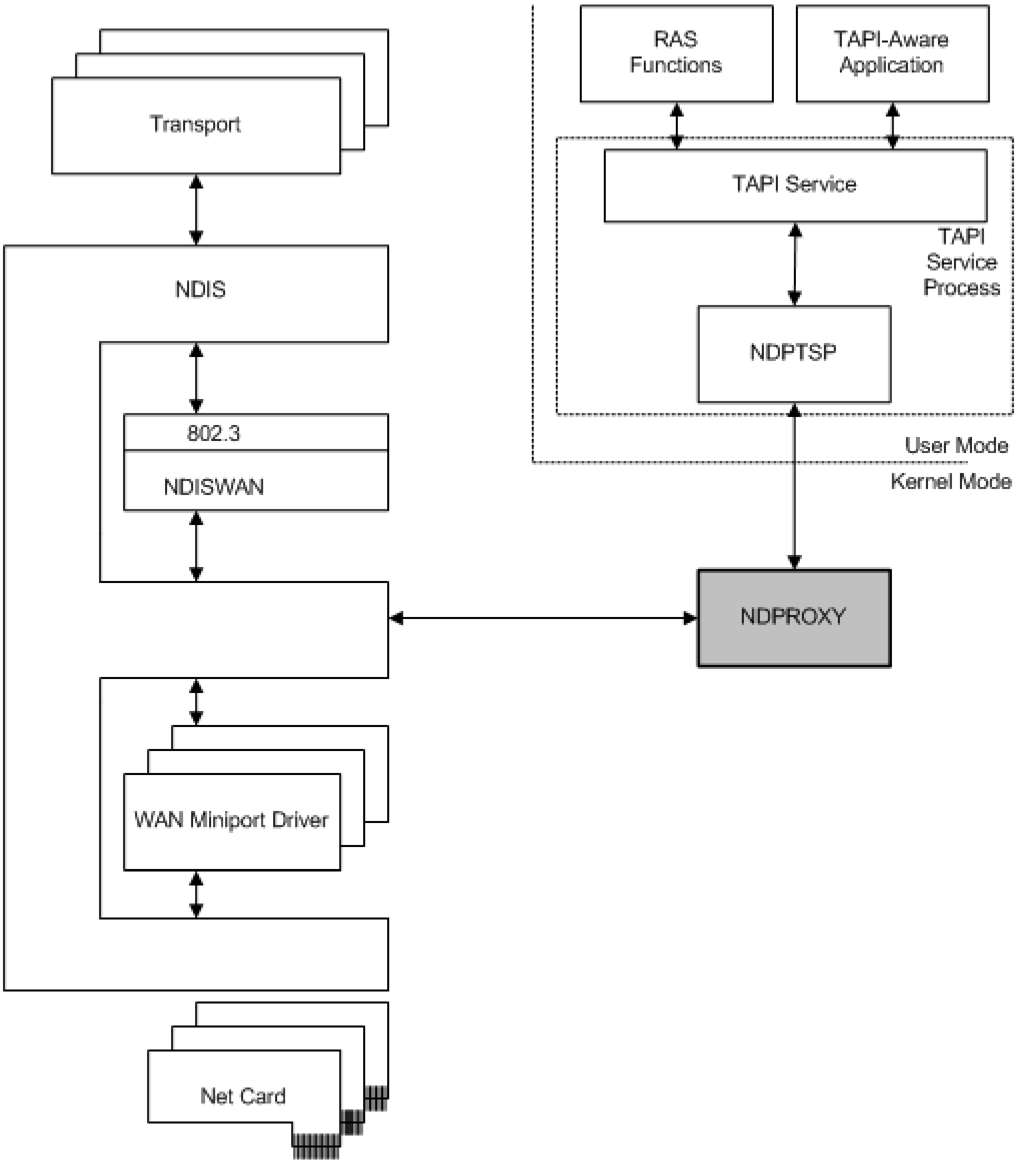
NDPROXY Overview

Article • 12/15/2021

Note If you are reading this page because of the 27 November 2013 Microsoft Security Advisory (2914486) affecting Windows XP and Windows Server 2003, you may find this Trustworthy Computing blog post helpful.

NDPROXY is a system-provided driver that interfaces NDISWAN and CoNDIS WAN drivers (WAN miniport drivers, call managers, and miniport call managers) to the TAPI services. This topic introduces NDPROXY operations that are further documented in CoNDIS WAN Operations that Support Telephonic Services.

The following figure shows how NDPROXY interfaces with other components in the RAS architecture.



NDPROXY provides the kernel-mode component of the service provider interface (SPI) for CoNDIS WAN. TAPI-aware applications make user-mode TAPI requests and the TAPI service routes these requests to NDPTSP. NDPTSP converts the user-mode TAPI service requests to kernel-mode SPI requests and passes the SPI requests to NDPROXY.

NDPROXY communicates through NDIS with the NDISWAN driver and one of the following:

- A miniport driver with a separate call manager
- An integrated miniport call manager (MCM)

The miniport driver interface and call manager interface to NDISWAN and NDPROXY are the same regardless of the configuration.

Note You can use the miniport driver with a separate call manager in situations where multiple hardware platforms need to be supported. In this situation, the same call manager can be used in combination with multiple miniport drivers to simplify development.

The following list summarizes the interfaces that exist between NDPROXY and the other components in the CoNDIS WAN driver stack:

- NDPROXY presents a connection-oriented client interface to CoNDIS WAN miniport drivers and a call manager interface to NDISWAN.
- NDISWAN presents a connection-oriented client interface to NDPROXY, CoNDIS WAN miniport drivers, and MCMs.
- CoNDIS WAN call managers or MCMs present a call manager interface to NDPROXY.
- CoNDIS WAN miniport drivers and MCMs present a CoNDIS miniport driver interface to NDISWAN.

For more information about connection-oriented clients, call managers, miniport drivers, and MCMs, see Connection-Oriented Environment.

NDPROXY calls the **NdisCoOidRequest** function with connection-oriented TAPI OIDs to determine the capabilities of a CoNDIS WAN miniport driver. NDPROXY also registers the TAPI-specific address family, creates virtual connections (VCs), makes and accepts calls, and activates VCs so that data can be sent and received on those VCs. For more information about handling OID requests in the CoNDIS WAN miniport driver, see Handling Queries in a CoNDIS WAN Miniport Driver and Setting CoNDIS WAN Miniport Driver Information.

Implementing CoNDIS WAN Miniport Drivers

Article • 03/14/2023

CoNDIS WAN miniport drivers are the same as other CoNDIS drivers with the exception of features added to support WAN operations. For more information about connection-oriented NDIS, see Connection-Oriented NDIS. For more information about features that support WAN operations, see WAN-Specific Capabilities of CoNDIS WAN Drivers.

The following topics provide information specific to CoNDIS WAN miniport drivers:

Registering CoNDIS WAN Drivers

Registering the WAN Address Family

Handling Queries in a CoNDIS WAN Miniport Driver

Setting CoNDIS WAN Miniport Driver Information

Sending Packets from a CoNDIS WAN Miniport Driver

Indicating Received Data from a CoNDIS WAN Miniport Driver

Indicating CoNDIS WAN Miniport Driver Status

WAN-Specific Capabilities of CoNDIS WAN Drivers

Article • 03/14/2023

CoNDIS WAN drivers differ from a non-WAN CoNDIS drivers as follows:

- CoNDIS WAN drivers that support TAPI services use the CO_ADDRESS_FAMILY_TAPI_PROXY address family.
- CoNDIS WAN drivers support WAN-specific OIDs: OID_WAN_PERMANENT_ADDRESS, OID_WAN_CURRENT_ADDRESS, and OID_WAN_MEDIUM_SUBTYPE.
- CoNDIS WAN miniport drivers support a set of CoNDIS WAN OIDs to set and query operating characteristics. For more information about CoNDIS WAN OIDs, see CoNDIS WAN Objects.
- CoNDIS WAN miniport drivers that provide TAPI services support a set of CoNDIS TAPI OIDs to set and query operating characteristics. For more information about CoNDIS TAPI OIDs, see TAPI Extensions for Connection-Oriented NDIS.
- CoNDIS WAN miniport drivers support a set of WAN-specific status indications that denote changes in the status of a link. For more information about CoNDIS WAN miniport driver status indications, see Indicating CoNDIS WAN Miniport Driver Status.
- CoNDIS WAN miniport drivers keep a WAN-specific set of statistics. The OID_WAN_CO_GET_STATS_INFO OID requests the miniport driver to return the statistics information.
- CoNDIS WAN miniport drivers never attempt to loop back any packets; NDISWAN provides loop-back support.

Registering CoNDIS WAN Drivers

Article • 03/14/2023

A CoNDIS WAN miniport driver or MCM calls **NdisMRegisterMiniportDriver** from its **DriverEntry** function to register its standard *MiniportXxx* functions with NDIS. For more information about registering *MiniportXxx* functions, see Initializing a Miniport Driver.

A CoNDIS WAN call manager is an NDIS protocol driver. As such, a call manager calls **NdisRegisterProtocolDriver** to register its standard *ProtocolXxx* functions. For more information about registering an NDIS protocol driver, see Initializing a Protocol Driver. For information about other differences between call manager initialization and MCM initialization, see Differences in Initialization.

The call to **NdisMRegisterMiniportDriver** provides an NDIS_MINIPORT_DRIVER_CHARACTERISTICS structure from the miniport driver. You must specify the correct NDIS version number. For more information about setting the NDIS version number, see **NDIS_MINIPORT_DRIVER_CHARACTERISTICS**.

CoNDIS WAN drivers must indicate NDIS version 5.0 or later.

NDIS 6.0 and later drivers must register CoNDIS callback functions as follows:

- To register CoNDIS *ProtocolXxx* and *MiniportXxx* functions, all CoNDIS drivers must call the **NdisSetOptionalHandlers** function.

- To register its CoNDIS *MiniportXxx* functions, a miniport driver or miniport call manager (MCM) must call the **NdisSetOptionalHandlers** function from its *MiniportSetOptions* function and pass it an **NDIS_MINIPORT_CO_CHARACTERISTICS** structure. To register call manager *ProtocolXxx* functions, MCMs also provide an **NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS** structure.

- To register its CoNDIS *ProtocolXxx* functions, a client or call managers must call the **NdisSetOptionalHandlers** function from its *ProtocolSetOptions* function and must provide an **NDIS_PROTOCOL_CO_CHARACTERISTICS** structure. Clients must also provide an **NDIS_CO_CLIENT_OPTIONAL_HANDLERS** structure and call managers must also provide an **NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS** structure.

For more information about CoNDIS driver registration, see CoNDIS Registration.

.

Registering the WAN Address Family

Article • 12/15/2021

This topic describes how to register the TAPI address family from a CoNDIS WAN miniport call manager (MCM) or a separate call manager.

Either type of call manager calls the **NdisCmRegisterAddressFamilyEx** function to register its call manager entry points and the address family type CO_ADDRESS_FAMILY_TAPI_PROXY. By doing so, the driver indicates that it provides TAPI services. For more information about registering an address family in a CoNDIS driver, see Registering and Opening an Address Family.

NDIS notifies NDPROXY of the newly-registered address family. NDPROXY determines that it can use the TAPI services that the call manager provides. NDPROXY opens the TAPI-proxy address family that is associated with the driver and registers NDPROXY's connection-oriented entry points with NDIS. These entry points are used to communicate with the driver.

NDPROXY can enumerate the TAPI capabilities of the miniport driver and later send TAPI requests that are encapsulated in NDIS structures. For details about using CoNDIS extensions for TAPI support, see CoNDIS WAN Operations That Support Telephonic Services.

Handling Queries in a CoNDIS WAN Miniport Driver

Article • 03/14/2023

This topic provides an overview of the requirements for handling queries in a CoNDIS WAN miniport driver. An upper-layer driver calls **NdisCoOidRequest** with a query request to determine WAN-specific capabilities and current status of a CoNDIS WAN miniport driver and the miniport driver's NIC.

After the NDISWAN intermediate driver forwards the query request, NDIS calls the miniport driver's **MiniportCoOidRequest** function. In a CoNDIS WAN miniport driver, this function is the same as in any connection-oriented miniport driver, except that the CoNDIS WAN miniport driver supports CoNDIS WAN Objects.

If the CoNDIS WAN miniport driver completes *MiniportCoOidRequest* asynchronously by returning a status of NDIS_STATUS_PENDING, it must complete the query later by calling **NdisCoOidRequestComplete**.

When NDIS calls *MiniportCoOidRequest*, NDIS passes a pointer to the **NDIS_OID_REQUEST** structure that contains the query OID and a buffer to hold the information retrieved from the miniport driver. The miniport driver controls this buffer until the request completes. If the number of bytes specified in the **InformationBufferLength** member of NDIS_OID_REQUEST is insufficient for the information that the OID requires, the miniport driver should fail the query request and set the **BytesNeeded** member of NDIS_OID_REQUEST to the number of bytes that the OID requires.

No other requests will be submitted to the particular WAN miniport driver until the current query request completes.

The following table summarizes the OIDs used to get or set operational characteristics for CoNDIS WAN miniport drivers.

Name	Optional or Required
OID_WAN_CO_GET_INFO Get information about virtual connections (VCs).	Required
OID_WAN_CO_GET_LINK_INFO Get information about a VC.	Required
OID_WAN_CO_GET_STATS_INFO Get statistics information for a VC.	Optional

A CoNDIS WAN miniport driver can support all of the NDIS General Objects. To learn more about setting information in a CoNDIS miniport driver, see Querying or Setting Information.

Setting CoNDIS WAN Miniport Driver Information

Article • 03/14/2023

This topic provides an overview of the requirements for setting information in a CoNDIS WAN miniport driver. An upper-layer driver calls **NdisCoOidRequest** with a set request to change information that a CoNDIS WAN miniport driver and the miniport driver's NIC maintain.

After the NDISWAN intermediate driver forwards the set request, NDIS calls the WAN miniport driver's **MiniportCoOidRequest** function. In a CoNDIS WAN miniport driver, this function is the same as in any CoNDIS miniport driver, except that the CoNDIS WAN miniport driver supports CoNDIS WAN Objects.

No other requests will be submitted to the CoNDIS WAN miniport driver until the current set request is complete. If the miniport driver does not immediately complete the set request, it returns NDIS_STATUS_PENDING from *MiniportCoOidRequest* and must later call **NdisCoOidRequestComplete** to complete the request.

A CoNDIS WAN miniport driver must recognize and respond properly to the following CoNDIS WAN OIDs.

Name	Optional or Required
OID_WAN_CO_SET_LINK_INFO Set information for a VC.	Required

A CoNDIS WAN miniport driver also supports the NDIS General Objects. To learn more about setting information in a CoNDIS miniport driver, see Querying or Setting Information.

Sending Packets from a CoNDIS WAN Miniport Driver

Article • 03/14/2023

An upper-layer driver calls NdisCoSendNetBufferLists to send network data packets to an underlying CoNDIS WAN miniport driver in a list of NET_BUFFER_LIST structures. The NDISWAN intermediate driver forwards those NET_BUFFER_LIST structures from the upper-layer driver. NDISWAN repackages the structures before sending them. NDISWAN forwards packets in new NET_BUFFER_LIST structures.

The NDISWAN intermediate driver calls NDIS to forward the new NET_BUFFER_LIST structures, NDIS calls the WAN miniport driver's MiniportCoSendNetBufferLists function.

The CoNDIS WAN miniport driver owns both the NET_BUFFER_LIST structures and associated data until the send completes. The miniport driver must later call NdisMSendNetBufferListsComplete to complete the send request.

A completion call does not necessarily indicate that the network datahas been transmitted; however with the exception of intelligent NICs, the network data usually has been transmitted. A completion call does however, indicate that the miniport driver is ready to release ownership of the NET_BUFFER_LIST structures.

After the CoNDIS WAN miniport driver receives NET_BUFFER_LIST structure that contains a network data packet, it should send the packet out on an active virtual connection (VC).

A CoNDIS WAN miniport driver specifies the number of outstanding packets that it can have per VC in the **MaxSendWindow** member of the NDIS_WAN_CO_INFO structure. The miniport driver provides this structure when the miniport driver responds to the OID_WAN_CO_GET_INFO request from the protocol driver. However, the miniport driver can adjust this number dynamically and on a per-VC basis by using the **SendWindow** member in the WAN_CO_LINKPARAMS structure. The miniport driver passes this structure to the NdisMCoIndicateStatusEx function. NDISWAN uses the current **SendWindow** value as its limit on outstanding sends. The miniport driver can set the value of the **SendWindow** member to zero to specify that it cannot handle any outstanding packets. That is, if the **SendWindow** member is set to zero, the send window is shut down and NDISWAN stops sending packets for the particular VC.

Packets that a WAN miniport driver sends contain simple HDLC PPP framing if PPP framing is set. For SLIP or RAS framing, packets contain only the data portion with no framing whatsoever. For more information about WAN packet framing, see WAN Packet Framing.

A WAN miniport driver must not attempt to provide software loopback or promiscuous-mode loopback. Both of these loopback types are fully supported by the NDISWAN driver.

Indicating Received Data from a CoNDIS WAN Miniport Driver

Article • 03/14/2023

The following operations occur when a CoNDIS WAN miniport driver receives a network data packet:

1. The driver removes driver-specific encapsulation from the network data packet, if necessary before calling NdisMCoIndicateReceiveNetBufferLists to indicate the received data in a NET_BUFFER_LIST structure. For example, the driver can remove PPPoE encapsulation. However, the miniport driver should leave encapsulated data, such as PPP header and payload, intact.

2. The driver calls the **NdisMCoIndicateReceiveNetBufferLists** function to indicate to NDISWAN that a packet has arrived.

3. NDISWAN processes the packet and calls NdisMIndicateReceiveNetBufferLists to indicate the arrival of the packet.

4. To forward the packet, NDIS calls the ProtocolReceiveNetBufferLists function of bound overlying protocol drivers.

Indicating CoNDIS WAN Miniport Driver Status

Article • 03/14/2023

A CoNDIS WAN miniport driver calls **NdisMCoIndicateStatusEx** to indicate status changes up to bound protocol drivers. For more information about indicating status from a CoNDIS miniport driver or MCM, see Indicating Miniport Driver Status.

Bound protocol drivers can ignore these status indications. However, processing these indications typically results in improved performance for protocol drivers and the miniport driver.

The NDISWAN intermediate driver forwards status indications to NDIS. NDIS calls the **ProtocolCoStatusEx** functions of bound protocol drivers or a configuration manager. These protocol drivers or configuration manager can log these indications and possibly take corrective action, if necessary.

For a CoNDIS WAN miniport driver, a call to **NdisMCoIndicateStatusEx** is the same as in any CoNDIS miniport driver, except that the CoNDIS WAN miniport driver indicates a WAN-specific status for each virtual connection (VC) on the miniport driver's NIC. The miniport driver calls **NdisMCoIndicateStatusEx** with an explicit VC handle to indicate these changes up to a protocol driver that shares this VC. If the driver specifies a **NULL***NdisVcHandle*, the status pertains to a general change in the state of the NIC.

Each status indication provides two basic pieces of information:

- A status code that specifies the general status. There are a limited number of defined general status codes; this list is subject to future expansion.

- A buffer that contains the status information. This status information can be specific to a NIC, or, for a CoNDIS WAN miniport driver, specific to a VC on a NIC. For example, a buffer might contain the new transmit speed of an X.25 connection, which recently decreased by a factor of two.

The CoNDIS WAN VC status indications are:

- NDIS_STATUS_WAN_CO_LINKPARAMS

  A CoNDIS WAN miniport driver calls **NdisMCoIndicateStatusEx** to indicate that the parameters for a particular VC that is active on the NIC have changed. In this call, the miniport driver passes the handle to the VC in the *NdisVcHandle* parameter, NDIS_STATUS_WAN_CO_LINKPARAMS in the *GeneralStatus* parameter, and a pointer to a **WAN_CO_LINKPARAMS** structure in the *StatusBuffer* parameter. WAN_CO_LINKPARAMS describes new parameters for the VC.

- NDIS_STATUS_WAN_CO_FRAGMENT

  A CoNDIS WAN miniport driver calls **NdisMCoIndicateStatusEx** to indicate that it has received a partial packet from the endpoint of a VC. In this call, the miniport driver passes the handle to the VC in the *NdisVcHandle* parameter, NDIS_STATUS_WAN_CO_FRAGMENT in the *GeneralStatus* parameter, and a pointer to an **NDIS_WAN_CO_FRAGMENT** structure in the *StatusBuffer* parameter. NDIS_WAN_CO_FRAGMENT describes the reason that the partial packet was received.

  After this indication occurs, a connection-oriented client should send frames to the connection-oriented client at the other end of the VC. These frames will notify the opposite endpoint of the partial-packet situation, so that the opposite endpoint is not required to wait for a time-out to occur.

  NDISWAN monitors dropped packets by counting the number of fragment indications on each VC.

CoNDIS WAN Operations that Support Telephonic Services

Article • 03/14/2023

This section describes how CoNDIS WAN miniport drivers implement telephonic services using NDIS functions in a connection-oriented environment. CoNDIS WAN miniport drivers communicate through NDIS with the NDPROXY and NDISWAN drivers. The NDPROXY driver communicates with telephony applications through a telephony service provider. For more information, see the Telephony Application Programming Interface (TAPI).

The following topics describe the NDPROXY driver more fully. These topics also describe how a CoNDIS WAN miniport driver registers and enumerates its TAPI capabilities, how it brings up lines, and how it sets up and closes calls that are initiated by TAPI requests:

NDPROXY Overview

CoNDIS TAPI Registration

CoNDIS TAPI Initialization

Making Outgoing Calls

Accepting Incoming Calls

CoNDIS TAPI Shutdown

Call Manager Requirements for Voice Streaming

Non-WAN-Specific Extensions to Support Telephonic Services Over Connection-Oriented NDIS

These descriptions briefly discuss the concepts embodied in TAPI, but the reader should consult the Windows SDK for details about TAPI. For more information about how TAPI models line devices and how all WAN miniport drivers should maintain the state of their connections, see Line Devices, Addresses, and Calls (NDIS 5.1) and Maintaining State Information (NDIS 5.1).

CoNDIS TAPI Registration

Article • 03/14/2023

This section discusses how a CoNDIS WAN miniport driver indicates that it supports TAPI services and how it sets up TAPI-specific communications with the NDISWAN and NDPROXY drivers.

After a CoNDIS WAN miniport driver has registered its miniport driver entry points for one or more NICs, the following operations cause the NDISWAN and NDPROXY drivers to become associated, in a TAPI-specific way, with those NICs.

- The CoNDIS WAN miniport driver calls the **NdisMCmRegisterAddressFamilyEx** function from within its *MiniportInitializeEx* function to register its call manager entry points and the address family type CO_ADDRESS_FAMILY_TAPI_PROXY. By doing so, the miniport driver advertises that it provides TAPI services.

- NDIS calls NDPROXY's **ProtocolCoAfRegisterNotify** function to notify NDPROXY of the newly registered address family. NDPROXY's *ProtocolCoAfRegisterNotify* examines the address-family data and determines that it can use the TAPI services provided by the call manager that is integrated into the CoNDIS WAN miniport driver. A TAPI-capable CoNDIS WAN miniport driver is an integrated miniport call manager (MCM) driver.

- NDPROXY calls the **NdisClOpenAddressFamilyEx** function to open the TAPI-proxy address family that is associated with the CoNDIS WAN miniport driver. **NdisClOpenAddressFamilyEx** registers NDPROXY's connection-oriented entry points with NDIS. These entry points are used to communicate with a TAPI-capable CoNDIS WAN miniport driver.

- NDPROXY calls **NdisCmRegisterAddressFamilyEx** to register its call manager entry points and the address family type CO_ADDRESS_FAMILY_TAPI. By doing so, NDPROXY advertises that it implements TAPI services.

- NDIS calls NDISWAN's *ProtocolCoAfRegisterNotify* function to notify NDISWAN of the newly registered address family. NDISWAN's *ProtocolCoAfRegisterNotify* examines the address-family data and determines that NDISWAN can use the TAPI services provided by NDPROXY.

- NDISWAN calls the **NdisClOpenAddressFamilyEx** function to open the TAPI address family that is associated with NDPROXY. **NdisClOpenAddressFamilyEx** registers NDISWAN's connection-oriented entry points with NDIS. These entry points are used to communicate with NDPROXY.

- NDISWAN calls the **NdisClRegisterSap** function to inform NDPROXY that NDISWAN can accept incoming calls on a particular Service Access Point (SAP). In this call, NDISWAN passes a **CO_SAP** structure that describes the SAP. NDISWAN sets the **SapType** member of CO_SAP to AF_TAPI_SAP_TYPE to specify that the SAP will be used for TAPI calls. NDISWAN sets the **Sap** member of CO_SAP to a string for a particular TAPI device class. A TAPI application provides this string when the application calls the TAPI **lineGetID** function. NDPROXY should notify NDISWAN about all incoming calls addressed to the SAP.

CoNDIS TAPI Initialization

Article • 03/14/2023

This section discusses how a CoNDIS WAN miniport driver enumerates its TAPI capabilities for applications. These TAPI capabilities consist of:

- Number of line devices the miniport driver supports--line devices include, for example, a modem, a fax board, and an ISDN card.
- Information for specific lines--line information includes, for example, a line identifier and the number of channel addresses (telephone numbers) the line supports for simultaneous transmission of voice and data.
- Information for specific channel addresses on lines of devices--address information includes, for example, the identity of a caller (Caller ID) and the number of active calls possible.

To retrieve information about underlying hardware, NDPROXY issues requests for line and channel-address capabilities. That is, the NDPROXY driver queries the TAPI capabilities of a CoNDIS WAN miniport driver. The NDPROXY driver calls the NdisCoOidRequest function to query the TAPI capabilities of the miniport driver. In this call, NDPROXY passes an NDIS_OID_REQUEST structure. NDPROXY specifies the following in NDIS_OID_REQUEST:

- **NdisRequestQueryInformation** value in the **RequestType** member
- Object identifier (OID) that specifies the TAPI capability to retrieve from the miniport driver in the **Oid** member
- Buffer to hold the TAPI-capability information that is returned in the **InformationBuffer** member

All queries sent to a CoNDIS WAN miniport driver by the NDPROXY driver can be completed either synchronously or asynchronously. If a CoNDIS WAN miniport driver determines that it cannot complete the query immediately, then it can simply return NDIS_STATUS_PENDING and call the NdisMCmOidRequestComplete function from within its *ProtocolCoOidRequest* function when it has completed the query.

After a CoNDIS WAN miniport driver notifies NDPROXY about the registration of a new address family as specified in CoNDIS TAPI Registration, NDPROXY queries the following OIDs to determine the TAPI-specific capabilities of the CoNDIS WAN miniport driver and the miniport driver's NIC.

- NDPROXY queries the miniport driver with OID_CO_TAPI_CM_CAPS to determine the number of lines supported by the miniport driver's device (the device for which it provides TAPI services). This OID also requests the miniport driver to indicate whether these lines have dissimilar line capabilities.

- NDPROXY next queries the miniport driver with OID_CO_TAPI_LINE_CAPS to determine the telephony capabilities for the specified line. This OID also requests the miniport driver to indicate whether addresses on this line have dissimilar address capabilities.
    - If the previous query of OID_CO_TAPI_CM_CAPS indicated that the miniport driver's device supports only one line, or if the device supports multiple lines that have the same line capabilities, NDPROXY has to query OID_CO_TAPI_LINE_CAPS only once to obtain the line capabilities of the device. In this case, the line capabilities returned by the miniport driver apply to all lines on the device.
    - If the device supports multiple lines with dissimilar line capabilities, NDPROXY must query OID_CO_TAPI_LINE_CAPS once for each line to obtain the line capabilities of each line.

- Finally, NDPROXY queries the miniport driver with OID_CO_TAPI_ADDRESS_CAPS to determine the telephony capabilities for a specified address on a specified line.
    - If the previous query of OID_CO_TAPI_LINE_CAPS indicated that the line supports only one address or that all addresses on the line have the same address capabilities, NDPROXY queries OID_CO_TAPI_ADDRESS_CAPS only once to determine the capabilities of all the addresses on the line.
    - If a line supports multiple addresses that have dissimilar capabilities, NDPROXY queries OID_CO_TAPI_ADDRESS_CAPS once for each address on the line.

The NDPROXY driver uses the information obtained with the TAPI enumeration OIDs to do the following:

- Create TAPI parameters for subsequent TAPI calls.

- Determine whether to accept or reject subsequent incoming TAPI calls.

- Register one or more TAPI service access points (SAPs) on which to receive subsequent incoming TAPI calls.

Making Outgoing Calls

Article • 12/15/2021

If an application attempts to make an outgoing call, it must first open a line. A line is opened as a result of an application calling the TAPI **lineOpen** function. To place a telephony call on the previously opened line, the application calls the TAPI **lineMakeCall** function and passes a pointer to the specific destination address. If anything but default call-setup parameters are requested, the application also passes a pointer to a LINECALLPARAMS structure. If the application uses default call-setup parameters, **lineMakeCall** provides those parameters in a LINECALLPARAMS structure. Members of this structure specify how the telephony call should be set up.

These TAPI-function calls cause the NDPROXY driver to first create a virtual connection (VC) with the CoNDIS WAN miniport driver and then to encapsulate TAPI parameters in NDIS structures in order to make the outgoing call. The miniport driver will use these TAPI parameters to set up the outgoing call. The following describes how the outgoing call is connected, set up, and made:

- NDPROXY calls NdisCoCreateVc to initiate the creation of the VC with the miniport driver. After NDPROXY calls **NdisCoCreateVc**, NDIS calls, as a synchronous operation, the *ProtocolCoCreateVc* function of the call manager integrated into the miniport driver. NDIS passes to *ProtocolCoCreateVc* a handle that represents the VC. If the call to **NdisCoCreateVc** is successful, NDIS fills and returns the VC handle. *ProtocolCoCreateVc* performs any necessary allocations of dynamic resources and structures that the miniport call manager (MCM) driver requires to perform subsequent operations on the VC that will later be activated. Such resources include, but are not limited to, memory buffers, data structures, events, and other such similar resources.

- NDPROXY specifies the TAPI parameters for an outgoing call in a CO_AF_TAPI_MAKE_CALL_PARAMETERS structure. NDPROXY fills this structure's members with the following information that was passed in the TAPI **lineMakeCall** function:
  - The destination address in the **DestAddress** member
  - The open-line identifier in the **ulLineID** member
  - The LINECALLPARAMS structure in the **LineCallParams** member

- NDPROXY overlays the CO_AF_TAPI_MAKE_CALL_PARAMETERS structure on the **Parameters** member of a CO_SPECIFIC_PARAMETERS structure and sets the **Length** member of CO_SPECIFIC_PARAMETERS to the size of CO_AF_TAPI_MAKE_CALL_PARAMETERS.

- NDPROXY sets the CO_SPECIFIC_PARAMETERS structure to the **MediaSpecific** member of a CO_MEDIA_PARAMETERS structure.

- NDPROXY sets a pointer to the CO_MEDIA_PARAMETERS structure to the **MediaParameters** member of a CO_CALL_PARAMETERS structure.

- Once NDPROXY encapsulates TAPI parameters, NDPROXY calls the NdisClMakeCall function to initiate the outgoing call. In this function call, NDPROXY passes a pointer to the filled CO_CALL_PARAMETERS structure. NDIS in turn calls the ProtocolCmMakeCall function of the CoNDIS WAN miniport driver's call manager. The miniport driver should examine only the CO_AF_TAPI_MAKE_CALL_PARAMETERS structure embedded in CO_CALL_PARAMETERS. No other call parameters are meaningful in this case. If the miniport driver subsequently activates the VC for the outgoing call, the miniport driver calls the NdisMCmActivateVc function and passes a pointer to the filled CO_CALL_PARAMETERS.

- After the miniport driver has negotiated with the network to establish the telephony-call parameters for the VC and set up a NIC for those call parameters, the miniport driver calls the NdisMCmMakeCallComplete function to indicate that it is ready to make data transfers on the VC. In this call, the miniport driver must pass the handle to the VC and modifications made to telephony-call parameters.

- The miniport driver must modify the **CallMgrParameters** member of the CO_CALL_PARAMETERS structure to specify the quality of service (QoS) of transferring packets, such as the bandwidth. To set this **CallMgrParameters** member, the miniport driver fills members of a CO_CALL_MANAGER_PARAMETERS structure and points this structure to **CallMgrParameters**. For example, to identify the transmit and receive speeds in bytes per second for the VC, the miniport driver must set the **PeakBandwidth** members of the **Transmit** and **Receive** members of CO_CALL_MANAGER_PARAMETERS. The **Transmit** and **Receive** members are FLOWSPEC structures. For more information about the FLOWSPEC structure, see the Microsoft Windows SDK.

- If the miniport driver has modified telephony-call parameters, it must set the **Flags** member in the CO_CALL_PARAMETERS structure with CALL_PARAMETERS_CHANGED. As a result of the **NdisMCmMakeCallComplete** call made by the miniport driver, NDIS calls NDPROXY's *ProtocolClMakeCallComplete* function to complete the asynchronous operations that were initiated with **NdisClMakeCall**.

- After the miniport driver successfully completes the outgoing call, NDPROXY notifies a TAPI application that the call is connected. This TAPI application then calls the TAPI **lineGetID** function to inform NDPROXY to locate the appropriate CoNDIS client. In this **lineGetID** call, the TAPI application supplies a string for a particular TAPI device class to which the application requires a handle. NDPROXY uses this string to locate the CoNDIS client that previously registered a SAP for the particular TAPI device class. If the CoNDIS client is NDISWAN, the string is NDIS. If NDPROXY locates a SAP with a string that matches the string passed by the TAPI application, NDPROXY calls NdisMCmCreateVc to set up a connection endpoint with NDISWAN on which it can dispatch notification of the outgoing call that was made. NDIS in turn calls NDISWAN's *ProtocolCoCreateVc* function and passes a handle that represents the VC.

- After NDPROXY sets up the connection endpoint with NDISWAN, it calls the NdisCmDispatchIncomingCall function to notify NDISWAN about the outgoing call. In this call, NDPROXY passes the encapsulated CO_AF_TAPI_MAKE_CALL_PARAMETERS structure that contains the outgoing call parameters. NDIS in turn calls NDISWAN's *ProtocolClIncomingCall* function, within which NDISWAN either accepts or rejects the requested connection. If NDISWAN changes the call parameters passed to it, it must set the **Flags** member in the CO_CALL_PARAMETERS structure with CALL_PARAMETERS_CHANGED.

- After deciding whether to accept the connection and after possibly changing the call parameters, NDISWAN calls the NdisClIncomingCallComplete function. NDIS in turn calls the miniport driver's *ProtocolCmIncomingCallComplete* function. Depending on whether NDISWAN accepted the outgoing call and whether the miniport driver accepts or rejects NDISWAN's proposed changes to the call parameters, the miniport driver calls either NdisCmDispatchCallConnected or NdisCmDispatchIncomingCloseCall functions. **NdisCmDispatchCallConnected** notifies NDISWAN that data transfers can begin on the VC that NDPROXY created for the outgoing call. **NdisCmDispatchIncomingCloseCall** informs NDISWAN and NDPROXY to tear down the proposed outgoing call.

- After NDISWAN accepts the outgoing call, NDPROXY calls the NdisCoGetTapiCallId function to retrieve a string that identifies NDISWAN's context for the VC. NDPROXY passes this string back to the TAPI application. The TAPI application uses this VC-context string to complete its call to **lineGetID**.

Accepting Incoming Calls

Article • 12/15/2021

Before an application can accept an incoming call, it first must have a line open. A line is opened as a result of an application calling the TAPI **lineOpen** function. This TAPI-function call causes underlying drivers to encapsulate TAPI parameters in NDIS structures in order to prepare to receive an incoming call. After the CoNDIS WAN miniport driver receives an incoming call, the miniport driver must first create a virtual connection (VC) with the NDPROXY driver and then notify NDPROXY of the incoming call. NDPROXY in turn notifies the application through TAPI. The following list describes how the incoming call is set up, connected, and made:

- NDPROXY specifies the TAPI parameters for an incoming connection in a CO_AF_TAPI_SAP structure. NDPROXY fills this structure's members with the following information that was passed in the TAPI **lineOpen** function:
  - Open-line identifier in the **ulLineID** member
  - Address of the incoming connection in the **ulAddressID** member
  - Media mode of the incoming connection's information stream in the **ulMediaModes** member

- NDPROXY overlays the CO_AF_TAPI_SAP structure on the **Sap** member of a CO_SAP structure and sets the **SapLength** member of CO_SAP to the size of CO_AF_TAPI_SAP. NDPROXY must also set the **SapType** member of CO_SAP to AF_TAPI_SAP_TYPE.

- Once NDPROXY encapsulates TAPI parameters, NDPROXY calls the NdisClRegisterSap function to make itself ready to receive incoming calls. In this function call, NDPROXY passes a pointer to the filled CO_SAP structure that specifies the Service Access Point (SAP) on which NDPROXY can receive incoming calls. NDIS forwards the CO_SAP structure to the *ProtocolCmRegisterSap* function of the CoNDIS WAN miniport call manager (MCM) driver. *ProtocolCmRegisterSap* communicates with network control devices or other media-specific agents, as necessary, to register the SAP on the network for NDPROXY. After the miniport driver has registered the SAP, it can accept an incoming-call offer directed to that SAP.

- A CoNDIS WAN miniport driver is alerted to an incoming call by signaling messages from the network. From these signaling messages, the miniport driver extracts the call parameters for the call, including the SAP to which the incoming call is addressed.

- Before indicating an incoming call to NDPROXY, the miniport driver calls the NdisMCmCreateVc function to initiate the creation of a VC with NDPROXY. NDPROXY allocates and initializes resources required for the VC and stores the handle to the VC.

- The CoNDIS WAN miniport driver sets the TAPI parameters for an incoming call in a CO_AF_TAPI_INCOMING_CALL_PARAMETERS structure. The miniport driver fills this structure's members with the following information that was extracted from signaling messages:
  - Line identifier in the **ulLineID** member
  - Address of the incoming call in the **ulAddressID** member
  - CO_TAPI_FLAG_INCOMING_CALL bit in the **ulFlags** member. All other bits of **ulFlags** are reserved and must be set to 0.
  - LINECALLPARAMS structure in the **LineCallInfo** member. Members of LINECALLPARAMS specify TAPI call parameters for an incoming call.

- The miniport driver overlays CO_AF_TAPI_INCOMING_CALL_PARAMETERS on the **Parameters** member of a CO_SPECIFIC_PARAMETERS structure and sets the **Length** member of CO_SPECIFIC_PARAMETERS to the size of CO_AF_TAPI_INCOMING_CALL_PARAMETERS.

- The miniport driver sets the CO_SPECIFIC_PARAMETERS structure to the **MediaSpecific** member of a CO_MEDIA_PARAMETERS structure.

- The miniport driver sets a pointer to the CO_MEDIA_PARAMETERS structure to the **MediaParameters** member of a CO_CALL_PARAMETERS structure.

- The miniport driver must also set the **CallMgrParameters** member of the CO_CALL_PARAMETERS structure to specify the quality of service (QoS) of transferring packets, such as the bandwidth. To set this **CallMgrParameters** member, the miniport driver fills members of a CO_CALL_MANAGER_PARAMETERS structure and points this structure to **CallMgrParameters**. For example, to identify the transmit and receive speeds in bytes per second for the VC, the miniport driver must set the **PeakBandwidth** members of the **Transmit** and **Receive** members of CO_CALL_MANAGER_PARAMETERS. The **Transmit** and **Receive** members are FLOWSPEC structures. For more information about the FLOWSPEC structure, see the Microsoft Windows SDK.

- After the miniport driver encapsulates TAPI parameters and fills the **CallMgrParameters** member of CO_CALL_MANAGER_PARAMETERS, it calls the NdisMCmDispatchIncomingCall function to indicate the incoming call to NDPROXY. In this call, the miniport driver passes the following:

  - A handle that identifies the SAP to which the incoming call is addressed
  - A handle that identifies the VC for the incoming call
  - A pointer to the filled CO_CALL_PARAMETERS structure

- NDPROXY returns NDIS_STATUS_PENDING to the miniport driver so NDPROXY can complete **NdisMCmDispatchIncomingCall** asynchronously.

- After the TAPI application answers the incoming call with the **lineAnswer** function, NDPROXY calls the NdisClIncomingCallComplete function. NDIS in turn calls the miniport driver's *ProtocolCmIncomingCallComplete* function. If NDPROXY returns an NDIS_STATUS_SUCCESS code, it indicates acceptance of the call parameters. If NDPROXY finds the call parameters unacceptable, it can request a change in the call parameters by setting the **Flags** member in the CO_CALL_PARAMETERS structure to CALL_PARAMETERS_CHANGED and by supplying revised call parameters. If NDPROXY accepts the incoming call, the miniport driver should send signaling messages to indicate to the calling entity that the call has been accepted. Otherwise, the miniport driver should send signaling messages to indicate that the call has been rejected. If NDPROXY is requesting a change in call parameters, the miniport driver sends signaling messages to request a change in call parameters.

- The miniport driver activates the VC that the miniport driver created with NDPROXY and must also call the NdisMCmActivateVc function to notify NDPROXY that the miniport driver is ready to transfer packets on the VC.

- If NDPROXY rejects the call, the miniport driver calls the NdisMCmDeactivateVc function to deactivate the VC that the miniport driver created for the incoming call. After the VC is deactivated, the miniport driver calls the NdisMCmDeleteVc function to delete the VC.

- Depending on whether NDPROXY accepted the incoming call and whether the end-to-end connection was successfully established, the miniport driver calls either NdisMCmDispatchCallConnected or NdisMCmDispatchIncomingCloseCall functions. Note that if the remote calling entity tore down the call, it sends signaling messages to indicate that the end-to-end connection was not successfully established. **NdisMCmDispatchCallConnected** notifies NDPROXY that data transfers can begin on the VC that the miniport driver created and activated for the incoming call. **NdisMCmDispatchIncomingCloseCall** informs NDPROXY to tear down the incoming call.

- If NDPROXY is directed to tear down the incoming call, it calls the NdisClCloseCall function to acknowledge that it will neither attempt to send nor expect to receive data on the VC. NDIS in turn calls the miniport driver's *ProtocolCmCloseCall* function. The miniport driver then calls the **NdisMCmDeactivateVc** function to deactivate the VC. After the VC is deactivated, the miniport driver calls the **NdisMCmDeleteVc** function to delete the VC.

- After the TAPI application accepts the incoming call and NDPROXY notifies the application that the call is connected, the application calls the TAPI **lineGetID** function to inform NDPROXY to locate the appropriate CoNDIS client. In this **lineGetID** call, the TAPI application supplies a string for a particular TAPI device class to which the application requires a handle. NDPROXY uses this string to locate the CoNDIS client that previously registered a SAP for the particular TAPI device class. If the CoNDIS client is NDISWAN, the string is NDIS. If NDPROXY locates a SAP with a string that matches the string passed by the TAPI application, NDPROXY calls **NdisMCmCreateVc** to set up a connection endpoint with NDISWAN on which it can dispatch notification of the incoming call. NDIS in turn calls NDISWAN's *ProtocolCoCreateVc* function and passes a handle that represents the VC.

- After NDPROXY sets up the connection endpoint with NDISWAN, it calls the NdisCmDispatchIncomingCall function to notify NDISWAN about the incoming call. In this call, NDPROXY passes the encapsulated CO_AF_TAPI_INCOMING_CALL_PARAMETERS structure that contains the incoming call parameters. NDIS in turn calls NDISWAN's *ProtocolClIncomingCall* function, within which NDISWAN either accepts or rejects the requested connection.

- After deciding whether to accept the connection and after possibly changing the call parameters, NDISWAN calls the **NdisClIncomingCallComplete** function. NDIS in turn calls the miniport driver's *ProtocolCmIncomingCallComplete* function. Depending on whether NDISWAN accepted the incoming call and whether the miniport driver accepts or rejects NDISWAN's proposed changes to the call parameters, the miniport driver calls either **NdisCmDispatchCallConnected** or **NdisCmDispatchIncomingCloseCall** functions. **NdisCmDispatchCallConnected** notifies NDISWAN that data transfers can begin on the VC that the miniport driver created for the incoming call. **NdisCmDispatchIncomingCloseCall** informs NDISWAN and NDPROXY to tear down the incoming call.

- After NDISWAN accepts the incoming call, NDPROXY calls the NdisCoGetTapiCallId function to retrieve a string that identifies NDISWAN's context for the VC. NDPROXY passes this string back to the TAPI application. The TAPI application uses this VC-context string to complete its call to **lineGetID**.

CoNDIS TAPI Shutdown

Article • 03/14/2023

A TAPI session begins after a CoNDIS WAN miniport driver has enumerated its TAPI capabilities to an application. Within a session, one or more lines can be opened and one or more calls can be established. During the time a line is open, many calls can be established and then closed or dropped. During a session, one or more lines can go through transitions from open to closed many times. How a miniport driver handles such transitions is described in this section.

Closing a Call

An in-process call can be closed either by the local node or by the remote node. The call can be closed on the local node, either because the last application with a handle to the call has closed the handle, or perhaps because the miniport driver's *MiniportHaltEx* or *MiniportResetEx* has been called. If the remote node hangs up an in-process call, the miniport driver must inform upper layers to tear down the call.

If an application on the local node closes the call, it must disconnect the call. A call is disconnected as a result of an application calling the TAPI **lineDrop** function. This TAPI-function call causes the NDPROXY driver to call the **NdisClCloseCall** function and to pass a handle that represents the VC for the call. NDIS in turn calls the CoNDIS WAN miniport driver's **ProtocolCmCloseCall** function. The miniport driver should return NDIS_STATUS_PENDING to NDPROXY so the miniport driver can complete **NdisClCloseCall** asynchronously.

The miniport driver's *ProtocolCmCloseCall* must communicate with network control devices to terminate a connection between the local node and a remote node. The miniport driver must then call the **NdisMCmDeactivateVc** function to initiate deactivation of the VC used for the call.

After the miniport driver terminates the connection, its *ProtocolCmCloseCall* can call the **NdisMCmCloseCallComplete** function to complete the call closure.

If the remote node hangs up an in-process call, the miniport driver calls the **NdisCmDispatchIncomingCloseCall** function to inform NDISWAN and NDPROXY to tear down the incoming call.

Closing a Line

A line is closed when the last application with an open handle to the line has closed the handle. A line is closed as a result of an application calling the TAPI **lineClose** function. This TAPI-function call causes the NDPROXY driver to initiate the closure of all calls on that line as described in the preceding section. The miniport driver should drop those calls and clean up their state.

Closing a Session

Session termination can be initiated by either the upper layers or a CoNDIS WAN miniport driver. After the last client process has detached from the higher-level Telephony module, the NDPROXY driver will be informed that it must terminate its session with each of the registered adapters. To do so, the NDPROXY driver calls the NdisClCloseAddressFamily function and passes the handle to the TAPI address family. NDIS in turn calls the miniport driver's ProtocolCmCloseAf function. The miniport driver should terminate any related activities it has in progress on the specified adapter and release any relevant resources. After calling **NdisClCloseAddressFamily**, the client should consider the handle to the TAPI address family invalid.

Driver-initiated session termination can occur if the miniport driver is being unloaded in its *MiniportHaltEx* function. Typically, the miniport driver would complete any outstanding NDPROXY requests and notify NDISWAN that all calls are closing. If the miniport driver were reloaded again later, it would go through the same initialization process described previously.

The CoNDIS WAN miniport driver might also initiate session termination if it underwent some dynamic reconfiguration that necessitated a complete reinitialization of all clients and drivers. For example, if an adapter's line-device modeling (for example, the number of line devices supported) was changed on the fly.

Responding to an OID_CO_TAPI_LINE_CAPS Query

Article • 12/06/2022

In response to an OID_CO_TAPI_LINE_CAPS query, a call manager or MCM returns a CO_TAPI_LINE_CAPS structure that contains a LINE_DEV_CAPS structure. To support voice streaming, a call manager or MCM must specify the following values in the LINE_DEV_CAPS structure:

- **ulMediaModes**

  This field should contain LINEMEDIAMODE_AUTOMATEDVOICE, which maps to TAPIMEDIAMODE_AUDIO in TAPI 3.0.

- **ulAddressTypes**

  This field must be filled in appropriately. For a description of valid values, see the description of **dwAddressTypes**. This field must not be zero.

- **ulGenerateDigitModes**

  This field must be filled in with a bitwise OR of the LINEDIGITMODE_constants that specify the digit modes that can be generated on the line. For a description of the LINEDIGITMODE_constant, see the description of **dwGenerateDigitModes**.

- **ulMonitorDigitModes**

  This field must be filled in with a bitwise OR of the LINEDIGITMODE_constants that specify the digit modes than can be detected on this line. For a description of the LINEDIGITMODE_constants, see the description of **dwMonitorDigitModes**.

Specifying Parameters for an Outgoing Call

Article • 12/15/2021

When making an outgoing call, a call manager or MCM that supports voice streaming must supply the following values in the **CO_CALL_MANAGER_PARAMETERS** structure:

- Maximum transmit SDU size (CallMgrParameters->Transmit.MaxSduSize)

- Maximum receive SDU size (CallMgrParameters->Receive.MaxSduSize)

A call manager or MCM that supports an address family other than CO_ADDRESS_FAMILY_TAPI_PROXY fills in these values when translating TAPI call parameters to NDIS call parameters in response to a query of OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS.

A call manager or MCM that supports the CO_ADDRESS_FAMILY_TAPI_PROXY family writes these values to the CO_CALL_MANAGER_PARAMETERS structure in the context of its **ProtocolCmMakeCall** function. Note that the maximum SDU sizes passed to the *ProtocolCmMakeCall* function are incorrect. The *ProtocolCmMakeCall* function must overwrite the incorrect values with correct values.

Specifying Parameters for an Incoming Call

Article • 12/06/2022

When indicating an incoming call with **Ndis(M)CmDispatchIncomingCall**, a call manager or MCM that supports voice streaming must specify the following values in the CO_CALL_MANAGER_PARAMETERS structure:

- Maximum transmit SDU size (CallMgrParameters->Transmit.MaxSduSize)
- Maximum receive SDU size (CallMgrParameters->Receive.MaxSduSize)

In addition, a call manager or an MCM must specify the following values in the LINE_CALL_INFO structure:

- **ulMediaMode**

  This field should contain LINEMEDIAMODE_AUTOMATEDVOICE, which maps to TAPIMEDIAMODE_AUDIO in TAPI 3.0.

- **ulCallerIDFlags**
- **ulCallerIDSize**
- **ulCallerIDOffset**
- **ulCallerIDNameSize**
- **ulCallerIDNameOffset**
- **ulCalledIDFlags**
- **ulCalledIDSize**
- **ulCalledIDOffset**
- **ulCalledIDNameSize**
- **ulCalledDNameOffset**
- **ulCallerIDAddressType**
- **ulCalledIDAddressType**

A call manager or MCM that supports an address family other than CO_ADDRESS_FAMILY_TAPI_PROXY specifies the preceding LINE_CALL_INFO members when responding to an OID_CO_TAPI_TRANSLATE_NDIS_CALLPARAMS query.

A call manager or an MCM that supports the CO_ADDRESS_FAMILY_TAPI_PROXY family specifies the above-listed LINE_CALL_INFO members in the media-specific portion of the CO_CALL_MANAGER_PARAMETERS structure that it supplies to **Ndis(M)CmDispatchIncomingCall**.

Non-WAN-Specific Extensions to Support Telephonic Services Over Connection-Oriented NDIS

Article • 03/14/2023

This topic describes non-WAN-specific extensions for TAPI support over connection-oriented NDIS. These extensions are the NDIS/TAPI translation OIDs. These extensions allow non-WAN-specific call managers and integrated miniport call manager (MCM) drivers to translate TAPI parameters to NDIS parameters or TAPI parameters to NDIS parameters. These extensions allow call managers and MCMs that support ATM, for example, to provide TAPI access over connection-oriented media. For information about WAN-specific extensions for TAPI support over connection-oriented NDIS, see CoNDIS WAN Operations that Support Telephonic Services.

The NDIS/TAPI translation OIDs should not be used for call managers or MCMs that respectively register CO_ADDRESS_FAMILY_TAPI_PROXY with **NdisCmRegisterAddressFamilyEx** or **NdisMCmRegisterAddressFamilyEx**. Instead, such call managers and MCMs, as well as their TAPI clients, should encapsulate TAPI parameters inside connection-oriented structures, as described in CoNDIS WAN Operations that Support Telephonic Services.

The NDIS/TAPI translation OIDs are as follows:

- OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS

  This OID requests a call manager or MCM to translate TAPI call parameters supplied by the client to NDIS call parameters. The client typically uses the NDIS call parameters returned by the call manager or MCM as an input (formatted as a **CO_CALL_PARAMETERS** structure) to **NdisClMakeCall**. The client uses **NdisClMakeCall** to initiate a connection-oriented call.

- OID_CO_TAPI_TRANSLATE_NDIS_CALLPARAMS

  This OID requests a call manager or MCM to translate NDIS call parameters for an incoming call (passed in a CO_CALL_PARAMETERS structure to the client's **ProtocolClIncomingCall** function) to TAPI call parameters. The client uses the translated TAPI call parameters returned by the call manager or MCM to determine whether to accept or reject the incoming call.

- OID_CO_TAPI_TRANSLATE_SAP

This OID requests a call manager or MCM to prepare one or more NDIS SAPs from TAPI call parameters that are supplied by the client. The client typically uses an NDIS SAP returned by the call manager or MCM as an input (formatted as a **CO_SAP** structure) to **NdisClRegisterSap**, with which the client registers a SAP on which to receive incoming calls.

WAN Packet Framing Overview

Article • 12/15/2021

This section provides information about WAN packet framing.

The NDISWAN intermediate driver retrieves information about the WAN packet framing performed by a WAN miniport driver from the miniport driver's response to the OID_WAN_MEDIUM_SUBTYPE query information request.

NDISWAN converts an out-going packet from LAN to PPP format. NDISWAN uses simple HDLC framing. Most of the media-specific framing must be done by the miniport driver.

Before sending packets to the WAN miniport driver's send function, NDISWAN does simple PPP HDLC framing. Simple PPP HDLC framing is PPP's HDLC framing without the FCS, bit or byte stuffing, and any beginning or end flags.

The following topics provide additional information about WAN packet framing:

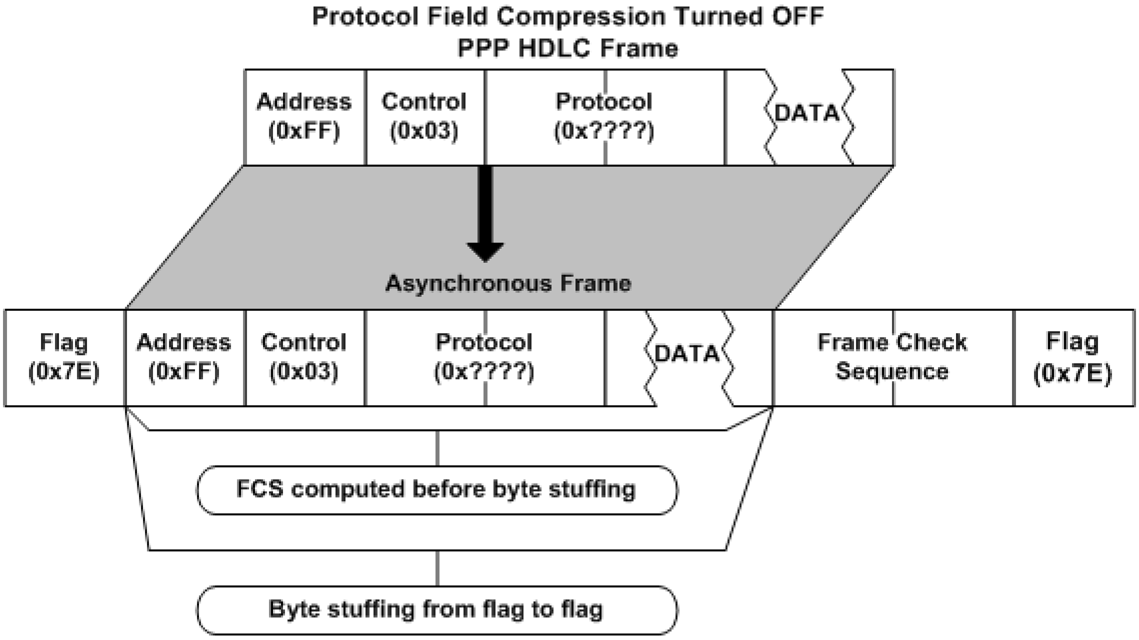
Asynchronous Framing

ISDN and Switched-56K Framing

Asynchronous Framing

Article • 12/15/2021

The following diagram illustrates asynchronous framing with compression turned off.



ISDN and Switched 56K Framing

Article • 12/15/2021

Initially, ISDN B channels (not D channels) should be used. Multiple B channel support is done through NDISWAN's multilink support. Initially, bit-synchronous HDLC framing with NRZ encoding should be used. Transparency should be provided by the driver or ISDN hardware. It is also the responsibility of the ISDN driver or hardware to provide for NRZ encoding, to calculate the FCS to add the PPP end flag (0x7E), and to insert any inter-frame time fill. Switched 56K drivers should frame in the same manner as ISDN.

WAN Miniport Driver Build Parameters

Article • 12/15/2021

This topics provides some information about defining build parameters for NDIS and CoNDIS WAN miniport drivers.

Add the following line to your Sources file before building to identify your driver as a miniport driver.

Text

```
C_DEFINES=/DNDIS_MINIPORT_DRIVER
```

If you are writing an NDIS WAN miniport driver that supports connections through TAPI, you must add the following line to your Sources file before building to identify the TAPI version that your driver supports.

Text

```
C_DEFINES=-DNDIS_TAPI_CURRENT_VERSION=0x00010003
```

If you are writing a CoNDIS WAN miniport driver that is an integrated miniport call manager (MCM) and that supports the CoNDIS address family type CO_ADDRESS_FAMILY_TAPI_PROXY, you must add the following line to your sources file before building to identify the TAPI version that your driver supports.

Text

```
C_DEFINES=-DNDIS_TAPI_CURRENT_VERSION=0x00030000
```

For WAN miniport drivers, the include paths should include Ndiswan.h as well as Ndis.h.

If the WAN miniport driver supports connections through TAPI, the driver should also include Ndistapi.h.

Standardized INF Keywords for Network Devices

Article • 12/15/2021

This section provides information about standardized keywords that appear in the registry and are specified in INF files. NDIS 6.0 and later versions of NDIS support standardized keywords for miniport drivers in network devices.

Standardized keywords provide:

- Standardized user interface properties for end users.
- The ability for both home network users and large-scale enterprises to easily configure networks that include devices from multiple hardware manufacturers.
- The ability to programmatically test for all advanced network device features.

The following standard INF keywords are mandatory for connectionless NDIS 6.0 and later miniport drivers:

- ***IfType**
- ***MediaType**
- ***PhysicalMediaType**

If the mandatory keywords are missing from the driver's INF file, NDIS does not call the miniport driver's *MiniportInitializeEx* function.

Standardized keywords are required for NDIS 6.0 and later miniport drivers if both of the following are true:

- An INF setting must be exposed in the **Advanced** properties page of the user interface.
- The device fully supports the specified properties.

Note Standardized keywords are optional but recommended for NDIS 5.1 and earlier NDIS miniport drivers.

This section specifies the INF keywords that are exposed in the user interface. However, miniport drivers must read the registry settings during initialization to determine the current configuration settings.

Within an INF file, definitions for these keywords are placed with the other definitions for the advanced properties page. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

All standardized keyword names start with an asterisk (*). This naming convention enables you to easily distinguish standardized names from non-standard names.

There are three types of standardized keyword data that are exposed in the user interface:

Enum
Values that can be selected from a list that appears in a drop down menu in the **Advanced** properties page.

Int
Numerical values that you can edit.

Edit
Text values that you can edit.

The following topics include descriptions for the standardized keywords that are common to all networking technologies:

Enumeration Keywords

Keywords That Can Be Edited

Keywords Not Displayed in the User Interface

In addition, standardized keywords that are specific to networking technologies are described in the following topics:

INF File Settings for Filter Drivers

INF Requirements for NDKPI

MB Miniport Driver INF Requirements

Standardized INF Keywords for Header-Data Split

Standardized INF Keywords for NDIS Quality of Service (QoS)

Standardized INF Keywords for NDIS Selective Suspend

Standardized INF Keywords for NVGRE Task Offload

Standardized INF Keywords for NDIS Packet Timestamping

Standardized INF Keywords for Packet Coalescing

Standardized INF Keywords for Power Management

Standardized INF Keywords for RSC

Standardized INF Keywords for RSS

Standardized INF Keywords for Single Root I/O Virtualization (SR-IOV)

Standardized INF Keywords for Virtual Machine Queue (VMQ)

Enumeration Keywords

Article • 12/15/2021

NDIS 6.0 and later versions of NDIS provide standardized enumeration keywords for miniport drivers of network devices. Enumeration keywords are associated with values that appear as a list in a menu.

The following example shows an INF file definition for an enumeration keyword.

```INF
HKR, Ndi\params\<SubkeyName>, ParamDesc, 0, "%<SubkeyName>%"
HKR, Ndi\params\<SubkeyName>, Type, 0, "enum"
HKR, Ndi\params\<SubkeyName>, Default, 0, "3"
HKR, Ndi\params\<SubkeyName>, Optional, 0, "0"
HKR, Ndi\params\<SubkeyName>\enum, "0", 0, "%Disabled%"
HKR, Ndi\params\<SubkeyName>\enum, "1", 0, "%Tx Enabled%"
HKR, Ndi\params\<SubkeyName>\enum, "2", 0, "%Rx Enabled%"
HKR, Ndi\params\<SubkeyName>\enum, "3", 0, "%Rx & Tx Enabled%"
```

The general enumeration keywords are:

***SpeedDuplex**
Speed and duplex settings that a device supports. The device INF file should list only the settings that the associated device supports. That is, for an Ethernet 10/100 device that can support only full-duplex mode, settings for Gigabit or higher speeds or half duplex should not be listed in the associated INF file.

Speed values that are not specifically defined already with enumerated values of 0 through 10 may be set as a number that is the value directly in Mbps. Direct values must be at least 1,000 Mbps (1 Gbps) and above. Here are a few examples for specifying the speed directly:

SpeedDuplex value	Resulting speed
1,000	1 Gbps
10,000	10 Gbps
25,000	25 Gbps
50,000	50 Gbps
100,000	100 Gbps

***FlowControl**
The ability for the device to enable or disable flow control in the send or receive path.

Note Ethernet devices today support flow control, and the Windows 8 in-box drivers for LAN have flow control enabled by default. When a kernel debugger attaches to one of these LAN adapters, the NIC will start pushing flow control pause frames into the network. Most network switches will react by temporarily taking down the network for all other computers that are connected to the same hub. This is a common development scenario, and the end-user experience is both undesirable and difficult to diagnose.

Note Client and Server defaults are not the same; refer to the table of defaults below.

For this reason, in Windows 8 and later, NDIS will disable flow control automatically when debugging is enabled on the computer (for example, by typing **bcdedit /set debug on** at the command line). When kernel debugging is enabled and the miniport calls NdisReadConfiguration and passes "*FlowControl" for the *Keyword* parameter, NDIS will override the configured value and return zero.

If you need to enable flow control while debugging, NDIS provides the **AllowFlowControlUnderDebugger** registry value to allow you to do that. The **AllowFlowControlUnderDebugger** registry value prevents NDIS from disabling flow control, and allows NICs to keep their configured behavior. It can be found under the following registry key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NDIS\Parameters

Set this registry value to 0x00000001.

If it does not exist, you can create a value with the name **AllowFlowControlUnderDebugger** and the type **REG_DWORD** and set it to 0x00000001.

***PriorityVLANTag**
A value that indicates whether the device has enabled or disabled the ability to insert the 802.1Q tags for packet priority and virtual LANs (VLANs). This keyword does not indicate whether the device enabled or disabled packet priority or VLAN tags. Instead, it describes the following:

- Whether the device inserts 802.1Q tags during a send operation
- Whether 802.1Q tag information is available in the NET_BUFFER_LIST out-of-band (OOB) information
- Whether the device copies 802.1Q tags to OOB during receive operations

The miniport driver should remove the 802.1Q header from all receive packets regardless of the ***PriorityVLANTag** setting. If the 802.1Q header is left in a packet, other drivers might not be able to parse the packet correctly.

If the Rx flag is enabled on the receive path, the miniport driver should copy the removed 802.1Q header into OOB.

Otherwise, if the Rx flag is disabled, the miniport driver should not copy the removed 802.1Q header into OOB.

If the Tx flag is enabled on the transmit path, the miniport driver should do the following:

- Insert the 802.1Q header into each outgoing packet and fill it up with the data from OOB (if any non-zero data exists in OOB).
- Advertise appropriate **MacOptions** in NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES (**NDIS_MAC_OPTION_8021P_PRIORITY** and **NDIS_MAC_OPTION_8021Q_VLAN**).

Otherwise, if the Tx flag is disabled, then:

- The miniport filter should not honor 802.1Q information in OOB (and therefore not insert any tag).
- The miniport filter should not advertise appropriate **MacOptions** in NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES.

Note If the miniport driver supports NDIS quality of service (QoS), it must also read the ***QOS** keyword value. Based on the ***QOS** keyword value, the ***PriorityVLANTag** keyword values are interpreted differently. For more information, see Standardized INF Keywords for NDIS QoS.

***InterruptModeration**
A value that describes whether the device enabled or disabled interrupt moderation. Interrupt moderation algorithms are device-dependent. The device manufacturer can use non-standardized keywords to support algorithmic settings. For more information about interrupt moderation, see Interrupt Moderation.

***RSS**
A value that describes whether the device enabled or disabled receive side scaling (RSS). For more information about RSS, see Receive Side Scaling.

***HeaderDataSplit**
A value that describes whether the device enabled or disabled header-data split. For more information about header-data split, see Header-Data Split.

The following keywords are associated with connection offload services:

***TCPConnectionOffloadIPv4**

***TCPConnectionOffloadIPv6**

For more information about the connection offload keywords, see Using Registry Values to Enable and Disable Connection Offloading.

The following keywords are associated with task offload services:

***IPChecksumOffloadIPv4**

***TCPChecksumOffloadIPv4**

***TCPChecksumOffloadIPv6**

***UDPChecksumOffloadIPv4**

***UDPChecksumOffloadIPv6**

***LsoV1IPv4**

***LsoV2IPv4**

Note For devices that support both large send offload version 1 (LSOv1) and LSOv2 over IPv4, only the ***LsoV2IPv4** keyword should be used in the INF file and registry values. If, for example, the ***LsoV2IPv4** keyword appears in the INF file and the ***LsoV1IPv4** keyword appears in the registry (or vice versa), the ***LsoV2IPv4** keyword always takes precedence.

***LsoV2IPv6**

***IPsecOffloadV1IPv4**

***IPsecOffloadV2**

***IPsecOffloadV2IPv4**

***TCPUDPChecksumOffloadIPv4**

***TCPUDPChecksumOffloadIPv6**

For more information about the TCP/IP offload keywords, see Using Registry Values to Enable and Disable Task Offloading.

The columns in the table at the end of this topic describe the following attributes for enumeration keywords:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with **SubkeyName**.

Value
The enumeration integer value that is associated with each option in the list. This value is stored in **NDI\params\***SubkeyName*\*Value*.

EnumDesc
The display text that is associated with each value that appears in the menu.

Default
The default value for the menu.

The following table lists all of the keywords and describes the values that a driver must use for the preceding attributes. For more information about a keyword, search for the keyword in the WDK documentation.

SubkeyName	ParamDesc	Value	EnumDesc
***SpeedDuplex**	Speed & Duplex	0 (Default)	Auto Negotiation
		1	10 Mbps Half Duplex
		2	10 Mbps Full Duplex
		3	100 Mbps Half Duplex
		4	100 Mbps Full Duplex
		5	1.0 Gbps Half Duplex
		6	1.0 Gbps Full Duplex
		7	10 Gbps Full Duplex
		8	20 Gbps Full Duplex
		9	40 Gbps Full Duplex
		10	100 Gbps Full Duplex
***FlowControl**	Flow Control	0 (Server Default)	Tx & Rx Disabled
		1	Tx Enabled

SubkeyName	ParamDesc	Value	EnumDesc
		2	Rx Enabled
		3 (Client Default)	Rx & Tx Enabled
		4	Auto Negotiation
***PriorityVLANTag**	Packet Priority & VLAN	0	Packet Priority & VLAN Disabled
		1	Packet Priority Enabled
		2	VLAN Enabled
		3 (Default)	Packet Priority & VLAN Enabled
***InterruptModeration**	Interrupt Moderation	0	Disabled
		1 (Default)	Enabled
***RSS**	Receive Side Scaling	0	Disabled
		1 (Default)	Enabled
***HeaderDataSplit**	Header Data Split	0 (Default)	Disabled
		1	Enabled
***TCPConnectionOffloadIPv4**	TCP Connection Offload (IPv4)	0	Disabled
		1 (Default)	Enabled
***TCPConnectionOffloadIPv6**	TCP Connection Offload (IPv6)	0	Disabled
		1 (Default)	Enabled
***IPChecksumOffloadIPv4**	IPv4 Checksum Offload	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
***TCPChecksumOffloadIPv4**	TCP Checksum Offload (IPv4)	0	Disabled

SubkeyName	ParamDesc	Value	EnumDesc
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
***TCPChecksumOffloadIPv6**	TCP Checksum Offload (IPv6)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
***UDPChecksumOffloadIPv4**	UDP Checksum Offload (IPv4)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
***UDPChecksumOffloadIPv6**	UDP Checksum Offload (IPv6)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
***LsoV1IPv4**	Large Send Offload Version 1 (IPv4)	0	Disabled
		1 (Default)	Enabled
***LsoV2IPv4**	Large Send Offload Version 2 (IPv4)	0	Disabled
		1 (Default)	Enabled
***LsoV2IPv6**	Large Send Offload Version 2 (IPv6)	0	Disabled
		1 (Default)	Enabled
***IPsecOffloadV1IPv4**	IPsec Offload Version 1 (IPv4)	0	Disabled

SubkeyName	ParamDesc	Value	EnumDesc
		1	Auth Header Enabled
		2	ESP Enabled
		3 (Default)	Auth Header & ESP Enabled
***IPsecOffloadV2**	IPsec Offload	0	Disabled
		1	Auth Header Enabled
		2	ESP Enabled
		3 (Default)	Auth Header & ESP Enabled
***IPsecOffloadV2IPv4**	IPsec Offload (IPv4 only)	0	Disabled
		1	Auth Header Enabled
		2	ESP Enabled
		3 (Default)	Auth Header & ESP Enabled
***TCPUDPChecksumOffloadIPv4**	TCP/UDP Checksum Offload (IPv4)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Tx and Rx Enabled
***TCPUDPChecksumOffloadIPv6**	TCP/UDP Checksum Offload (IPv6)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Tx and Rx Enabled

Keywords That Can Be Edited

Article • 12/15/2021

NDIS 6.0 and later versions of NDIS provide standardized keywords that can be edited for miniport drivers of network devices. These standardized keywords are associated with numeric or text values that you can edit in the user interface.

The following example shows an INF file definition for a keyword that can be edited.

INF

```
HKR, Ndi\params\<SubkeyName>,ParamDesc, 0, "<ParamDesc>"
HKR, Ndi\params\<SubkeyName>,Type, 0, "int"
HKR, Ndi\params\<SubkeyName>,Default, 0, "<IHV defined>"
HKR, Ndi\params\<SubkeyName>,Optional, 0, "0"
HKR, Ndi\params\<SubkeyName>,Min, 0, "0"
HKR, Ndi\params\<SubkeyName>,Max, 0, "<IHV defined>"
```

The standard keywords that can be edited are:

__*JumboPacket__ The size, in bytes, of the largest supported Jumbo Packet (an Ethernet frame that is greater than 1514 bytes) that the hardware can support. This is also known as a Jumbo Frame. _*JumboPacket_'s range of values and maximum value are IHV-defined. For more info, check with your IHV.

__*ReceiveBuffers__
The number of receive descriptors used by the miniport adapter. The miniport driver can choose any default value that is appropriate for performance-tuning. Note that if the value is too small, the miniport adapter may run out of receive buffers under heavy load. If the value is too large, system resources are wasted.

__*TransmitBuffers__
The size, in bytes, of the transmit buffers that the hardware can support. This size is hardware-dependent and can include data buffers, buffer descriptors, and so on. Hardware vendors can assign any value that is appropriate for their purposes.

__*NetworkAddress__
The network address of the device. The format for a MAC address is: XX-XX-XX-XX-XX-XX. The hyphens (-) are optional.

The columns in the table at the end of this topic describe the following attributes for keywords that can be edited:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with SubkeyName.

Type
The type of value that can be edited. The value can be either numeric (**Int**) or text that can be edited (**Edit**).

Default value
The default value for the integer or text. <IHV defined> indicates that the value is associated with the particular independent hardware vendor (IHV) requirements.

Min
The minimum value that is allowed for an integer. <IHV defined> indicates that the minimum value is associated with the particular IHV requirements.

Max
The maximum value that is allowed for an integer. <IHV defined> indicates that the minimum value is associated with the particular IHV requirements.

The following table lists all of the keywords and describes the values that a driver must use for the preceding attributes. For more information about a keyword, search for the keyword in the WDK documentation.

SubkeyName	ParamDesc	Type	Default value	Min	Max
***JumboPacket**	Jumbo Packet	Int	1514	1514	
***ReceiveBuffers**	Receive Buffers	Int		1	
***TransmitBuffers**	Transmit Buffers	Int		0	
***NetworkAddress**	Network Address	Edit	N/A	N/A	N/A

Keywords Not Displayed in the User Interface

Article • 12/15/2021

NDIS 6.0 and later versions of NDIS provide some standardized keywords for miniport drivers of network devices. These standardized keywords appear in INF files but not in the user interface.

These general keywords are described in the following list. For more information about a particular keyword, search for the keyword in the WDK documentation.

***IfType**
The NDIS interface type for a device. For more information about the NDIS interface type, see NDIS Interface Types.

***MediaType**
The media type for a device. For more information about the media type of the miniport adapter, see OID_GEN_MEDIA_SUPPORTED.

***PhysicalMediaType**
The physical media type for a device. For more information about the physical media type of the miniport adapter, see OID_GEN_PHYSICAL_MEDIUM.

***NdisDeviceType**
The type of the device. The default value is zero, which indicates a standard networking device that connects to a network. Set ***NdisDeviceType** to NDIS_DEVICE_TYPE_ENDPOINT (1) if this device is an endpoint device and is not a true network interface that connects to a network. For example, you must specify NDIS_DEVICE_TYPE_ENDPOINT for devices such as smart phones that use a networking infrastructure to communicate to the local computer system but do not provide connectivity to an external network. However, you must ***not*** set this keyword to NDIS_DEVICE_TYPE_ENDPOINT for virtual adapters such as VPN interfaces, because they provide connectivity to an external network.

Note Windows Vista automatically identifies and monitors the networks a computer connects to. If the NDIS_DEVICE_TYPE_ENDPOINT flag is set, the device is an endpoint device and is not a connection to a true external network. Consequently, Windows ignores the endpoint device when it identifies networks. The Network Awareness APIs indicate that the device does not connect the computer to a network. For end users in this situation, the Network and Sharing Center and the network icon in the notification area do not show the NDIS endpoint device as connected. However, the connection is shown in the Network Connections Folder.

Roadmap for Developing NDIS Protocol Drivers

Article • 03/14/2023

To create a Network Driver Interface Specification (NDIS) protocol driver package, follow these steps:

- Step 1: Learn about Windows architecture and drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- Step 2: Learn about NDIS.

  For general information about NDIS and NDIS drivers, see the following topics:

  Windows Network Architecture and the OSI Model

  Network Driver Programming Considerations

  Driver Stack Management

  NET_BUFFER Architecture

- Step 3: Determine additional Windows driver design decisions.

  For more information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- Step 4: Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user-mode application. For more information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Lab Kit (HLK) testing, see Building, Debugging, and Testing Drivers. For more information about building, testing, verifying, and debugging tools, see Driver Development Tools.

- Step 5: Read the protocol driver introduction topics. Introduction to NDIS Protocol Drivers Protocol Driver Design Concepts

- Step 6: Read the writing protocol drivers section.

This section provides an overview of the primary protocol driver interfaces. These interfaces included functions that protocol drivers provide (*ProtocolXxx* functions) and NDIS calls to initiate operations. NDIS provides **Ndis*Xxx*** functions that protocol drivers call to perform NDIS operations.

- Step 7: Review the NDIS protocol driver sample in the Windows driver samples repository on GitHub.

- Step 8: Develop (or port), build, test, and debug your NDIS driver.

  See the porting guides if you are porting an existing driver:
  - Porting NDIS 5.x Drivers to NDIS 6.0
  - Porting NDIS 6.x Drivers to NDIS 6.20
  - Porting NDIS 6.x Drivers to NDIS 6.30

  For more information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process will help ensure that you build a driver that works.

- Step 9: Create a driver package for your driver.

  For more information about how to install drivers, see Providing a Driver Package. For more information about how to install an NDIS driver, see Components and Files Used for Network Component Installation and Notify Objects for Network Components.

- Step 10: Sign and distribute your driver.

  The final step is to sign (optional) and distribute the driver. If your driver meets the quality standards that are defined for the Windows Hardware Lab Kit (HLK), you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual driver.

Introduction to NDIS Protocol Drivers

Article • 03/14/2023

An NDIS protocol driver exports a set of *ProtocolXxx* functions at its lower edge. Such a protocol driver communicates with NDIS to send and receive network data. The protocol driver binds to an underlying miniport driver or intermediate driver that exports a *MiniportXxx* interface at its upper edge.

Note The miniport driver upper edge of an intermediate driver (virtual miniport) does not manage physical devices. Underlying miniport drivers manage physical devices.

Protocol drivers always use NDIS-provided functions to communicate with underlying NDIS drivers to send and receive network data. For example, a protocol driver that has a connectionless lower-edge (which communicates with underlying drivers for connectionless media, such as Ethernet) must call **NdisSendNetBufferLists** to send network data to an underlying NDIS driver. The protocol driver can call **NdisOidRequest** to query or set OIDs that underlying connectionless drivers support. A protocol driver that has a connection-oriented lower edge (which communicates with underlying drivers for connection-oriented media, such as ISDN) must call **NdisCoSendNetBufferLists** to send network data to a lower-level NDIS driver. It can also call **NdisCoOidRequest** to query or set OIDs that are supported by underlying connection-oriented drivers.

NDIS also provides a set of **Ndis*Xxx*** functions that hide the details of the underlying operating system. For example, a protocol driver can call **NdisInitializeEvent** to create an event for synchronization purposes and **NdisInitializeListHead** to create a linked list. Protocol drivers that use the NDIS versions of such functions are more portable across Microsoft operating systems. However, protocol drivers can also call kernel-mode support routines, such as **IoCreateDevice**. For more information, see Summary of Kernel-Mode Support Routines.

Developers of protocol drivers should use the same programming considerations that are applied to other NDIS drivers.

Initializing a Protocol Driver

Article • 12/15/2021

The system calls a protocol driver's DriverEntry routine after it loads the driver. Protocol drivers load as system services. They can load at any time before, during, or after the miniport drivers load.

Protocol drivers allocate driver resources and register *ProtocolXxx* functions in **DriverEntry**. This includes CoNDIS clients and stand-alone call managers. To register its *ProtocolXxx* functions with NDIS, a protocol driver calls the NdisRegisterProtocolDriver function.

DriverEntry returns STATUS_SUCCESS, or its equivalent NDIS_STATUS_SUCCESS, if the driver registered as an NDIS protocol driver successfully. If **DriverEntry** fails initialization by propagating an error status that was returned by an **NdisXxx** function or by a kernel-mode support routine, the driver will not remain loaded. **DriverEntry** must execute synchronously; that is, it cannot return STATUS_PENDING or its equivalent NDIS_STATUS_PENDING.

The DriverEntry function of an NDIS protocol driver must call the NdisRegisterProtocolDriver function. To register the driver's *ProtocolXxx* entry points with the NDIS library, a protocol driver initializes an **NDIS_PROTOCOL_DRIVER_CHARACTERISTICS** structure and passes it to **NdisRegisterProtocolDriver**.

Drivers that call NdisRegisterProtocolDriver must be prepared for an immediate call to any of their ProtocolXxx functions.

NDIS protocol drivers provide the following *ProtocolXxx* functions, which are updated versions of the functions that legacy drivers provide:

ProtocolSetOptions

ProtocolBindAdapterEx

ProtocolUnbindAdapterEx

ProtocolOpenAdapterCompleteEx

ProtocolCloseAdapterCompleteEx

ProtocolNetPnPEvent

ProtocolUninstall

NDIS protocol drivers provide the following *ProtocolXxx* functions for send and receive operations:

ProtocolReceiveNetBufferLists

ProtocolSendNetBufferListsComplete

All types of NDIS protocol drivers should register fully functional *ProtocolBindAdapterEx* and *ProtocolUnbindAdapterEx* functions to support Plug and Play (PnP). In general, a DriverEntry function should call NdisRegisterProtocolDriver immediately before it returns control with a status value of STATUS_SUCCESS or NDIS_STATUS_SUCCESS.

Any protocol driver that exports a set of standard kernel-mode driver routines in addition to its NDIS-defined *ProtocolXxx* functions must set the entry points for those driver routines in the given driver object that is passed in to its DriverEntry function. For more information about the functionality of such a protocol driver's **DriverEntry** function, see Writing a DriverEntry Routine.

If an attempt to allocate resources that the driver needs to carry out network I/O operations fails, DriverEntry should release all resources that it already allocated before it returns control with a status other than STATUS_SUCCESS or NDIS_STATUS_SUCCESS.

If an error occurs after a successful call to NdisRegisterProtocolDriver, the driver must call the NdisDeregisterProtocolDriver function before **DriverEntry** returns.

To allow a protocol driver to configure optional services, NDIS calls the *ProtocolSetOptions* function within the context of the protocol driver's call to **NdisRegisterProtocolDriver**. For more information about optional services, see Configuring Optional Protocol Driver Services.

CoNDIS client drivers must call the NdisSetOptionalHandlers function from the *ProtocolSetOptions* function. The driver initializes an **NDIS_CO_CLIENT_OPTIONAL_HANDLERS** structure and passes it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

CoNDIS stand-alone call managers must also call the NdisSetOptionalHandlers function from the *ProtocolSetOptions* function. The driver initializes an **NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS** structure and passes it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

MCMs are not protocol drivers. Therefore, they must call the NdisSetOptionalHandlers function from the MiniportSetOptions function. The MCM initializes an **NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS** structure and passes it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

To unregister with NDIS, a protocol driver calls NdisDeregisterProtocolDriver from its Unload routine.

To perform cleanup operations before a protocol driver is uninstalled, a protocol driver can register a *ProtocolUninstall* function. The *ProtocolUninstall* function is optional. For example, the protocol lower edge of an intermediate driver might require a *ProtocolUninstall* function. The intermediate driver can release its protocol edge resources in *ProtocolUninstall* before NDIS calls its *MiniportDriverUnload* function.

Protocol Binding States and Operations

Article • 12/15/2021

An NDIS protocol driver must support the following operational states for each binding that the driver manages:

Unbound
The Unbound state is the initial state of a binding. In this state, the protocol driver waits for NDIS to call the *ProtocolBindAdapterEx* function.

Opening
In the Opening state, a protocol driver allocates resources for the binding and attempts to open the adapter.

Running
In the Running state, a protocol driver performs send and receive processing for a binding.

Closing
In the Closing state, the protocol driver closes the binding to the adapter and then releases the resources for the binding.

Pausing
In the Pausing state, a protocol driver completes any operations that are required to stop send and receive operations for a binding.

Paused
In the Paused state, the protocol driver does not perform send or receive operations for a binding.

Restarting
In the Restarting state, a protocol driver completes any operations that are required to restart send and receive operations for a binding.

In the following table, the headings represent the binding states, and events are listed in the first column. The rest of the entries in the table specify the next state that the binding enters after an event occurs within a state. The blank entries represent invalid event/state combinations.

Event \ State	Unbound	Opening	Closing	Paused	Restarting	Running	Pausing
ProtocolBindAdapterEx	Opening						
Bind failed		Unbound					
Bind is complete		Paused					

Event \ State	Unbound	Opening	Closing	Paused	Restarting	Running	Pausing
ProtocolUnbindAdapterEx				Closing			
Unbind is complete			Unbound				
PnP pause						Pausing	
Pause is complete							Paused
PnP restart				Restarting			
Restart is complete					Running		
Restart failed					Paused		
Send and receive operations						Running	Pausing
OID requests			Closing	Paused	Restarting	Running	Pausing

Note The events listed in the preceding table are the primary events for an NDIS protocol binding. Additional events will be added to this table as the information becomes available.

The primary binding events are defined as follows:

ProtocolBindAdapterEx
After NDIS calls the driver's *ProtocolBindAdapterEx* function, the binding enters the Opening state. For more information, see Binding to an Adapter.

Bind failed
If the protocol driver fails to bind to the adapter, the binding returns to the Unbound state.

Bind is complete
If the driver successfully opens the adapter, the binding enters the Paused state. The driver completes the bind operation.

ProtocolUnbindAdapterEx
After NDIS calls the driver's *ProtocolUnbindAdapterEx* hander, the binding enters the *Closing* state. For more information, see Unbinding from an Adapter.

Unbind is complete
After the driver completes the unbind operation, the binding enters the Unbound state.

PnP Pause
After NDIS sends the protocol driver a network Plug and Play (PnP) pause event notification, the binding enters the Pausing state. For more information see Pausing a Binding.

Pause is complete
After the driver has completed all operations that are required to stop send and receive

operations, the pause operation is complete and the binding is in the Paused state.

Note The driver must wait for all its outstanding send requests to complete before the pause operation is complete.

PnP Restart
After NDIS sends the protocol driver a network PnP restart event notification, the binding enters the Restarting state. For more information, see Restarting a Binding.

Restart is complete
After the driver is ready to handle send and receive operations, the restart operation is complete and the binding is in the Running state.

Restart failed
If NDIS sends the protocol driver a network PnP restart event notification and the restart attempt fails, the binding returns to the Paused state.

Send and Receive Operations
A protocol driver must handle send and receive operations in the Running and Pausing states. For more information about send and receive operations, see Protocol Driver Send and Receive Operations.

OID Requests
A protocol driver can initiate OID requests to set or query information in underlying drivers. A protocol driver can initiate OID requests from all states, except Unbound and Opening.

Related topics

Writing NDIS Protocol Drivers

Binding to an Adapter

Article • 12/15/2021

NDIS calls a protocol driver's *ProtocolBindAdapterEx* function to open a binding whenever an underlying adapter to which the driver can bind becomes available. After NDIS calls *ProtocolBindAdapterEx*, the binding enters the Opening state. In the *Opening* state, the protocol driver allocates resources for the binding and opens the adapter.

NDIS passes to *ProtocolBindAdapterEx* the NDIS context for the binding operation as well as a pointer to an **NDIS_BIND_PARAMETERS** structure. This structure contains information about the adapter such as:

- The name of the adapter.
- The registry location for parameters specific to this binding under protocol service entry in the registry.
- The physical device object for the adapter.

To open an adapter, protocol drivers call the **NdisOpenAdapterEx** function. The protocol driver passes the following to **NdisOpenAdapterEx**:

- The handle that NDIS returned to the driver at the *NdisProtocolHandle* parameter of the **NdisRegisterProtocolDriver** function.
- The protocol driver's context for this binding.
- A pointer to a structure of type **NDIS_OPEN_PARAMETERS**.

NDIS_OPEN_PARAMETERS contains information such as name of the adapter that **NdisOpenAdapterEx** should open, an array of medium types that the protocol driver supports and, optionally, an array of frame types that the driver can receive on this binding.

If a protocol driver returns NDIS_STATUS_PENDING from *ProtocolBindAdapterEx*, it must call **NdisCompleteBindAdapterEx** with the final status to complete the bind request.

If NDIS returns NDIS_STATUS_PENDING from **NdisOpenAdapterEx**, NDIS later calls the protocol driver's *ProtocolOpenAdapterCompleteEx* function with the final status after the open request has been completed.

After the driver successfully opens the binding to the adapter, the binding is in the Paused state.

A protocol driver calls the **NdisCloseAdapterEx** function to close the adapter. The driver can call **NdisCloseAdapterEx** from the *ProtocolBindAdapterEx* function or *ProtocolUnbindAdapterEx* function.

If after opening the adapter and before completing the bind request, *ProtocolBindAdapterEx* encounters a failure and must close the binding to the adapter, it can call **NdisCloseAdapterEx**. For more information about closing an adapter, see Unbinding from an Adapter.

Unbinding from an Adapter

Article • 12/15/2021

NDIS calls a protocol driver's *ProtocolUnbindAdapterEx* function to request that the driver unbind from an underlying adapter. As the reciprocal of *ProtocolBindAdapterEx*, NDIS calls *ProtocolUnbindAdapterEx* to close the binding to the adapter and to release the resources that the driver allocated for the binding.

In *ProtocolUnbindAdapterEx*, a protocol driver calls **NdisCloseAdapterEx** to close the binding to an underlying adapter. The protocol driver passes **NdisCloseAdapterEx** the handle that **NdisOpenAdapterEx** provided at its *NdisBindingHandle* parameter. This handle identifies the binding that NDIS should close.

Protocol drivers must close an adapter from the *ProtocolBindAdapterEx* function or *ProtocolUnbindAdapterEx* function.

If a protocol driver must initiate an operation to close a binding, the driver can call **NdisUnbindAdapter**. **NdisUnbindAdapter** schedules a work item that results in an NDIS call to *ProtocolUnbindAdapterEx*. This work item can run before the call to **NdisUnbindAdapter** returns. Therefore, driver writers must assume that the binding handle is invalid after **NdisUnbindAdapter** returns.

If a protocol driver returns NDIS_STATUS_PENDING from *ProtocolUnbindAdapterEx*, it must call **NdisCompleteUnbindAdapterEx** with the final status to complete the bind request.

If NDIS returns NDIS_STATUS_PENDING from **NdisCloseAdapterEx**, NDIS later calls the protocol driver's *ProtocolCloseAdapterCompleteEx* function.

NDIS can call *ProtocolUnbindAdapterEx* if the binding is in the Paused state.

After all the unbind operations are complete, the binding is in the Unbound state.

Starting and Pausing a Binding

Article • 12/15/2021

NDIS pauses a binding to stop data flow that could interfere with Plug and Play (PnP) operations, for example, adding or removing a filter module in a driver stack, or, to add a new binding. For more information about how to modify a running driver stack, see Modifying a Running Driver Stack.

NDIS starts a binding from the *Paused* state. The binding enters the *Paused* state after the bind operation is complete or after a pause operation is complete.

The following topics provide more information about starting and pausing a binding:

Restarting a Binding

Pausing a Binding

Restarting a Binding

Article • 12/15/2021

To restart a binding that is paused, NDIS sends the protocol driver a network Plug and Play (PnP) restart event notification. After the protocol driver receives the restart notification, the affected binding enters the Restarting state.

To send a restart notification, NDIS calls a protocol driver's *ProtocolNetPnPEvent* function. The **NET_PNP_EVENT_NOTIFICATION** structure that NDIS passes to *ProtocolNetPnPEvent* specifies **NetEventRestart** in the **NetEvent** member and the **Buffer** member contains a pointer to the **NDIS_PROTOCOL_RESTART_PARAMETERS** structure. NDIS provides a pointer to an **NDIS_RESTART_ATTRIBUTES** structure in the **RestartAttributes** member of the NDIS_PROTOCOL_RESTART_PARAMETERS structure.

Note While the binding was paused, NDIS could have reconfigured the driver stack. The new stack configuration can support a different set of capabilities for the underlying adapter. These new capabilities can affect how the protocol driver communicates on a binding.

The protocol driver should use the information in the **NDIS_PROTOCOL_RESTART_PARAMETERS** structure to avoid unnecessary OID requests.

In the Restarting state, the protocol driver can:

- Use OID requests to query the driver stack. For example, the driver can find out about support for receive side scaling by using OID_GEN_RECEIVE_SCALE_CAPABILITIES.

- Reallocate **NET_BUFFER** and **NET_BUFFER_LIST** pools, if necessary.

- Enumerate the list of the underlying filter modules.

- Use OID requests to reveal new adapter capabilities.

After the driver is ready to resume send and receive operations for the binding, the binding enters the Running state.

Pausing a Binding

Article • 12/15/2021

After NDIS sends a protocol driver a network Plug and Play (PnP) pause event notification for a binding, the binding enters the Pausing state.

To notify the protocol driver of the PnP pause event, NDIS calls the *ProtocolNetPnPEvent* function with the **NetEvent** member of the **NET_PNP_EVENT_NOTIFICATION** structure is set to **NetEventPause**. The **Buffer** member contains an **NDIS_PROTOCOL_PAUSE_PARAMETERS** structure.

For a binding in the Pausing state, the protocol driver:

- Should not initiate any new send requests.

- Must wait for outstanding send requests to complete. The pause operation is not complete until NDIS calls the **ProtocolSendNetBufferListsComplete** function for all of the driver's outstanding send requests.

- Should handle receive indications as usual. The underlying miniport driver waits for outstanding receive data to return before completing a pause operation. This ensures that there are no ongoing receive operations in the driver stack after the miniport driver is paused.

- Should return new receive indications to NDIS immediately. If necessary, the driver can copy such receive indications before it returns them.

For more information about protocol driver send and receive operations, see Protocol Driver Send and Receive Operations.

A binding enters the Paused state after the protocol driver is done returning outstanding receive indications for the binding and NDIS has completed all of the outstanding send requests for the binding.

For a binding in the Paused state, the protocol driver:

- Must not make any send requests.

- Should return receive indications immediately. If necessary, the driver can copy such receive indications before it returns them.

Configuring Optional Protocol Driver Services

Article • 12/15/2021

NDIS calls a protocol driver's *ProtocolSetOptions* function to allow a protocol driver to configure optional services. NDIS calls *ProtocolSetOptions* within the context of the protocol driver's call to the **NdisRegisterProtocolDriver** function

ProtocolSetOptions registers default entry points for optional *ProtocolXxx* functions and can allocate other driver resources. To register optional *ProtocolXxx* functions, the protocol driver calls the **NdisSetOptionalHandlers** function and passes a characteristics structure at the *OptionalHandlers* parameter. In this case, the protocol driver passes the handle from the *NdisDriverHandle* parameter of *ProtocolSetOptions* at the *NdisHandle* parameter of **NdisSetOptionalHandlers**.

A protocol driver can also call **NdisSetOptionalHandlers** from the *ProtocolBindAdapterEx* function or the *ProtocolOpenAdapterCompleteEx* function after the protocol driver has a valid handle from the **NdisOpenAdapterEx** function. In this case, the protocol driver passes the handle from the *NdisBindingHandle* parameter of **NdisOpenAdapterEx** at the *NdisHandle* parameter of **NdisSetOptionalHandlers**.

In this case, the valid characteristics structures are:

NDIS_PROTOCOL_CO_CHARACTERISTICS

NDIS_CO_CLIENT_OPTIONAL_HANDLERS

NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS

NDIS_CLIENT_CHIMNEY_OFFLOAD_GENERIC_CHARACTERISTICS (see NDIS 6.0 TCP chimney offload documentation)

NDIS_CLIENT_CHIMNEY_OFFLOAD_TCP_CHARACTERISTICS (see NDIS 6.0 TCP chimney offload documentation)

Protocol Driver Send and Receive Operations

Article • 12/15/2021

Protocol drivers originate send requests and handle the receive indications of underlying drivers. In a single function call, NDIS protocol drivers can send multiple **NET_BUFFER_LIST** structures with multiple **NET_BUFFER** structures on each NET_BUFFER_LIST structure. In the receive path, protocol drivers can receive a list of NET_BUFFER_LIST structures.

Protocol drivers must manage send buffer pools. Proper management of such pools requires preallocation of sufficient buffer space to optimize system performance.

The following topics provide more information about protocol driver buffer management, send operations, and receive operations:

Protocol Driver Buffer Management

Sending Data from a Protocol Driver

Receiving Data in Protocol Drivers

Protocol Driver Buffer Management

Article • 12/15/2021

A protocol driver must manage NET_BUFFER_LIST structure pools and NET_BUFFER structure pools for send operations. To create these pools, drivers call the following functions:

NdisAllocateNetBufferListPool

NdisAllocateNetBufferPool

Protocol drivers can use the following functions to allocate structures from the pools:

NdisAllocateNetBufferAndNetBufferList

NdisAllocateNetBufferList

NdisAllocateNetBuffer

Calling **NdisAllocateNetBufferAndNetBufferList** is more efficient than calling **NdisAllocateNetBufferList** followed by **NdisAllocateNetBuffer**. However, **NdisAllocateNetBufferAndNetBufferList** only creates one NET_BUFFER structure on the NET_BUFFER_LIST structure. To use **NdisAllocateNetBufferAndNetBufferList**, the driver must set the *AllocateNetBuffer* parameter to **TRUE** when it calls **NdisAllocateNetBufferListPool**.

Protocol drivers can use OID requests to query the back-fill and context space requirements of the underlying drivers. A protocol driver should determine the back-fill and context requirements for a binding in the *Opening* or *Restarting* states. The driver should allocate sufficient back-fill and context space for the entire stack. If necessary, a protocol driver can free the pools and reallocate them in the *Restarting* state.

Protocol drivers use the following functions to free the pools:

NdisFreeNetBufferListPool

NdisFreeNetBufferPool.

Protocol drivers use the following functions to free the structures allocated from the pools:

NdisFreeNetBufferList

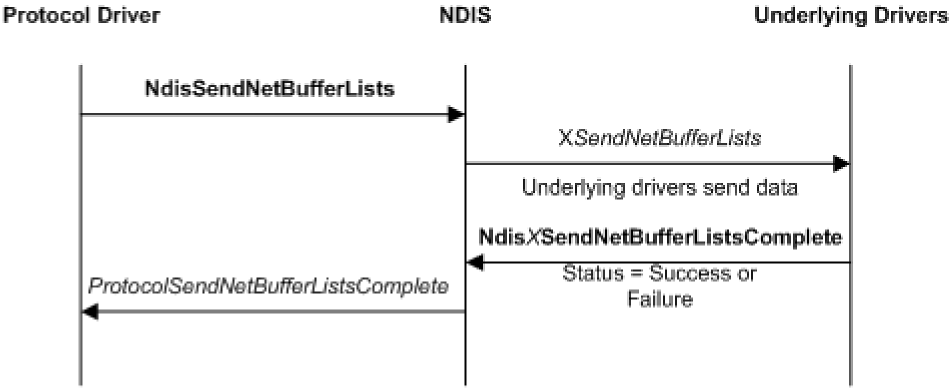
NdisFreeNetBuffer

Drivers should free NET_BUFFER structures allocated with **NdisAllocateNetBuffer** before freeing the associated NET_BUFFER_LIST structure. NET_BUFFER structures allocated with **NdisAllocateNetBufferAndNetBufferList** are freed when the driver calls **NdisFreeNetBufferList** for the associated NET_BUFFER_LIST structure.

Sending Data from a Protocol Driver

Article • 12/15/2021

The following figure illustrates a protocol driver send operation, which involves a protocol driver, NDIS, and underlying drivers in a driver stack.


Protocol Driver
NDIS
Underlying Drivers
NdisSendNetBufferLists
XSendNetBufferLists
Underlying drivers send data
NdisXSendNetBufferListsComplete
Status = Success or Failure
ProtocolSendNetBufferListsComplete

Protocol drivers call the NdisSendNetBufferLists function to send the network data that is defined in a list of NET_BUFFER_LIST structures.

A protocol driver must set the **SourceHandle** member of each NET_BUFFER_LIST structure to the same value that it passes to the *NdisBindingHandle* parameter. The binding handle provides the information that NDIS requires to return the NET_BUFFER_LIST structure to the protocol driver after the underlying miniport driver calls NdisMSendNetBufferListsComplete.

Before calling **NdisSendNetBufferLists**, a protocol driver can set information that accompanies the send request with the NET_BUFFER_LIST_INFO macro. The underlying drivers can retrieve this information with the NET_BUFFER_LIST_INFO macro.

As soon as a protocol driver calls **NdisSendNetBufferLists**, it relinquishes ownership of the NET_BUFFER_LIST structures and all associated resources. NDIS calls the ProtocolSendNetBufferListsComplete function to return the structures and data to the protocol driver. NDIS can collect the structures and data from multiple send requests into a single linked list of NET_BUFFER_LIST structures before it passes the list to *ProtocolSendNetBufferListsComplete*.

Until NDIS calls *ProtocolSendNetBufferListsComplete*, the current status of a protocol-driver-initiated send is unknown. A protocol driver temporarily releases ownership of all resources it allocated for a send request when it calls **NdisSendNetBufferLists**. A protocol driver should never attempt to examine the NET_BUFFER_LIST structures or any associated data before NDIS returns the structures to *ProtocolSendNetBufferListsComplete*.

ProtocolSendNetBufferListsComplete performs whatever postprocessing is necessary to complete a send operation. For example, the protocol driver can notify the clients, that requested the protocol driver to send the network data, that the send operation is complete.

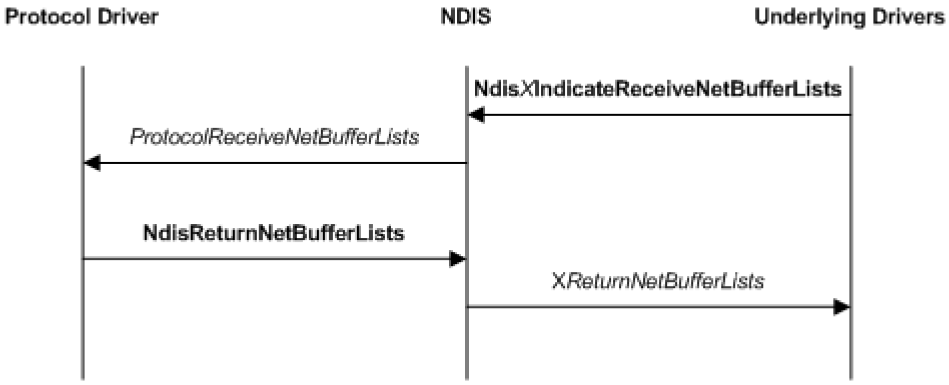
When NDIS calls *ProtocolSendNetBufferListsComplete*, the protocol driver regains ownership of all of the resources associated with the NET_BUFFER_LIST structures that are specified by the *NetBufferLists* parameter. *ProtocolSendNetBufferListsComplete* can either free these resources (for example, by calling **NdisFreeNetBuffer** and **NdisFreeNetBufferList**) or prepare them for reuse in a subsequent call to **NdisSendNetBufferLists**.

Although NDIS always submits protocol-supplied network data to the underlying miniport driver in the protocol-determined order as passed to **NdisSendNetBufferLists**, the underlying driver can complete the send requests in random order. That is, every bound protocol driver can rely on NDIS to submit the network data that the protocol driver passes to **NdisSendNetBufferLists** in FIFO order to the underlying driver. However, no protocol driver can rely on the underlying driver to call **NdisMSendNetBufferListsComplete** in the same order.

Receiving Data in Protocol Drivers

Article • 12/15/2021

The following figure illustrates a basic receive operation, which involves a protocol driver, NDIS, and underlying drivers in a driver stack.



NDIS calls a protocol driver's *ProtocolReceiveNetBufferLists* function to process receive indications that come from underlying drivers. NDIS calls *ProtocolReceiveNetBufferLists* after an underlying driver calls a receive indication function (for example, **NdisMIndicateReceiveNetBufferLists**) to indicate received network data or loop-back data.

If the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *ProtocolReceiveNetBufferLists* is not set, the protocol driver retains ownership of the **NET_BUFFER_LIST** structures until it calls the **NdisReturnNetBufferLists** function. If NDIS sets the **NDIS_RECEIVE_FLAGS_RESOURCES** flag, the protocol driver cannot retain the **NET_BUFFER_LIST** structure and the associated resources. The set **NDIS_RECEIVE_FLAGS_RESOURCES** flag indicates that an underlying driver is running low on receive resources. In this case, the *ProtocolReceiveNetBufferLists* function should copy the received data into protocol-allocated storage and return as quickly as possible.

Note NDIS can change the flags that an underlying driver indicates. For example, if a miniport driver sets the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of the **NdisMIndicateReceiveNetBufferLists** function, NDIS can copy the indicated data and pass the copy to *ProtocolReceiveNetBufferLists* with the **NDIS_RECEIVE_FLAGS_RESOURCES** flag cleared.

Note If the **NDIS_RECEIVE_FLAGS_RESOURCES** flag is set, the protocol driver must retain the original set of **NET_BUFFER_LIST** structures in the linked list. For example, when this flag is set the driver might process the structures and indicate them up the stack one at a time but before the function returns it must restore the original linked list.

Protocol drivers call the **NdisReturnNetBufferLists** function to release ownership of a list of **NET_BUFFER_LIST** structures, along with the associated **NET_BUFFER** structures, and network data.

Protocol Driver OID Requests

Article • 12/15/2021

NDIS defines object identifier (OID) values to identify adapter parameters which include operating parameters such as device characteristics, configurable settings and statistics. For more information about OIDs, see NDIS OIDs.

Protocol drivers can query or set the operating parameters of underlying drivers.

NDIS also provides a direct OID request interface for NDIS 6.1 and later protocol drivers. The *direct OID request path* supports OID requests that are queried or set frequently. For example, the IPsec offload version 2 (IPsecv2) interface provides the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID for direct OID requests. The direct OID request interface is optional for NDIS drivers.

The following topics provide more information about protocol driver OID requests:

Generating OID Requests from an NDIS Protocol Driver

Protocol Driver Direct OID Requests

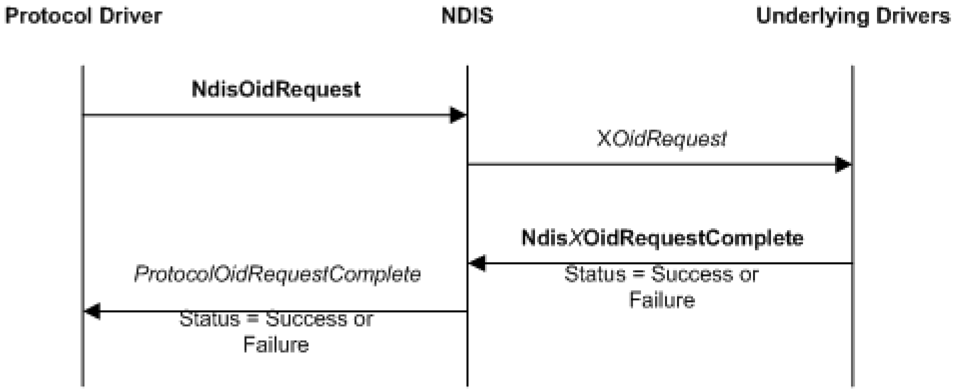
Protocol Driver Synchronous OID Requests

Generating OID Requests from an NDIS Protocol Driver

Article • 03/14/2023

To originate an OID request to underlying drivers, a protocol calls the **NdisOidRequest** function.

The following figure illustrates an OID request that is originated by a protocol driver.



After a protocol driver calls the **NdisOidRequest** function, NDIS calls the request function of the next underlying driver. For more information about how a miniport driver handles OID requests, see OID Requests for an Adapter. For more information about how a filter driver handles OID requests, see Filter Module OID Requests.

To complete synchronously, **NdisOidRequest** returns NDIS_STATUS_SUCCESS or an error status. To complete asynchronously, **NdisOidRequest** returns NDIS_STATUS_PENDING.

If **NdisOidRequest** returns NDIS_STATUS_PENDING, NDIS calls the **ProtocolOidRequestComplete** function after the underlying drivers complete the OID request. In this case, NDIS passes the results of the request at the *OidRequest* parameter of *ProtocolOidRequestComplete*. NDIS passes the final status of the request at the *Status* parameter of *ProtocolOidRequestComplete*.

If **NdisOidRequest** returns NDIS_STATUS_SUCCESS, it returns the results of a query request in the **NDIS_OID_REQUEST** structure at the *OidRequest* parameter. In this case, NDIS does not call the *ProtocolOidRequestComplete* function.

To determine what information was successfully handled by an underlying driver, protocol drivers that issue OID requests must check the value in the **SupportedRevision** member in the NDIS_OID_REQUEST structure after the OID request returns. For more information about NDIS version information, see Specifying NDIS Version Information.

If the underlying driver should associate the OID request with a subsequent status indication, the protocol driver should set the **RequestId** member in the NDIS_OID_REQUEST structure. When the underlying driver makes a status indication, it sets the **RequestId** member in the NDIS_STATUS_INDICATION structure to the value that is provided in the OID request.

A driver can call **NdisOidRequest** when a binding is in the *Restarting*, *Running*, *Pausing*, or *Paused* state.

Protocol Driver Direct OID Requests

Article • 12/15/2021

To support the direct OID request path, protocol drivers provide *ProtocolXxx* function entry points in the NDIS_PROTOCOL_DRIVER_CHARACTERISTICS structure and NDIS provides **Ndis*Xxx*** functions for protocol drivers.

The *direct OID request interface* is similar to the standard OID request interface. For example, the **NdisDirectOidRequest** and **ProtocolDirectOidRequestComplete** functions are similar to the **NdisOidRequest** and **ProtocolOidRequestComplete** functions.

Note NDIS 6.1 and later support specific OIDs for use with the direct OID request interface. OIDs that existed before NDIS 6.1 and some NDIS 6.1 OIDs are not supported. To determine if an OID can be used in the direct OIDs interface, see the OID reference page. For example, see the note in the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID.

To support the direct OIDs request interface, use the documentation for the standard OID request interface. The following table shows the relationship between the functions in the direct OID request interface and the standard OID request interface.

Direct OID function	Standard OID function
ProtocolDirectOidRequestComplete	ProtocolOidRequestComplete
NdisDirectOidRequest	NdisOidRequest
NdisCancelDirectOidRequest	NdisCancelOidRequest

Protocol Driver Synchronous OID Requests

Article • 12/15/2021

To support the Synchronous OID request path, protocol drivers call the **NdisSynchronousOidRequest** function to issue a Synchronous OID.

For protocol drivers, the *Synchronous OID request interface* differs from the Regular and Direct OID request interfaces in that protocol drivers do not have to implement an asynchronous *complete* callback function. This is because of the synchronous nature of the path. For more info about the differences between Regular, Direct, and Synchronous OIDs in general, see Synchronous OID Request Interface in NDIS 6.80.

> ⓘ Note
>
> NDIS 6.80 supports specific OIDs for use with the Synchronous OID request interface. OIDs that existed before NDIS 6.80 and some NDIS 6.80 OIDs are not supported. To determine if an OID can be used in the Synchronous OID request interface, see the OID reference page.

To support the Synchronous OID request interface, use the documentation for the standard OID request interface. The following table shows the relationship between the functions in the Synchronous OID request interface and the standard OID request interface.

Synchronous OID function	Standard OID function
NdisSynchronousOidRequest	*NdisOidRequest*

Handling Status Indications in a Protocol Driver

Article • 01/18/2023

Protocol drivers must supply a **ProtocolStatusEx** function that NDIS calls when an underlying driver reports status.

NDIS calls a protocol driver's *ProtocolStatusEx* function, after an underlying driver calls a status indication function (**NdisMIndicateStatus** or **NdisFIndicateStatus**). For more information about indicating status from a miniport driver, see Adapter Status Indications.

For more information about indicating status from a filter driver, see Filter Module Status Indications.

If the status indication is associated with an OID request, the underlying driver can set the **DestinationHandle** and **RequestId** members so that NDIS can provide the status indication to a specific protocol binding. For more information about OID requests, see Protocol Driver OID Requests.

Handling PnP Event Notifications in a Protocol Driver

Article • 12/15/2021

NDIS 6.0 and later protocol drivers handle the same Plug and Play (PnP) event notifications as NDIS 5.x drivers in addition to event notifications that are specific to NDIS 6.0 and later. The handling of PnP event notifications is driver specific.

To notify a protocol driver of a network PnP event, NDIS calls the driver's *ProtocolNetPnPEvent* function. To define the type of event and characteristics of the event, NDIS passes a **NET_PNP_EVENT_NOTIFICATION** structure at the *NetPnPEvent* event parameter of *ProtocolNetPnPEvent*.

Protocol drivers should handle driver stack changes. For more information about driver stack changes, see Modifying a Running Driver Stack.

Protocol drivers that do not handle stack change notifications are unbound from the adapter and rebound. Bindings for protocol drivers that handle driver stack notifications successfully are not affected.

Protocol drivers should handle driver stack pause notifications. For more information about these notifications, see Pausing a Driver Stack.

Protocol drivers should handle driver stack restart notifications. For more information about these notifications, see Restarting a Driver Stack.

NDIS protocol driver installation

Article • 05/30/2023

To install a protocol driver, you must first provide a single INF file. The configuration manager reads configuration information about the protocol driver from the INF file and copies it to the registry.

For more information about protocol driver INF files, see Installation Requirements for Network Protocols. For an example protocol driver INF file, see the ndisprot 630 sample driver.

Once you have provided your protocol driver INF file, to install or uninstall your protocol driver you must use the `INetCfg` family of Network Configuration Interfaces. For example, to install or remove network components, call into the INetCfgClassSetup interface. You can either call into these interfaces programmatically or you can indirectly call them with netcfg.exe, which calls `INetCfg` for you. You can't install a driver package through the `INetCfg` and use the Driver Store feature on older Windows versions. To successfully install the driver package in this scenario, you need to have a minimum OS build number of 25319. You can't use SetupAPI to install or uninstall an NDIS protocol driver.

For an example of calling into `INetCfg` through code, see the Bindview Network Configuration Utility sample .

Pageable and Discardable Code in a Protocol Driver

Article • 12/15/2021

Driver developers should designate code as pageable whenever possible, freeing system space for code that must be memory-resident. You can mark functions as pageable with the **NDIS_PAGEABLE_FUNCTION** macro. The IRQL, resource management features, and other characteristics of a function might prohibit the function from being pageable.

Every *ProtocolXxx* function runs at an IRQL in the range from PASSIVE_LEVEL to DISPATCH_LEVEL. Functions that run exclusively at IRQL = PASSIVE_LEVEL should be marked as pageable.

A driver function that runs at IRQL = PASSIVE_LEVEL can be made pageable as long as it neither calls nor is called by any function that runs at IRQL >= DISPATCH_LEVEL--such as a function that acquires a spin lock. Acquiring a spin lock causes the IRQL of the acquiring thread to be raised to DISPATCH_LEVEL. A driver function, such as *ProtocolBindAdapterEx*, that runs at IRQL = PASSIVE_LEVEL must not call any **Ndis*Xxx*** functions that run at IRQL >= DISPATCH_LEVEL if that driver function is marked as pageable code. For more information about the IRQL for each **Ndis*Xxx*** function, see NDIS Library Functions.

The **DriverEntry** function of NDIS protocol drivers, as well as code that is called only from **DriverEntry**, should be specified as initialization-only code, by using the **NDIS_INIT_FUNCTION** macro. Code that is identified with this macro is assumed to run only once at system initialization time, and, as a result, is mapped only during that time. After a function marked as initialization-only returns, it is discarded.

Protocol Driver Reset Operations

Article • 12/15/2021

Protocol drivers cannot initiate a reset operation in NDIS 6.0 and later versions.

Typically, an underlying miniport driver resets a NIC because the NIC is timing out during send or request operations. This condition causes NDIS to call the miniport driver's *MiniportCheckForHangEx* and subsequently *MiniportResetEx* functions. Alternatively, the miniport driver determines a NIC's receive capability is dysfunctional.

If a reset is initiated by NDIS and *MiniportResetEx* returns NDIS_STATUS_PENDING, NDIS calls the **ProtocolStatusEx**(or **ProtocolCoStatusEx**) function of each bound protocol driver with a status of NDIS_STATUS_RESET_START. When the miniport driver calls **NdisMResetComplete**, NDIS again calls *ProtocolStatusEx*(or *ProtocolCoStatusEx*) with a status of NDIS_STATUS_RESET_END.

A protocol driver must handle the possibility that outstanding sends on a binding to an underlying NIC can be canceled because the NIC is reset. If a bound protocol driver has any transmit requests pending, NDIS will indicate a send complete to the protocol driver with an appropriate status. The protocol driver must resubmit the send requests when the reset operation is completed, assuming the NIC becomes operational again.

When a protocol driver receives a status of NDIS_STATUS_RESET_START, it should:

- Hold any network data that is ready to be transmitted until *Protocol(Co)Status* receives an NDIS_STATUS_RESET_END notification.

- Not make any NDIS calls that are directed to the underlying miniport driver, except calls to return resources such as returning network data with **NdisReturnNetBufferLists**.

After *ProtocolStatusEx*(or *ProtocolCoStatusEx*) receives an NDIS_STATUS_RESET_END message, the protocol driver can resume sending network data and OID requests.

Handling PnP Events and Power Management Events in a Protocol Driver

Article • 12/15/2021

When the operating system issues a Plug and Play (PnP) I/O request packet (IRP) or a power management IRP to a target device object that represents a network interface card (NIC), NDIS intercepts the IRP. NDIS indicates the event to each bound protocol driver and each bound intermediate driver by calling the driver's *ProtocolNetPnPEvent* function. In the call to *ProtocolNetPnPEvent*, NDIS passes a pointer to a **NET_PNP_EVENT_NOTIFICATION** that contains a NET_PNP_EVENT structure. The NET_PNP_EVENT structure describes the PnP event or power management event being indicated. For more information about the protocol driver PnP interface, see Handling PnP Event Notifications in a Protocol Driver.

The following list contains PnP and power management events, as indicated by the **NetEvent** code in the NET_PNP_EVENT structure:

- **NetEventSetPower**

  Indicates a Set Power request, which specifies that the miniport adapter should transition to a particular power state. A power management–aware protocol driver should always succeed this event by returning NDIS_STATUS_SUCCESS. An old protocol driver can return NDIS_STATUS_NOT_SUPPORTED to indicate that NDIS should unbind it from the miniport adapter.

  After issuing the set power request, NDIS pauses the driver stack if the miniport adapter is transitioning to a low-power state. NDIS restarts the driver stack before the set-power request if the miniport adapter is transitioning to the working state (D0). For more information about pausing and restarting the driver stack, see Pausing a Driver Stack.

  If the miniport adapter is in a low-power state, the protocol driver cannot issue any OID requests. This requirement is an additional power management restriction that is added to the other restrictions that apply when the driver stack is in the Paused state.

  If the underlying miniport adapter is not power management–aware, the miniport driver sets the **PowerManagementCapabilities** member of **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** to **NULL** and NDIS sets the **PowerManagementCapabilities** member of **NDIS_BIND_PARAMETERS** to **NULL**.

Note Starting with NDIS 6.30, after being notified of this event, the protocol driver must stop generating new I/O requests and should not wait for the completion of any pending I/O requests within the context of the call to *ProtocolNetPnPEvent*.

For more information about set-power events, see Handling PnP Events and Power Management Events in an Intermediate Driver.

- **NetEventQueryPower**

  Indicates a Query Power request, which queries whether the underlying miniport adapter can make a transition to a particular power state. A protocol driver should always succeed a **NetEventQueryPower** . After establishing an active connection, a protocol driver can call **PoRegisterSystemState** to register a continuous busy state. As long as the state registration is in effect, the power manager does not attempt to put the system to sleep. After the connection becomes inactive, the protocol driver cancels the state registration by calling **PoUnregisterSystemState**. A protocol driver should never try to prevent the system from transitioning to the sleeping state by failing a **NetEventQueryRemoveDevice**. Note that a **NetEventQueryPower** is always followed by a **NetEventSetPower**. A **NetEventSetPower** that sets the device's current power state in effect cancels the **NetEventQueryPower**.

  Note Starting with NDIS 6.30, after being notified of this event, the protocol driver should not wait for the completion of any pending I/O requests within the context of the call to *ProtocolNetPnPEvent*.

- **NetEventQueryRemoveDevice**

  Indicates a Query Remove Device request, which queries whether the NIC can be removed without disrupting operations. If a protocol driver cannot release a device (for example, because the device is in use), it must fail a **NetEventQueryRemoveDevice** by returning NDIS_STATUS_FAILURE.

- **NetEventCancelRemoveDevice**

  Indicates a Cancel Remove Device request, which cancels the removal of an underlying NIC. The protocol driver should always succeed this event by returning NDIS_STATUS_SUCCESS.

- **NetEventReconfigure**

  Indicates that the configuration has changed for a network component. For example, if a user changes the IP address for TCP/IP, NDIS indicates this event to the TCP/IP protocol with the **NetEventReconfigure** code. The protocol driver can, in rare circumstances, return a failure code if it is not able to apply the indicated configuration changes and there are no available default values. A failed attempt to allocate memory is an example of a case in which the protocol returns a failure code. Returning an error code can result in prompting the user to restart the system.

A protocol should validate **NetEventReconfigure**-related data passed to its *ProtocolNetPnPEvent* function. For more information about such data, see **NET_PNP_EVENT for Protocol Drivers**.

- **NetEventBindList**

  Indicates to a protocol driver that its bind list processing order has been reconfigured. This list indicates a relative order to be applied to the protocol's bindings when processing, for example, a user request that might be routed to one of several bindings. The buffer passed with this event contains a list of device names formatted as NULL-terminated Unicode strings. The format of each device name is identical to the *DeviceName* parameter that is passed to a call to *ProtocolBindAdapterEx*.

  A protocol should validate **NetEventBindList**-related data passed to its *ProtocolNetPnPEvent* function. For more information about such data, see **NET_PNP_EVENT for Protocol Drivers**.

  A protocol should validate **NetEventBindList**-related data passed to its *ProtocolNetPnPEvent* function. For more information about such data, see **NET_PNP_EVENT for Protocol Drivers**.

- **NetEventBindsComplete**

  Indicates that a protocol driver has bound to all the NICs to which it can bind. NDIS will not indicate any more bindings to the protocol driver unless, for example, a PnP NIC is plugged into the system.

- **NetEventPnPCapabilities**

  Indicates that the user enabled or disabled the wake-up capabilities of the underlying adapter. (The *ProtocolBindingContext* parameter that NDIS passes to *ProtocolNetPnPEvent* specifies the binding .)

- **NetEventPause**

  Indicates that the specified protocol binding should enter thePausing state. The binding will enter the Paused state after NDIS has completed all of the outstanding send requests for the binding. For more information about pausing a binding, see Pausing a Binding.

- **NetEventRestart**

  Indicates that the specified protocol binding has entered the Restarting state. After the protocol driver is ready to resume send and receive operations for the binding, the binding enters the Running state. For more information about restarting a binding, see Restarting a Binding.

- **NetEventPortActivation**

  Indicates the activation of a list of ports that are associated with the specified binding. For more information about pausing a binding, see Handling the Port Activation PnP Event.

- **NetEventPortDeactivation**

  Indicates the deactivation of a list of ports that are associated with the specified binding. For more information about pausing a binding, see Handling the Port Deactivation PnP Event.

- **NetEventIMReEnableDevice**

  Indicates that the configuration has changed for a virtual miniport of an NDIS 6.0 or later intermediate driver. **NetEventIMReEnableDevice** is similar to the **NetEventReconfigure** event except that the intermediate driver receives this event for a single virtual miniport and the **NetEventReconfigure** event applies to all of the intermediate driver's virtual miniports. For example, an intermediate driver receives the **NetEventIMReEnableDevice** event when a user disables and then enables a single virtual miniport from the Device Manager or another source. For examples of intermediate driver power management, see the NDIS MUX Intermediate Driver and Notify Object driver sample available in the Windows driver samples repository on GitHub.

The **Buffer** member of the NET_PNP_EVENT structure points to a buffer that contains information specific to the event being indicated.

A protocol driver can complete the call to *ProtocolNetPnPEvent* asynchronously with **NdisCompleteNetPnPEvent**.

Send and Receive Operations in Protocol Drivers

Article • 12/15/2021

There are two different interfaces for send and receive operations in NDIS protocol drivers. Protocol drivers with a connectionless lower edge call the **NdisSendNetBufferLists** function to send network data. A connectionless protocol driver must supply a **ProtocolReceiveNetBufferLists** function. NDIS calls *ProtocolReceiveNetBufferLists* when an underlying connectionless miniport driver calls the **NdisMIndicateReceiveNetBufferLists** function to indicate received network data. For more information about sending and receiving data in connectionless protocol drivers, see Protocol Driver Send and Receive Operations.

Connection-oriented NDIS (CoNDIS) protocol drivers call the **NdisCoSendNetBufferLists** function to send network data. A CoNDIS protocol driver must supply a **ProtocolCoReceiveNetBufferLists** function. NDIS calls *ProtocolCoReceiveNetBufferLists* when an underlying CoNDIS miniport driver calls the **NdisMCoIndicateReceiveNetBufferLists** function to indicate received network data. For more information about send and operations in connection-oriented protocol drivers, see Connection-Oriented Operations.

For an introduction to send and receive operations, see Send and Receive Operations.

OID Request Operations in a Protocol Driver

Article • 12/15/2021

There are two different interfaces for OID request operations in a protocol driver. NDIS protocol drivers with a connectionless lower edge call the **NdisOidRequest** function to initiate an OID request. An NDIS protocol driver with a connectionless lower edge must supply a **ProtocolOidRequestComplete** function. NDIS calls *ProtocolOidRequestComplete* when the underlying drivers complete a pending OID request. For more information about OID requests in connectionless protocol drivers, see Protocol Driver OID Requests.

Connection-oriented NDIS (CoNDIS) protocol drivers call the **NdisCoOidRequest** function to initiate an OID request. A CoNDIS protocol driver must supply a **ProtocolCoOidRequestComplete** function. NDIS calls *ProtocolOidRequestComplete* when the underlying drivers complete a pending OID request. For more information OID requests in connection-oriented protocol drivers, see Connection-Oriented Operations.

For more information about OIDs, see NDIS OIDs.

Status Indications in a Protocol Driver

Article • 12/15/2021

There are two different interfaces for status indications in a protocol driver. An NDIS protocol driver with a connectionless lower edge is required to supply a **ProtocolStatusEx** function. NDIS calls *ProtocolStatusEx* when an underlying connectionless miniport driver calls **NdisMIndicateStatusEx** to report a change in its hardware status. NDIS calls *ProtocolStatusEx* when the status change begins. For more information about status indications in connectionless protocol drivers, see Handling Status Indications in a Protocol Driver.

A connection-oriented protocol driver must supply a **ProtocolCoStatusEx** function. NDIS calls *ProtocolCoStatusEx* when an underlying connection-oriented miniport driver calls **NdisMCoIndicateStatusEx** to report a change in its hardware status. NDIS calls *ProtocolCoStatusEx* when the status change begins. For more information about status indications in connection-oriented protocol drivers, see Connection-Oriented Operations

For a complete list of the possible status indications, see Status Indications.

Roadmap for Developing NDIS Filter Drivers

Article • 03/14/2023

To create a Network Driver Interface Specification (NDIS) filter driver package, follow these steps:

- Step 1: Learn about Windows architecture and drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- Step 2: Learn about NDIS.

  For general information about NDIS and NDIS drivers, see the following topics:

  Windows Network Architecture and the OSI Model

  Network Driver Programming Considerations

  Driver Stack Management

  NET_BUFFER Architecture

- Step 3: Determine additional Windows driver design decisions.

  For more information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- Step 4: Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user-mode application. For more information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Compatibilty testing, see Building, Debugging, and Testing Drivers. For more information about building, testing, verifying, and debugging tools, see Driver Development Tools.

- Step 5: Read the filter driver introduction topics.

- Step 6: Read the writing protocol drivers section.

This section provides an overview of the primary protocol driver interfaces. These interfaces included functions that protocol drivers provide (*ProtocolXxx* functions) and NDIS calls to initiate operations. NDIS provides **Ndis*Xxx*** functions that protocol drivers call to perform NDIS operations.

- Step 7: Review the NDIS filter driver sample in the Windows driver samples repository on GitHub.

- Step 8: Develop (or port), build, test, and debug your NDIS driver.

  See the porting guides if you are porting an existing driver:
  - Porting NDIS 5.x Drivers to NDIS 6.0
  - Porting NDIS 6.x Drivers to NDIS 6.20
  - Porting NDIS 6.x Drivers to NDIS 6.30

  For more information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process will help ensure that you build a driver that works.

- Step 9: Create a driver package for your driver.

  For more information about how to install drivers, see Providing a Driver Package. For more information about how to install an NDIS driver, see Components and Files Used for Network Component Installation and Notify Objects for Network Components.

- Step 10: Sign and distribute your driver.

  The final step is to sign (optional) and distribute the driver. If your driver meets the quality standards that are defined for the Windows Hardware Compatibilty Program, you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual driver.

Getting started with NDIS Filter Drivers

Article • 03/14/2023

Filter drivers provide filtering services for miniport drivers. NDIS driver stacks must include miniport drivers and protocol drivers and optionally include filter drivers. For more information about NDIS drivers and the driver stack, see Driver Stack Management.

The following applications might require a filter driver:

- Data filtering applications for security or other purposes.
- Applications that monitor and collect network data statistics.

The following topics provide an introduction to filter driver characteristics and services:

Filter Driver Characteristics

Filter Driver Services

Types of Filter Drivers

Mandatory Filter Drivers

Filter Driver Characteristics

Article • 12/15/2021

Filter drivers have the following characteristics:

- An instance of a filter driver is called a *filter module*. Filter modules are attached to an underlying miniport adapter. Multiple filter modules from the same filter driver or different filter drivers can be stacked over an adapter.

- Overlying protocol drivers are not required to provide alternate functionality when filter modules are installed between such drivers and the underlying miniport drivers (otherwise stated, filter modules are transparent to overlying protocol drivers).

- Because filter drivers do not implement virtual miniports like an intermediate driver, filter drivers are not associated with a device object. A miniport adapter with overlying filter modules functions as a modified version of the miniport adapter. For more information about the driver stack, see NDIS 6.0 Driver Stack.

- NDIS uses configuration information to attach the filter modules to the adapter in the correct driver stack order. For more information about the driver stack order of filter modules, see INF File Settings for Filter Drivers.

- NDIS can dynamically insert or delete filter modules in the driver stack, or reconfigure the filter modules, without tearing down the entire stack. For more information, see Modifying a Running Driver Stack.

- Protocol drivers can obtain the list of filter modules in a driver stack when NDIS restarts the driver stack.

  For more information about the list of filter modules, see **NDIS_PROTOCOL_RESTART_PARAMETERS**.

- Filter drivers can filter most communication to and from the underlying miniport adapter. Filter modules are not associated with any particular binding between overlying protocol drivers and the miniport adapter. For more information about the types of filtering services that a filter driver can provide, see Filter Driver Services.

- Filter drivers can select the services that are filtered and can be bypassed for the services that are not filtered. The selection of the services that are bypassed and the services that are filtered can be reconfigured dynamically. For more information, see Data Bypass Mode.

- NDIS guarantees the availability of context space (see NET_BUFFER_LIST_CONTEXT structure) for filter drivers. Therefore, filter drivers are not required to include the code to copy buffers to obtain context space. For more information about how to manage buffers, see Filter Driver Buffer Management.

Filter Driver Services

Article • 12/15/2021

Filter drivers can provide the following services:

- Originate send requests and receive indications.
- Modify data buffer ordering or timing in the send and receive data paths.
- Add, modify, delete network data buffers in both the send and receive data paths of a driver stack. For more information about filtering send and receive data, see Filter Module Send and Receive Operations.
- Originate query and set OID requests to the underlying drivers.
- Filter query and set OID requests to the underlying drivers.
- Filter responses of OID requests from the underlying drivers. For more information about OID requests, see Filter Module OID Requests.
- Originate status indications to the overlying drivers.
- Filter status indications from the underlying drivers. For more information, see Filter Module Status Indications.
- Manage configuration parameters in the registry for each miniport adapter with which it interfaces. For more information, see Accessing Configuration Information for a Filter Driver.

Types of Filter Drivers

Article • 12/15/2021

There are two primary types of filter drivers:

Monitoring
These filter drivers monitor the behavior in a driver stack. However, they only pass on information and do not modify the behavior of the driver stack. Monitoring filter drivers cannot modify or originate data.

Modifying
These filter drivers modify the behavior of the driver stack. The type of modification is driver-specific.

The **FilterType** entry in the INF file is 0x00000001 for monitoring filter drivers and 0x00000002 for modifying filter drivers.

You can specify that a filter driver is mandatory. This feature is generally used with modifying filter drivers. If a mandatory filter driver does not load, the associated driver stack will be torn down. For more information about mandatory filter drivers, see Mandatory Filter Drivers.

Mandatory Filter Drivers

Article • 12/15/2021

Mandatory filter drivers are filter drivers that must be present for a driver stack to function properly. If the mandatory filter module does not attach, the rest of the driver stack will be torn down. Modifying or monitoring filter drivers can be mandatory. All filter intermediate drivers are optional.

To attach a mandatory filter module to a driver stack, NDIS unbinds all the protocol bindings, attaches the filter module, and then reestablishes all the protocol bindings. If the driver does not attach, NDIS tears down the underlying driver stack.

To detach a mandatory filter module from a driver stack, NDIS unbinds all the protocol bindings detaches the filter module, and then reestablishes the protocol bindings. To detach an optional filter module, NDIS pauses the stack and restarts it without unbinding the protocol drivers.

When a computer restarts, NDIS will not bind any protocol drivers to a miniport adapter if any mandatory filter module that is associated with the adapter does not attach to the miniport adapter.

To install a mandatory filter driver, you must specify a value of 0x00000001 for **FilterRunType** in the INF file. To install an optional filter driver, you must specify a value of 0x00000002 for **FilterRunType** in the INF file.

Initializing a Filter Driver

Article • 12/15/2021

Filter driver initialization occurs immediately after the system loads the driver. Filter drivers load as system services. The system can load the filter drivers at any time before, during, or after the miniport drivers load. NDIS can attach a filter module to a miniport adapter after a miniport adapter of the type supported by the filter driver becomes available and the filter driver initialization is complete.

While a driver stack is starting, the system loads the filter drivers if they are not already loaded. For more information about starting a driver stack that includes filter modules, see Starting a Driver Stack.

After a filter driver loads, the system calls the driver's DriverEntry routine.

The system passes two arguments to DriverEntry:

- A pointer to the driver object, which was created by the I/O system.
- A pointer to the registry path, which specifies where driver-specific parameters are stored.

DriverEntry returns STATUS_SUCCESS, or its equivalent NDIS_STATUS_SUCCESS, if the driver successfully registered as an NDIS filter driver. If **DriverEntry** fails initialization by propagating an error status returned by an **NdisXxx** function or by a kernel-mode support routine, the driver will not remain loaded. **DriverEntry** must execute synchronously; that is, it cannot return STATUS_PENDING or its equivalent NDIS_STATUS_PENDING.

The filter driver passes the driver object to the NdisFRegisterFilterDriver function when it registers with NDIS as a filter driver. The driver can use the registry path to obtain configuration information. For more information about how to access filter driver configuration information, see Accessing Configuration Information for a Filter Driver.

A filter driver calls **NdisFRegisterFilterDriver** from its **DriverEntry** routine. Filter drivers export a set of *FilterXxx* functions by passing an **NDIS_FILTER_DRIVER_CHARACTERISTICS** structure to **NdisFRegisterFilterDriver** at the *FilterCharacteristics* parameter.

The NDIS_FILTER_DRIVER_CHARACTERISTICS structure specifies entry points for mandatory and optional *FilterXxx* functions. Some optional functions can be bypassed. For more information about bypassing functions, see Data Bypass Mode.

Drivers that call NdisFRegisterFilterDriver must be prepared for an immediate call to any of their *FilterXxx* functions.

The NDIS_FILTER_DRIVER_CHARACTERISTICS structure specifies the entry points for these mandatory *FilterXxx* functions:

FilterAttach

FilterDetach

FilterRestart

FilterPause

The NDIS_FILTER_DRIVER_CHARACTERISTICS structure specifies the entry points for these optional, and not changeable at run-time, *FilterXxx* functions:

FilterSetOptions

FilterSetModuleOptions

FilterOidRequest

FilterOidRequestComplete

FilterStatus

FilterNetPnPEvent

FilterDevicePnPEventNotify

FilterCancelSendNetBufferLists

The NDIS_FILTER_DRIVER_CHARACTERISTICS structure specifies the default entry points for these optional, and changeable at run-time, *FilterXxx* functions:

FilterSendNetBufferLists

FilterSendNetBufferListsComplete

FilterReturnNetBufferLists

FilterReceiveNetBufferLists

The preceding four functions are also defined in the **NDIS_FILTER_PARTIAL_CHARACTERISTICS** structure. This structure specifies the functions that can be changed at a run time by calling the **NdisSetOptionalHandlers** function from the *FilterSetModuleOptions* function. If a filter driver will change these partial characteristics at runtime, it must provide the entry point for *FilterSetModuleOptions*. The partial characteristics can be different for each filter module. For more information, see Starting a Filter Module.

NDIS calls the *FilterSetOptions* function within the context of the call to **NdisFRegisterFilterDriver**. *FilterSetOptions* registers optional services with NDIS. For more information, see Configuring Optional Filter Driver Services.

If the call to **NdisFRegisterFilterDriver** succeeds, NDIS fills the variable at *NdisFilterDriverHandle* with a filter driver handle. The filter driver saves this handle and later passes this handle to NDIS functions, such as **NdisFDeregisterFilterDriver**, that require a filter driver handle as an input parameter. When the driver unloads, it must call the **NdisFDeregisterFilterDriver** function to release the driver resources allocated by **NdisFRegisterFilterDriver**.

After *FilterSetOptions* returns, the filter modules are in the *Detached* state. NDIS can call the filter driver's *FilterAttach* function at any time after the call to *FilterSetOptions* returns. The driver performs filter module-specific initialization in the *FilterAttach* function. For more information about attaching a filter module to a driver stack, see Attaching a Filter Module.

A filter driver also performs any other driver-specific initialization that it requires in **DriverEntry**. The filter driver must release the driver-specific resources that it allocates in its *FilterDriverUnload* routine. For more information, see Unloading a Filter Driver.

Unloading a Filter Driver

Article • 12/15/2021

The driver object that is associated with an NDIS filter driver specifies an *Unload* routine called *FilterDriverUnload*. The system can call the *FilterDriverUnload* routine when all the miniport adapters that the filter driver services have been removed.

Unload should release any driver-specific resources. Any device objects that the filter driver created must be destroyed. The system can complete a driver unload operation after *FilterDriverUnload* returns.

The functionality of the unload function is driver-specific. As a general rule, *Unload* should undo the operations that were performed during driver initialization. For more information about driver initialization, see Initializing a Filter Driver.

A filter driver must call the **NdisFDeregisterFilterDriver** function from *Unload*. **NdisFDeregisterFilterDriver** calls *FilterDetach* to detach all currently attached filter modules that are associated with this filter driver.

For more information about unloading filter drivers, see Stopping a Driver Stack.

Filter Module States and Operations

Article • 12/15/2021

A filter driver must support the following operational states for each filter module (instance of a filter driver) that the driver manages:

Detached
The *Detached* state is the initial state of a filter module. When a filter module is in this state, NDIS can call the filter driver's *FilterAttach* function to attach the filter module to the driver stack.

Attaching
In the *Attaching* state, a filter driver prepares to attach the filter module to the driver stack.

Paused
In the *Paused* state, the filter driver does not perform send or receive operations.

Restarting
In the *Restarting* state, a filter driver completes any operations that are required to restart send and receive operations for a filter module.

Running
In the *Running* state, a filter driver performs normal send and receive processing for a filter module.

Pausing
In the *Pausing* state, a filter driver completes any operations that are required to stop send and receive operations for a filter module.

In the following table, the headings are the filter module states. Major events are listed in the first column. The rest of the entries in the table specify the next state that the filter module enters after an event occurs within a state. The blank entries represent invalid event/state combinations.

Event/State	Detached	Attaching	Paused	Restarting	Running	Pausing
Filter attach	Attaching					
Attach is complete		Paused				
Filter detach			Detached			

Event/State	Detached	Attaching	Paused	Restarting	Running	Pausing
Filter restart			Restarting			
Restart is complete				Running		
Filter pause					Pausing	
Pause is complete						Paused
Attach failed		Detached				
Restart failed				Paused		
Send and Receive					Running	Pausing
OID Requests			Paused	Restarting	Running	Pausing

The primary filter driver events are defined as follows:

Filter attach
NDIS called the driver's *FilterAttach* function to attach a filter module to a driver stack. For more information about attaching a filter module, see Attaching a Filter Module.

Attach is complete
When a filter module is in the *Attaching* state and the filter driver completes initialization of all the resources that the filter module requires, the filter module enters the *Paused* state.

Filter detach
NDIS called the driver's *FilterDetach* function to detach a filter module from a driver stack. For more information, see Detaching a Filter Module.

Filter restart
NDIS called the driver's *FilterRestart* function to restart a paused filter module. For more information, see Starting a Filter Module.

Restart is complete
When the filter module is in the *Restarting* state and the driver is ready to perform send and receive operations, the filter module enters the *Running* state.

Filter pause
NDIS called the driver's *FilterPause* function to pause a filter module. For more

information, see Pausing a Filter Module.

Pause is complete
After the driver has completed all operations that are required to stop send and receive operations, the pause operation is complete and the filter module is in the *Paused* state.

Attach failed
If NDIS calls a driver's *FilterAttach* function and the attach operation fails (for example, because the required resources are not available), the filter module returns to the *Detached* state.

Restart failed
If NDIS calls a driver's *FilterRestart* function and the restart attempt fails, the filter module returns to the *Paused* state.

Send and Receive Operations
A driver can handle send and receive operations in the *Running* and *Pausing* states. For more information about send and receive operations, see Filter Module Send and Receive Operations.

OID Requests
A driver can handle OID requests in the *Running*, *Restarting*, *Paused*, and *Pausing* states. For more information about OID requests, see Filter Module OID Requests.

Attaching a Filter Module

Article • 12/15/2021

To initiate the process of inserting a filter module into a driver stack, NDIS calls a filter driver's *FilterAttach* function. At the start of execution in the *FilterAttach* function, the filter module enters the *Attaching* state. For more information about attaching a filter module to a driver stack, see Starting a Driver Stack.

A filter driver uses the handle, that NDIS passes at the *NdisFilterHandle* parameter of *FilterAttach* in all future **NdisXxx** function calls that refer to this filter module. Such functions include status indications, send requests, receive indications, and OID requests.

While a filter module is in the *Attaching* state, the driver:

- Creates a context area for the filter module and allocates buffer pools and other filter module-specific resources. For more information about buffer pools, see Filter Driver Buffer Management.
- Calls the **NdisFSetAttributes** function by using the *NdisFilterHandle* value that NDIS passed to *FilterAttach*. The *FilterModuleContext* parameter of **NdisFSetAttributes** specifies the filter driver's context area for this filter module. NDIS passes this context area to the filter driver's *FilterXxx* functions.
- Optionally, reads configuration parameters for this filter module from the registry. For more information, see Accessing Configuration Information for a Filter Driver.
- If the preceding operations completed successfully, the filter module is in the *Paused* state.
- If the preceding operations failed, the filter driver must release any resources that it allocated in the *FilterAttach* function and return the filter module to the *Detached* state.
- Returns NDIS_STATUS_SUCCESS or an appropriate failure code. If the driver returns a failure code, NDIS terminates the driver stack.

Note The registry can contain a flag, which specifies that a filter module is optional. If an optional filter module does not attach, NDIS does not terminate the rest of the driver stack.

A filter driver cannot make send requests, indicate received data, make OID requests, or make status indications from the *Attaching* state. Send and receive operations are supported in the *Running* and *Pausing* states. OID requests and status indications are supported in the *Paused*, *Restarting*, *Running*, and *Pausing* states.

NDIS calls the *FilterDetach* function to detach a filter module that NDIS attached with *FilterAttach*. For more information, see Detaching a Filter Module.

Detaching a Filter Module

Article • 12/15/2021

To initiate the process of detaching a filter module from a driver stack, NDIS calls a filter driver's *FilterDetach* function. At the start of execution in the *FilterDetach* function, the filter module enters the *Detached* state. Before detaching a filter module, NDIS must pause the driver stack. For more information about pausing the driver stack, see Pausing a Driver Stack.

In its *FilterDetach* function, the driver frees its context areas and other resources (such as buffer pools) for the affected filter module. A filter driver cannot fail the call to *FilterDetach*. Therefore, filter drivers should preallocate, during the attach operation, all the resources required to perform the detach operation successfully. For more information about attaching a filter module, see Attaching a Filter Module.

After the filter module returns from *FilterDetach*, NDIS can start the paused driver stack. For more information about starting a driver stack, see Starting a Driver Stack.

Starting and Pausing a Filter Module

Article • 12/15/2021

NDIS pauses a filter module to stop data flow that could interfere with Plug and Play operations, for example, adding or removing a filter module in a driver stack, or to add a new binding. For more information about how to modify a running driver stack, see Modifying a Running Driver Stack.

NDIS starts a filter module from the *Paused* state. The filter module enters the *Paused* state after the attach operation is complete or after a pause operation is complete.

The following topics provide more information about starting and pausing a filter module:

Starting a Filter Module

Pausing a Filter Module

Starting a Filter Module

Article • 12/15/2021

To start a paused filter module, NDIS calls the filter driver's *FilterSetModuleOptions* function, if any, followed by a call to the *FilterRestart* function. The filter module enters the *Restarting* state at the start of execution in the *FilterRestart* function.

If the driver provided an entry point for *FilterSetModuleOptions*, the driver can change the partial characteristic for a filter module. For more information, see Data Bypass Mode.

When it calls a filter driver's *FilterRestart* function, NDIS passes a pointer to an **NDIS_RESTART_ATTRIBUTES** structure to filter driver in the **RestartAttributes** member of the **NDIS_FILTER_RESTART_PARAMETERS** structure. Filter drivers can modify the restart attributes that are specified by underlying drivers. For more information about how to modify restart attributes, see *FilterRestart*.

Note NDIS calls *FilterSetModuleOptions* for all filter modules in a stack before NDIS calls the *FilterRestart* function for any filter module in the stack.

NDIS starts a filter module as part of a Plug and Play operation to restart a driver stack. For an overview of restarting the driver stack, see Restarting a Driver Stack.

On behalf of a filter module that is in the *Restarting* state, the filter driver:

- Completes any operations that are required to restart normal send and receive operations.

  For more information about send and receive operations, see Filter Module Send and Receive Operations.

- Can read or write configurable parameters for the filter module.

- Can receive network data indications. The driver can copy and queue such data and indicate it to overlying drivers later, or it can discard the data.

- Should not initiate any new receive indications.

- Should reject all new send requests made to its *FilterSendNetBufferLists* function immediately by calling the **NdisFSendNetBufferListsComplete** function. It should set the complete status in each NET_BUFFER_LIST to NDIS_STATUS_PAUSED.

- Can provide status indications with the **NdisFIndicateStatus** function.

For more information about status indications, see Filter Module Status Indications.

- Should handle OID requests in the *FilterOidRequest* function.

  For more information about OID requests, see Filter Module OID Requests.

- Should not initiate any new send requests.

- Should return new receive indications to NDIS immediately by calling the **NdisFReturnNetBufferLists** function. If necessary, the driver can copy such receive indications before it returns them.

- Can make OID requests to the underlying drivers to set or query updated configuration information.

- Should handle status indications in its *FilterStatus* function.

- Should Indicate NDIS_STATUS_SUCCESS or a failure status. If a filter module does not restart, NDIS will detach it and if it is a mandatory filter, NDIS terminates the entire driver stack.

After the filter driver successfully restarts the send and receive operations, it must complete the restart operation. The filter driver can complete the restart operation synchronously or asynchronously by returning NDIS_STATUS_SUCCESS or NDIS_STATUS_PENDING respectively from *FilterRestart*.

If the driver returns NDIS_STATUS_PENDING, it must call the **NdisFRestartComplete** function after it completes the restart operation. In this case, the driver passes the final status of the restart operation to **NdisFRestartComplete**.

After the restart operation is complete, the filter module is in the *Running* state. The driver resumes normal send and receive processing.

NDIS does not initiate other Plug and Play operations, such as, attach, detach, or pause requests, while the filter driver is in the *Restarting* state. NDIS can initiate pause requests after a filter driver is in the *Running* state. For more information about pausing a filter module, see Pausing a Filter Module.

Pausing a Filter Module

Article • 12/15/2021

To pause a running filter module, NDIS calls the filter driver's *FilterPause* function. The filter module enters the *Pausing* state at the start of execution in the *FilterPause* function.

NDIS pauses a filter module as part of a Plug and Play operation to pause a driver stack. For an overview of pausing the driver stack, see Pausing a Driver Stack.

On behalf of a filter module that is in the *Pausing* state, the filter driver:

- Should not originate any new receive indications.

  For more information about send and receive operations, see Filter Module Send and Receive Operations.

- If there are receive operations that the filter driver originated and that NDIS has not completed, the filter driver must wait for NDIS to complete such operations. The pause operation is not complete until NDIS calls the *FilterReturnNetBufferLists* function for all such outstanding receive indications.

- Should return any outstanding receive indications that underlying drivers originated to NDIS immediately. The pause operation is not complete until the driver calls the **NdisFReturnNetBufferLists** function for such outstanding receive indications. These outstanding receive indications can exist if the driver queues the buffers that it receives from underlying drivers.

- Should return new receive indications that underlying drivers originate to NDIS immediately by calling the **NdisFReturnNetBufferLists** function. If necessary, the driver can copy receive indications and queue them before it returns them.

  Note **NdisFReturnNetBufferLists** should not be called for NBLs indicated with NDIS_RECEIVE_FLAGS_RESOURCES flag set in a corresponding *FilterReceiveNetBufferLists* call. Such NBLs are returned to NDIS synchronously by returning from the *FilterReceiveNetBufferLists* routine.

- Should not originate any new send requests.

- If there are send operations that the filter driver originated and that NDIS has not completed, the filter driver must wait for NDIS to complete such operations. The pause operation is not complete until NDIS calls the *FilterSendNetBufferListsComplete* function for all such outstanding send requests.

- Should return all new send requests made to its *FilterSendNetBufferLists* function immediately by calling the **NdisFSendNetBufferListsComplete** function. The filter driver should set the **Status** member in each NET_BUFFER_LIST structure to NDIS_STATUS_PAUSED.

- Can provide status indications with the **NdisFIndicateStatus** function.

  For more information about status indications, see Filter Module Status Indications.

- Should handle status indications in its *FilterStatus* function.

- Should handle OID requests in the *FilterOidRequest* function.

  For more information about OID requests, see Filter Module OID Requests.

- Can initiate OID requests.

- Should not free the resources the driver allocated during the attach operation.

- Should cancel timers, if required to stop send and receive operations.

  For more information about timers, see NDIS 6.0 Timer Services.

After the filter driver successfully pauses the send and receive operations, it must complete the pause operation. The filter driver can complete the pause operation synchronously or asynchronously by returning NDIS_STATUS_SUCCESS or NDIS_STATUS_PENDING respectively from *FilterPause*.

If the driver returns NDIS_STATUS_PENDING, it must call the **NdisFPauseComplete** function after it completes the pause operation.

On behalf of a filter module that is in the *Paused* state, the filter driver:

- Should not originate new receive indications.

- Should return new receive indications that underlying drivers originate to NDIS immediately by calling the **NdisFReturnNetBufferLists** function. If necessary, the driver can copy receive indications and queue them before it returns them.

- Should not originate new send requests.

- Should return all new send requests made to its *FilterSendNetBufferLists* function immediately by calling the **NdisFSendNetBufferListsComplete** function. The filter driver should set the **Status** member in each NET_BUFFER_LIST structure to NDIS_STATUS_PAUSED.

- Can provide status indications with the **NdisFIndicateStatus** function.

- Should handle status indications in its *FilterStatus* function.

- Should handle OID requests in the *FilterOidRequest* function.

- Can initiate OID requests.

NDIS does not initiate other Plug and Play operations, such as, attach, detach, or a restart requests, while the filter driver is in the *Pausing* state. NDIS can initiate detach or restart requests after a filter driver is in the *Paused* state. For more information about how to detach a filter module, see Detaching a Filter Module. For more information about how to restart a filter module, see Starting a Filter Module.

Data Bypass Mode

Article • 12/15/2021

The filter driver *data bypass mode* can provide improved system performance. NDIS does not call *FilterXxx* functions that are bypassed. For example, if the send and receive services are not required for a given filter application, the filter driver can bypass the send and receive functions.

A filter driver specifies the default entry points, for functions that can be bypassed, during driver initialization when it calls the **NdisFRegisterFilterDriver** function. The entry points are **NULL** for functions that are bypassed by default. For more information about initialization, see Initializing a Filter Driver.

To change the bypass state at runtime, the driver must specify an entry point for the *FilterSetModuleOptions* function during driver initialization. The driver can initialize an **NDIS_FILTER_PARTIAL_CHARACTERISTICS** structure and pass the new characteristics to the **NdisSetOptionalHandlers** function from within the context of *FilterSetModuleOptions*.

NDIS calls the *FilterSetModuleOptions* function, if any, at the start of a restart operation. A filter driver can set bypass mode independently for each filter module. For more information, see Starting a Filter Module.

Filter drivers can bypass the following optional *FilterXxx* functions that are specified in the **NDIS_FILTER_DRIVER_CHARACTERISTICS** structure:

FilterSendNetBufferLists

FilterSendNetBufferListsComplete

FilterCancelSendNetBufferLists

FilterReturnNetBufferLists

FilterReceiveNetBufferLists

To set a *FilterXxx* function to bypass mode, a filter driver specifies **NULL** for that function's entry point. However, if a driver calls any NDIS function that has an associated *FilterXxx* function, it must provide an entry point for that *FilterXxx* function. For example, if a driver calls the **NdisFIndicateReceiveNetBufferLists** function, it must provide a *FilterReturnNetBufferLists* function.

If a filter driver specifies a *FilterSendNetBufferLists* function and it queues send requests, it must also specify a *FilterCancelSendNetBufferLists* function.

If a filter driver specifies a *FilterReceiveNetBufferLists* or **FilterReturnNetBufferLists** function, the driver must also specify a *FilterStatus* function.

To change its bypass mode settings at run time, a filter driver can call the **NdisFRestartFilter** function. **NdisFRestartFilter** schedules a pause operation that is followed by a restart operation for the specified filter module. When NDIS calls *FilterSetModuleOptions*, the filter driver can change the functions for that filter module by calling **NdisSetOptionalHandlers** and specifying a new set of entry points.

Note Pause and restart could cause some network packets to be dropped on the transmit path, or receive path, or both. Network protocols that provide a reliable transport mechanism might retry the network I/O operation in the case of a lost packet, but other protocols that do not guarantee reliability do not retry the operation.

A filter driver can register additional optional functions that support optional driver services. The driver registers these optional services in the *FilterSetOptions* function. For more information about these optional services, see Configuring Optional Filter Driver Services.

Configuring Optional Filter Driver Services

Article • 12/15/2021

NDIS calls a filter driver's *FilterSetOptions* function to configure optional filter driver services. NDIS calls *FilterSetOptions* within the context of the filter driver's call to the **NdisFRegisterFilterDriver** function

FilterSetOptions registers the default entry points for optional *FilterXxx* functions that are required for optional services, and can allocate other driver resources. To register optional services, the filter driver calls the **NdisSetOptionalHandlers** function and passes a characteristics structure at the *OptionalHandlers* parameter.

There are no optional filter driver services in the current Windows version.

Filter drivers can also call **NdisSetOptionalHandlers** to set the some *FilterXxx* function entry points for a given filter module. For more information, see Data Bypass Mode.

If the filter driver calls **NdisSetOptionalHandlers** from *FilterRestart*, the configuration changes only affect the filter module that NDIS is restarting. The configuration of other filter modules is not affected.

Filter Module Send and Receive Operations

Article • 12/15/2021

This section documents send and receive operations for NDIS 6.0 filter drivers. Filter drivers can initiate send requests and receive indications or filter the requests and indications of other drivers.

Filter modules are stacked over a miniport adapter. For more information about the driver stack, see NDIS 6.0 Driver Stack.

The filter modules in the driver stack can filter all send requests and receive indications that are associated with the underlying adapter. This is true for all protocol bindings to an adapter. For more information about NDIS 6.0 send and receive operations, see Send and Receive Operations.

Filter drivers do not provide direct support for legacy send and receive operations that are based on the **NDIS_PACKET** structure. Instead, NDIS converts receive indications from legacy miniport drivers to **NET_BUFFER** structures. Also, NDIS handles the required conversions from send requests that are based on NET_BUFFER structures to legacy send requests that are based on NDIS_PACKET structures.

Note A filter driver can change the send and receive *FilterXxx* functions for a filter module dynamically. For more information, see Data Bypass Mode.

The following topics provide additional information about filter driver send and receive operations:

Filter Driver Buffer Management

Sending Data from a Filter Driver

Canceling a Send Request in a Filter Driver

Receiving Data in a Filter Driver

Filter Driver Buffer Management

Article • 12/15/2021

Filter drivers create buffers to copy network data obtained from other drivers, or to initiate send or receive operations.

If a filter driver does not create buffers, the driver does not manage buffer pools. Such a driver simply passes on the buffers that it receives from other drivers.

A filter driver that creates buffers to support send or receive operations must manage NET_BUFFER_LIST structure pools and NET_BUFFER structure pools.

To create these pools, drivers call the following functions:

NdisAllocateNetBufferListPool

NdisAllocateNetBufferPool

Filter drivers can use the following functions to allocate structures from the pools:

NdisAllocateNetBufferAndNetBufferList

NdisAllocateNetBufferList

NdisAllocateNetBuffer

Calling **NdisAllocateNetBufferAndNetBufferList** is more efficient than calling **NdisAllocateNetBufferList** followed by **NdisAllocateNetBuffer**. However, **NdisAllocateNetBufferAndNetBufferList** only creates one NET_BUFFER structure on the NET_BUFFER_LIST structure. To use **NdisAllocateNetBufferAndNetBufferList**, the driver must set the *AllocateNetBuffer* parameter to **TRUE** when it calls **NdisAllocateNetBufferListPool**.

Filter drivers that originate send requests should determine the context and backfill space requirements of the underlying drivers. Filter drivers use restart attributes to determine the backfill requirements of underlying drivers. A filter driver should determine the backfill and context requirements in the *Restarting* state. The driver should allocate sufficient backfill and context space for the entire stack. If necessary, a filter driver can free the pools and reallocate them in the *Restarting* state.

Filter drivers use the following functions to free the pools:

NdisFreeNetBufferListPool

NdisFreeNetBufferPool

Filter drivers use the following functions to free the structures allocated from the pools:

NdisFreeNetBufferList

NdisFreeNetBuffer

Drivers should free NET_BUFFER structures allocated with **NdisAllocateNetBuffer** before freeing the associated NET_BUFFER_LIST structure. NET_BUFFER structures allocated with **NdisAllocateNetBufferAndNetBufferList** are freed when the driver calls **NdisFreeNetBufferList** for the associated NET_BUFFER_LIST structure.

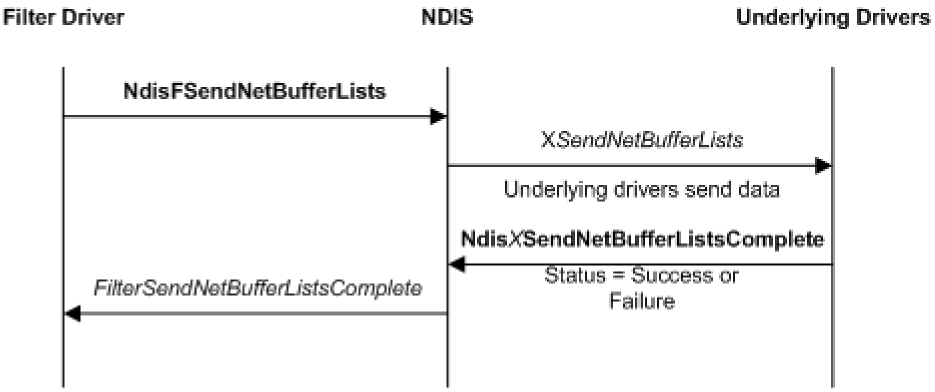
Sending Data from a Filter Driver

Article • 12/15/2021

Filter drivers can initiate send requests or filter send requests that overlying drivers initiate. When a protocol driver calls the NdisSendNetBufferLists function, NDIS submits the specified NET_BUFFER_LIST structure to the topmost filter module in the driver stack.

Send Requests Initiated by a Filter Driver

The following figure illustrates a send operation that is initiated by a filter driver.


Filter Driver
NDIS
Underlying Drivers
NdisFSendNetBufferLists
XSendNetBufferLists
Underlying drivers send data
NdisXSendNetBufferListsComplete
Status = Success or Failure
FilterSendNetBufferListsComplete

Filter drivers call the NdisFSendNetBufferLists function to send the network data that is defined in a list of NET_BUFFER_LIST structures.

A filter driver must set the **SourceHandle** member of each NET_BUFFER_LIST structure that it creates to the same value that it passes to the *NdisFilterHandle* parameter of **NdisFSendNetBufferLists**. NDIS drivers should not modify the **SourceHandle** member for NET_BUFFER_LIST structures that the driver did not originate.

Before calling **NdisFSendNetBufferLists**, a filter driver can set information that accompanies the send request with the NET_BUFFER_LIST_INFO macro. The underlying drivers can retrieve this information with the NET_BUFFER_LIST_INFO macro.

As soon as a filter driver calls **NdisFSendNetBufferLists**, it relinquishes ownership of the NET_BUFFER_LIST structures and all associated resources. NDIS can handle the send request or pass the request to underlying drivers.

NDIS calls the *FilterSendNetBufferListsComplete* function to return the structures and data to the filter driver. NDIS can collect the structures and data from multiple send requests into a single linked list of NET_BUFFER_LIST structures before it passes the list to *FilterSendNetBufferListsComplete*

Until NDIS calls *FilterSendNetBufferListsComplete*, the current status of a send request is unknown. A filter driver should *never* try to examine the NET_BUFFER_LIST structures or any associated data before NDIS returns the structures to *FilterSendNetBufferListsComplete*.

FilterSendNetBufferListsComplete performs whatever postprocessing is necessary to complete a send operation.

When NDIS calls *FilterSendNetBufferListsComplete*, the filter driver regains ownership of all the resources associated with the NET_BUFFER_LIST structures that are specified by the *NetBufferLists* parameter. *FilterSendNetBufferListsComplete* can either free these resources (for example, by calling the NdisFreeNetBuffer and NdisFreeNetBufferList functions) or prepare them for reuse in a subsequent call to **NdisFSendNetBufferLists**.

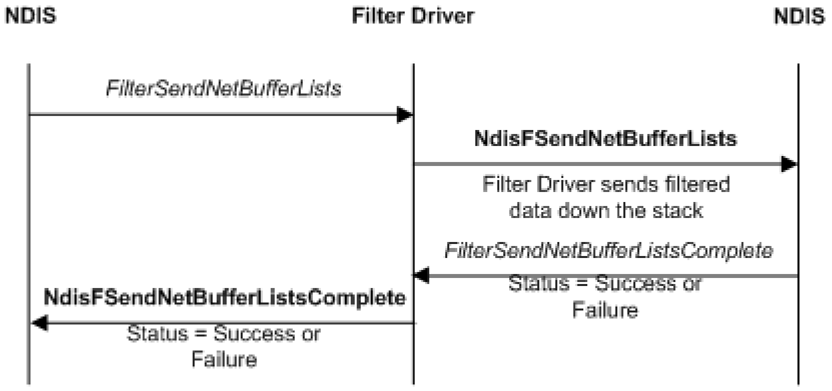
NDIS always submits filter-supplied network data to the underlying drivers in the filter-driver-determined order as passed to **NdisFSendNetBufferLists**. However, after sending the data in the specified order, the underlying drivers can return the buffers in any order.

A filter driver can request loopback for send requests that it originates. To request loopback, the driver sets the NDIS_SEND_FLAGS_CHECK_FOR_LOOPBACK flag in the *SendFlags* parameter of NdisFSendNetBufferLists. NDIS indicates a received packet that contains the send data.

Note A filter driver should keep track of send requests that it originates and make sure that it does not call the NdisFSendNetBufferListsComplete function when such requests are complete.

Filtering Send Requests

The following figure illustrates filtering a send request that is initiated by an overlying driver.


NDIS
Filter Driver
NDIS
FilterSendNetBufferLists
NdisFSendNetBufferLists
Filter Driver sends filtered data down the stack
FilterSendNetBufferListsComplete
Status = Success or Failure
NdisFSendNetBufferListsComplete
Status = Success or Failure

NDIS calls a filter driver's *FilterSendNetBufferLists* function to filter the send request of an overlying driver.

The filter driver must not modify the **SourceHandle** member in the NET_BUFFER_LIST structures that it receives from other drivers.

The filter driver can filter the data and send the filtered data to underlying drivers. For each NET_BUFFER structure submitted to *FilterSendNetBufferLists*, a filter driver can do the following:

- Pass the buffer on to the next underlying driver by calling the **NdisFSendNetBufferLists** function. NDIS guarantees the availability of context space (see NET_BUFFER_LIST_CONTEXT structure) for filter drivers. The filter driver can modify the buffer contents before calling **NdisFSendNetBufferLists**. The processing of the filtered data proceeds as with a send operation initiated by a filter driver.

- Drop the buffer by calling the **NdisFSendNetBufferListsComplete** function.

- Queue the buffer in a local data structure for later processing. The design of the filter driver dictates what causes the driver to process a queued buffer. Some examples include processing after a time-out or processing after a specific buffer is received.

  Note If the driver queues send requests for later processing, it must support send cancellation requests. For more information about send cancellation requests, see Canceling Send Requests in a Filter Driver.

- Copy the buffer and originate a send request with the copy. The send operation is similar to a filter driver initiated send request. In this case, the driver must return the original buffer to the overlying driver by calling the **NdisFSendNetBufferListsComplete** function.

Completion of send requests proceeds up the driver stack. When the miniport driver calls the **NdisMSendNetBufferListsComplete** function, NDIS calls the *FilterSendNetBufferListsComplete* function for the lowest overlying filter module.

After the send operation is complete, the filter driver reverses the modifications to the overlying driver's buffer descriptors that the filter driver made in *FilterSendNetBufferLists*. The driver calls the **NdisFSendNetBufferListsComplete** function to return the linked list of NET_BUFFER_LIST structures to the overlying drivers and to return the final status of the send request.

When the topmost filter module calls **NdisFSendNetBufferListsComplete**, NDIS calls the originating protocol driver's **ProtocolSendNetBufferListsComplete** function.

A filter driver that does not provide a *FilterSendNetBufferLists* function can still initiate a send request. If such a driver does initiate a send request, it must provide a *FilterSendNetBufferListsComplete* function and it must not pass the complete event up the driver stack.

A filter driver can pass on or filter the loopback request of an overlying driver. To pass on a loopback request, if NDIS set NDIS_SEND_FLAGS_CHECK_FOR_LOOPBACK in the *SendFlags* parameter of *FilterSendNetBufferLists*, the filter driver sets NDIS_SEND_FLAGS_CHECK_FOR_LOOPBACK in the *SendFlags* parameter when it calls **NdisFSendNetBufferLists**. NDIS indicates a received packet that contains the send data.

In general, if a filter driver modifies any behavior in such a way that NDIS cannot provide a standard service (such as loopback), the filter driver must provide that service for NDIS. For example, a filter driver that modifies a request for the hardware address (see OID_802_3_CURRENT_ADDRESS), should handle loopback of buffers directed to the new hardware address. In this case, NDIS cannot provide the loopback service it typically provides because the filter altered the address. Also, if the filter driver sets promiscuous mode (see OID_GEN_CURRENT_PACKET_FILTER), it should not pass on the extra data that it receives to overlying drivers.

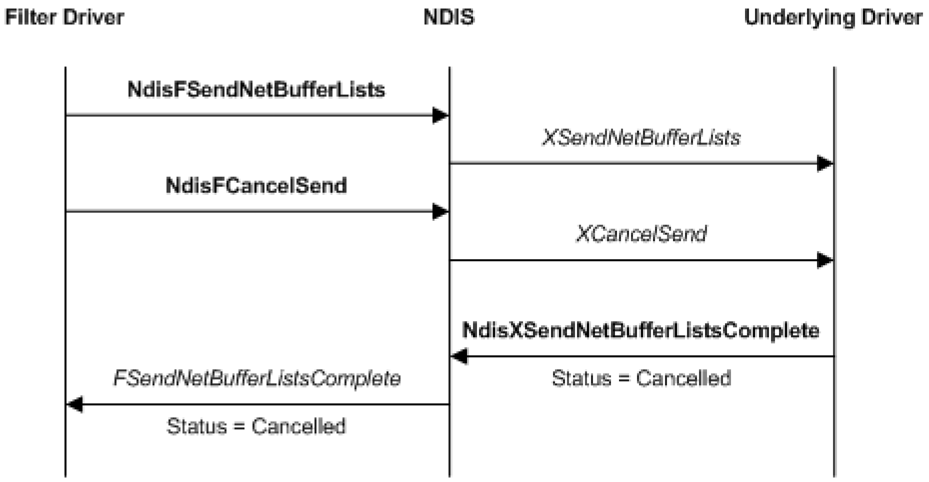
Canceling a Send Request in a Filter Driver

Article • 12/15/2021

Filter drivers can cancel send requests that were originated by the filter driver or that were originated by overlying drivers.

Canceling Filter Driver Send Requests

The following figure illustrates canceling a send request that was originated by a filter driver.



A filter driver calls the NDIS_SET_NET_BUFFER_LIST_CANCEL_ID macro for each NET_BUFFER_LIST structure that it creates for send operations. The NDIS_SET_NET_BUFFER_LIST_CANCEL_ID function marks the specified data with a cancellation identifier.

Before assigning cancellation IDs to network data, a filter driver must call NdisGeneratePartialCancelId to obtain the high-order byte of each cancellation ID that it assigns. This ensures that the driver does not duplicate cancellation IDs assigned by other drivers in the system. Drivers typically call **NdisGeneratePartialCancelId** one time from the DriverEntry routine. However, drivers can obtain more than one partial cancellation identifier by calling **NdisGeneratePartialCancelId** multiple times.

To cancel the pending transmission of data in a marked NET_BUFFER_LIST structure, a filter driver passes the cancellation ID to the NdisFCancelSendNetBufferLists function. Drivers can obtain a NET_BUFFER_LIST structure's cancellation ID by calling the NDIS_GET_NET_BUFFER_LIST_CANCEL_ID macro.

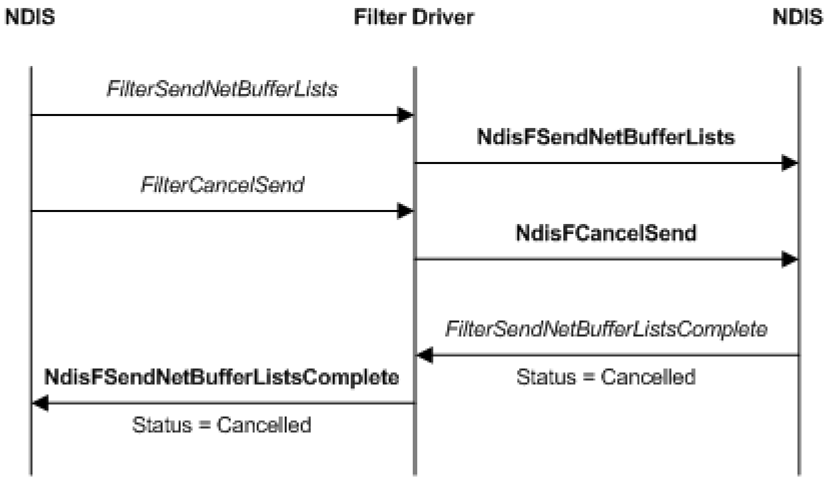
If a filter driver marks all NET_BUFFER_LIST structures with the same cancellation identifier, it can cancel all pending transmissions with a single call to **NdisFCancelSendNetBufferLists**. If a filter driver marks all NET_BUFFER_LIST structures within a subgroup of NET_BUFFER_LIST structures with a unique identifier, it can cancel all pending transmissions within that subgroup with a single call to **NdisFCancelSendNetBufferLists**.

NDIS calls the cancel send function of the underlying drivers. After aborting the pending transmission, the underlying drivers call a send complete function (for example **NdisMSendNetBufferListsComplete**) to return the NET_BUFFER_LIST structures with a completion status of NDIS_STATUS_SEND_ABORTED. NDIS, in turn, calls the filter driver's *FilterSendNetBufferListsComplete* function.

In *FilterSendNetBufferListsComplete*, a filter driver can call NDIS_SET_NET_BUFFER_LIST_CANCEL_ID with *CancelId* set to **NULL**. This prevents the NET_BUFFER_LIST from accidentally being used again with a stale cancellation ID.

Canceling Send Requests Originated by Overlying Drivers

The following figure illustrates canceling a send request that was originated by an overlying driver.



Overlying drivers call a cancel send function (**NdisFCancelSendNetBufferLists** or **NdisCancelSendNetBufferLists**) to cancel outstanding send requests. These overlying drivers must mark the send data with a cancellation ID before making a send request.

NDIS calls a filter driver's *FilterCancelSendNetBufferLists* function to cancel the transmission of all **NET_BUFFER_LIST** structures that are marked with a specified cancellation identifier.

FilterCancelSendNetBufferLists performs the following operations:

1. Traverses the filter driver's list of queued NET_BUFFER_LIST structures for the specified filter module and calls the **NDIS_GET_NET_BUFFER_LIST_CANCEL_ID** macro to obtain the cancellation identifier for each structure. The filter driver compares the cancellation ID that NDIS_GET_NET_BUFFER_LIST_CANCEL_ID returns with the cancellation ID that NDIS passed to *FilterCancelSendNetBufferLists*.

2. Removes from the send queue (unlinks) all NET_BUFFER_LIST structures whose cancellation identifiers match the specified cancellation identifier.

3. Calls the **NdisFSendNetBufferListsComplete** function for all unlinked NET_BUFFER_LIST structures to return the structures. The filter driver sets the status field of the NET_BUFFER_LIST structures to NDIS_STATUS_SEND_ABORTED.

4. Calls the **NdisFCancelSendNetBufferLists** function to pass the cancel send request to underlying drivers. The filter driver passes on the cancellation identifier that it received from the overlying driver. The cancel operation proceeds as with a filter-driver-originated cancel send operation.

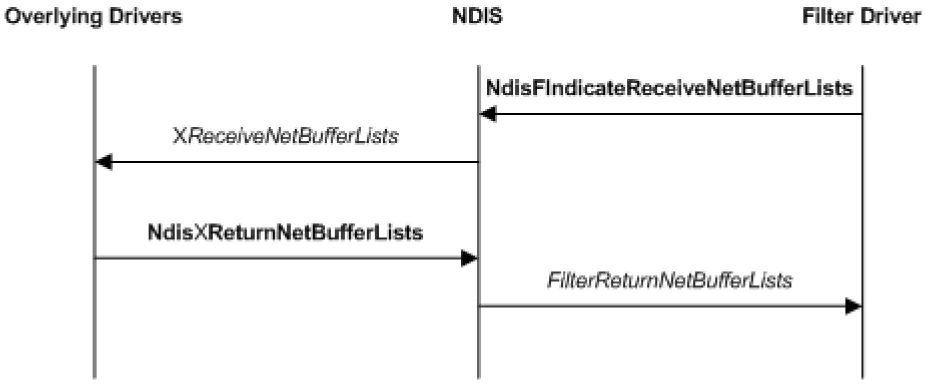
Receiving Data in a Filter Driver

Article • 12/15/2021

Filter drivers can initiate receive indications or filter receive indications from underlying drivers. When a miniport driver calls the NdisMIndicateReceiveNetBufferLists function, NDIS submits the specified NET_BUFFER_LIST structure to the lowest overlying filter module in the driver stack.

Receive Indications Initiated by a Filter Driver

The following figure illustrates a receive indication that is initiated by a filter driver.


Overlying Drivers
NDIS
Filter Driver
NdisFIndicateReceiveNetBufferLists
XReceiveNetBufferLists
NdisXReturnNetBufferLists
FilterReturnNetBufferLists

Filter drivers call the NdisFIndicateReceiveNetBufferLists function to indicate received data. The **NdisFIndicateReceiveNetBufferLists** function passes the indicated list of NET_BUFFER_LIST structures up the stack to overlying drivers. The filter driver allocates the structures from pools that it created during initialization.

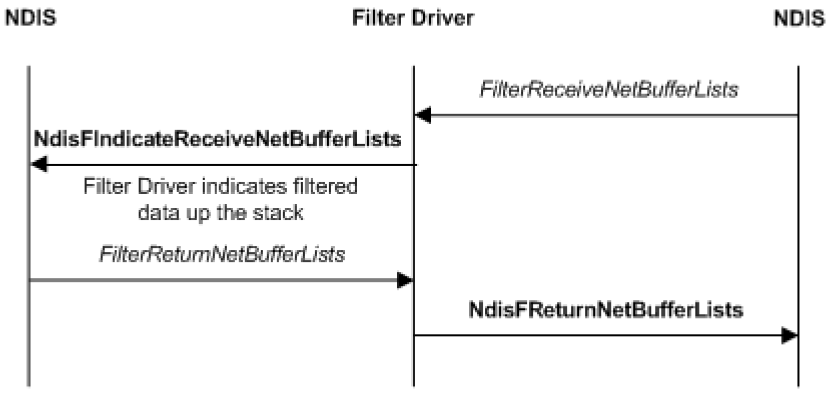
If a filter driver sets the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of NdisFIndicateReceiveNetBufferLists, this indicates that the filter driver must regain ownership of the NET_BUFFER_LIST structures immediately. In this case, NDIS does not call the filter driver's *FilterReturnNetBufferLists* function to return the **NET_BUFFER_LIST** structures. The filter driver regains ownership immediately after **NdisFIndicateReceiveNetBufferLists** returns.

If a filter driver does not set the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of NdisFIndicateReceiveNetBufferLists, NDIS returns the indicated NET_BUFFER_LIST structures to the filter driver's *FilterReturnNetBufferLists* function. In this case, the filter driver relinquishes ownership of the indicated structures until NDIS returns them to *FilterReturnNetBufferLists*.

Note A filter driver should keep track of receive indications that it initiates and make sure that it does not call the NdisFReturnNetBufferLists function when the receive operation is complete.

Filtering Receive Indications

The following figure illustrates a filtered receive indication that is initiated by an underlying driver.


NDIS
Filter Driver
NDIS
FilterReceiveNetBufferLists
NdisFIndicateReceiveNetBufferLists
Filter Driver indicates filtered data up the stack
FilterReturnNetBufferLists
NdisFReturnNetBufferLists

NDIS calls a filter driver's *FilterReceiveNetBufferLists* function to process receive indications that come from underlying drivers. NDIS calls *FilterReceiveNetBufferLists* after an underlying driver calls a receive indication function (for example, **NdisMIndicateReceiveNetBufferLists**) to indicate received network data or loopback data.

If the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists* is not set, the filter driver keeps ownership of the **NET_BUFFER_LIST** structures until it calls the **NdisFReturnNetBufferLists** function.

If the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter is set, the filter driver cannot keep the **NET_BUFFER_LIST** structure and the associated underlying driver-allocated resources. This flag can indicate that the underlying driver is running low on receive resources. The *FilterReceiveNetBufferLists* function should return as quickly as possible.

Note If the **NDIS_RECEIVE_FLAGS_RESOURCES** flag is set, the filter driver must retain the original set of **NET_BUFFER_LIST** structures in the linked list. For example, when this flag is set, the driver might process the structures and indicate them up the stack one at a time but before the function returns, it must restore the original linked list.

Filter drivers can perform filter operations on received data before indicating the data to overlying drivers. For each buffer submitted to its *FilterReceiveNetBufferLists* function a filter driver can do the following:

- Pass it on to the next overlying driver by calling **NdisFIndicateReceiveNetBufferLists**. The driver can modify the contents of the buffer. NDIS guarantees the availability of context space (see NET_BUFFER_LIST_CONTEXT structure).

  A filter driver can change the status that NDIS passed to *FilterReceiveNetBufferLists* or simply pass it on to **NdisFIndicateReceiveNetBufferLists**.

  Note A filter driver can pass on a buffer with **NdisFIndicateReceiveNetBufferLists** even if NDIS sets the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists*. In this case, the filter driver must not return from *FilterReceiveNetBufferLists* until it regains ownership of the buffer.

- Discard the buffer. If NDIS cleared the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists*, call the **NdisFReturnNetBufferLists** function to discard the buffer. If NDIS set the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists*, take no action and return from *FilterReceiveNetBufferLists* to discard the buffer.

- Queue the buffer in a local data structure for later processing. If NDIS set the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists*, the filter driver must create a copy before returning from *FilterReceiveNetBufferLists*.

- Copy the buffer and originate a receive indication with the copy. The receive indication is similar to a filter-driver-initiated receive indication. In this case, the driver must return the original buffer to the underlying driver.

The **NdisFIndicateReceiveNetBufferLists** function passes the indicated list of **NET_BUFFER_LIST** structures up the driver stack to overlying drivers. The receive operation proceeds similarly to a filter-driver-initiated receive operation.

If an overlying driver retained ownership of the buffer, NDIS calls the *FilterReturnNetBufferLists* function for the filter module. In its *FilterReturnNetBufferLists* function, the filter driver will undo the operations that it performed on the buffer on the receive indication path.

When the lowest layer filter module indicates that it is done with a buffer, NDIS returns the buffer to the miniport driver. If NDIS cleared the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists*, the filter driver calls **NdisFReturnNetBufferLists** to return the buffer. If NDIS set the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists*, returning from *FilterReceiveNetBufferLists* returns the buffer.

Filter Module OID Requests

Article • 12/15/2021

NDIS defines object identifier (OID) values to identify adapter parameters, which include operating parameters such as device characteristics, configurable settings and statistics. For more information about OIDs, see NDIS OIDs.

Filter drivers can query or set the operating parameters of underlying drivers or filter the OID requests of overlying drivers.

NDIS also provides a direct OID request interface for NDIS 6.1 and later filter drivers. The *direct OID request path* supports OID requests that are queried or set frequently. For example, the IPsec offload version 2 (IPsecv2) interface provides the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID for direct OID requests. The direct OID request interface is optional for NDIS drivers.

For NDIS 6.81 and later filter drivers, NDIS provides a Synchronous OID Request Interface. The *Synchronous OID request path* supports OIDs that require synchronization or OIDs that should not be queued by filter drivers, such as RSSv2 OIDs. The Synchronous OID Request Interface is optional for NDIS drivers but is required if the filter driver advertises support for RSSv2.

The following topics provide more information about filter driver OID requests:

Filtering OID Requests in an NDIS Filter Driver

Generating OID Requests from an NDIS Filter Driver

Filter Module Direct OID Requests

Filter Module Synchronous OID Requests

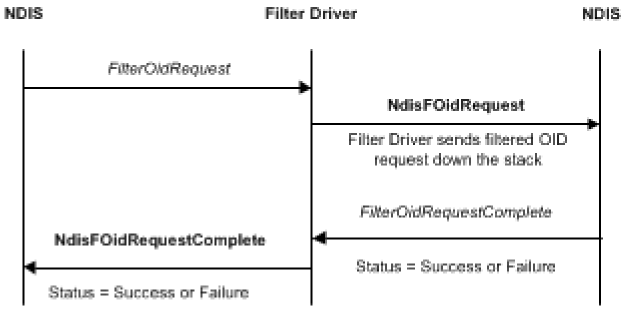
Filtering OID Requests in an NDIS Filter Driver

Article • 03/14/2023

Filter drivers can process OID requests that are originated by overlying drivers. NDIS calls the *FilterOidRequest* function to process each OID request. Filter drivers can forward OID requests to underlying drivers by calling the **NdisFOidRequest** function.

NDIS can call a filter driver's *FilterCancelOidRequest* function to cancel an OID request. When NDIS calls *FilterCancelOidRequest*, the filter driver should try to call the **NdisFOidRequest** function as soon as possible.

The following figure illustrates a filtered OID request.


NDIS
Filter Driver
NDIS
FilterOidRequest
NdisFOidRequest
Filter Driver sends filtered OID request down the stack
FilterOidRequestComplete
NdisFOidRequestComplete
Status = Success or Failure
Status = Success or Failure

The filter driver can complete the OID request synchronously or asynchronously by returning NDIS_STATUS_SUCCESS or NDIS_STATUS_PENDING, respectively, from *FilterOidRequest*. *FilterOidRequest* can also complete synchronously with an error status.

A filter driver that successfully handles an OID set request must set the **SupportedRevision** member in the **NDIS_OID_REQUEST** structure upon return from the OID set request. The **SupportedRevision** member notifies the initiator of the OID request about which revision the driver supported. For more information about version information in NDIS structures, see Specifying NDIS Version Information.

If *FilterOidRequest* returns NDIS_STATUS_PENDING, it must call the **NdisFOidRequestComplete** function after it completes the OID request. In this case, the driver passes the results of the request at the *OidRequest* parameter of **NdisFOidRequestComplete**. The driver passes the final status of the request at the *Status* parameter of **NdisFOidRequestComplete**.

If *FilterOidRequest* returns NDIS_STATUS_SUCCESS, it returns the results of a query request in the **NDIS_OID_REQUEST** structure at the *OidRequest* parameter. In this case, the driver does not call the **NdisFOidRequestComplete** function.

To forward an OID request to underlying drivers, a filter driver calls the **NdisFOidRequest** function. If a request should not be forwarded to the underlying drivers, a filter driver can complete the request immediately. To complete the request without forwarding, the driver can return NDIS_STATUS_SUCCESS (or an error status) from *FilterOidRequest*, or it can call **NdisFOidRequestComplete** after returning NDIS_STATUS_PENDING.

Note Before the driver calls **NdisFOidRequest**, the driver must allocate an **NDIS_OID_REQUEST** structure and transfer the request information to the new structure by calling **NdisAllocateCloneOidRequest**.

The forwarded request proceeds the same as a request initiated by a filter driver. For more information, see Generating OID Requests from an NDIS Filter Driver.

After the underlying drivers complete a forwarded request, the filter driver can modify the response, if necessary, and pass it to overlying drivers.

A filter driver can receive OID requests from overlying drivers when it is in the *Restarting*, *Running*, *Pausing*, or *Paused* state.

Note Like miniport drivers, filter drivers can receive only one OID request at a time. Because NDIS serializes requests that are sent to a filter module, a filter driver cannot be called at *FilterOidRequest* before it completes the previous request.

The following is an example of a filter driver modifying an OID request:

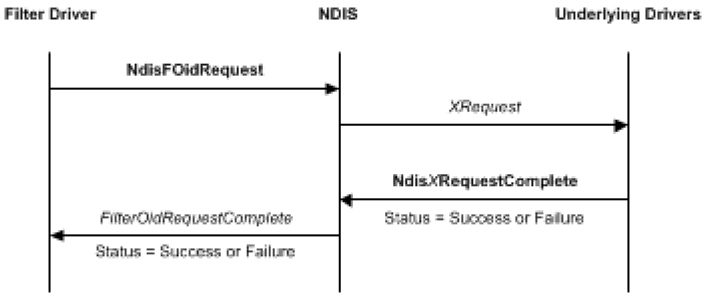
- A filter driver adds a header. In this case, after the driver receives a response to a query for OID_GEN_MAXIMUM_FRAME_SIZE from the underlying drivers, the filter subtracts the size of its header from the response. The driver subtracts its header size because the driver inserts a header in front of each sent packet and removes the header in each received packet.

Generating OID Requests from an NDIS Filter Driver

Article • 03/14/2023

A filter driver can originate OID query or set requests to underlying drivers by calling the **NdisFOidRequest** function.

The following figure illustrates an OID request that is originated by a filter driver.



After a filter driver calls the **NdisFOidRequest** function, NDIS calls the request function of the next underlying driver. For more information about how a miniport driver handles OID requests, see OID Requests for an Adapter.

To complete synchronously, **NdisFOidRequest** returns NDIS_STATUS_SUCCESS or an error status. To complete asynchronously, **NdisFOidRequest** returns NDIS_STATUS_PENDING.

To determine what information was successfully handled by an underlying driver, filter drivers that issue OID requests must check the value in the **SupportedRevision** member in the NDIS_OID_REQUEST structure after the OID request returns. For more information about NDIS version information, see Specifying NDIS Version Information.

If **NdisFOidRequest** returns NDIS_STATUS_PENDING, NDIS calls the *FilterOidRequestComplete* function after the underlying drivers complete the OID request. In this case, NDIS passes the results of the request at the *OidRequest* parameter of *FilterOidRequestComplete*. NDIS passes the final status of the request at the *Status* parameter of *FilterOidRequestComplete*.

If **NdisFOidRequest** returns NDIS_STATUS_SUCCESS, it returns the results of a query request in the **NDIS_OID_REQUEST** structure at the *OidRequest* parameter. In this case, NDIS does not call the *FilterOidRequestComplete* function.

A driver can call **NdisFOidRequest** when it is in the *Restarting*, *Running*, *Pausing*, or *Paused* state.

Note A filter driver should keep track of OID requests that it originates and make sure that it does not call the **NdisFOidRequestComplete** function when such requests are complete.

Filter Module Direct OID Requests

Article • 12/15/2021

To support the direct OID request path, filter drivers provide *FilterXxx* function entry points in the **NDIS_FILTER_DRIVER_CHARACTERISTICS** structure and NDIS provides **NdisF*Xxx*** functions for filter drivers.

The *direct OID request interface* is similar to the standard OID request interface. For example, the **NdisFDirectOidRequest** and *FilterDirectOidRequest* functions are similar to the **NdisFOidRequest** and *FilterOidRequest* functions.

Note NDIS 6.1 and later support specific OIDs for use with the direct OID request interface. OIDs that existed before NDIS 6.1 and some NDIS 6.1 OIDs are not supported. To determine if an OID can be used in the direct OIDs interface, see the OID reference page. For example, see the note in the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID.

Filter drivers must be able to handle direct OID requests that are not serialized. Unlike the standard OID request interface, NDIS does not serialize direct OID requests with other requests that are sent with the direct OID interface or with the standard OID request interface. Also, filter drivers must be able to handle direct OID requests at IRQL <= DISPATCH_LEVEL.

To support the direct OIDs request interface, use the documentation for the standard OID request interface. The following table shows the relationship between the functions in the direct OID request interface and the standard OID request interface.

Direct OID function	Standard OID function
FilterDirectOidRequest	*FilterOidRequest*
FilterCancelDirectOidRequest	*FilterCancelOidRequest*
FilterDirectOidRequestComplete	*FilterOidRequestComplete*
NdisFDirectOidRequest	**NdisFOidRequest**
NdisFDirectOidRequestComplete	**NdisFDirectOidRequestComplete**
NdisFCancelDirectOidRequest	**NdisFCancelOidRequest**

Filter Module Synchronous OID Requests

Article • 12/15/2021

To support the Synchronous OID request path, filter drivers provide a *FilterSynchronousOidRequest* function entry point in the **NDIS_FILTER_DRIVER_CHARACTERISTICS** structure when they call the **NdisFRegisterFilterDriver** function.

ⓘ Note

NDIS 6.81 supports specific OIDs for use with the Synchronous OID request interface. OIDs that existed before NDIS 6.80 and some NDIS 6.80 OIDs are not supported. To determine if an OID can be used in the Synchronous OID request interface, see the OID reference page.

To support the Synchronous OID request interface, use the documentation for the standard OID request interface. The following table shows the relationship between the functions in the Synchronous OID request interface and the standard OID request interface.

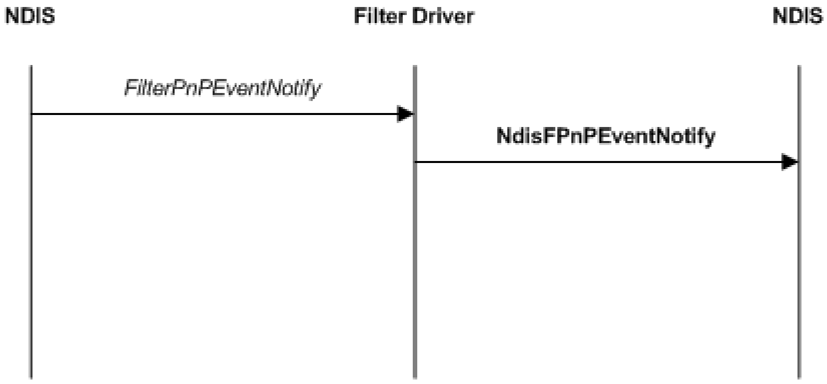
Synchronous OID function	Standard OID function
FilterSynchronousOidRequest	*FilterOidRequest*
FilterSynchronousOidRequestComplete	*FilterOidRequestComplete*

Filter Module PnP Event Notifications

Article • 12/15/2021

Filter drivers can receive all the device Plug and Play (PnP) notifications that underlying miniport drivers receive. Also, filter drivers can receive all the network PnP notifications that overlying protocol drivers receive.The handling of PnP notifications is driver specific.

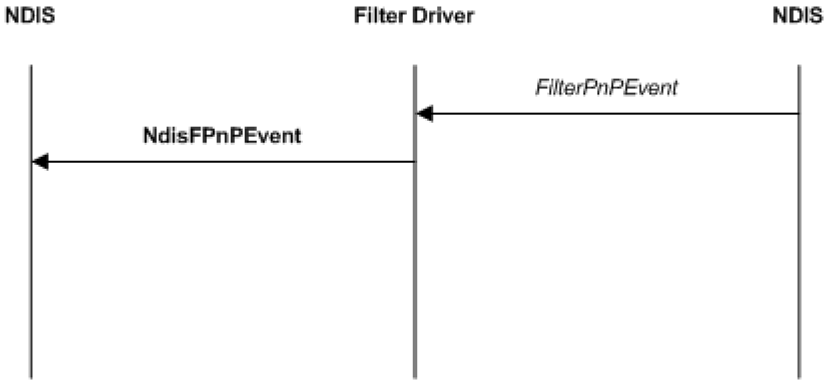
The following figure illustrates a filtered device PnP event notification.



Filter drivers provide a *FilterDevicePnPEventNotify* function that NDIS calls to pass in device PnP and Power Management event notifications. This is similar to the *MiniportDevicePnPEventNotify* function.

Filter drivers can forward device PnP and Power Management events to underlying drivers. To forward a device PnP or Power Management event, call the **NdisFDevicePnPEventNotify** function.

The following figure illustrates a filtered network PnP event notification.



Filter drivers provide a *FilterNetPnPEvent* function that NDIS calls to pass in network PnP and Power Management event notifications. This is similar to the *ProtocolNetPnPEvent* function.

Filter drivers can forward network PnP and Power Management events to overlying drivers. To forward a network PnP or Power Management event, call the

NdisFNetPnPEvent function.

Filter drivers should handle driver stack changes. For more information about driver stack changes, see Modifying a Running Driver Stack.

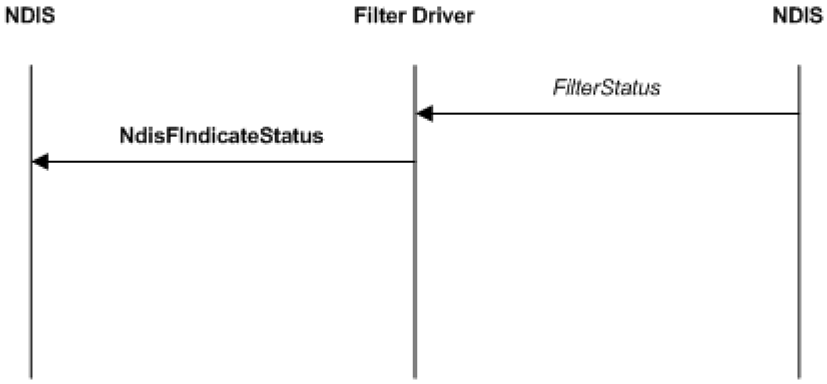
If necessary to allow handling of these events, NDIS can initiate a pause operation after the PnP or Power Management notification. For more information, see Pausing a Driver Stack.

Filter Module Status Indications

Article • 12/15/2021

Filter drivers can supply a *FilterStatus* function that NDIS calls when an underlying driver reports status. Filter drivers can also initiate status indications.

The following figure illustrates a filtered status indication.



NDIS calls a filter driver's *FilterStatus* function, after an underlying driver calls a status indication function (**NdisMIndicateStatusEx** or **NdisFIndicateStatus**). For more information about how to indicate status from a miniport driver, see Adapter Status Indications.

A filter driver calls **NdisFIndicateStatus** in its *FilterStatus* function, to pass on a filtered status indication to overlying drivers. A filter driver can filter out status indications (by not calling **NdisFIndicateStatus**) or modify the indicated status before it calls **NdisFIndicateStatus**.

To originate status indications, filter drivers call **NdisFIndicateStatus** without a prior call to *FilterStatus*.

In this case, the filter driver should set the **SourceHandle** member to the handle that NDIS passed to the *NdisFilterHandle* parameter of the *FilterAttach* function. If the status indication is associated with an OID request, the filter driver can set the **DestinationHandle** and **RequestId** members so that NDIS can provide the status indication to a specific protocol binding.

After a filter driver calls **NdisFIndicateStatus**, NDIS calls the status function (**ProtocolStatusEx** or *FilterStatus*) of the next overlying driver.

NDIS Filter Driver Installation

Article • 05/30/2023

This section provides information about installing NDIS filter drivers. Lightweight Filter drivers are different from filter intermediate drivers. The configuration manager supplies NDIS with a list of filter modules for each miniport adapter. There's no virtual device (or virtual miniport) that is associated with a filter driver as there is with an NDIS filter intermediate driver.

To install a filter driver, you must provide a single INF file. The configuration manager reads configuration information about the filter driver from the INF file and copies it to the registry.

The filter driver INF file defines a network service. Filter drivers don't have a miniport INF file. For an example filter driver INF file, see the ndislwf sample driver.

Once you have provided your filter driver INF file, to install or uninstall your filter driver you must use the `INetCfg` family of Network Configuration Interfaces. For example, to install or remove network components, call into the INetCfgClassSetup interface. You can either call into these interfaces programmatically or you can indirectly call them with netcfg.exe, which calls `INetCfg` for you. You can't install a driver package through the `INetCfg` and use the Driver Store feature on older Windows versions. To successfully install the driver package in this scenario, you need to have a minimum OS build number of 25319. You can't use SetupAPI to install or uninstall an NDIS filter driver.

For an example of calling into `INetCfg` through code, see the Bindview Network Configuration Utility sample .

This section includes:

Specifying Filter Driver Binding Relationships

INF File Settings for Filter Drivers

Accessing Configuration Information for a Filter Driver

Specifying Filter Driver Binding Relationships

Article • 12/15/2021

In a network driver INF file, the **UpperRange** entry lists the possible upper bindings and the **LowerRange** entry lists the possible lower bindings. These entries can contain various system-defined values.

For filter drivers, you must set the value of the **UpperRange** and **LowerRange** entries to **noupper** and **nolower**, respectively. The following example illustrates these INF file entries for a filter driver.

```INF
HKR, Ndi\Interfaces,UpperRange,,"noupper"
HKR, Ndi\Interfaces,LowerRange,,"nolower"
```

In a filter driver, the **FilterMediaTypes** entry in the filter INF file defines the driver's bindings to other drivers. **FilterMediaTypes** specifies the media types that the filter driver services. For a list of possible media types, see the list of Microsoft-supplied **LowerRange** values in Specifying Binding Interfaces. The following example illustrates a **FilterMediaTypes** entry for a filter driver.

```INF
HKR, Ndi\Interfaces, FilterMediaTypes,,"ethernet"
```

When the computer loads a filter driver, the driver is inserted into all of the existing protocol-to-adapter bindings, depending on the media types that **FilterMediaTypes** lists.

INF File Settings for Filter Drivers

Article • 12/15/2021

The filter driver INF file settings specify the characteristics of the filter driver. For example, filter drivers can be modifying or monitoring and can be mandatory or optional. The INF file also specifies general configuration parameters and other information to associate the filter driver with particular miniport adapters. Filter drivers are identified by a unique filter identification GUID (FID GUID).

This section includes:

Configuring an INF File for a Monitoring Filter Driver

Configuring an INF File for a Modifying Filter Driver

Configuring an INF File for a Monitoring Filter Driver

Article • 05/30/2023

The following NDIS filter driver installation issues are associated with monitoring filter drivers:

- Set the **Class** INF file entry to **NetService** in the INF file. The following example shows a sample **Class** entry for the INF file.

  INF

  ```
  Class = NetService
  ```

- The *DDInstall* section in a filter driver INF file must have a **Characteristics** entry. The following example shows how you should define the **Characteristics** entry in your filter INF file.

  INF

  ```
  Characteristics=0x40000
  ```

  The 0x40000 value indicates that NCF_LW_FILTER (0x40000) is set. Filter drivers must not set the NCF_FILTER (0x400) flag. The values of the NCF_ *Xxx* flags are defined in *Netcfgx.h*. For more information about NCF_ *Xxx* flags, see DDInstall Section in a Network INF File.

- Set the **NetCfgInstanceId** INF file entry in the INF file, as the following example shows.

  INF

  ```
  NetCfgInstanceId="{5cbf81bf-5055-47cd-9055-a76b2b4e3697}"
  ```

  You can use the *Uuidgen.exe* tool to create the GUID for the **NetCfgInstanceId** entry.

- The *DDInstall* section of the INF file for a filter driver must include an **Addreg** directive for an **Ndi** key. The INF file must specify the **Service** entry under the **Ndi** key. The **ServiceBinary** entry in the *service-install* section of the INF file specifies the path to the binary for the filter driver. For more information, see Adding

Service-Related Values to the Ndi Key and DDInstall.Services Section in a Network INF File.

- The *DDInstall* section in a filter driver INF file must have **FilterType** and **FilterRunType** entries. To specify a monitoring filter, define the **FilterType** entry in your INF file, as the following example shows.

  INF

  ```
  HKR, Ndi,FilterType,0x00010001 ,0x00000001
  ```

  The **FilterType** value 0x00000001 indicates that the filter is a monitoring filter.

- Define the **FilterRunType** entry in your INF file, as the following example shows.

  INF

  ```
  HKR, Ndi,FilterRunType,0x00010001 ,0x00000002
  ```

  The 0x00000002 value in the preceding example indicates that the filter module is optional. To install a mandatory filter module, set the **FilterRunType** entry to 0x00000001. For more information, see Mandatory Filter Drivers.

  Note We highly recommend that a monitoring lightweight filter (LWF) driver should not be mandatory, unless it is to be used in a controlled environment where there will be no optional modifying LWF drivers. This is because a mandatory monitoring LWF driver can cause optional modifying LWF drivers to fail *FilterAttach*. A monitoring LWF driver is bound over every modifying filter and binding by design to facilitate monitoring of networking traffic at all levels. Consider the following scenario:

  - An instance of a mandatory monitoring LWF driver is installed over an optional modifying LWF driver.
  - The lower modifying optional LWF driver fails to attach to a lower component. This will cause the mandatory monitoring LWF driver's *FilterAttach* handler not to be called.
  - Because now an instance of a mandatory LWF driver is not loaded, NDIS will not bind any protocols (such as TCP/IP) to the interface or NIC, thus rendering the interface to be unusable.

- The following example shows how a filter driver INF file specifies the name of the service.

  INF

```
HKR, Ndi,Service,,"NdisMon"
```

In this example, "NdisMon" is the name of the driver's service as it is reported to NDIS. Note that the name of a filter driver's service can be different from the name of the binary for the driver, but typically they are the same.

- The following example shows how the filter INF file references the name of the filter driver's service when it adds that service.

  INF

  ```
  [Install.Services]
  AddService=NdisMon,,NdisMon_Service_Inst

  [NdisMon_Service_Inst]
  DisplayName     = %NdisMon_Desc%
  ServiceType     = 1 ;SERVICE_KERNEL_DRIVER
  StartType       = 1 ;SERVICE_SYSTEM_START
  ErrorControl    = 1 ;SERVICE_ERROR_NORMAL
  ServiceBinary   = %13%\ndisMon.sys
  LoadOrderGroup  = NDIS
  Description     = %NdisMon_Desc%
  AddReg          = Common.Params.Reg
  ```

- A filter INF file must specify at least the primary service name of the filter for the **CoServices** attribute, as the following example shows.

  INF

  ```
  HKR, Ndi,CoServices,0x00010000,"NdisMon"
  ```

  For more information about the **CoServices** attribute, see Adding Service-Related Values to the Ndi Key.

- The **FilterClass** value in the INF file for a filter driver determines its order in a stack of modifying filters. However, monitoring filter drivers do not define the **FilterClass** key. Instead the monitoring filter module that is installed first is closest to the miniport adapter.

- You must define the following entries in the monitoring filter driver INF file to control the driver bindings:

  INF

  ```
  HKR, Ndi\Interfaces,UpperRange,,"noupper"
  HKR, Ndi\Interfaces,LowerRange,,"nolower"
  ```

```
HKR, Ndi\Interfaces, FilterMediaTypes,,"ethernet"
```

For more information about controlling the driver bindings, see Specifying Filter Driver Binding Relationships.

- A monitoring filter INF file can specify common parameter definitions for the filter driver, parameters that are associated with a specific adapter, and parameters that are associated with a particular instance (filter module). The following example shows some common parameter definitions.

> ⊗ **Caution**
>
> Using **HKR AddReg** to put keys directly under the service state is a compliance violation. These keys need to be added under the Parameters key of the service to be compliant.

INF

```
[Common.Params.reg]

HKR, FilterDriverParams\DriverParam, ParamDesc, ,"Driverparam for
filter"
HKR, FilterDriverParams\DriverParam, default, ,"5"
HKR, FilterDriverParams\DriverParam, type,   ,"int"

HKR, FilterAdapterParams\AdapterParam, ParamDesc, ,"Adapterparam for
filter"
HKR, FilterAdapterParams\AdapterParam, default, ,"10"
HKR, FilterAdapterParams\AdapterParam, type,   ,"int"

HKR, FilterInstanceParams\InstanceParam, ParamDesc, ,"Instance param
for filter"
HKR, FilterInstanceParams\InstanceParam, default, ,"15"
HKR, FilterInstanceParams\InstanceParam, type,   ,"int"
```

Configuring an INF File for a Modifying Filter Driver

Article • 11/30/2022

The following NDIS filter driver installation issues are associated with modifying filter drivers. To create your own modifying filter driver INF file, you can also adapt the sample NDIS 6.0 filter driver.

- Set the **Class** INF file entry to **NetService** in the INF file. The following example shows a sample **Class** entry for the INF file.

  INF

  ```
  Class = NetService
  ```

- The *DDInstall* section in a filter driver INF file must have a **Characteristics** entry. The following example shows how you should define the **Characteristics** entry in your filter INF file.

  INF

  ```
  Characteristics=0x40000
  ```

  The 0x40000 value indicates that NCF_LW_FILTER (0x40000) is set. Filter drivers must not set the NCF_FILTER (0x400) flag. The values of the NCF_ *Xxx* flags are defined in *Netcfgx.h*. For more information about NCF_ *Xxx* flags, see DDInstall Section in a Network INF File.

- Set the **NetCfgInstanceId** INF file entry in the INF file, as the following example shows.

  INF

  ```
  NetCfgInstanceId="{5cbf81bd-5055-47cd-9055-a76b2b4e3697}"
  ```

  You can use the *Uuidgen.exe* tool to create the GUID for the **NetCfgInstanceId** entry.

- The *DDInstall* section of the INF file for a filter driver must include an **Addreg** directive for an **Ndi** key. The INF file must specify the **Service** entry under the **Ndi** key. The **ServiceBinary** entry in the *service-install* section of the INF file specifies the path to the binary for the filter driver. For more information, see Adding Service Related Values to the Ndi Key and DDInstall.Services Section in a Network INF File.

- The *DDInstall* section in a filter driver INF file must have **FilterType** and **FilterRunType** entries. To specify a modifying filter, define the **FilterType** entry in your INF file, as the following example shows.

  INF

  ```
  HKR, Ndi,FilterType,0x00010001 ,0x00000002
  ```

  The **FilterType** value 0x00000002 indicates that the filter is a modifying filter.

- Define the **FilterRunType** entry in your INF file, as the following example shows.

  INF

  ```
  HKR, Ndi,FilterRunType,0x00010001 ,0x00000001
  ```

  The 0x00000001 value in the preceding example indicates that the filter module is mandatory. To install an optional filter module, set the **FilterRunType** entry to 0x00000002. For more information, see Mandatory Filter Drivers.

- The following example shows how a modifying filter driver INF file specifies the name of the service.

  INF

  ```
  HKR, Ndi,Service,,"NdisLwf"
  ```

  In this example, NdisLwf is the name of the driver's service as it is reported to NDIS. Note that the name of a filter driver's service can be different from the name of the binary for the driver—but typically they are the same.

- The following example shows how the filter INF file references the name of the filter driver's service when it adds that service.

  INF

  ```
  [Install.Services]
  AddService=NdisLwf,,NdisLwf_Service_Inst;, common.EventLog

  [NdisLwf_Service_Inst]
  DisplayName     = %NdisLwf_Desc%
  ```

```
ServiceType      = 1 ;SERVICE_KERNEL_DRIVER
StartType        = 1 ;SERVICE_SYSTEM_START
ErrorControl     = 1 ;SERVICE_ERROR_NORMAL
ServiceBinary    = %13%\ndislwf.sys
LoadOrderGroup   = NDIS
Description      = %NdisLwf_Desc%
AddReg           = Common.Params.reg
```

- A filter INF file must specify at least the primary service name of the filter for the **CoServices** attribute, as the following example shows.

  INF

  ```
  HKR, Ndi,CoServices,0x00010000,"NdisLwf"
  ```

  For more information about the **CoServices** attribute, see Adding Service Related Values to the Ndi Key.

- The **FilterClass** value in the INF file for a filter driver determines its order in a stack of filters. Filter drivers must define the **FilterClass** key. The class of the driver can be one of the values in the following table.

Value	Description
scheduler	Packet scheduling filter service. This class of filter driver is the highest-level driver that can exist above encryption class filters in a driver stack. A packet scheduler detects the 802.1p priority classification that is given to packets by quality of service (QoS) signaling components and the scheduler sends those packets levels to underlying drivers according to their priority.
encryption	Encryption class filter drivers exist between scheduler and compression class filters.
compression	Compression class filter drivers exist between encryption and vpn class filters.
vpn	VPN class filter drivers exist between compression and load balance filter drivers.

Value	Description
loadbalance	Load balancing filter service. This class of filter driver exists between packet scheduling and failover drivers. A load balancing filter service balances its workload of packet transfers by distributing the workload over its set of underlying miniport adapters.
failover	Failover filter service. This class of filter driver exists between load balance and diagnostics drivers.
diagnostic	Diagnostic filter drivers exist below failover drivers in the stack.
custom	Filter drivers in custom class exist below diagnostic drivers.
provider_address	Provider address filter drivers exist below the in-box Hyper-V Network Virtualization ms_wnv filter and operate on provider address (PA) packets.

Note If multiple filter drivers have the same FilterClass, they will all be added to the layered stack of filter drivers. The system assigns a layering order to each modifying filter driver with the same FilterClass. In some cases, the system administrator can rearrange the relative order of filter drivers that have the same FilterClass.

The following example shows a sample **FilterClass** .

```INF
HKR, Ndi,FilterClass,, compression
```

- Only Hyper-V switch extension filter drivers are valid in the Hyper-V Extensible Switch. Hyper-V extensible switch filter drivers must define the FilterClass key with one of the values in the following table.

Value	Description

Value	Description
ms_switch_capture	Starting with NDIS 6.30, capture drivers monitor packet traffic in the Hyper-V extensible switch driver stack. This class of filter driver exists below custom drivers in the stack. For more information about this class of driver, see Capturing Extensions.
ms_switch_filter	Starting with NDIS 6.30, filtering drivers filter packet traffic and enforce port or switch policy for packet delivery through the extensible switch driver stack. This class of filter driver exists below **ms_switch_capture** drivers in the stack. For more information about this class of driver, see Filtering Extensions.
ms_switch_forward	Starting with NDIS 6.30, forwarding drivers filter perform the same functions as a filtering driver. Forwarding drivers also forward packets to and from extensible switch ports. This class of filter driver exists below **ms_switch_filter** drivers in the stack. For more information about this class of driver, see Forwarding Extensions.

- You must define the following entries in the modifying filter driver INF file to control the driver bindings.

  INF

  ```
  HKR, Ndi\Interfaces,UpperRange,,"noupper"
  HKR, Ndi\Interfaces,LowerRange,,"nolower"
  HKR, Ndi\Interfaces, FilterMediaTypes,,"ethernet"
  ```

  For more information about controlling the driver bindings, see Specifying Filter Driver Binding Relationships.

- A modifying filter INF file can specify common parameter definitions for the driver and parameters that are associated with a specific adapter. The following example shows some common parameter definitions.

⊗ Caution

Using **HKR AddReg** to put keys directly under the service state is a compliance violation. These keys need to be added under the Parameters key of the service to be compliant.

INF

```
[Common.Params.reg]

HKR, FilterDriverParams\DriverParam,  ParamDesc, , "Driverparam for lwf"
HKR, FilterDriverParams\DriverParam,  default, , "5"
HKR, FilterDriverParams\DriverParam,  type,  , "int"

HKR, FilterAdapterParams\AdapterParam,  ParamDesc, , "Adapterparam for lwf"
HKR, FilterAdapterParams\AdapterParam,  default, , "10"
HKR, FilterAdapterParams\AdapterParam,  type,  , "int"
```

Accessing Configuration Information for a Filter Driver

Article • 12/15/2021

NDIS supports a set of functions that provide access to filter driver registry parameters. Filter drivers can access these parameters during the attach or restart operations or when they are processing a Plug and Play (PnP) notification. For more information about PnP notifications, see Filter Module PnP Event Notifications. For more information about attaching a filter module, see Attaching a Filter Module. For more information about restart operations, see Starting a Filter Module.

Filter drivers call the **NdisOpenConfigurationEx** function to access the registry settings. If a filter driver obtained the handle in the **NdisHandle** member of the **NDIS_CONFIGURATION_OBJECT** structure by calling the **NdisFRegisterFilterDriver** function, the **NdisOpenConfigurationEx** function provides a handle to the registry location where the filter driver's configuration parameters are stored. Filter drivers can use the configuration handle until they call the **NdisFDeregisterFilterDriver** function.

If a filter driver obtained the handle in **NdisHandle** from the *NdisFilterHandle* parameter of the *FilterAttach* function, **NdisOpenConfigurationEx** provides a handle to the registry location where a filter module's configuration parameters are stored. The filter driver can use the configuration handle until NDIS detaches the filter module and the *FilterDetach* function returns. If a monitoring filter driver specifies the NDIS_CONFIG_FLAG_FILTER_INSTANCE_CONFIGURATION flag in the **Flags** member of the **NDIS_CONFIGURATION_OBJECT** structure, the driver can access the filter module configuration for a specific filter module when there are multiple filter modules that are configured over the same miniport adapter. Modifying filter drivers must not use this flag.

After a driver is done accessing the configuration information, the driver must call the **NdisCloseConfiguration** function to release the configuration handle and related resources.

NDIS Intermediate Drivers Guide

Article • 03/14/2023

NDIS intermediate drivers interface between upper-level protocol drivers and miniport drivers. Some applications that might require an intermediate driver include:

- Media translation between an old transport driver and a miniport driver that manages a media type unknown to the transport driver.
- Data filtering for security or other purposes.
- Load Balancing Failover (LBFO) solutions.
- Monitoring and collecting of network data statistics.

Before attempting to write an intermediate driver, you should read about NDIS miniport and protocol drivers. For more information about NDIS miniport drivers, see NDIS Miniport Drivers. For more information about NDIS protocol drivers, see NDIS Protocol Drivers.

The following sections introduce intermediate drivers and describe how to create and install such drivers:

Roadmap for Developing NDIS Intermediate Drivers

Article • 03/14/2023

To create a Network Driver Interface Specification (NDIS) intermediate driver package, follow these steps:

- Step 1: Learn about Windows architecture and drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- Step 2: Learn about NDIS.

  For general information about NDIS and NDIS drivers, see the following topics:

  Windows Network Architecture and the OSI Model

  Network Driver Programming Considerations

  Driver Stack Management

  NET_BUFFER Architecture

- Step 3: Determine additional Windows driver design decisions.

  For more information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- Step 4: Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user-mode application. For more information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Lab Kit (HLK) testing, see Building, Debugging, and Testing Drivers. For more information about building, testing, verifying, and debugging tools, see Driver Development Tools.

- Step 5: Read the intermediate driver, miniport driver, and protocol driver introduction topics. Introduction to NDIS Intermediate Drivers Introduction to NDIS Miniport Drivers NDIS Protocol Drivers

- Step 6: Read the writing intermediate drivers section.

  Intermediate drivers use a combination of protocol driver and miniport driver interfaces in addition to some intermediate driver specific interfaces. As an option, you can also read the miniport driver and protocol driver design guides.

- Step 7: Review the NDIS intermediate driver sample in the Windows driver samples repository on GitHub.

- Step 8: Develop (or port), build, test, and debug your NDIS driver.

  See the porting guides if you are porting an existing driver:

  - Porting NDIS 5.x Drivers to NDIS 6.0
  - Porting NDIS 6.x Drivers to NDIS 6.20
  - Porting NDIS 6.x Drivers to NDIS 6.30

    For more information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process will help ensure that you build a driver that works.

- Step 9: Create a driver package for your driver.

  For more information about how to install drivers, see Providing a Driver Package. For more information about how to install an NDIS driver, see Components and Files Used for Network Component Installation and Notify Objects for Network Components.

- Step 10: Sign and distribute your driver.

  The final step is to sign (optional) and distribute the driver. If your driver meets the quality standards that are defined for the Windows Hardware Lab Kit (HLK), you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual driver.

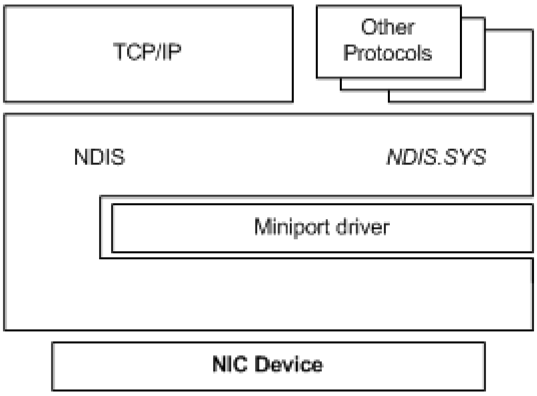
NDIS Intermediate Drivers Overview

Article • 03/14/2023

NDIS driver stacks must include miniport drivers and protocol drivers and can also include intermediate drivers. Because intermediate drivers are optional, you must understand the relationships between miniport drivers, protocol drivers, and NDIS before addressing intermediate drivers.

Miniport drivers control NIC devices and communicate with the lower edge of protocol drivers. Protocol drivers implement protocols, such as TCP/IP, and are above the miniport drivers in the driver stack. NDIS provides services to simplify development and maintenance of miniport drivers and protocol drivers.

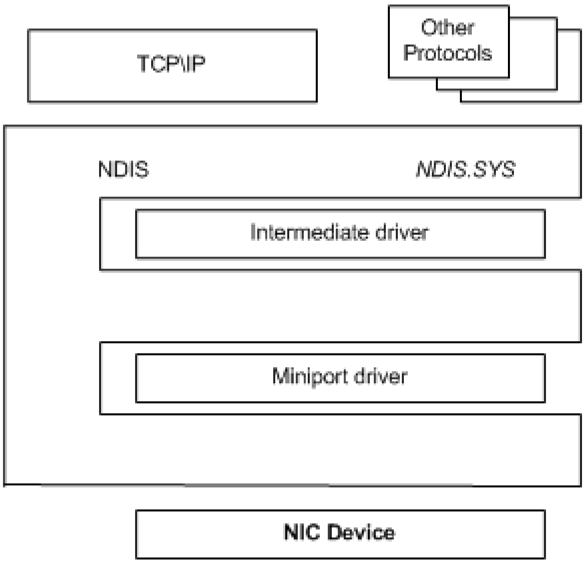
The relationships between miniport drivers, protocol drivers, and NDIS are illustrated in the following figure.



NDIS miniport drivers and protocol drivers are bound together through standard NDIS interfaces.

NDIS intermediate drivers include a protocol driver interface at their upper edge and a miniport driver interface at their lower edge. The intermediate driver's protocol interface allows it to load above a driver with a miniport driver interface. Therefore, intermediate drivers can load above miniport drivers or other intermediate drivers. The intermediate driver's miniport interface allows it to load below a driver with a protocol lower edge interface. Therefore, intermediate drivers can load below protocol drivers or below other intermediate drivers.

The relationships between miniport drivers, protocol drivers, intermediate drivers, and NDIS are illustrated in the following figure.

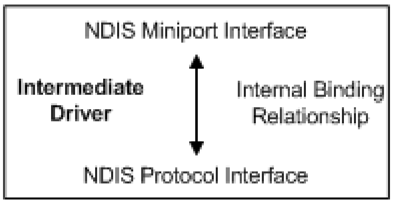


The intermediate driver miniport interface is called a virtual miniport. It is virtual in that it does not control a physical device directly. Instead, it relies on an underlying miniport driver to communicate with the physical device.

Bindings between the intermediate driver and other drivers are called external bindings. NDIS controls external bindings. The upper edge of the virtual miniport binds with the next-higher driver, which can be a protocol driver or another intermediate driver. The lower edge of the intermediate driver protocol binds to the next lower driver, which can be another intermediate driver or an underlying miniport driver.

The lower edge of the virtual miniport and the upper edge of the intermediate driver protocol do not require external bindings. Instead, the intermediate driver binds its virtual miniport and its protocol internally. These internal bindings, which are implementation specific, are not controlled by NDIS.

The following figure illustrates the internal bindings between the virtual miniport and the intermediate driver protocol.



There are two types of NDIS intermediate drivers: filter intermediate drivers and MUX intermediate drivers. The following topics describe these driver types:

NDIS Filter Intermediate Drivers

NDIS MUX Intermediate Drivers

NDIS Filter Intermediate Drivers

Article • 03/14/2023

Note Filter intermediate drivers are not supported in NDIS 6.0 and later. You should use the NDIS filter driver interface instead. For more information about NDIS filter drivers, see NDIS Filter Drivers.

In NDIS 5.*x*, an *NDIS filter intermediate driver* exposes one virtual miniport for each underlying miniport driver that is bound to the intermediate driver's lower (protocol) edge.

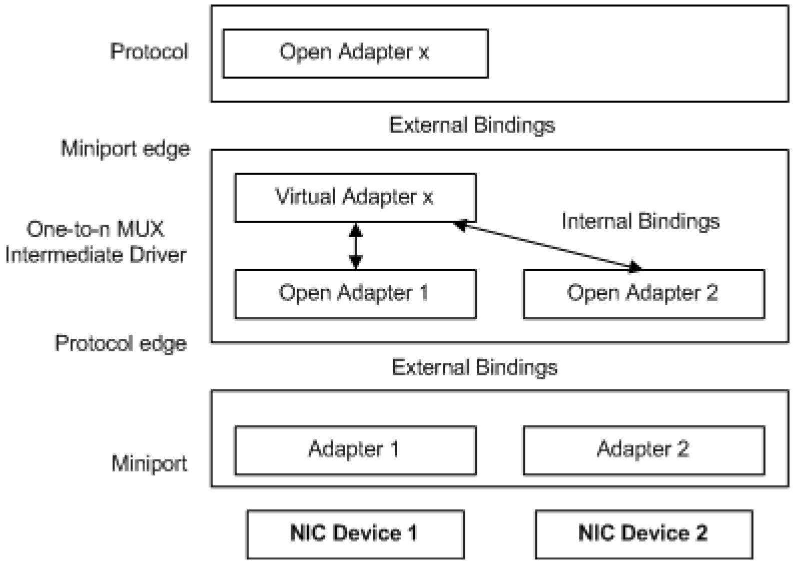
NDIS MUX Intermediate Drivers

Article • 03/14/2023

The number of virtual miniports that are exposed by a MUX intermediate driver can be different than the number of lower physical adapters that are bound to the driver. A MUX intermediate driver exposes virtual miniports in a one-to-*n*, *n*-to-one, or even an *m*-to-*n* relationship with underlying adapters. This variety results in complicated internal bindings and data paths.

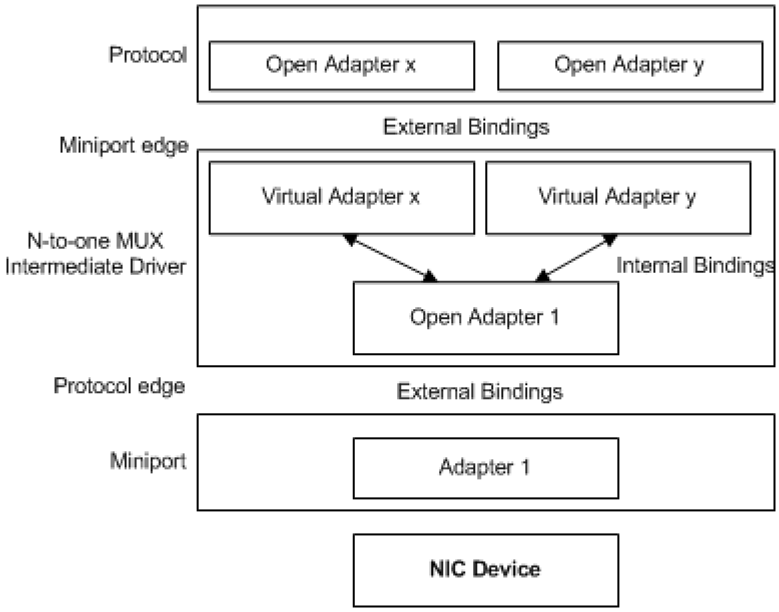
In a one-to-*n* configuration, a single MUX intermediate driver can bind to many physical adapters below. Transport drivers bind to the virtual miniport of the MUX intermediate driver in the same way that they bind to nonvirtual miniports. The MUX intermediate driver repackages and passes down all requests and send packets that are submitted to the intermediate driver for a specific connection. A Load Balancing Failover (LBFO) driver is an example of this type of MUX intermediate driver.

The following figure illustrates a one-to-*n* MUX intermediate driver configuration.



In an *n*-to-one configuration, a MUX intermediate driver can expose many virtual miniports for a single physical adapter below. Overlying protocol drivers bind to these virtual miniports of the MUX intermediate driver in the same way that they bind to nonvirtual miniports. The MUX intermediate driver handles requests and sends that are submitted to the driver for specific connections at each virtual miniport. The driver repackages and transfers these requests and sends down to the NDIS miniport driver for the bound physical adapter.

The following figure illustrates an *n*-to-one MUX intermediate driver configuration.



MUX intermediate drivers require a notify object DLL. When a MUX intermediate driver is initialized, its bindings are determined by the configuration established by its notify object DLL. For more information about installing MUX intermediate drivers, see MUX Intermediate Driver Installation.

The following list describes examples of *n*-to-one MUX intermediate drivers:

- 802 and proprietary virtual LANs are technologies that could be implemented as intermediate drivers similar to the MUX sample.
- The MUX Intermediate Driver Sample is an n-to-one MUX intermediate driver. MUX creates multiple virtual miniports layered above a single underlying miniport adapter.

Getting started writing NDIS Intermediate Drivers

Article • 03/14/2023

Unless noted otherwise, NDIS intermediate drivers provide the same services as miniport drivers and protocol drivers. The intermediate driver's miniport edge provides miniport driver services and the protocol edge provides protocol driver services. (For more information, see Writing NDIS Miniport Drivers and Writing NDIS Protocol Drivers.)The initialization for NDIS 6.0 and later intermediate drivers is different from the initialization for legacy intermediate drivers. Also, NDIS 6.0 and later drivers can register as a combined miniport-intermediate driver.

The following topics provide more information about intermediate driver initialization:

- Initializing an Intermediate Driver
- Initializing a Miniport-Intermediate Driver
- Unloading an Intermediate Driver
- Initializing a Virtual Miniport
- Halting a Virtual Miniport

Initializing an Intermediate Driver

Article • 12/15/2021

An NDIS intermediate driver registers its *MiniportXxx* functions and its *ProtocolXxx* functions in the context of its DriverEntry routine. To register its *MiniportXxx* functions, an intermediate driver must call the NdisMRegisterMiniportDriver function with the NDIS_INTERMEDIATE_DRIVER flag set. This flag is in the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure that the driver passes at *MiniportDriverCharacteristics*. To register its *ProtocolXxx* functions, an intermediate driver must call the NdisRegisterProtocolDriver function.

DriverEntry returns STATUS_SUCCESS, or its equivalent NDIS_STATUS_SUCCESS, if the driver registered as an NDIS intermediate driver successfully. If DriverEntry fails initialization by propagating an error status that was returned by an **NdisXxx** function or by a kernel-mode support routine, the driver will not remain loaded. **DriverEntry** must execute synchronously; that is, it cannot return STATUS_PENDING or its equivalent NDIS_STATUS_PENDING.

To register the intermediate driver with NDIS, the DriverEntry routine must, at a minimum:

- Call the NdisMRegisterMiniportDriver function with the NDIS_INTERMEDIATE_DRIVER flag set to register the driver's *MiniportXxx* functions.
- Call the NdisRegisterProtocolDriver function to register the driver's *ProtocolXxx* functions if the driver subsequently binds itself to an underlying NDIS driver.
- Call the NdisIMAssociateMiniport function to inform NDIS about the association between the driver's miniport upper edge and protocol lower edge.

If an error occurs in **DriverEntry** after **NdisMRegisterMiniportDriver** returns successfully, the driver must call the NdisMDeregisterMiniportDriver function before **DriverEntry** returns. If **DriverEntry** succeeds, the driver must call **NdisMDeregisterMiniportDriver** from its MiniportDriverUnload function.

Intermediate drivers share most of the **DriverEntry** requirements of protocol drivers and miniport drivers.

The initialization of an intermediate driver's virtual miniport occurs when the driver calls the NdisIMInitializeDeviceInstanceEx function from its ProtocolBindAdapterEx function.

NDIS calls the *ProtocolBindAdapterEx* function after all underlying miniport drivers have initialized.

In effect, the **DriverEntry** function of an NDIS intermediate driver can ignore the *RegistryPath* pointer after passing it to **NdisMRegisterMiniportDriver**. Such a driver can also ignore the *DriverObject* pointer after passing it to **NdisMRegisterMiniportDriver**. However, the driver should save the miniport driver handle value that is returned by **NdisMRegisterMiniportDriver** at *NdisMiniportDriverHandle* and the protocol handle value that is returned by **NdisRegisterProtocolDriver** at *NdisProtocolHandle* for subsequent calls to **NdisXxx** functions. The intermediate driver's *ProtocolBindAdapterEx* function binds the driver to each underlying miniport driver before its *MiniportInitializeEx* function is called to initialize the intermediate driver's virtual miniport. Still higher level protocol drivers subsequently bind themselves to the virtual miniport that it creates. This strategy enables an NDIS intermediate driver to allocate resources at the creation of the virtual miniport according to the features of the underlying miniport driver to which it is bound.

Initializing a Miniport-Intermediate Driver

Article • 12/15/2021

A miniport-intermediate driver combines a miniport driver for a virtual device, a protocol driver, and a miniport driver for a physical device. A miniport-intermediate driver functions similarly to an intermediate driver layered over a miniport driver. Such a driver allows an intermediate driver to communicate directly with an underlying miniport driver without incurring the performance penalties that might result with two separate drivers.

To register its physical miniport driver, a miniport-intermediate driver calls the NdisMRegisterMiniportDriver function with appropriate parameters just as for any miniport driver. To register its virtual miniport, the driver calls **NdisMRegisterMiniportDriver** again, but with the NDIS_INTERMEDIATE_DRIVER flag set in the structure at *MiniportDriverCharacteristics* .

For each virtual or physical device instance of a miniport-intermediate driver, if the **IMMiniport** registry key is set to **DWORD:0x0000001**, NDIS calls the *MiniportInitializeEx* function that the driver registered for the virtual device. Otherwise, NDIS calls the driver's *MiniportInitializeEx* function that the driver registered for the physical device.

Unloading an Intermediate Driver

Article • 12/15/2021

NDIS calls the *MiniportDriverUnload* function to unload an intermediate driver. Intermediate drivers must perform the same operations in *MiniportDriverUnload* as other miniport drivers. In addition to calling the **NdisMDeregisterMiniportDriver** function, an intermediate driver also calls **NdisDeregisterProtocolDriver**. *MiniportDriverUnload* should also perform any necessary cleanup operations, such as deallocating any protocol driver resources.

To perform cleanup operations before a intermediate driver is uninstalled, an intermediate driver can register a *ProtocolUninstall* function. For example, the protocol lower edge of an intermediate driver might require a *ProtocolUninstall* function. The intermediate driver can release its protocol edge resources in *ProtocolUninstall* before NDIS calls its *MiniportDriverUnload* function.

A miniport-intermediate driver calls **NdisMDeregisterMiniportDriver** twice, once for its physical device interface, and again for its virtual device interface.

Initializing a Virtual Miniport

Article • 04/29/2024

To initiate the initialization of a virtual miniport, an intermediate driver calls the **NdisIMInitializeDeviceInstanceEx** function. The intermediate driver usually makes this call from its *ProtocolBindAdapterEx* function. After the intermediate driver calls **NdisIMInitializeDeviceInstanceEx** and the Plug and Play manager requests NDIS to start the virtual device, NDIS calls the driver's *MiniportInitializeEx* function.

The call to *MiniportInitializeEx* can be in the context of **NdisIMInitializeDeviceInstanceEx** if the Plug and Play manager starts the virtual device before **NdisIMInitializeDeviceInstanceEx** returns. If the intermediate driver provides more than one virtual miniport, the driver must call **NdisIMInitializeDeviceInstanceEx** for each virtual miniport that it makes available.

NDIS passes initialization parameters to *MiniportInitializeEx* in an **NDIS_MINIPORT_INIT_PARAMETERS** structure at *MiniportInitParameters* . The **IMDeviceInstanceContext** member of the structure specifies a pointer to the context area for a virtual device. The driver passed this pointer to the **NdisIMInitializeDeviceInstanceEx** function at the *DeviceContext* parameter.

In *MiniportInitializeEx*, the intermediate driver performs the operations required to initialize a virtual miniport. This initialization is similar to the initialization of any other miniport adapter.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Halting a Virtual Miniport

Article • 12/15/2021

If an NDIS intermediate driver calls the **NdisIMDeinitializeDeviceInstance** function, NDIS calls the *MiniportHaltEx* function for the affected virtual miniport. An intermediate driver usually calls **NdisIMDeInitializeDeviceInstance** from its *ProtocolUnbindAdapterEx* function.

NDIS sets the *HaltAction* parameter to **NdisHaltDeviceInstanceDeInitialized** to indicate that NDIS is halting the adapter in response to an intermediate driver's call to the **NdisIMDeInitializeDeviceInstance** function.

The intermediate driver's *MiniportHaltEx* function must release all driver-allocated resources that are associated with a virtual miniport.

Intermediate Driver Design Concepts

Article • 12/15/2021

This section provides some basic information to help you start writing an NDIS intermediate driver. To write an NDIS intermediate driver, you must understand the NDIS miniport driver and protocol driver operations and functions.

The MUX intermediate driver sample in the Microsoft Windows Driver Kit (WDK) provides a basic example of an *n*-to-one MUX intermediate driver that you can adapt to your specific needs.

The virtual miniport of an NDIS intermediate driver must be deserialized. *Deserialized drivers* serialize the operation of their own *MiniportXxx* functions and queue internally all incoming send network data instead of relying on NDIS to perform these operations. This action results in significantly better full-duplex performance, if the driver's critical sections (code that can be executed by only one thread at a time) are kept small. For more information about deserialized drivers, see Deserialized NDIS Miniport Drivers.

An NDIS intermediate driver can support only connectionless communication at its virtual miniport. At its protocol interface, however, an NDIS intermediate driver can support either connectionless communication or connection-oriented communication. For more information about connection-oriented communication, see Connection-Oriented NDIS.

An intermediate driver is typically layered above one or more NDIS miniport drivers and below a transport driver. Intermediate drivers can also be layered with other intermediate drivers.

The following topics provide additional information about writing NDIS intermediate drivers:

Intermediate Driver DriverEntry Function

Dynamic Binding in an Intermediate Driver

Intermediate Driver Query and Set Operations

Intermediate Driver Network Data Management

Receiving Data in an Intermediate Driver

Transmitting Network Data Through an Intermediate Driver

Handling PnP Events and Power Management Events in an Intermediate Driver

Intermediate Driver Reset Operations

Status Indications in an Intermediate Driver

Intermediate Driver DriverEntry Function

Article • 12/15/2021

An intermediate driver's initial required entry point must be explicitly named **DriverEntry** so that the loader can properly identify it. All other exported driver functions, which are described in this section as *MiniportXxx* and *ProtocolXxx*, can have any vendor-specified name because they are passed as addresses to NDIS.

In an intermediate driver, **DriverEntry** must at a minimum:

1. Call **NdisMRegisterMiniportDriver** and save the handle that is returned in the *NdisMiniportDriverHandle* parameter.

2. Call **NdisRegisterProtocolDriver** to register the driver's *ProtocolXxx* functions if the driver subsequently binds itself to an underlying NDIS driver.

3. Call **NdisIMAssociateMiniport** to inform NDIS about the association between the driver's miniport upper edge and protocol lower edge.

An intermediate driver must register a *MiniportDriverUnload* unload handler. This unload handler is called when the system unloads the intermediate driver. If **DriverEntry** fails, this unload handler is not called; instead, the driver is simply unloaded. For more information about the unload handler, see Unloading an Intermediate Driver.

The unload handler should call **NdisDeregisterProtocolDriver** to deregister the protocol portion of the intermediate driver. The unload handler should also perform any necessary cleanup operations, such as reallocating resources used by the protocol portion of the driver.

Note that an unload handler differs from a *MiniportHaltEx* function: the unload handler has a more global scope, and the scope of the *MiniportHaltEx* function is restricted to a particular miniport adapter. The intermediate driver should clean up state information and reallocate resources when each underlying miniport driver that is bound to it is halted. For information about handling the halt operation for virtual miniports, see Halting a Virtual Miniport.

ProtocolUninstall is an optional unload handler. Register an entry point for this function in the *ProtocolCharacteristics* structure that you pass to **NdisRegisterProtocolDriver**. NDIS calls *ProtocolUninstall* in response to a user request to uninstall an intermediate driver. NDIS calls *ProtocolUnbindAdapterEx* once for each bound adapter, and then NDIS calls *ProtocolUninstall*. This handler is called before the system actually unloads the driver. This timing provides a chance to release any device objects or other resources that might otherwise prevent the system from calling the unload handler that is registered with **NdisMRegisterMiniportDriver** and unloading the driver.

DriverEntry can initialize spin locks to protect any globally-shared resources that the intermediate driver allocates, such as state variables, structures, and memory areas. The driver uses these resources to track connections and to track sends in progress or driver-allocated queues.

If **DriverEntry** fails to allocate any resources that the driver needs to carry out network I/O operations, it should release any previously allocated resources and return an appropriate error status.

The following topics further describe how to register intermediate drivers:

Registering as an NDIS Intermediate Driver

Registering an Intermediate Driver as a Miniport Driver

Registering an Intermediate Driver as a Protocol Driver

Registering as an NDIS Intermediate Driver

Article • 03/14/2023

An NDIS intermediate driver must register its *MiniportXxx* functions and its *ProtocolXxx* functions with NDIS in the context of its **DriverEntry** function. To register its *MiniportXxx* functions, an intermediate driver must call **NdisMRegisterMiniportDriver** with the NDIS_INTERMEDIATE_DRIVER flag set. This flag is in the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure that the driver passes at *MiniportDriverCharacteristics* . This call exports the intermediate driver's *MiniportXxx* functions. For more information about registering *MiniportXxx* functions, see Registering an Intermediate Driver as a Miniport Driver.

Note that the intermediate driver controls when its virtual miniports are initialized, and thus, when the driver is ready to accept sends and requests on an adapter. NDIS calls the intermediate driver's *MiniportInitializeEx* function after the Plug and Play (PnP) manager has started the virtual miniport device and after the intermediate driver has called **NdisIMInitializeDeviceInstanceEx** for that device. The call to *MiniportInitializeEx* can happen at a later time and therefore is not necessarily within the context of the call to **NdisIMInitializeDeviceInstanceEx**. If the intermediate driver exports more than one virtual miniport, the driver must call **NdisIMInitializeDeviceInstanceEx** for each virtual miniport that it makes available for network requests.

To register its *ProtocolXxx* functions, an intermediate driver must call the **NdisRegisterProtocolDriver** function. For more information about registering *ProtocolXxx* functions, see Registering an Intermediate Driver as a Protocol Driver.

Registering an Intermediate Driver as a Miniport Driver

Article • 12/15/2021

An intermediate driver calls NdisMRegisterMiniportDriver to export its *MiniportXxx* functions. The *NdisMiniportDriverHandle* that is returned by **NdisMRegisterMiniportDriver** must be retained by the intermediate driver and input to NDIS when the driver calls NdisIMInitializeDeviceInstanceEx.

The intermediate driver must:

1. Zero-initialize an NDIS_MINIPORT_DRIVER_CHARACTERISTICS structure with NdisZeroMemory.

2. Store the addresses of the mandatory *MiniportXxx* functions, as well as any optional *MiniportXxx* functions that the driver exports.

An intermediate driver that supports NDIS 6.0 features must register as a version 6.0 miniport driver. For more information about specifying miniport driver version numbers, see NDIS_MINIPORT_DRIVER_CHARACTERISTICS.

You must set the following entries in *MiniportCharacteristics* to a valid *MiniportXxx* function address unless the function is optional and is not exported. If the driver does not export the function, set the address to **NULL**.

SetOptionsHandler
MiniportSetOptions is an optional function. NDIS calls *MiniportSetOptions* so the intermediate driver can specify optional handlers.

InitializeHandlerEx
NDIS calls *MiniportInitializeEx* as a result of the intermediate driver calling **NdisIMInitializeDeviceInstanceEx** to initialize its miniport adapter operations for the virtual miniport being initialized.

HaltHandlerEx
MiniportHaltEx is a required function. NDIS calls *MiniportHaltEx* if the virtual miniport device that the intermediate driver exposed is disabled or stopped, or if the intermediate driver called NdisIMDeInitializeDeviceInstance to initiate its removal.

UnloadHandler
MiniportDriverUnload is a required function. NDIS calls *MiniportDriverUnload* to unload the intermediate driver.

PauseHandler
MiniportPause is a required function. NDIS calls *MiniportPause* to stop the flow of network data through a specified virtual miniport of the intermediate driver.

RestartHandler
MiniportRestart is a required function. NDIS calls *MiniportRestart* to restart the flow of network data through a specified virtual miniport of the intermediate driver.

OidRequestHandler
MiniportOidRequest receives OID_*XXX* requests originating from an overlying driver that has called **NdisOidRequest** or from NDIS. The intermediate driver might handle a request or pass it on to the underlying miniport driver.

SendNetBufferListsHandler
MiniportSendNetBufferLists receives an array of one or more pointers to **NET_BUFFER_LIST** structures that specify network data for transmission over the network. Every intermediate driver should supply a *MiniportSendNetBufferLists* function. For more information, see Transmitting Network Data Through an Intermediate Driver.

ReturnNetBufferListsHandler
MiniportReturnNetBufferLists receives a returned **NET_BUFFER_LIST** structure that it previously indicated to a higher-level driver by calling **NdisMIndicateReceiveNetBufferLists**. The call to **NdisMIndicateReceiveNetBufferLists** relinquishes control of the resources indicated to the higher-level driver. After the higher-level driver consumes each indication, the intermediate driver allocated NET_BUFFER_LIST structure and the resources it describes are returned to the *MiniportReturnNetBufferLists* function.

CancelSendHandler
MiniportCancelSend is a required function. NDIS calls *MiniportCancelSend* to cancel a send request.

CheckForHangHandler
MiniportCheckForHangEx is not required for intermediate drivers, so they should set this entry point to **NULL**.

ResetHandlerEx
MiniportResetEx is not required for intermediate drivers, so they should set this entry point to **NULL**.

DevicePnPEventNotifyHandler
The entry point for the *MiniportDevicePnPEventNotify* function.

ShutdownHandlerEx
MiniportShutdownEx is a required function. *MiniportShutdownEx* restores the virtual miniport to its initial state (before the intermediate driver's **DriverEntry** routine runs).

CancelOidRequestHandler
MiniportCancelOidRequest is a required function. NDIS calls *MiniportCancelOidRequest* to cancel an OID request.

An intermediate driver might require other *MiniportXxx* functions that are implementation specific. For information about registering optional, see Configuring Optional Miniport Driver Services.

Certain miniport driver handler functions are never supplied by an intermediate driver. Reasons for this include: such drivers do not manage interrupting devices, or such drivers do not allocate buffers at raised IRQL.

Note Intermediate drivers must include pause and restart functionality. Include support for pause and restart of virtual miniports, if needed, when NDIS pauses an underlying driver stack. For more information about pause and restart, see Driver Stack Management.

Registering an Intermediate Driver as a Protocol

Article • 12/15/2021

An intermediate driver registers its *ProtocolXxx* functions with NDIS in the context of its **DriverEntry** function by calling **NdisRegisterProtocolDriver**.

Registering an intermediate driver as a protocol is nearly identical to registering as a protocol driver. For more information, see Initializing a Protocol Driver.

An intermediate driver with a connection-oriented lower edge must register as a connection-oriented client. A connection-oriented client uses the call-set-up and tear-down services of a call manager or integrated miniport call manager (MCM). A connection-oriented client also uses the send and receive capabilities of a connection-oriented miniport driver or an MCM to send and receive data. For more information, see Connection-Oriented Operations Performed by Clients.

An intermediate driver might require other *ProtocolXxx* functions that are implementation specific. For information about registering optional *ProtocolXxx* functions, see Configuring Optional Protocol Driver Services.

Dynamic Binding in an Intermediate Driver

Article • 12/15/2021

An intermediate driver must support dynamic binding to underlying miniport adapters by providing both a *ProtocolBindAdapterEx* and a *ProtocolUnbindAdapterEx* function.

When a miniport adapter becomes available, NDIS calls the *ProtocolBindAdapterEx* function of any intermediate driver that can bind to that miniport adapter. As part of the binding operation, the intermediate driver should initialize a virtual miniport that is associated with that miniport adapter. When a miniport adapter is removed, NDIS calls the *ProtocolUnbindAdapterEx* function of any intermediate driver that is bound to that miniport adapter.

The following topics contain additional information about dynamic binding operations in intermediate drivers:

Intermediate Driver Binding Operations

Opening an Adapter Underlying an Intermediate Driver

Initializing Virtual Miniports

Intermediate Driver Unbinding Operations

Intermediate Driver Binding Operations

Article • 12/15/2021

When a miniport adapter becomes available, NDIS calls the *ProtocolBindAdapterEx* function of any intermediate driver that can bind to that miniport adapter.

An intermediate driver must provide the protocol binding operations documented in Binding to an Adapter.

Binding-time actions include allocating and initializing an adapter-specific context area for the binding, initializing any virtual miniports, and calling **NdisOpenAdapterEx** to bind to the adapter.

Intermediate drivers are not required to allocate separate **NET_BUFFER_LIST** structure pools for each binding. Intermediate drivers are required to allocate NET_BUFFER_LIST structure pools only if the drivers design requires it to allocate its own structures. Otherwise, the driver can just pass on the structures that it receives from other drivers. Such drivers should allocate different pools for send and receive.

For information about the requirements to allocate and manage network data, see Intermediate Driver Network Data Management.

Opening an Adapter Underlying an Intermediate Driver

Article • 12/15/2021

Opening an adapter underlying an intermediate driver is the same as opening an underlying adapter in a protocol driver. For more information about opening an underlying adapter in an intermediate driver, see Binding to an Adapter.

Initializing Virtual Miniports

Article • 12/15/2021

An intermediate driver initializes its virtual miniports after it has successfully opened an underlying miniport adapter and is ready to accept requests and sends on its virtual miniports. An intermediate driver calls **NdisIMInitializeDeviceInstanceEx** from its *ProtocolBindAdapterEx* function one or more times to request initialization of one or more virtual miniports.

Note An intermediate driver is not required to call **NdisIMInitializeDeviceInstanceEx** when it opens an underlying miniport adapter. There does not have to be a one-to-one relationship between virtual miniports and open adapters.

Set the *DriverInstance* parameter of **NdisIMInitializeDeviceInstanceEx** to the device name for the virtual miniport being initialized. The intermediate driver obtains the device name from the **UpperBindings** registry key.

For an *n*-to-one MUX intermediate driver that layers multiple virtual miniports over a single physical NIC, there must be a device name for every virtual miniport. The MUX intermediate driver requires a notify object that maintains the list of virtual miniport device names. The recommended location for the list is the **UpperBindings** registry key. In this case, the **UpperBindings** registry key is a MULTI_SZ entry that contains the list of device names. The MUX intermediate driver calls **NdisIMInitializeDeviceInstanceEx** once for each device name that is specified in the device name list.

Calling **NdisIMInitializeDeviceInstanceEx** results in a call to the intermediate driver's *MiniportInitializeEx* function to perform the initialization of the specified virtual miniport, provided that NDIS receives an IRP_MN_START_DEVICE to start the device. If NDIS does not receive such an IRP, NDIS will not call the intermediate driver's *MiniportInitializeEx* function. The call to *MiniportInitializeEx* can happen at a later time and therefore is not necessarily within the context of the call to **NdisIMInitializeDeviceInstanceEx**. If NDIS never calls *MiniportInitializeEx* for the virtual miniport referenced in a call to **NdisIMInitializeDeviceInstanceEx**, and the intermediate driver no longer requires the virtual miniport, the intermediate driver should call **NdisIMCancelInitializeDeviceInstance** to cancel the initialization of the virtual miniport. For example, suppose that an intermediate driver creates a virtual miniport in response to a successful binding to an underlying miniport. If that binding is removed before NDIS calls *MiniportInitializeEx*, the intermediate driver should call **NdisIMCancelInitializeDeviceInstance** to cancel the initialization of the miniport.

MiniportInitializeEx must allocate and initialize a virtual-miniport-specific context area. For more information about specifying the context area, see Initializing a Virtual Miniport.

An intermediate driver must operate as a deserialized driver. For more information about deserialized drivers, see Deserialized NDIS Miniport Drivers.

An intermediate driver should verify that the state information it maintains is properly initialized. If the driver requires send-related resources--for example, new **NET_BUFFER_LIST** structures for network data that *MiniportSendNetBufferLists* will transmit to the next lower layer--the NET_BUFFER_LIST structure pool can be allocated at this time.

Intermediate Driver Unbinding Operations

Article • 12/15/2021

An intermediate driver unbinds from an underlying miniport driver by calling **NdisCloseAdapterEx** from its *ProtocolUnbindAdapterEx* function. NDIS calls *ProtocolUnbindAdapterEx* if the underlying miniport adapter is no longer available.

An intermediate driver's *ProtocolUnbindAdapterEx* function might be called when the driver has an outstanding call to **NdisIMInitializeDeviceInstanceEx**. This situation occurs when NDIS has not yet called *MiniportInitializeEx* to initialize the corresponding virtual miniports. In this case, the intermediate driver must call **NdisIMCancelInitializeDeviceInstance** to attempt to cancel the initialization of these virtual miniports.

If the binding that is being closed is mapped to a device exported by the intermediate driver, and if that device was initialized by calling **NdisIMInitializeDeviceInstanceEx**, the intermediate driver can call **NdisIMDeInitializeDeviceInstance** to close the device. The result is that the intermediate driver's virtual miniport becomes no longer available for sends or requests made by higher-level drivers.

If an NDIS intermediate driver calls the **NdisIMDeInitializeDeviceInstance** function, NDIS calls the *MiniportHaltEx* function for the affected virtual miniport. For information about handling the halt operation for virtual miniports, see Halting a Virtual Miniport.

After an intermediate driver calls **NdisCloseAdapterEx**, it should fail any send requests for that binding with an appropriate error status.

For additional information about intermediate driver unbinding operations, see Unbinding from an Adapter.

Intermediate Driver Query and Set Operations

Article • 12/15/2021

After it has successfully bound to an underlying miniport adapter and initialized its virtual miniports, an intermediate driver queries the operating characteristics of the underlying miniport adapter and sets its own internal state. If appropriate, the intermediate driver also negotiates such parameters as lookahead buffer size for the binding with the underlying miniport adapter. Most of the attributes that are associated with an underlying miniport adapter are passed to the intermediate driver at the *BindParameters* parameter of the *ProtocolBindAdapterEx* function. Intermediate drivers should use the values that are passed to *ProtocolBindAdapterEx*, if possible, instead of issuing OID queries. However, an intermediate driver with a connectionless lower edge can issue OID queries by calling **NdisOidRequest**. An intermediate driver with a connection-oriented lower edge can issue OID queries by calling **NdisCoOidRequest**.

An intermediate driver can also receive query and set requests from higher level drivers through its *MiniportOidRequest* function. The driver can either respond to those requests or pass them down to the underlying driver. How an intermediate driver responds to queries and sets depends on the implementation.

Note The behavior of intermediate drivers can also be affected by the power state of the virtual miniport and the underlying miniport driver. To learn more about the effects of the power state on query and set operations, see Handling a Set Power Request.

The Network Reference section contains information about all of the general, connection-oriented, nonmedia-specific OIDs and about required media-specific OIDs of interest to intermediate driver developers.

The following topics provide additional information about issuing and responding to queries and sets in an intermediate driver:

Issuing Set and Query Requests from an Intermediate Driver

Responding to Sets and Queries in an Intermediate Driver

Issuing Set and Query Requests from an Intermediate Driver

Article • 12/15/2021

The protocol edge of an intermediate driver can issue set and query information requests to the underlying miniport driver. The virtual miniport edge of an intermediate driver can use the information obtained from the underlying driver to determine how to respond to set and query requests.

To cancel an OID request, call the **NdisCancelOidRequest** function.

For more information about responding to set and query requests, see Responding to Sets and Queries in an Intermediate Driver. For more information about issuing OID requests, see OID Request Operations in a Protocol Driver.

Responding to Sets and Queries in an Intermediate Driver

Article • 12/15/2021

Because an NDIS intermediate driver is bound to an overlying NDIS driver, it can also receive queries and sets from its *MiniportOidRequest* function. In some cases, the intermediate driver just passes such requests through to the underlying miniport driver. Otherwise, it can respond to these queries and sets as appropriate to the medium that it exports at its upper edge. Note that an intermediate driver must always pass through any OID_PNP_*Xxx* requests that it receives from an overlying NDIS driver to the underlying miniport driver. NDIS 6.0 intermediate drivers can also cancel OID requests.

To forward a request down to the underlying drivers, an NDIS intermediate driver calls **NdisAllocateCloneOidRequest** to allocate a cloned **NDIS_OID_REQUEST** structure. The driver calls the **NdisOidRequest** function to send the request. When the request is complete, the driver must call the **NdisFreeCloneOidRequest** function to free the NDIS_OID_REQUEST structure.

To cancel an OID request, call the **NdisCancelOidRequest** function.

Typically, the general OIDs that an intermediate driver receives are the same or similar to those that the intermediate driver sends to the underlying miniport driver. The medium-specific OIDs that an intermediate driver receives are the type of the medium that the overlying driver requires.

If an intermediate driver itself processes the setting of an OID rather than passing the set request to an underlying miniport, it should validate the value to be set. If the intermediate driver determines that the value to be set is out of bounds, it should fail the set request.

Note If an intermediate driver modifies the contents of TCP network data that it forwards down to an underlying miniport driver such that TCP offload functions cannot be performed on the network data, the intermediate driver should respond to OID_TCP_OFFLOAD_CURRENT_CONFIG queries with a status of NDIS_STATUS_NOT_SUPPORTED instead of passing the request down to the underlying miniport.

For additional information about responding to sets and queries in an intermediate driver, see Obtaining and Setting Miniport Driver Information and NDIS Support for WMI.

Intermediate Driver Network Data Management

Article • 12/15/2021

An intermediate driver receives **NET_BUFFER_LIST** structures with one or more associated MDLs from a higher-level driver to send over the network. The intermediate driver can pass the data through to the underlying driver by calling **NdisSendNetBufferLists** if the driver has a connectionless lower edge, or by calling **NdisCoSendNetBufferLists** if the driver has a connection-oriented lower edge. Alternatively, the intermediate driver can take some actions to modify either the contents of the chained buffers or the ordering or timing of the incoming data relative to other transmissions.

Depending on the purpose of the intermediate driver, such a driver can repackage buffers that are chained to incoming NET_BUFFER_LIST structures. For example, an intermediate driver repackages network data in the following circumstances:

- The intermediate driver receives a larger data buffer from an overlying protocol driver than can be sent in a single buffer over the underlying medium. Consequently, the intermediate driver must divide the incoming data into smaller buffers.

- The intermediate driver changes the length or content of the network data by compressing or encrypting the data before forwarding each send to the underlying driver.

For information about creating network data management, see Protocol Driver Buffer Management.

NDIS provides interfaces to clone and fragment **NET_BUFFER_LIST** structures. For more information about cloning and fragmenting structures, see Derived NET_BUFFER_LIST Structures.

NET_BUFFER_LIST structures can be allocated as needed, at driver initialization time, or in the *ProtocolBindAdapterEx* function. An intermediate driver developer can, if necessary and for performance reasons, allocate a number of structures at initialization time so that **ProtocolReceiveNetBufferLists** has preallocated resources into which to copy incoming data for indicating to a higher-level driver, and so that *MiniportSendNetBufferLists* has available **NET_BUFFER_LIST** structures (and possibly buffers) to pass incoming send network data on to the next lower driver.

If an intermediate driver copies send data or received data to a new buffer or buffers, and the length of actual data in the last buffer is less than the allocated length of the buffer, the intermediate driver can call **NdisAdjustMdlLength** to adjust the buffer to the actual length of the data.

An intermediate driver with a connectionless lower edge always receives incoming data from an underlying miniport adapter from its ProtocolReceiveNetBufferLists function.

An intermediate driver with a connection-oriented lower edge always receives incoming data from an underlying miniport adapter from its ProtocolCoReceiveNetBufferLists function.

Receiving Data in an Intermediate Driver with a Connectionless Lower Edge

Article • 12/15/2021

An intermediate driver with a connectionless lower edge must have a ProtocolReceiveNetBufferLists function to receive network data.

Underlying connectionless miniport drivers call the **NdisMIndicateReceiveNetBufferLists**, passing a linked list of one or more NET_BUFFER_LIST structures, relinquishing ownership of the indicated structures to higher level drivers. When the higher level drivers have consumed the data, they return the NET_BUFFER_LIST structures (and the resources they specify) to the miniport driver.

For more information about receiving data in an intermediate driver with a connectionless lower edge, see Protocol Driver Send and Receive Operations.

Receiving Data in an Intermediate Driver with a Connection-Oriented Lower Edge

Article • 12/15/2021

If an intermediate driver is layered above a connection-oriented miniport driver, NDIS then calls the intermediate driver's **ProtocolCoReceiveNetBufferLists** function to indicate received data.

An underlying connection-oriented miniport driver indicates network data by calling **NdisMCoIndicateReceiveNetBufferLists**, passing a linked list of one or more **NET_BUFFER_LIST** structures.

For more information about receiving data in an intermediate driver with a connection-oriented lower edge, see Connection-Oriented Operations.

Indicating Receive Network Data to Higher Level Drivers

Article • 12/15/2021

A connectionless intermediate driver indicates receive network data to the next higher driver by calling the NdisMIndicateReceiveNetBufferLists function. A connection-oriented intermediate driver indicates receive network data to the next higher driver by calling the NdisMCoIndicateReceiveNetBufferLists function.

Before indicating the receive network data, the driver processes the data, perhaps converting it to the format expected by a higher-level driver, and if required, copying relevant data into MDLs that are associated with an intermediate-driver-allocated NET_BUFFER structure.

Transmitting Network Data Through an Intermediate Driver

Article • 12/15/2021

As discussed in Registering an Intermediate Driver as a Miniport Driver, an intermediate driver must provide a *MiniportSendNetBufferLists* function when it registers with **NdisMRegisterMiniportDriver**. The *MiniportSendNetBufferLists* function can forward incoming **NET_BUFFER_LIST** structures by calling **NdisSendNetBufferLists** if the driver has a connectionless lower edge . *MiniportSendNetBufferLists* can send the list of NET_BUFFER_LIST structures it receives with **NdisSendNetBufferLists** without regard to the capabilities of the underlying miniport driver.

MiniportSendNetBufferLists receives a list of NET_BUFFER_LIST structures arranged in an order determined by an overlying caller of **NdisSendNetBufferLists**. In most cases, the intermediate driver should maintain this ordering as it passes an incoming array of NET_BUFFER_LIST structures on to the underlying miniport driver. An intermediate driver that modifies data in network data before passing them on to the underlying driver can reorder a list.

NDIS always preserves the ordering of **NET_BUFFER_LIST** structure pointers as passed as a linked list to **NdisSendNetBufferLists**. The underlying miniport driver also assumes that list that is passed in to its *MiniportSendNetBufferLists* function implies the network data should be transmitted in the same order.

Handling PnP Events and Power Management Events in an Intermediate Driver

Article • 12/15/2021

An intermediate driver must be able to handle Plug and Play (PnP) events and power management events. Specifically:

- An intermediate driver must set the NDIS_MINIPORT_ATTRIBUTES_NO_HALT_ON_SUSPEND flag in the **AttributeFlags** member of the **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure that is passed to **NdisMSetMiniportAttributes**. For more information, see Initializing as a Miniport.

- The virtual miniport of an intermediate driver must handle OID_PNP_*Xxx* requests.

- The protocol section of an intermediate driver should propagate appropriate OID_PNP_*Xxx* requests to the underlying miniport drivers. The virtual miniport of the intermediate driver should pass the underlying miniport driver's responses to these requests back to the protocol driver that originated the requests. The intermediate driver does not have to pass requests that are not required by design. For example, when there is not a one-to-one relationship between virtual miniports and underlying miniport adapters as in Load Balancing Failover (LBFO) applications.

- The protocol portion of an intermediate driver must supply a *ProtocolNetPnPEvent* function.

Intermediate driver protocol and miniport event handlers are not called in any particular order. Event handlers for intermediate drivers should be implemented accordingly.

This section includes the following topics:

Initializing Intermediate Drivers to Handle PnP and Power Management Events

Handling OID_PNP_Xxx Queries and Sets

Implementing a ProtocolNetPnPEvent Handler in an Intermediate Driver

Handling a Set Power Request

Initializing Intermediate Drivers to Handle PnP and Power Management Events

Article • 12/15/2021

To handle Plug and Play (PnP) and power management events, NDIS intermediate drivers must do the following:

- When NDIS calls the intermediate driver's *ProtocolBindAdapterEx* function, the *BindParameters* parameter points to an **NDIS_PM_CAPABILITIES** structure that contains the capabilities of the underlying miniport adapter. The power management capabilities are reported in one of the following members:

  - **PowerManagementCapabilities**

    For NDIS 6.0 and NDIS 6.1 intermediate drivers, this member contains the power management capabilities within an NDIS_PNP_CAPABILITIES structure. For more information about this structure, see OID_PNP_CAPABILITIES.

    Note For NDIS 6.20 and later intermediate drivers, the **PowerManagementCapabilities** member is set to **NULL** and the power management capabilities are reported in the **PowerManagementCapabilitiesEx** member.

  - **PowerManagementCapabilitiesEx**

    For NDIS 6.20 and later intermediate drivers, this member contains the power management capabilities within an **NDIS_PM_CAPABILITIES** structure.

    Note For NDIS 6.0 and NDIS 6.1 intermediate drivers, the **PowerManagementCapabilitiesEx** member is set to **NULL** and the power management capabilities are reported in the **PowerManagementCapabilities** member.

Note If the underlying miniport adapter does not support power management events, the **PowerManagementCapabilities** and **PowerManagementCapabilitiesEx** members are set to **NULL**.

- When NDIS calls MiniportInitializeEx for each virtual miniport supported by the NDIS intermediate driver, the driver reports its power management capabilities by calling **NdisMSetMiniportAttributes** in the following ways:

1. Depending on the version of the NDIS intermediate driver, the power management capabilities are reported in either the **PowerManagementCapabilities** member (for NDIS 6.0 and NDIS 6.1 intermediate drivers) or **PowerManagementCapabilitiesEx** member (for NDIS 6.20 and later intermediate drivers) of NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES. If either the **PowerManagementCapabilities** or **PowerManagementCapabilitiesEx** member of the NDIS_BIND_PARAMETERS structure is not **NULL**, the intermediate driver must do the following:

    - Save the original values of the **MinMagicPacketWakeUp**, **MinPatternWakeUp**, and **MinLinkChangeWakeUp** members of the **PowerManagementCapabilities**(NDIS 6.0 and NDIS 6.1) or **PowerManagementCapabilitiesEx**(NDIS 6.20 and later) members.

    - Disable the power management functionality by setting the **MinMagicPacketWakeUp**, **MinPatternWakeUp**, and **MinLinkChangeWakeUp** members to **NdisDeviceStateUnspecified**.

    - Pass the address of the modified NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure as the *MiniportAttributes* parameter in the call to NdisMSetMiniportAttributes.

2. An intermediate driver must set the NDIS_MINIPORT_ATTRIBUTES_NO_HALT_ON_SUSPEND flag in the **AttributeFlags** member of the NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES structure. The driver must pass the address of this structure as the *MiniportAttributes* parameter in the call to NdisMSetMiniportAttributes.

For more information about the initialization requirements of NDIS intermediate drivers, see Initializing Virtual Miniports.

Handling OID_PNP_Xxx Queries and Sets

Article • 12/15/2021

The virtual miniport of an intermediate driver must export the *MiniportOidRequest* function. NDIS calls the intermediate driver's *MiniportOidRequest* function when an overlying driver that is bound to the intermediate driver's virtual miniport calls **NdisOidRequest** to query or set information objects (OID_*Xxx*). NDIS can also call *MiniportOidRequest* on its own behalf. For more information about miniport driver handling of sets and queries to information objects, see Obtaining and Setting Miniport Driver Information and NDIS Support for WMI.

The intermediate driver should retain information about the capabilities of the underlying miniport adapters that it receives in the *ProtocolBindAdapterEx* function. If the miniport adapter is not power management-aware, NDIS sets the **PowerManagementCapabilities** member of **NDIS_BIND_PARAMETERS** to **NULL**.

The intermediate driver can query or set an OID_*Xxx* that is maintained by the underlying miniport driver. It does this with **NdisOidRequest**(if the intermediate driver has a connectionless lower edge), or with **NdisCoOidRequest**(if the intermediate driver has a connection-oriented lower edge).

An intermediate driver should handle queries and sets as follows:

- OID_PNP_CAPABILITIES

  In response to this OID query, intermediate drivers must report the PnP capabilites of the underlying physical miniport adapters. Note that miniport adapters for physical devices do not receive this OID query.

  The intermediate driver receives the PnP capabilities of the underlying miniport adapters in the bind parameters. It should pass them to overlying drivers as appropriate for the intermediate driver's intended use. Intermediate drivers and miniport drivers report PnP capabilities in miniport adapter attributes. The intermediate driver does not issue OID_PNP_CAPABILITIES requests to the underlying miniport driver. If the underlying miniport adapter is power management-aware, in the NDIS_PM_WAKE_UP_CAPABILITIES structure in the virtual miniport attributes, the intermediate driver must specify a device power state of **NdisDeviceStateUnspecified** for each wake-up capability:

  - MinMagicPacketWakeUp
  - MinPatternWakeUp

    - MinLinkChangeWakeUp

  Such a setting indicates that the intermediate driver is power management-aware but cannot wake up the system.

- OID_PNP_QUERY_POWER and OID_PNP_SET_POWER

  The intermediate driver must always return NDIS_STATUS_SUCCESS to a query of OID_PNP_QUERY_POWER or a set of OID_PNP_SET_POWER. The intermediate driver must never propagate either of these OID requests to the underlying miniport driver.

- "Wake-up OIDs"

  If an underlying NIC is power management-aware, the intermediate driver must pass to the underlying miniport driver (by calling **NdisOidRequest** or **NdisCoOidRequest**) the following OID_PNP_*Xxx* that relate to wake-up events:

  OID_PNP_ENABLE_WAKE_UP

  OID_PNP_ADD_WAKE_UP_PATTERN

  OID_PNP_REMOVE_WAKE_UP_PATTERN

  OID_PNP_WAKE_UP_PATTERN_LIST

  OID_PNP_WAKE_UP_ERROR

  OID_PNP_WAKE_UP_OK

The intermediate driver must also propagate the underlying miniport driver's response to these OIDs to the overlying protocol drivers.

If the underlying miniport adapter is not power management-aware, the miniport driver sets the **PowerManagementCapabilities** member of **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** to **NULL** and NDIS sets the **PowerManagementCapabilities** member of **NDIS_BIND_PARAMETERS** to **NULL**.

If an underlying miniport adapter is not power management-aware, the intermediate driver should return NDIS_STATUS_NOT_SUPPORTED in response to a query or set of these OIDs.

Implementing a ProtocolNetPnPEvent Handler in an Intermediate Driver

Article • 12/15/2021

The implementation of a *ProtocolNetPnPEvent* function in intermediate drivers is nearly identical to the implementation in protocol drivers. For more information about implementing a *ProtocolNetPnPEvent* handler in an intermediate driver, see Handling PnP Events and PM Events in a Protocol Driver.

NDIS intermediate drivers pass on PnP events to higher layer drivers by calling the **NdisMNetPnPEvent** function.

Handling a Set Power Request

Article • 12/15/2021

An intermediate driver must handle requests to set power to the working state (a network device power state of D0) and to sleeping states (a network device power state of D1, D2, or D3). The intermediate driver should also maintain power state variables and a StandBy flag. These issues are discussed further in this topic.

For examples of intermediate driver power management, see the NDIS MUX Intermediate Driver and Notify Object driver sample in the Windows driver samples repository on GitHub.

Handling a Set Power Request to a Sleeping State

There are two cases where an intermediate driver must handle a set power request to a sleeping state:

- NDIS requests that the virtual miniport upper edge of the intermediate driver go to a sleeping state.
- The intermediate driver protocol lower edge handles the underlying miniport driver transition to a sleeping state when it receives a Plug and Play (PnP) event notification.

These events can happen in any order and one event does not necessarily accompany the other.

When the virtual miniport upper edge of the intermediate driver receives a request to set power to a sleeping state, the sequence of events for handling the request is as follows:

1. NDIS calls the *ProtocolNetPnPEvent* function of each protocol driver bound to the virtual miniport. The call to *ProtocolNetPnPEvent* specifies a **NetEventSetPower** event for a sleeping state. Protocol drivers that are bound to the intermediate driver stop sending network data and making OID requests to the intermediate driver virtual miniport. The protocol lower edge of the intermediate driver can continue to send network data and requests down until NDIS indicates that the underlying miniport driver is making the transition to a sleeping state.
2. NDIS pauses the overlying drivers and then the virtual miniport after issuing the **NetEventSetPower** event. The specified reason for the pause is a transition to a

low-power state. For more information about pausing a virtual miniport, see Pausing an Adapter.

Note No OID requests can be sent to the virtual miniport while it is in a low-power state, with the exception of OID_PNP_SET_POWER.

3. NDIS issues an OID_PNP_SET_POWER request to the virtual miniport of the intermediate driver. The intermediate driver accepts the request by returning NDIS_STATUS_SUCCESS. The intermediate driver must not propagate the OID_PNP_SET_POWER request to the underlying miniport driver. After the intermediate driver completes this request, it should not indicate any more received network data or indicate status, even if it keeps receiving network data and status indications from the underlying miniport driver.

When the protocol lower edge of the intermediate driver transitions the underlying miniport driver to a sleeping state, the sequence of events for handling the transition is as follows:

1. NDIS calls the *ProtocolNetPnPEvent* function of the intermediate driver protocol lower edge. The call to *ProtocolNetPnPEvent* specifies a **NetEventSetPower** event for a sleeping state. The intermediate driver must stop sending network data and making OID requests to the underlying miniport driver. If there are outstanding requests or sends, the intermediate driver should return NDIS_STATUS_PENDING from the call to *ProtocolNetPnPEvent*. The intermediate driver calls **NdisCompleteNetPnPEvent** to complete the call to *ProtocolNetPnPEvent*. The protocol edge of an intermediate driver can still get received packet and status indications from the underlying miniport driver. Received network data can be ignored. If an intermediate driver's implementation depends upon monitoring the status of the underlying miniport driver, status indications should still be monitored.

2. NDIS pauses the protocol edge of the intermediate driver and then pauses the underying miniport adapter after issuing the **NetEventSetPower** event. The specified reason for the pause is a transition to a low-power state. For more information about pausing a protocol binding, see Pausing a Binding.

   Note No OID requests can be sent to the underlying miniport adapter while it is in a low-power state, with the exception of OID_PNP_SET_POWER.

3. NDIS issues an OID_PNP_SET_POWER request to the underlying miniport driver. However, if the underlying miniport driver does not support power management, it will be halted. In this case, even though NDIS halts the underlying miniport driver, it does not request the intermediate driver protocol to unbind from the underlying

miniport driver and NIC. After the underlying miniport driver has successfully completed processing the OID (or the miniport driver is halted), it will not indicate any more network data or status.

Handling a Set Power Request to the Working State

There are two cases where an intermediate driver handles a set power request to the working state:

- NDIS requests that the virtual miniport upper edge of the intermediate driver go to the working state.
- The intermediate driver protocol lower edge handles the underlying miniport driver transition to the working state, when it receives a Plug and Play (PnP) event notification.

These events can occur in any order and one event does not necessarily accompany the other.

When the virtual miniport upper edge of the intermediate driver receives a request to set power to a working state, the sequence of events for handling the request is as follows:

1. NDIS issues an OID_PNP_SET_POWER to the virtual miniport of the intermediate driver. The intermediate driver returns NDIS_STATUS_SUCCESS to the set power request. The intermediate driver must not propagate the OID_PNP_SET_POWER request to the underlying miniport driver.
2. NDIS restarts the virtual miniport and then restarts the overlying drivers after issuing the set power OID. For more information about restarting a virtual miniport, see Starting an Adapter.
3. NDIS calls the *ProtocolNetPnPEvent* function of the overlying protocol drivers. The call to *ProtocolNetPnPEvent* specifies a **NetEventSetPower** event to set the working state (D0). Bound protocol drivers can start sending network data to the intermediate driver's virtual miniport.

When the protocol lower edge of the intermediate driver transitions the underlying miniport driver to a working state, the sequence of events for handling the transition is as follows:

1. NDIS issues an OID_PNP_SET_POWER to the underlying miniport driver or calls its *MiniportInitializeEx* handler if the underlying miniport driver was halted.

2. NDIS restarts the underlying miniport driver and then the protocol edge of the intermediate NDIS and the underlying miniport adapter after issuing the OID. For more information about pausing a protocol binding, see Restarting a Binding.

3. NDIS calls the *ProtocolNetPnPEvent* function of the intermediate driver. The call to *ProtocolNetPnPEvent* specifies a **NetEventSetPower** event to set the working state (D0). The intermediate driver can start sending network data to the underlying miniport driver.

Power States and the Standby Flag

The intermediate driver should maintain a separate power state variable for each virtual miniport instance and for each underlying miniport driver to which the driver is bound. The intermediate driver should also maintain a StandingBy flag for each virtual miniport that is:

- Set to **TRUE** when the power state of either the virtual miniport or the underlying miniport driver leaves D0.

- Set to **FALSE** when the power state of either the virtual miniport or the underlying miniport driver returns to D0.

Note For MUX intermediate drivers, there can be multiple virtual miniports that are associated with an underlying miniport driver or multiple underlying miniports that are associated with each virtual miniport. When the power state of any miniport adapter changes, the behavior of all of the associated miniports are also affected. How the behavior is affected is implementation-specific. For example, a driver that implements a Load Balancing Failover (LBFO) solution might not deactivate the virtual miniports when a single underlying miniport driver is deactivated. However, a driver implementation that depends on all underlying miniport drivers would require deactivation of virtual miniports when any underlying miniport driver is deactivated.

The intermediate driver should use the StandingBy flag and power state variables when processing requests as follows:

- The driver's *MiniportSendNetBufferLists* function should fail unless the virtual miniport and its underlying miniport adapter are both in D0.

- The driver's *MiniportOidRequest* function should always succeed OID_PNP_QUERY_POWER to ensure that the driver receives the subsequent OID_PNP_SET_POWER requests.

- The driver's *MiniportOidRequest* function should fail if the virtual miniport is not in D0 or if StandingBy is **TRUE**. Otherwise, it should queue a single request if the underlying miniport driver is not in D0. A queued request should be processed when the underlying miniport driver state becomes D0.

- The intermediate driver virtual miniport should report status only if both the underlying miniport driver and virtual miniport are in D0.

Intermediate Driver Reset Operations

Article • 12/15/2021

An intermediate driver must be prepared to handle the situation where its outstanding sends on a binding to an underlying driver can be dropped because the underlying NIC is reset.

An underlying driver typically resets a NIC because NDIS calls the miniport driver's *MiniportResetEx* function when NDIS times out queued sends or requests that are bound for the NIC. If an underlying NIC is reset, NDIS calls the **ProtocolStatusEx**(or **ProtocolCoStatusEx**) function of each bound protocol and intermediate driver with a status of NDIS_STATUS_RESET_START. When the miniport driver completes the reset, NDIS again calls *ProtocolStatusEx*(or *ProtocolCoStatusEx*) with a status of NDIS_STATUS_RESET_END.

When a NIC is reset, if a bound intermediate driver has any transmit network data that is pending to that NIC, NDIS completes those network data back to the intermediate driver with an appropriate status. The intermediate driver must resubmit these network data again when the reset is completed.

When an intermediate driver receives a status of NDIS_STATUS_RESET_START, it should:

- Hold any network data ready to be transmitted until *ProtocolStatusEx* or *ProtocolCoStatusEx* receives an NDIS_STATUS_RESET_END notification.

- Hold any received network data that are ready to be indicated up to the next higher driver until *ProtocolStatusEx*(or *ProtocolCoStatusEx*) receives an NDIS_STATUS_RESET_END notification.

- Clean up any internal state it maintains for in-progress operations and NIC status.

After *ProtocolStatusEx*(or *ProtocolCoStatusEx*) receives NDIS_STATUS_RESET_END, the intermediate driver can resume sending network data, making requests and making indications to higher-level drivers.

An intermediate driver does not provide a *MiniportResetEx* function.

Status Indications in an Intermediate Driver

Article • 12/15/2021

The implementation of status indications in intermediate drivers is nearly identical to the implementation in protocol drivers. For more information about intermediate driver status indications, see Status Indications in a Protocol Driver.

When an intermediate driver receives a status indication, it can indicate the status indication up to the higher-level drivers by calling **NdisMIndicateStatusEx**. An intermediate driver should indicate status changes to overlying drivers as appropriate for its specific design requirements.

Installing an Intermediate Driver

Article • 12/15/2021

Intermediate drivers require two INF files. One of the INF files defines the installation parameters for the protocol lower edge. The other INF file defines the installation parameters for the virtual miniport upper edge.

The protocol INF file is the primary INF file. After the protocol lower edge is installed, the virtual miniport upper edge is installed, based on references to the miniport driver INF file that are defined in the protocol INF file.

On Windows Vista, you can use a notify object or a custom setup application to copy the miniport driver INF file to the system INF directory. For Windows Vista and later operating system versions, you should use the **INF CopyINF directive** in the protocol INF file to copy the miniport driver INF file. For more information about the notify object and copying INF files, see Intermediate Driver Notify Object.

The system-supplied device setup class for the protocol lower edge is **NetService** for filter intermediate drivers and **NetTrans** for MUX intermediate drivers. The driver class for the virtual miniport is always **Net**.

In addition to the INF files, you must also provide a notify object with a MUX intermediate driver. The notify object is optional for filter intermediate drivers.

The virtual miniport device is always removed from the user interface by using the **ExcludeFromSelect** directive. Therefore, the user only sees the protocol and installs the protocol from the protocol INF file.

Note The **ExcludeFromSelect** directive does not remove the virtual device from the **Connections** dialog box. However, the NCF_HIDDEN flag in the miniport driver INF file *DDInstall* section's **Characteristics** entry prevents the virtual miniport from being displayed in any part of the user interface, including the **Connections** dialog box.

This section provides information about intermediate INF files and notify objects. This information is described in the following topics:

Intermediate Driver UpperRange And LowerRange INF File Entries

MUX Intermediate Driver Installation

Intermediate Driver Notify Object

Intermediate Driver UpperRange And LowerRange INF File Entries

Article • 12/15/2021

This topic describes how to use the **UpperRange** and **LowerRange** INF file entries to define NDIS intermediate driver binding relationships.

In a network driver INF file, the **UpperRange** entry lists the possible upper bindings and the **LowerRange** entry lists the possible lower bindings. There are various system-defined values for these lists.

For filter intermediate drivers, you must set the value of the **UpperRange** and **LowerRange** entries to **noupper** and **nolower**, respectively. You should define these entries only in the protocol INF file; they are not required in the miniport driver INF file. The following code example illustrates these entries for a filter intermediate driver.

INF

```
HKR, Ndi\Interfaces, UpperRange, , noupper
HKR, Ndi\Interfaces, LowerRange, , nolower
```

In a filter intermediate driver, the **FilterMediaTypes** entry in the protocol INF file defines the driver's bindings to other drivers. **FilterMediaTypes** specifies the media types serviced by the filter intermediate driver. For a list of possible media types, see the list of Microsoft-supplied **LowerRange** values in Specifying Binding Interfaces. The following code example illustrates this entry for a filter intermediate driver.

INF

```
HKR, Ndi\Interfaces, FilterMediaTypes, , "ethernet, tokenring, fddi, wan"
```

When a filter intermediate driver is initialized, it inserts itself into all existing protocol-to-miniport bindings, as appropriate to the media types listed in **FilterMediaTypes**.

For MUX intermediate drivers, you should always set **UpperRange** in the protocol INF file to **noupper**. Set **LowerRange** to a list of values taken from those values allowed for **LowerRange,** as specified in Specifying Binding Interfaces. The following code example illustrates these entries for a MUX intermediate driver's lower edge.

INF

```
HKR, Ndi\Interfaces, UpperRange, 0, "noupper"
HKR, Ndi\Interfaces, LowerRange, 0, "ndis5"
```

For MUX intermediate drivers, you should always set **LowerRange** in the miniport driver INF file to **nolower**. Set the **UpperRange** to a list of values taken from those values allowed for the **UpperRange**, as specified in Specifying Binding Interfaces. The following code example illustrates these entries for a MUX intermediate driver virtual miniport.

INF

```
HKR, Ndi\Interfaces, UpperRange, 0, "ndis5"
HKR, Ndi\Interfaces, LowerRange, 0, "nolower"
```

MUX Intermediate Driver Installation

Article • 12/15/2021

This topic provides an overview of MUX intermediate driver installation issues. For additional information about the structure of intermediate driver INF files, see Installation Requirements for Network MUX Intermediate Drivers.

A MUX intermediate driver requires two INF files. The protocol INF file defines the installation parameters for the protocol lower edge. The miniport INF file defines the installation parameters for the virtual miniport upper edge. Set the **Class** INF file entry to **Net** in the virtual miniport INF file and **NetTrans** in the protocol INF file. The following code example shows a **Class** entry for the protocol INF file.

INF

```
Class = NetTrans
```

The *DDInstall* section in a MUX intermediate driver INF file must have a **Characteristics** entry. Define the **Characteristics** entry in your protocol INF file as demonstrated in the following code example.

INF

```
Characteristics = 0x80
```

NCF_HAS_UI (0x80) is required to enable custom property pages, which in this case is the notify object

Define the **Characteristics** entry in your miniport INF file as demonstrated in the following code example.

INF

```
Characteristics = 0x21
```

The **Characteristics** value 0x21 indicates the NCF_VIRTUAL (0x1) and NCF_NOT_USER_REMOVABLE (0x20) flags are set. NCF_VIRTUAL specifies that the device is a virtual adapter. NCF_NOT_USER_REMOVABLE is optional and specifies that the user cannot remove the intermediate driver. If you want to hide the virtual miniport from the user (you should not do this if your user must install devices manually) you can define the NCF_HIDDEN (0x8) flag. The NCF_*Xxx* flags are defined in Netcfgx.h. For more information about the **Characteristics** entry and NCF_*Xxx* flags, see DDInstall Section.

The *DDInstall* section of the protocol INF file for a MUX intermediate driver must include an **Addreg** directive for an **Ndi** key. For more information, see Adding Service-Related Values to the Ndi Key and DDInstall.Services Section.

In addition to the INF files, you must also provide a notify object with a MUX Intermediate driver. The notify object is responsible for installation of virtual miniports. Reference the notify object with the **ComponentDll** entry in the protocol INF as follows:

INF

```
HKR, Ndi,            ComponentDll,   , mux.dll
```

The user installs the protocol INF file which defines configuration parameters, copies installation files and also installs the notify object DLL. The user adds virtual miniports through the user interface provided by the notify object. The miniport INF file should define the **ExcludeFromSelect** entry to prevent the user from installing the miniport INF file instead of the protocol INF file.

The protocol name that the driver registers must match the service name.

INF

```
HKR, Ndi, Service, 0, MUXP
```

The **UpperRange** and **LowerRange** INF file entries determine the bindings for a MUX intermediate driver. The protocol INF file must define the protocol edge bindings, as the following code example shows.

INF

```
HKR, Ndi\Interfaces, UpperRange,    0,          "noupper"
HKR, Ndi\Interfaces, LowerRange,    0,          "ndis5"
```

The miniport INF file must define the upper edge bindings, as the following code example shows.

INF

```
HKR, Ndi\Interfaces,    UpperRange, 0,  "ndis5"
HKR, Ndi\Interfaces,    LowerRange, 0,  "nolower"
```

You should replace "ndis5" in the preceding code examples with the protocol bindings required by your driver. For more information about intermediate driver bindings and the **UpperRange/LowerRange** entries, see Intermediate Driver UpperRange And LowerRange INF File Entries.

Intermediate Driver Notify Object

Article • 05/30/2023

An *intermediate driver notify object* is an extension of the network class installer. The network class installer loads and initializes your notify object and sends it notifications of events (such as virtual miniport removal notifications) related to your driver. If you want an overview of notify objects in general or more information about notify objects, see Notify Objects for Network Components.

To include the notify object in your installation, you must reference it in your intermediate driver protocol INF. Filter intermediate drivers do not require a notify object. You can include a notify object with your filter intermediate driver if you would like to provide more flexible configuration options to your user.

On Windows Vista, you can use the notify object or a custom setup application to copy the miniport INF file to the system INF directory. For either of these, you use **SetupCopyOEMInf** to copy the INF. For Windows Vista and later operating system versions, you should use the **INF CopyINF directive** in the protocol INF to copy the miniport INF. On older versions of Windows you can't create a driver package with a notify object that is executed from the Driver Store. To successfully install a driver package in this scenario, you need to have a minimum OS build number of 25341. For more information about copying INF files, see Copying INFs.

A MUX intermediate driver notify object must provide services to install and remove virtual miniports. This can be done automatically or by providing a user interface. It must manage the virtual miniports' device name list in the registry. The device name list defines the bindings between virtual miniports and physical devices. For example, the n-to-one MUX intermediate driver sample notify object maintains a list of virtual miniports bound to each physical device in an **UpperBindings** registry entry. The MUX sample driver reads the **UpperBindings** list and initializes a virtual miniport for each entry.

Your MUX intermediate driver should use the **UpperRange/LowerRange** entries to control external bindings. However, you can control external bindings from your notify object if necessary. For more information about bindings in intermediate drivers, see Intermediate Driver UpperRange And LowerRange INF File Entries

Your notify object can optionally provide a user interface that allows the user to change or view your driver's configuration. The MUX intermediate driver sample includes an example user interface for a notify object.

Connection-Oriented NDIS

Article • 03/14/2023

This section describes connection-oriented NDIS (CoNDIS). Most CoNDIS 6.0 and later driver operations have not changed from their CoNDIS 5.*x* versions. For more information about the differences between CoNDIS 5.x and CoNDIS 6.0, see Porting CoNDIS 5.x Drivers to CoNDIS 6.0.

Unless noted otherwise, CoNDIS drivers provide the same services as connectionless NDIS drivers. You should be familiar with connectionless NDIS drivers before you attempt to write CoNDIS drivers. For more information about connectionless NDIS drivers, see Writing NDIS Miniport Drivers, Writing NDIS Protocol Drivers, and Writing NDIS Intermediate Drivers.

The following sections describe connection-oriented NDIS:

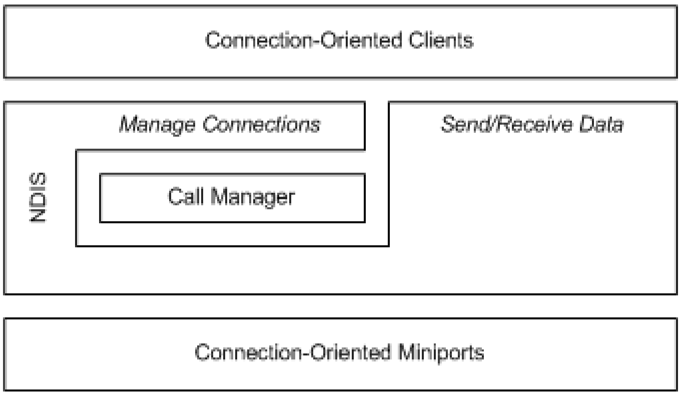
Connection-Oriented Environment

Article • 12/15/2021

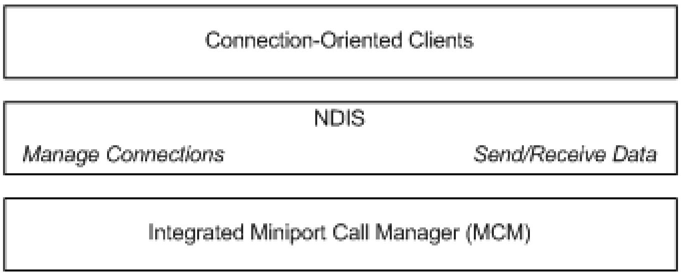
NDIS supports the following connection-oriented drivers:

- Connection-oriented client
- Call manager
- Integrated miniport call manager (MCM) driver
- Connection-oriented miniport driver

The following figure shows a configuration of connection-oriented clients, a call manager, and a miniport driver.



The following figure shows a configuration of connection-oriented clients and an integrated MCM driver.


Connection-Oriented Clients
NDIS
Manage Connections
Send/Receive Data
Integrated Miniport Call Manager (MCM)

A *connection-oriented miniport driver* controls one or more network interface cards (NICs) and provides an interface between connection-oriented protocol drivers (connection-oriented clients and call managers) and the NIC hardware.

For a summary of connection-oriented operations performed by a connection-oriented miniport driver, see Connection-Oriented Operations Performed by Miniport Drivers.

A *call manager* is an NDIS protocol driver that provides call setup and tear-down services for connection-oriented clients. A call manager:

- Uses the send and receive capabilities of a connection-oriented miniport driver to exchange signaling messages with network entities, such as network switches or remote peers.

- Supports one or more signaling protocol drivers. For a summary of connection-oriented operations performed by a call manager, see Connection-Oriented Operations Performed by Call Managers.

An *integrated MCM driver* is a connection-oriented miniport driver that also provides call manager services to connection-oriented clients. An MCM driver has the following characteristics:

- An MCM driver provides the same connection-oriented services to clients as a call manager that is paired with a connection-oriented miniport driver; however, the call manager-to-miniport driver interface is internal to the driver and therefore opaque to NDIS.

- Multiple call managers and MCM drivers can coexist in the same environment.

- Each call manager or MCM driver can support multiple signaling protocol drivers.

For a detailed comparison of MCM drivers and call managers, see How an MCM Driver Differs from a Call Manager.

A *connection-oriented client*:

- Uses the call setup and tear-down services of a call manager or MCM driver.

- Uses the send and receive capabilities of a connection-oriented miniport driver or an MCM driver to send and receive data.

- Can provide its own network and transport-layer services to a higher-layer application at its upper edge.

- Uses the services of a call manager and a connection-oriented miniport driver, or it uses the services of an MCM driver at its upper edge.

- Can be an adaptation layer, that resides between an old protocol and connection-oriented NDIS.

  Such adaptation layers use call management services to establish underlying connections but hide the connection-oriented nature of this interface from the connectionless protocols above it.

Note The definition of a connection-oriented client's upper-edge interface is beyond the scope of the NDIS documentation. If a client serves as an adaptation layer, its upper-edge interface is defined by the protocol that it adapts to connection-oriented NDIS.

For a summary of connection-oriented operations performed by a connection-oriented client, see Connection-Oriented Operations Performed by Clients.

Related topics

NDIS Miniport Drivers

Using AFs, VCs, SAPs, and Parties

Article • 12/15/2021

Connection-oriented drivers create and use entities including address families (AFs), virtual connections (VCs), service access points (SAPs), and parties.

When a connection-oriented driver registers an AF or creates a VC, SAP, or party, it passes a pointer to its local context area for that entity to NDIS. NDIS then returns to the driver (as well as to other appropriate connection-oriented drivers) a handle that represents the newly registered AF or the newly created VC, SAP, or party.

The following topics describe the entities that connection-oriented drivers create and use:

Address Families

Virtual Connections

Service Access Points

Parties

Address Families

Article • 12/15/2021

An *address family* (AF) represents an association between one of the following sets of drivers:

- A connection-oriented client, a call manager, and the underlying connection-oriented miniport driver.
- A connection-oriented client and an MCM driver for a specific signaling protocol.

An address family also specifies a particular signaling protocol.

A call manager or MCM driver advertises its call manager services for a specific signaling protocol by registering the address family with NDIS. NDIS then notifies each client on the binding of the newly registered address family. Before it can use the call manager services provided by a call manager or MCM driver, a connection-oriented client must open the address family with the call manager or MCM driver that advertised it.

For more information about operations on address families, see Registering and Opening an Address Family and Closing an Address Family.

Virtual Connections

Article • 12/15/2021

On a local computer, a *virtual connection (VC)* is an endpoint (or association) that can host a single call between a client, call manager or MCM driver, and a miniport driver. On the network, a VC refers to a connection between two communicating endpoints, such as two connection-oriented clients.

Many VCs can be active on a NIC at the same time, enabling the NIC to simultaneously service many calls. Each connection can be to different endpoints on different computers.

VCs on a network vary in the type of service that they provide to clients. For example, a VC can provide unidirectional or bidirectional service. Quality of service (QoS) parameters for each direction can guarantee specific performance thresholds, such as bandwidth and latency. Depending on the signaling protocol, the QoS for a VC may be negotiable. For more information about NDIS support of QoS, see Quality of Service.

A VC on a network can be a switched VC (SVC) or a permanent VC (PVC):

- An SVC is created as needed for a particular call. For example, a connection-oriented client initiates the creation of a VC for an outgoing call that it is going to make. Similarly, a call manager or MCM driver initiates the creation of a VC for an incoming call that it is going to indicate to a connection-oriented client. The call manager or MCM driver must communicate and sometimes negotiate the parameters for the VC with the remote party.

- A permanent VC is manually created and eventually deleted by an operator using a configuration utility, which is not supplied in NDIS. A client that monitors such manual creation and deletion of PVCs can use the OID_CO_ADD_PVC and OID_CO_DELETE_PVC OIDs to request that a call manager or MCM driver add or delete a PVC to or from its list of configured PVCs. The QoS for a PVC is configured by the operator and is not negotiable over the network.

In NDIS, a VC consists of resources that are allocated by a miniport driver to maintain state information about a VC on a network. These resources could include, but are not limited to, memory buffers, events, and data structures. The miniport driver is requested to create such a context for a VC by a connection-oriented client for an outgoing call or a call manager for an incoming call. For more information about the creation of VCs, see Creating a VC.

Before a created VC can be used for data transmission, it must be activated by a call manager or MCM driver. To activate a VC, a miniport driver or MCM driver sets up resources for the VC and communicates with a NIC as necessary to prepare the NIC to receive or transmit data on the VC. For more information about VC activation, see Activating a VC.

When tearing down a call, a call manager or MCM driver deactivates the VC used for the call.

After a call is torn down, the creator of the VC (a connection-oriented client, call manager, or MCM driver) can either initiate the deletion of the VC or use the VC for another call.

Service Access Points

Article • 12/15/2021

A *service access point* (SAP) identifies the characteristics of incoming calls of interest to a connection-oriented client. By registering a SAP with a call manager or MCM driver, a client indicates that the call manager or MCM driver should notify the client of all incoming calls addressed to that SAP.

A client does not always register a SAP, for example, if it does not handle incoming calls. A client can register multiple SAPs with a call manager or MCM driver.

For more information about SAPs, see Registering a SAP and Deregistering a SAP.

Parties

Article • 12/15/2021

A *party* represents one of possibly many leaves of a *point-to-multipoint connection*. When making an outgoing call, a connection-oriented client can specify a party. This makes the call a multipoint call, with the client acting as the *root* of the call and the remote party as a *leaf*. The client can then request that additional remote parties be added as *leaf nodes* to the call.

For more information about adding parties to a point-to-multipoint call, see Adding a Party to a Multipoint Call. For information about deleting parties from a point-to-multipoint call, see Dropping a Party from a Multipoint Call.

Quality of Service

Article • 12/15/2021

The originator of a call on an SVC can specify *quality of service* (QoS) parameters for the call that specify performance parameters for the call. Depending on the signaling protocol that is being used, a call manager or MCM driver that is setting up an outgoing or incoming call can negotiate the QoS with a network entity such as a network switch or a remote client. If allowed by the signaling protocol, a connection-oriented client might also request a change of QoS when determining whether to accept an incoming call.

The QoS parameters for a call are specified as call parameters in a **CO_CALL_PARAMETERS** structure. CO_CALL_PARAMETERS points to two other structures:

- **CO_CALL_MANAGER_PARAMETERS**, which specifies call manager parameters that a call manager or MCM driver use to set up a call.

- **CO_MEDIA_PARAMETERS**, which specifies media parameters that a miniport driver or MCM driver use to activate a VC.

Both CO_CALL_MANAGER_PARAMETERS and CO_MEDIA_PARAMETERS contain generic parameters (flags) that apply to all drivers that use the parameters. Each of these structures also points to a **CO_SPECIFIC_PARAMETERS** structure that specifies call manager-specific parameters (when pointed to by a CO_CALL_MANAGER_PARAMETERS structure) or media-specific parameters (when pointed to by a CO_MEDIA_PARAMETERS structure).

For more information about QoS operations, see Client-Initiated Request to Change Call Parameters and Incoming Request to Change Call Parameters.

MCM Drivers vs. Call Managers

Article • 12/15/2021

An integrated MCM driver is a connection-oriented miniport driver that also provides call manager services to connection-oriented clients. As such, an MCM driver performs all the connection-oriented functions of both a connection-oriented miniport driver and a call manager. Like all miniport drivers, MCM drivers must use **Ndis*Xxx*** calls to communicate with the underlying NIC hardware.

An MCM driver differs from a call manager in two major ways:

- A call manager is an NDIS connection-oriented *protocol driver* with added call manager functionality. An MCM driver is an NDIS connection-oriented *miniport driver* with added call manager functionality.

- The interface between a call manager and a connection-oriented miniport driver is fully exposed to NDIS--that is, all communication between the call manager and the miniport driver passes through NDIS. Except for the activation and deactivation of client VCs (VCs used for transmitting outgoing or incoming client data), the interface between the call manager part of an MCM driver and the miniport driver part of an MCM driver is opaque to NDIS. The activation and deactivation of client VCs must be accomplished through NDIS because NDIS keeps track of client VCs.

The differences between an MCM driver and a call manager are further described in the following sections:

Differences in Initialization

Differences in Calls to NdisXxx Functions

Differences in Virtual Connections

Differences in Initialization

Article • 12/15/2021

A call manager is an NDIS protocol; therefore, it follows the initialization sequence for a connection-oriented protocol, but with one additional step. In its *ProtocolBindAdapterEx* handler, immediately after completing the initialization steps for a connection-oriented protocol, a call manager must register an address family by calling **NdisCmRegisterAddressFamilyEx**. The call to **NdisCmRegisterAddressFamilyEx**, in which a call manager registers its call manager functions, identifies the protocol as a call manager. The call manager must register an address family for each NIC to which it binds itself.

An MCM driver is a miniport driver; therefore, it follows the initialization sequence for a connection-oriented miniport driver with the addition of the following step: an MCM driver must register an address family by calling **NdisMCmRegisterAddressFamilyEx** in its *MiniportInitializeEx* function, immediately after completing the miniport driver initialization sequence . The call to **NdisMCmRegisterAddressFamilyEx**, in which an MCM driver registers its call manager functions, distinguishes the MCM driver from a regular connection-oriented miniport driver. Although an MCM driver registers its miniport driver handlers only once during initialization by calling **NdisMRegisterMiniportDriver**, it must call **NdisMCmRegisterAddressFamilyEx** once for each NIC that it controls.

Differences in Calls to NdisXxx Functions

Article • 12/15/2021

A call manager calls a different set of call manager functions than an MCM driver. A call manager calls **NdisCm_Xxx_** functions, and an MCM driver calls **NdisMCm_Xxx_** functions.

An MCM driver does not call the **NdisCo_Xxx_** functions that both connection-oriented clients and call managers call. Instead, an MCM driver calls the following comparable **NdisMCm_Xxx_** functions:

- NdisMCmCreateVc instead of NdisCoCreateVc
- NdisMCmDeleteVc instead of NdisCoDeleteVc
- NdisMCmOidRequest instead of NdisCoOidRequest
- NdisMCmOidRequestComplete instead of NdisCoOidRequestComplete

An MCM driver does not require a call that is comparable to NdisCoSendNetBufferLists, because the send interface between the call manager and the miniport driver is internal to an MCM driver and therefore opaque to NDIS.

Differences in Virtual Connections

Article • 12/15/2021

A call manager uses *signaling VCs* to send and receive signaling messages to and from network entities, such as switches. A call manager's signaling VCs are visible to NDIS. The call manager must create, activate, deactivate, and delete all VCs with calls to NDIS. An MCM driver's signaling VCs, however, are opaque to NDIS. An MCM driver does not create, activate, deactivate, and delete signaling VCs with calls to NDIS. Instead, an MCM driver performs such operations internally. An MCM driver must call NDIS to perform operations on VCs that are used to send or receive client data. This is because NDIS must keep track of client VCs.

Because MCM driver is both a call manager and a miniport driver, certain connection-oriented functions are redundant. Specifically, MiniportCoCreateVc and MiniportCoDeleteVc are redundant and are therefore not supplied by an MCM driver. VC operations are handled by:

- An MCM driver's ProtocolCoCreateVc and ProtocolCoDeleteVc functions when a client requests the creation or deletion of a VC.
- NdisMCmCreateVc and NdisMCmDeleteVc when the MCM driver creates or deletes a VC.
- NdisMCmActivateVc and NdisCmDeactivateVc when the MCM driver activates or deactivates a VC.

An MCM driver must supply a MiniportCoOidRequest function for a client to use in querying or setting miniport driver information, and a MiniportCoSendNetBufferLists function to handle send operations from a client.

Connection-Oriented Timing Features

Article • 12/15/2021

Connection-oriented NDIS supports using a NIC's local time for scheduling the transmission of packets and for time-stamping send and receive packets.

Note These connection-oriented timing features are optional. These features are not supported by all CoNDIS NICs.

A connection-oriented protocol driver can call **NdisCoOidRequest** to query the local timing capabilities of a connection-oriented miniport driver or an MCM driver with OID_GEN_CO_GET_TIME_CAPS. In response to such a query, the miniport driver or MCM driver returns information about:

- Whether there is a readable clock on the NIC.
- Whether the NIC derives its time from the network connection.
- The precision of the local clock.
- Whether the NIC can timestamp received packets with its local time.
- Whether the NIC can schedule a send packet for transmission according to its local time.
- Whether the NIC can timestamp transmitted packets with its local time.

To obtain a NIC's local time, a connection-oriented protocol can call **NdisCoOidRequest** to query a connection-oriented miniport driver or MCM driver with OID_GEN_CO_GET_NETCARD_TIME. The connection-oriented miniport driver or MCM driver synchronously returns its local time, which the connection-oriented protocol can then use to schedule the transmission of packets.

Timing information for a send or receive packet is contained in the packet's out-of-band (OOB) data. For more information, see **NET_BUFFER_LIST**.

CoNDIS Miniport Driver Registration

Article • 03/14/2023

CoNDIS miniport drivers initialize like other miniport drivers and also must register additional CoNDIS entry points. For general information about miniport driver initialization, see Initializing a Miniport Driver.

To register CoNDIS entry points for *MiniportXxx* functions, CoNDIS miniport drivers call the **NdisSetOptionalHandlers** function from the *MiniportSetOptions* function. In *MiniportSetOptions*, the miniport driver initializes an **NDIS_MINIPORT_CO_CHARACTERISTICS** structure and passes it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

Miniport call managers (MCMs) also register *ProtocolXxx* functions in *MiniportSetOptions*. For more information about MCM driver registration, see CoNDIS MCM Registration.

For more information about configuring optional miniport driver services, see Configuring Optional Miniport Driver Services.

CoNDIS Client Registration

Article • 03/14/2023

CoNDIS clients initialize like other protocol drivers and also must register additional CoNDIS entry points. For general information about protocol driver initialization, see Initializing a Protocol Driver.

To register CoNDIS entry points for *ProtocolXxx* functions, CoNDIS clients call the **NdisSetOptionalHandlers** function from the *ProtocolSetOptions* function. In *ProtocolSetOptions*, all CoNDIS protocol drivers initialize an **NDIS_PROTOCOL_CO_CHARACTERISTICS** structure and pass it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

To specify entry points for a CoNDIS client, a protocol driver initializes an **NDIS_CO_CLIENT_OPTIONAL_HANDLERS** structure and passes it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

For more information about configuring optional protocol driver services, see Configuring Optional Protocol Driver Services.

CoNDIS Call Manager Registration

Article • 03/14/2023

CoNDIS stand-alone call managers initialize like other protocol drivers and also must register additional CoNDIS entry points. For general information about protocol driver initialization, see Initializing a Protocol Driver.

To register CoNDIS entry points for *ProtocolXxx* functions, call managers call the **NdisSetOptionalHandlers** function from the *ProtocolSetOptions* function. In *ProtocolSetOptions*, all CoNDIS protocol drivers initialize an **NDIS_PROTOCOL_CO_CHARACTERISTICS** structure and pass it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

To specify entry points for a call manager, a protocol driver initializes an **NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS** structure and passes it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

Miniport call managers (MCMs) also register call manager *ProtocolXxx* functions. For more information about MCM driver registration, see CoNDIS MCM Registration.

For more information about configuring optional protocol driver services, see Configuring Optional Protocol Driver Services.

CoNDIS MCM Registration

Article • 03/14/2023

CoNDIS miniport call managers (MCMs) initialize like other miniport drivers and also must register additional CoNDIS entry points. For general information about miniport driver initialization, see Initializing a Miniport Driver.

To register CoNDIS entry points for *MiniportXxx* functions and *ProtocolXxx* functions, CoNDIS MCMs call the **NdisSetOptionalHandlers** function from the *MiniportSetOptions* function. In *MiniportSetOptions*, an MCM initializes an **NDIS_MINIPORT_CO_CHARACTERISTICS** structure and passes it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

To register call manager entry points, MCMs initialize an **NDIS_CO_CALL_MANAGER_OPTIONAL_HANDLERS** structure and pass it at the *OptionalHandlers* parameter of **NdisSetOptionalHandlers**.

For more information about configuring optional miniport driver services, see Configuring Optional Miniport Driver Services.

Connection-Oriented Operations Performed by Clients

Article • 12/15/2021

A connection-oriented client:

- **Opens and closes an address family.**

  On receiving notification from NDIS that a call manager or MCM driver has registered an address family, a connection-oriented driver can open that address family with the call manager or MCM driver. The client can then use the call manager services provided by the call manager or MCM driver. A client releases the association between itself and a call manager or MCM driver by closing the address family.

- **Registers and deregisters SAPs.**

  After opening an address family with a call manager or an MCM driver, a connection-oriented client can register one or more SAPs with the call manager or MCM driver. The call manager or MCM driver will then indicate to the client any incoming calls addressed to the registered SAPs. A client releases an SAP by deregistering the SAP.

- **Adds and deletes PVCs.**

  A connection-oriented client can monitor when an operator manually configures or deconfigures a permanent VC (PVC). In response to such an action, the client can request a call manager or MCM driver to add a PVC to its list of configured PVCs or to delete a PVC from such a list (see OID_CO_ADD_PVC and OID_CO_DELETE_PVC).

- **Makes outgoing calls.**

  Before making an outgoing call, a client must initiate the creation of a VC for the call. A client can then make an outgoing call. To make a point-to-multipoint call, a client specifies a party when making the call.

- **Adds a party to or drops a party from a point-to-multipoint call.**

  A client can add a party to a point-to-multipoint call and delete a party from a point-to-multipoint call. A client can also respond to an incoming request to drop a party from a point-to-multipoint call.

- **Accepts or rejects an incoming call.**

  A client can accept or reject an incoming call that is addressed to a SAP that the client previously registered with a call manager or MCM driver.

- **Negotiates the call parameters for an active VC.**

  Depending on what is allowed by the signaling protocol, a client can negotiate the call parameters for an active VC. A client can request a change in quality of service (QoS) and respond to an incoming request to change the QoS for an active VC. A client can also respond to a request from the remote party to change the QoS for a call.

- **Sends and receives packets.**

  A client can send packets through a connection-oriented miniport driver or MCM driver. A client can also receive packets through a connection-oriented miniport driver or MCM driver.

- **Initiates the deletion of a VC.**

  A client can initiate the deletion of a VC that it created.

- **Initiates the tear-down of a call.**

  A client can initiate the tear-down of a outgoing call that it made or an incoming call that it accepted.

- **Queries or sets information.**

  A client can query or set information maintained by a bound call manager or the call manager portion of an MCM driver. A client can also respond to queries and sets from a bound call manager or MCM driver.

  In addition, a client can query or set information maintained by a bound miniport driver or the miniport driver portion of a bound MCM driver.

- **Inputsminiport driver status indications.**

  A client can input status indicated by a connection-oriented miniport driver or an MCM driver.

Connection-Oriented Operations Performed by Call Managers

Article • 12/15/2021

A call manager performs:

- **Registers and deregisters one or more address families (AFs).**

  A call manager registers one or more address families with NDIS . By registering an address family, a call manager advertises its call manager services (specifically, a signaling protocol) to bound connection-oriented clients. For information about registering entry points with NDIS, see CoNDIS Registration.

- **Registers and deregisters SAPs at the request of a connection-oriented client.**

  A call manager receives a bound connection-oriented client's requests to register SAPs and to deregister SAPs. The call manager sends signaling messages over the network to register or deregister SAPs on behalf of clients.

- **Sets up an outgoing call at the request of a connection-oriented client.**

  When a connection-oriented client makes an outgoing call, the call manager communicates (exchanges signaling messages) with network control devices, as necessary, to make a connection. If the call is accepted by the remote party, the call manager activates the VC that is created for the call.

- **Sets up and indicates an incoming call to a connection-oriented client.**

  A call manager indicates to a bound connection-oriented client all calls that are addressed to a SAP registered by that client. Before indicating the incoming call to the client, the call manager initiates creation of a VC for the call then initiates activation of the VC.

- **Communicates requests for a change in QoS.**

  Depending on the signaling protocol, the call manager can communicate a request from the local client to change the QoS for an outgoing or incoming call or a request from the remote party to change the QoS for a call.

- **Communicates requests to add and drop parties.**

  A call manager communicates a local client's request to add a party to or drop a party from a point-to-multipoint call. A call manager also communicates a remote party's incoming request to drop itself from a point-to-multipoint call.

- **Tears down a call.**

  At the request of a connection-oriented client, a call manager closes a call by communicating with network control devices to terminate a connection. At the request of a remote party, a call manager indicates to a local connection-oriented client a remote party's request to close a call. In the process of tearing down a call, the call manager deactivates the VC that is used for the call. If the call manager created the VC (for an incoming call), the call manager can also delete the VC.

- **Queries or sets information.**

  A call manager can query or set information maintained by a bound connection-oriented client. A call manager can also respond to query and set operations from a bound connection-oriented client.

  In addition, a call manager can query or set information maintained by a bound miniport driver or by the miniport driver portion of a bound MCM driver.

- **Inputsminiport driver status indications.**

  A call manager inputs status indications from a bound connection-oriented miniport driver.

Connection-Oriented Operations Performed by Miniport Drivers

Article • 12/15/2021

In addition to controlling NIC hardware, a connection-oriented miniport driver:

- **Sends and receives packets.**

  A connection-oriented miniport driver sends and receives packets on behalf of connection-oriented clients or call managers.

- **Creates (sets up) VCs.**

  At the request of a connection-oriented client, a connection-oriented miniport driver allocates and initializes the resources for a VC for an outgoing call. At the request of a call manager, a connection-oriented miniport driver allocates and initializes the resources for a VC for an incoming call or on which the call manager will send or receive signaling messages.

- **Activates VCs.**

  At the request of a call manager, a connection-oriented miniport driver communicates with a NIC to prepare the NIC to receive or transmit data across a VC (see Activating a VC).

- **Deactivates VCs.**

  At the request of a call manager, a connection-oriented miniport driver communicates with a NIC to terminate all communication across a VC (see Deactivating a VC).

- **Deletes VCs.**

  At the request of a connection-oriented client, a connection-oriented miniport driver deallocates the resources for a VC whose creation was initiated by that client (see Deleting a VC). At the request of a call manager, a connection-oriented miniport driver deallocates the resources for a VC whose creation was initiated by that call manager.

- **Responds to information queries or sets.**

  A connection-oriented miniport driver responds to query and set operations by a bound connection-oriented client or call manager.

- **Indicates status.**

  A connection-oriented miniport driver can indicate changes in its status or the status of a NIC to bound connection-oriented clients and call managers.

- **Resets the NIC.**

  At the request of NDIS, a connection-oriented miniport driver resets a NIC.

Operations on Address Families and SAPs

Article • 12/15/2021

A call manager or MCM driver must register its call manager entry points with NDIS and advertise its call manager services to connection-oriented clients. For more information about registering entry points with NDIS, see CoNDIS Registration.

To use the call manager services of a call manager or MCM driver, a connection-oriented client must open an address family with that call manager or MCM driver. To receive incoming calls, the client must also register one or more SAPs with the call manager or MCM driver.

The following connection-oriented operations pertain to address families and SAPs:

Registering and Opening an Address Family

Registering a SAP

Deregistering a SAP

Closing an Address Family

Registering and Opening an Address Family

Article • 12/15/2021

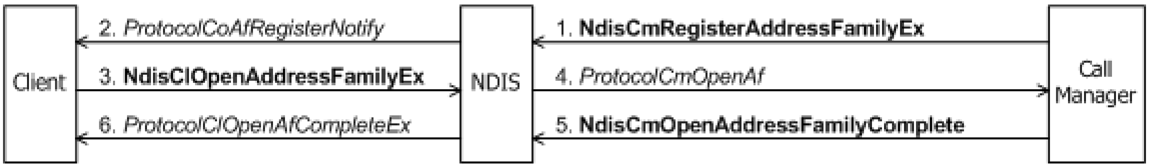
A call manager must register an address family for each NIC on which it provides call manager services to connection-oriented clients. Similarly, an MCM driver must register an address family for the NIC that it manages.

By registering an address family, a call manager or MCM driver causes NDIS to advertise the call manager's or MCM driver's services to all connection-oriented clients that bind to the adapter.

If a connection-oriented client can use the services advertised by a call manager or MCM driver, it can open an address family with the call manager or MCM driver.

Registering an Address Family from a Call Manager

After its *ProtocolBindAdapterEx* function binds to an underlying miniport driver with **NdisOpenAdapterEx**, a call manager calls **NdisCmRegisterAddressFamilyEx** to register an address family for the binding (see the following figure).

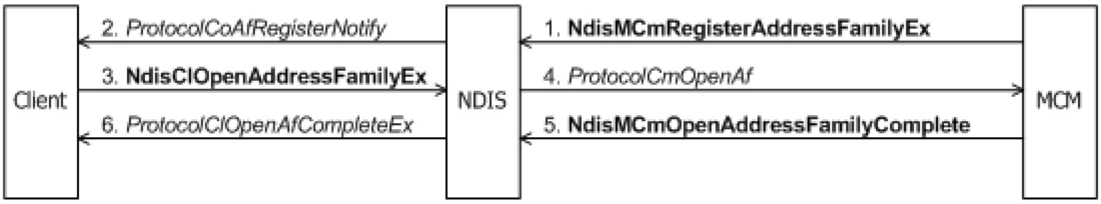

Client
2. ProtocolCoAfRegisterNotify
3. NdisClOpenAddressFamilyEx
6. ProtocolClOpenAfCompleteEx
NDIS
1. NdisCmRegisterAddressFamilyEx
4. ProtocolCmOpenAf
5. NdisCmOpenAddressFamilyComplete
Call Manager

The call to **NdisCmRegisterAddressFamilyEx** advertises the call manager's specific signaling services. A call manager must register an address family each time that its *ProtocolBindAdapterEx* function and is called and successfully binds to a NIC with **NdisOpenAdapterEx**.

The call manager can support more than one address family across all the miniport drivers to which it is bound. The call manager can also support more than one address family on a single NIC to which it is bound. The call manager must register the same entry points for each address family on the binding. Only one call manager can support a particular type of address family for clients bound to any particular miniport driver. For more information about registering entry points for a call manager, see CoNDIS Registration.

Registering an Address Family from an MCM Driver

An MCM driver calls **NdisMCmRegisterAddressFamilyEx** from its *MiniportInitializeEx* function after registering its miniport driver entry points with **NdisMRegisterMiniportDriver**. For more information about regsitering entry points see, CoNDIS Registration. An MCM driver calls **NdisMCmRegisterAddressFamilyEx** once to advertise its services to connection-oriented clients (see the following figure).


Client
2. ProtocolCoAfRegisterNotify
3. NdisClOpenAddressFamilyEx
6. ProtocolClOpenAfCompleteEx
NDIS
1. NdisMCmRegisterAddressFamilyEx
4. ProtocolCmOpenAf
5. NdisMCmOpenAddressFamilyComplete
MCM

A miniport driver of a NIC that has on-board connection-oriented signaling support can register itself as an MCM driver even though a call manager may be available. By doing so, such an MCM driver preempts the call manager as the call manager for that NIC.

Opening an Address Family

A call manager's or MCM driver's call to **Ndis(M)CmRegisterAddressFamily** causes NDIS to call the **ProtocolCoAfRegisterNotify** function of each connection-oriented client on the binding (as shown in two previous figures).

ProtocolCoAfRegisterNotify examines the address-family data to determine whether the client can use the services of this particular CM or MCM driver. Whether the client can make modifications in the (M)CM-supplied address-family data depends on the particular signaling-protocol support of the call manager or MCM driver.

If the client finds the offered call-management services acceptable, *ProtocolCoAfRegisterNotify* allocates a per-AF context area for the client and calls **NdisClOpenAddressFamilyEx**. **NdisClOpenAddressFamilyEx** does not register the client's connection-oriented entry points with NDIS. For more information about registering connection-oriented entry points with NDIS, see CoNDIS Registration.

The call to **NdisClOpenAddressFamilyEx** causes NDIS to call the call manager's or MCM driver's **ProtocolCmOpenAf** function (as shown already in the two earlier figures). *ProtocolCmOpenAf* ensures that the client passed in a valid address family and allocates and initializes the resources necessary to perform operations on behalf of the client that is opening this instance of the address family. *ProtocolCmOpenAf* also stores an NDIS-supplied *NdisAfHandle* that represents the association between the call manager and client for the open address family.

ProtocolCmOpenAf can complete synchronously or asynchronously. To complete asynchronously, the *ProtocolCmOpenAf* function of a call manager calls **NdisCmOpenAddressFamilyComplete**; the *ProtocolCmOpenAf* function of an MCM driver calls **NdisMCmOpenAddressFamilyComplete**. The call to **Ndis(M)CmOpenAddressFamilyComplete** causes NDIS to call the *ProtocolOpenAfComplete* function of the client that originally called **NdisClOpenAddressFamilyEx**.

If the client's call to **NdisClOpenAddressFamilyEx** is successful, NDIS returns to the client an *NdisAfHandle* that represents the association between the call manager and client for the open address family.

If a client accepts incoming calls, it usually registers one or more SAPs from its *ProtocolClOpenAfCompleteEx* function by calling **NdisClRegisterSap** following its successful call to **NdisClOpenAddressFamilyEx**.

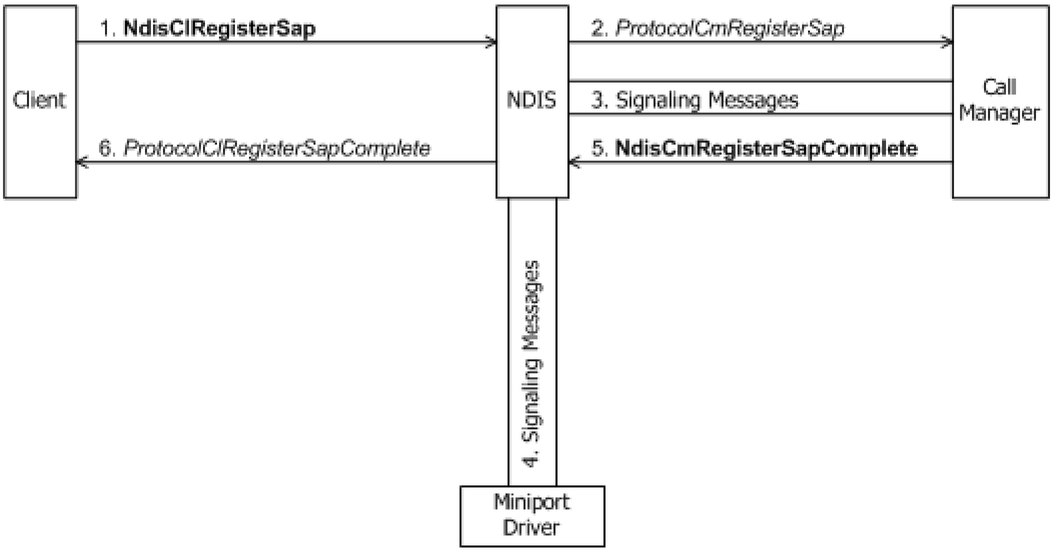
If a client makes outgoing calls, it could create one or more VCs in its *ProtocolClOpenAfCompleteEx* function in anticipation of a request by one or more its clients to make an outgoing call.

Registering a SAP

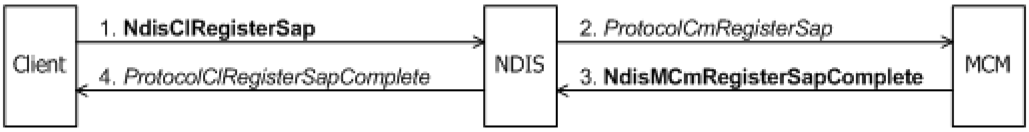
Article • 12/15/2021

If a client accepts incoming calls, its ProtocolClOpenAfCompleteEx function usually registers one or more SAPs with the call manager by calling NdisClRegisterSap.

The following figure shows a client of a call manager registering a SAP.



The following figure shows a client of an MCM driver registering a SAP.


Client
1. NdisClRegisterSap
NDIS
2. ProtocolCmRegisterSap
MCM
4. ProtocolClRegisterSapComplete
3. NdisMCmRegisterSapComplete

With the call to **NdisClRegisterSap**, a client requests notifications of incoming calls on a particular SAP. NDIS forwards the SAP information supplied by the client to the call manager's or MCM driver's ProtocolCmRegisterSap function for validation. If the given SAP is already in use or if the call manager or MCM driver does not recognize the client-supplied SAP specification, the call manager or MCM driver fails this request.

In *ProtocolCmRegisterSap*, the call manager or MCM driver might communicate with network control devices or other media-specific agents to register the SAP on the network for a connection-oriented client. *ProtocolCmRegisterSap* also stores an NDIS-supplied *NdisSapHandle* that represents the SAP.

ProtocolCmRegisterSap can complete synchronously or asynchronously. To complete asynchronously, the *ProtocolCmRegisterSap* function of a call manager calls NdisCmRegisterSapComplete. The *ProtocolCmRegisterSap* function of an MCM driver calls NdisMCmRegisterSapComplete. The call to **Ndis(M)CmRegisterSapComplete** causes NDIS to call the client's *ProtocolClRegisterSapComplete* function.

If the client's call to **NdisClRegisterSap** is successful, NDIS returns to the client an NdisSapHandle that represents the SAP.

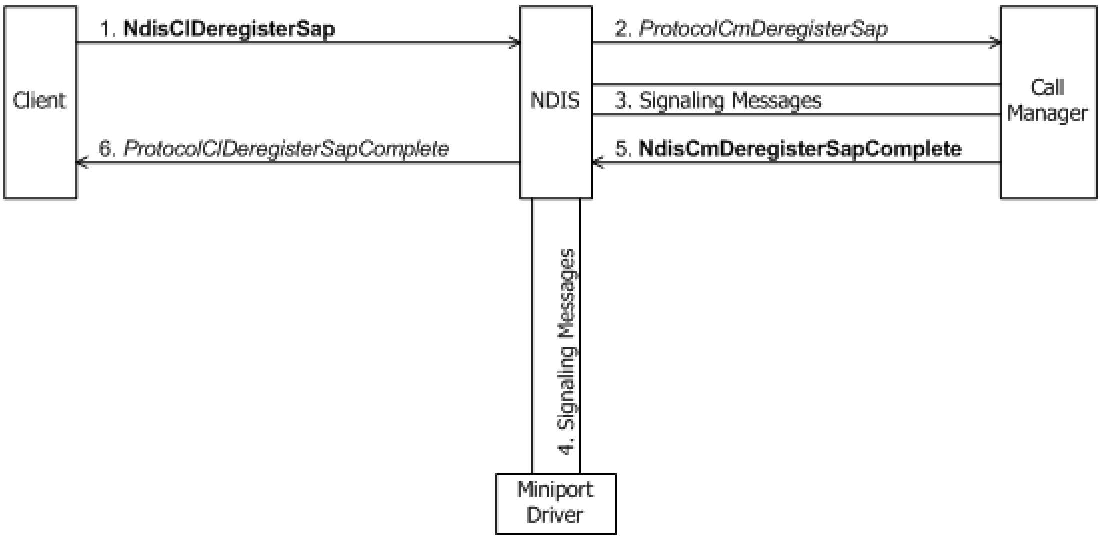
After a call manager registers a SAP on behalf of a connection-oriented client, it notifies that client of an incoming call offer directed to that SAP by calling **NdisCmDispatchIncomingCall**. An MCM driver calls **NdisMCmDispatchIncomingCall**(see Indicating an Incoming Call). A client can receive incoming calls on a SAP even while SAP registration is still pending; that is, before its *ProtocolClRegisterSapComplete* function is called.

Deregistering a SAP

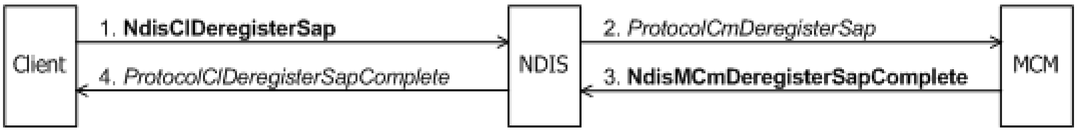
Article • 12/15/2021

A connection-oriented client deregisters a SAP with NdisClDeregisterSap.

The following figure shows a client of a call manager deregistering a SAP.



The following figure shows a client of an MCM driver deregistering a SAP.


Client
1. NdisClDeregisterSap
4. ProtocolClDeregisterSapComplete
NDIS
2. ProtocolCmDeregisterSap
3. NdisMCmDeregisterSapComplete
MCM

The call to **NdisClDeregisterSap** causes NDIS to call the call manager's or MCM driver's ProtocolCmDeregisterSap function. In *ProtocolCmDeregisterSap*, the call manager or MCM driver might communicate with network control devices or other media-specific agents to deregister the SAP on the network. In addition, *ProtocolCmDeregisterSap* must free any resources that it dynamically allocated for the SAP.

ProtocolCmDeregisterSap can complete synchronously or asynchronously. To complete asynchronously, the *ProtocolCmDeregisterSap* function of a call manager calls NdisCmDeregisterSapComplete. The *ProtocolCmDeregisterSap* function of an MCM driver calls NdisMCmDeregisterSapComplete. **Ndis(M)CmDegisterSapComplete** notifies both NDIS and the client that the call manager has completed the SAP-deregistration request for which its *ProtocolCmDeregisterSap* function previously returned NDIS_STATUS_PENDING.

A call to **Ndis(M)CmDeregisterSapComplete** causes NDIS to call the client's ProtocolClDeregisterSapComplete function. A call to *ProtocolClDeregisterSapComplete* indicates that the client's preceding call to **NdisClDeregisterSap** has been processed by the call manager or MCM driver.

Note that a client can deregister a SAP without affecting an incoming call that has already been received on that SAP and without affecting the VC for that incoming call.

Closing an Address Family Overview

Article • 12/15/2021

A connection-oriented client calls **NdisClCloseAddressFamily** to delete the association between itself, a call manager, and a particular underlying NIC.

When a CoNDIS stand-alone call manager is closing a binding to an underlying miniport adapter, or a miniport call manager (MCM) is halting a miniport adapter, the call manager or the MCM must notify NDIS if an associated address family (AF) should be closed. NDIS then notifies each CoNDIS client that has the AF open that the client should close the AF.

This section includes the following topics:

Closing a CoNDIS Call Manager or MCM

Closing an Address Family in a CoNDIS Client

Closing a CoNDIS Call Manager or MCM

Article • 03/14/2023

When a stand-alone call manager is unbinding from an underlying miniport adapter, the call manager must notify all of the affected CoNDIS clients that they must close the associated AF. To notify each client, NDIS stand-alone call managers call the **NdisCmNotifyCloseAddressFamily** function.

If a CoNDIS miniport adapter that an MCM manages is halting, the MCM must notify all of the affected clients that they must close the associated AF. To notify each client, the MCMs call the **NdisMCmNotifyCloseAddressFamily** function.

If a stand-alone call manager or MCM calls **NdisCmNotifyCloseAddressFamily** or **NdisMCmNotifyCloseAddressFamily**, respectively, NDIS calls the **ProtocolClNotifyCloseAf** function of the CoNDIS client that is associated with the handle in the *NdisAfHandle* parameter of **NdisCmNotifyCloseAddressFamily** or **NdisMCmNotifyCloseAddressFamily**. This call notifies the client to close the AF. If **NdisCmNotifyCloseAddressFamily** or **NdisMCmNotifyCloseAddressFamily** returns NDIS_STATUS_PENDING, NDIS will call the call manager's **ProtocolCmNotifyCloseAfComplete** function when the close notification operation is complete.

For more information about closing an address family in a CoNDIS client, see Closing an Address Family in a CoNDIS Client.

Closing an Address Family in a CoNDIS Client

Article • 03/14/2023

To close AFs, a CoNDIS client must provide a ProtocolClNotifyCloseAf function. NDIS calls *ProtocolClNotifyCloseAf* when a stand-alone call manager or MCM calls the NdisCmNotifyCloseAddressFamily function or the NdisMCmNotifyCloseAddressFamily function, respectively.

From within *ProtocolClNotifyCloseAf*, the client finishes closing the specified AF, or it returns NDIS_STATUS_PENDING and calls the NdisClNotifyCloseAddressFamilyComplete function to complete the operation. After the client calls **NdisClNotifyCloseAddressFamilyComplete**, NDIS calls the ProtocolCmNotifyCloseAfComplete function to notify the call manager that the client closed the AF.

To close the AF, the client should:

1. If the client has active multipoint connections, call the NdisClDropParty function as many times as necessary until only a single party remains active on each multipoint virtual connection (VC).

2. Call the NdisClCloseCall function as many times as necessary to close all of the calls that are still open and are associated with the address family.

3. Call the NdisClDeregisterSap function as many times as necessary to deregister all of the service access points (SAPs) that the client registered with the call manager.

4. Call the NdisClCloseAddressFamily function to close the AF.

Creating a VC

Article • 12/15/2021

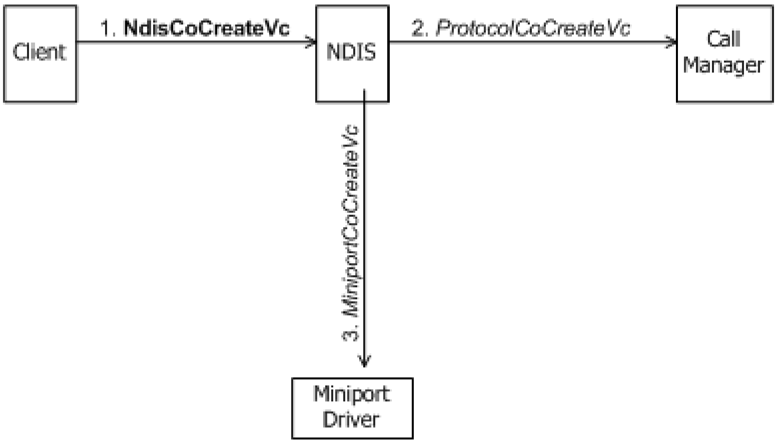
Before making an outgoing call, a connection-oriented client initiates the creation a virtual connection (VC). Before indicating an incoming call to a connection-oriented client, a call manager or an MCM driver initiates the creation of a VC . After the VC has been set up and activated, client data can be transmitted or received on the VC.

A call manager or an MCM driver can also initiate the creation of a VC on which signaling messages are exchanged with network components, such as a network switch.

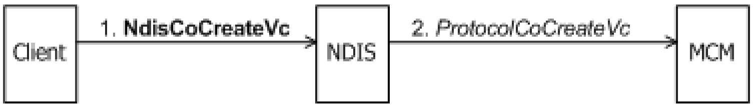
Client-Initiated Creation of a VC

Before making a call with **NdisClMakeCall**, a connection-oriented client calls **NdisCoCreateVc** to initiate the creation of a VC.

The following figure shows a client of a call manager initiating the creation of a VC.



The following figure shows a client of an MCM driver initiating the creation of a VC.



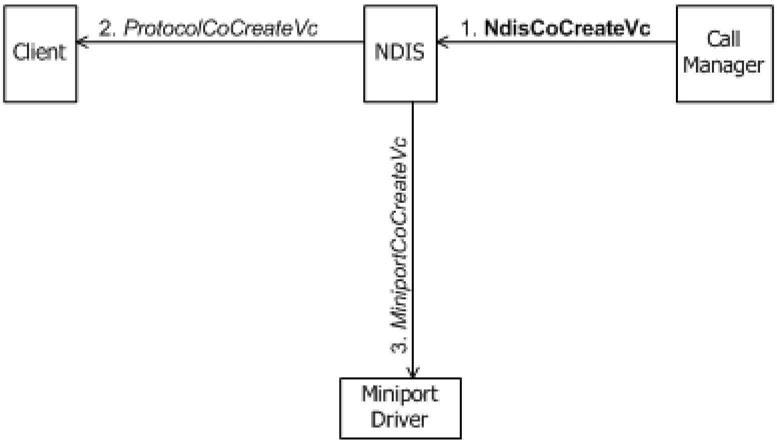
When a connection-oriented client of a call manager calls **NdisCoCreateVc**, NDIS calls, as a synchronous operation, the **ProtocolCoCreateVc** function of the call manager and the **MiniportCoCreateVc** function of the underlying miniport driver (see the first figure in this topic). NDIS passes an *NdisVcHandle* that represents the VC to both *ProtocolCoCreateVc* and *MiniportCoCreateVc*. If the call to **NdisCoCreateVc** is successful, NDIS returns the *NdisVcHandle* to **NdisCoCreateVc**.

ProtocolCoCreateVc allocates and initializes any dynamic resources and structures that the call manager requires to perform subsequent operations on a VC that will be activated. *MiniportCoCreateVc* allocates and initializes any resources that the miniport driver requires to maintain state information about the VC. Both *ProtocolCoCreateVc* and *MiniportCoCreateVc* store the *NdisVcHandle* .

When a connection-oriented client of an MCM driver, the call to **NdisCoCreateVc** causes NDIS to call the MCM driver's *ProtocolCoCreateVc* function (see Client-Initiated Creation of a VC (MCM Driver Present)). In this case, *ProtocolCoCreateVc* performs the necessary allocation and initialization of resources for the VC. There is no call (internal or otherwise) to *MiniportCoCreateVc*, because an MCM driver does not supply such a function.

Call Manager-Initiated Creation of a VC

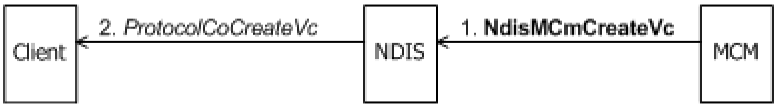
Before indicating an incoming call to a connection-oriented client with **NdisCmDispatchIncomingCall**, a call manager calls **NdisCoCreateVc** to initiate the creation of a VC (see the following figure).



When a call manager calls **NdisCoCreateVc**, NDIS calls, as a synchronous operation, the **ProtocolCoCreateVc** function of the connection-oriented client that registered the SAP on which the call is being received, as well as the **MiniportCoCreateVc** function of the underlying miniport. NDIS passes an *NdisVcHandle* that represents the VC to both *ProtocolCoCreateVc* and *MiniportCoCreateVc*. If the call to **NdisCoCreateVc** is successful, NDIS returns the *NdisVcHandle* to **NdisCoCreateVc**.

MCM Driver-Initiated Creation of a VC

Before indicating an incoming call to a connection-oriented client with **NdisMCmDispatchIncomingCall**, an MCM driver calls **NdisMCmCreateVc** to initiate the creation of a VC (see the following figure).



When an MCM driver calls **NdisMCmCreateVc**, NDIS calls, as a synchronous operation before **NdisMCmCreateVc** returns, the **ProtocolCoCreateVc** function of the connection-oriented client that registered the SAP on which the call is being received. NDIS passes an *NdisVcHandle* that represents the VC to *ProtocolCoCreateVc*. If the call to **NdisMCmCreateVc** is successful, NDIS returns the *NdisVcHandle* to **NdisMCmCreateVc**.

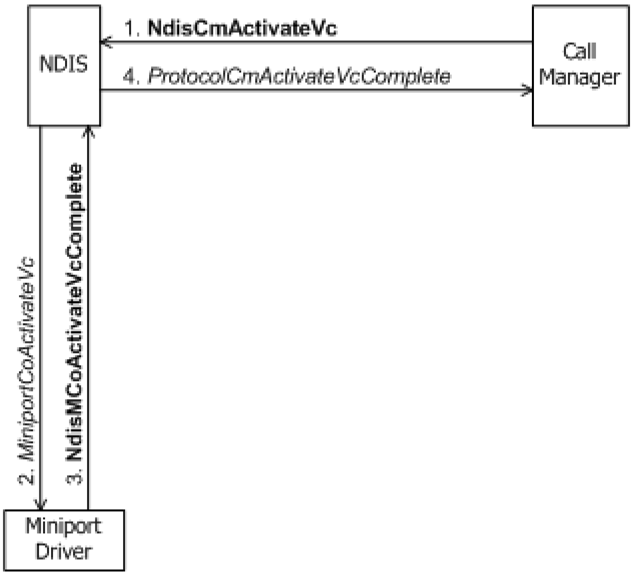
ProtocolCoCreateVc allocates and initializes any dynamic resources and structures that the client requires to perform subsequent operations on the VC. *ProtocolCoCreateVc* also stores the *NdisVcHandle* .

Note that when an MCM driver creates a VC for exchanging signaling messages with a network component, it does not use **Ndis*Xxx*** calls to create a VC. In fact, an MCM driver does not use **Ndis*Xxx*** calls to create, activate, deactivate, or delete such VCs. Instead, an MCM driver performs these operations internally. Such VCs are therefore opaque to NDIS.

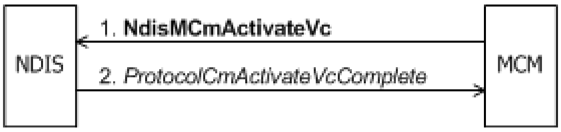
Activating a VC

Article • 12/15/2021

After a virtual connection (VC) has been created (see Creating a VC), it must be activated before data can be transmitted or received on it. A call manager initiates the activation of a VC by calling **NdisCmActivateVc**(see the following figure).



An MCM driver initiates the activation of a VC by calling **NdisMCmActivateVc**(see the following figure).



A call manager or MCM driver could initiate reactivation of an active VC if the local client or a remote party successfully negotiates a change in call parameters on that VC (see Client-Initiated Request to Close a Call and Incoming Request to Change Call Parameters). The call manager or MCM driver can call **Ndis(M)CmActivateVc** many times for a single VC to change the call parameters for an already active call.

For a client-initiated outgoing call, a call manager or an MCM driver usually calls **Ndis(M)CmActivateVc** immediately following the packet exchange confirming a negotiated agreement with the remote target of the call or successful call-setup at the switch. The call manager or MCM driver calls **Ndis(M)CmActivateVc** before it notifies NDIS (and the client) of outgoing call completion with **Ndis(M)CmMakeCallComplete**(see Making a Call). For an incoming call, a call manager or MCM driver usually calls **Ndis(M)CmActivateVc** after it has called **NdisCo(MCm)CreateVc** successfully and before it calls **Ndis(M)CmDispatchIncomingCall**(see Indicating an Incoming Call).

A call manager's call to **NdisCmActivateVc** causes NDIS to call the MiniportCoActivateVc function of the underlying miniport driver. *MiniportCoActivateVc* must validate the call parameters for this VC to verify that the adapter can support the requested call. If the call parameters are acceptable, *MiniportCoActivateVc* communicates with its adapter as necessary to prepare the adapter to receive or transmit data across the virtual connection (for example, programming receive buffers). If the requested call parameters cannot be supported, the miniport driver fails the request.

MiniportCoActivateVc can complete synchronously or asynchronously. The call to NdisMCoActivateVcComplete causes NDIS to call the call manager's ProtocolCmActivateVcComplete function. *ProtocolCmActivateVcComplete* must check the status returned by **NdisMCoActivateVcComplete** to ensure that the virtual connection has been activated successfully. If the miniport driver did not successfully activate the VC, the call manager must not attempt to communicate over the VC. *ProtocolCmActivateVcComplete* must also complete any processing required by the network media to ensure that the virtual connection is ready for data transmission before returning control to NDIS.

An MCM driver's call to **NdisMCmActivateVc** informs NDIS that it has set up call and media parameters on a newly created VC or changed the call parameters on an established VC. This action notifies NDIS that the MCM driver has made a NIC ready for transfers on the VC. NDIS completes the activation sequence by calling the MCM driver's *ProtocolCmActivateVcComplete* function.

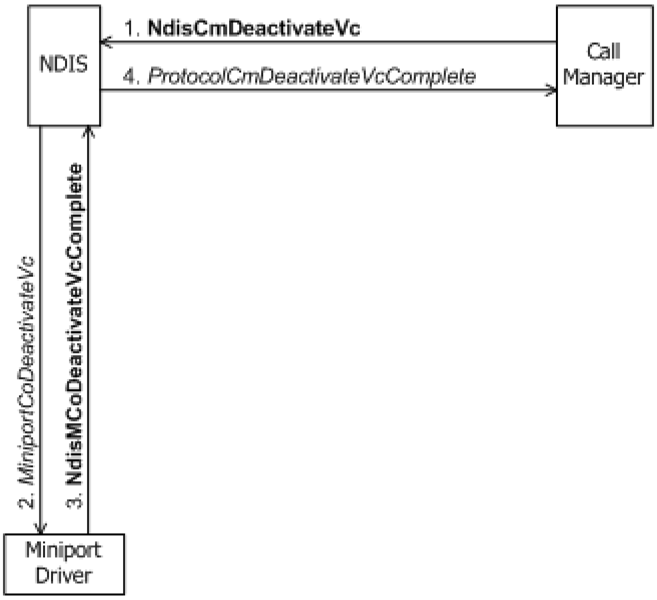
An MCM driver calls **NdisMCmActivateVc** to activate only VCs used for transmitting and/or receiving client data, but not to activate VCs used for exchanging signaling messages between the MCM driver and network components such as a switch. An MCM driver activates a signaling VC internally without calling any **Ndis*Xxx*** function. Any VC that an MCM driver sets up for its own signaling purposes is therefore opaque to NDIS.

Deactivating a VC

Article • 12/15/2021

A call manager calls **NdisCmDeactivateVc** as an essential step in closing either an outgoing or incoming call, typically after the packet exchange with network components that tears down the call (see Client-Initiated Request to Close a Call and Incoming Request to Close a Call). An MCM driver does the same thing by calling **NdisMCmDeactivateVc**.

The call to **NdisCmDeactivateVc** causes NDIS to call the underlying miniport driver's **MiniportCoDeactivateVc** function (see the following figure). *MiniportCoDeactivateVc* communicates with its network adapter to terminate all communication across this VC (for example, clearing receive or send buffers on the adapter).

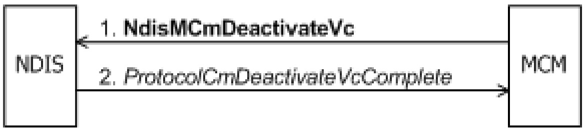

NDIS
1. NdisCmDeactivateVc
4. ProtocolCmDeactivateVcComplete
Call Manager
2. MiniportCoDeactivateVc
3. NdisMCoDeactivateVcComplete
Miniport Driver

Before it deactivates a VC, the miniport driver must complete any pending transfers on the VC. That is, the miniport driver must wait until it has completed all sends in progress and until all receive packets that it has indicated are returned to it. After deactivating the VC, the miniport driver cannot indicate receives or transmit sends on the VC.

Note that *MiniportCoDeactivateVc* does not delete the VC. The creator (client, call manager, or MCM driver) of a particular VC that will not be reused calls **NdisCoDeleteVc** to destroy that VC. A deactivated VC can be reactivated by a connection-oriented client, a call manager, or an MCM driver.

MiniportCoDeactivateVc can complete synchronously or asynchronously. A call to **NdisMCoDeactivateVcComplete**. causes NDIS to call the **ProtocolCmDeactivateVcComplete** function of the call manager that originally requested the VC deactivation. Completion of the deactivation means that all call parameters for the VC used on activation are no longer valid. Any further use of the VC is prohibited except to reactivate it with a new set of call parameters.

An MCM driver's call to **NdisMCmDeactivateVc** informs NDIS that it has deactivated a VC or changed the call parameters on an established VC (see the following figure). NDIS completes the deactivation sequence by calling the MCM driver's *ProtocolCmDeactivateVcComplete* function.



An MCM driver does not call **NdisMCmDeactivateVc** to deactivate VCs used for exchanging signaling messages between the MCM driver and network components such as a switch. An MCM driver deactivates a signaling VC internally without calling any **Ndis*Xxx*** function.

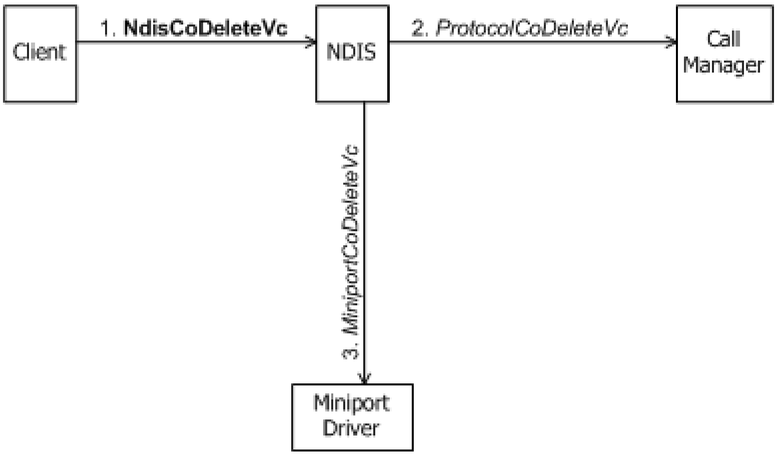
Deleting a VC

Article • 12/15/2021

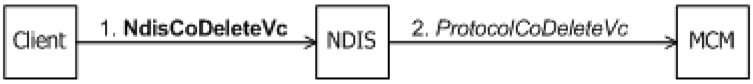
Only the connection-oriented client, call manager, or MCM driver that initiated the creation of a virtual circuit (VC) can initiate the deletion of that VC. A client therefore deletes a VC that it previously created for an outgoing call, a call manager or MCM driver deletes a VC that it previously created for an incoming call over the network, and a call manager deletes a VC that it previously created for exchanging signaling messages over the network. (An MCM driver does not call NDIS to delete a VC that it created for exchanging signaling messages. The MCM driver deletes such a VC with an internal operation that is opaque to NDIS.)

A connection-oriented client or call manager initiates the deletion of a VC with NdisCoDeleteVc.

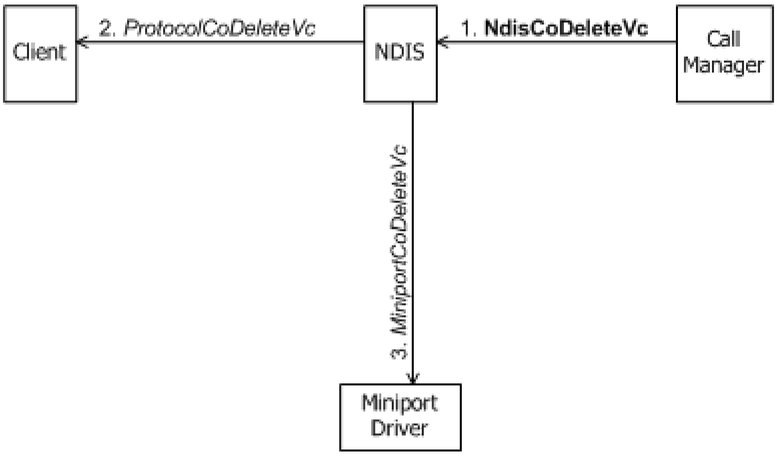
The following figure shows a client of a call manager initiating the deletion of a VC.



The following figure shows a client of an MCM driver initiating the deletion of a VC.



The following figure shows a call manager initiating the deletion of a VC.



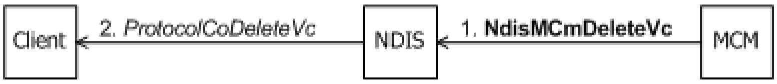
When a client or call manager calls **NdisCoDeleteVc** or when an MCM driver calls **NdisMCmDeleteVc**, there must be no outstanding calls on the given VC and that VC must already have been deactivated. To meet these requirements implies that the following conditions are satisfied:

- The client has already called **NdisClCloseCall** with the given *NdisVcHandle* and its close-call request has completed successfully.
- The call manager has already called **NdisCmDeactivateVc** or the MCM driver has already called **NdisMCmDeactivateVc** with the given *NdisVcHandle* and the deactivation request has completed successfully (see Incoming Request to Close a Call).

A client's or call manager's call to **NdisCoDeleteVc** causes NDIS to call both the underlying miniport driver's **MiniportCoDeleteVc** function and the **ProtocolCoDeleteVc** function of the client or call manager with which the caller shares the *NdisVcHandle* (see the three preceding figures).

MiniportCoDeleteVc frees any resources allocated for the VC, as well as the miniport driver's context for the VC. *ProtocolCoDeleteVc* releases any resources that the client or call manager used to perform operations on and track state for the VC. Both *MiniportCoDeleteVc* and *ProtocolCoDeleteVc* are synchronous functions that cannot return NDIS_STATUS_PENDING.

An MCM driver initiates the deletion of a VC with **NdisMCmDeleteVc**(see the following figure).



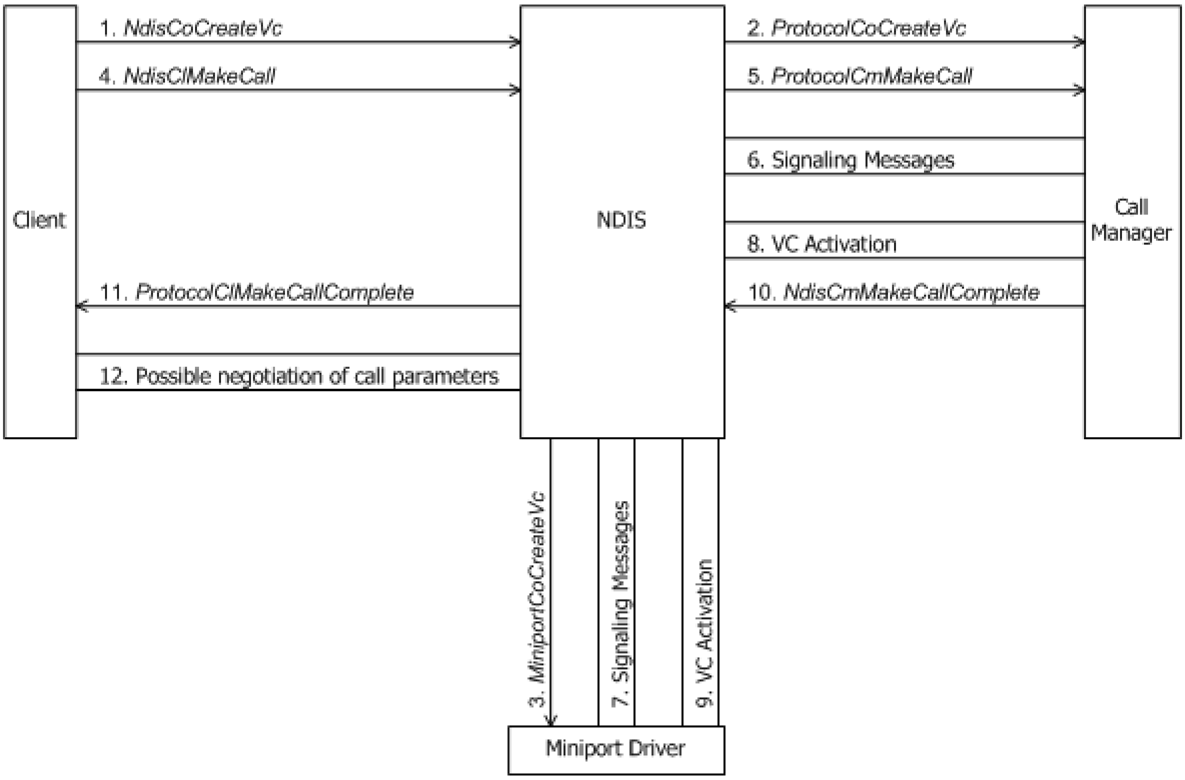
An MCM driver's call to **NdisMCmDeleteVc** causes NDIS to call the **ProtocolCoDeleteVc** function of the client with which the MCM driver shared the *NdisVcHandle* .

When **NdisCoDeleteVc** or **NdisMCmDeleteVc** returns control, the *NdisVcHandle* is no longer valid.

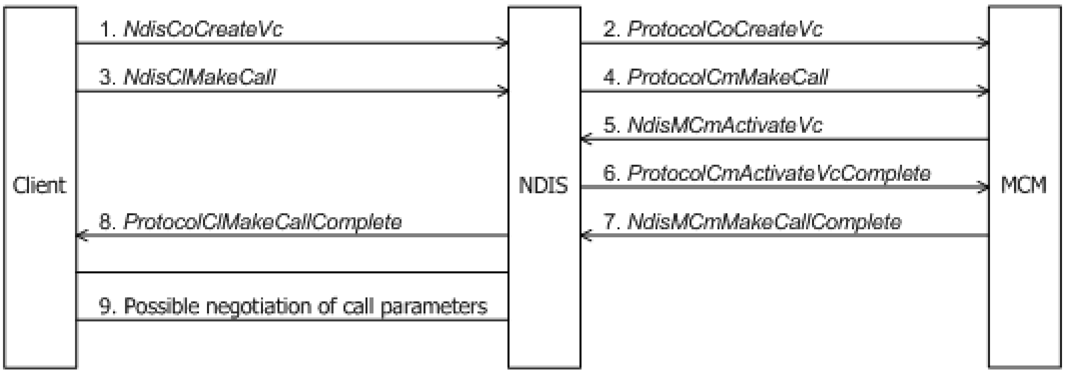
Making a Call

Article • 12/15/2021

The following figure shows a client making an outgoing call through a call manager.



The following figure shows a client making an outgoing call through an MCM driver.


Client
1. NdisCoCreateVc
3. NdisClMakeCall
NDIS
2. ProtocolCoCreateVc
4. ProtocolCmMakeCall
5. NdisMCmActivateVc
6. ProtocolCmActivateVcComplete
MCM
7. NdisMCmMakeCallComplete
8. ProtocolClMakeCallComplete
9. Possible negotiation of call parameters

Before making an outgoing call, a connection-oriented client must:

- Initialize call parameters in a structure of type CO_CALL_PARAMETERS. The call manager or MCM driver typically uses the call parameters the client specifies to set up the call and to derive media parameters for use by the miniport driver.
- Initiate the creation of a VC with NdisCoCreateVc.

On the successful return of **NdisCoCreateVc**, the client calls NdisClMakeCall to initiate the call (see the two figures in this section).

In its call to **NdisClMakeCall**, the client passes a pointer to the CO_CALL_PARAMETERS structure initialized previously. The client also passes an *NdisVcHandle* (returned by **NdisCoCreateVc**) that identifies the VC on which the client will transmit (and perhaps receive) data for the call. If the client is making a multipoint call (a call to more than one remote party), it also passes a *ProtocolPartyContext* that specifies a handle to a client-allocated resident context area in which the client will maintain per-party state for the initial party on the multipoint VC.

The call to **NdisClMakeCall** causes NDIS to forward this request to the **ProtocolCmMakeCall** function of the call manager or MCM driver with which the client shares the given *NdisVcHandle* . *ProtocolCmMakeCall* must validate the input call parameters that were set up by the client.

ProtocolCmMakeCall communicates (exchanges signaling messages) with network control devices to make a connection. A call manager calls **NdisCoSendNetBufferLists** to initiate such an exchange (see Sending NET_BUFFER Structures from CoNDIS Drivers). An MCM driver never calls **NdisCoSendNetBufferLists**. Instead, it transmits the data directly across the network.

The call manager or MCM driver can modify the client-supplied call parameters while negotiating with relevant network components and can return different traffic parameters than the client originally gave to **NdisClMakeCall**(see Incoming Request to Change Call Parameters).

An explicit *NdisPartyHandle* passed to *ProtocolCmMakeCall* indicates that the VC created by the client will be used for a multipoint call. The call manager or MCM driver must allocate and initialize any necessary resources required to maintain per-party state information and control the multipoint call.

After a call manager has done all the necessary communication with its networking hardware as required by its medium, it must call **NdisCmActivateVc** to initiate the activation of the VC on which call data will be sent and perhaps received. An MCM driver must call **NdisMCmActivateVc**.

When the underlying miniport driver is ready to make data transfers on the VC (that is, after the VC has been activated), a call manager calls **NdisCmMakeCallComplete**, and an MCM driver calls **NdisMCmMakeCallComplete**. At this point, the call manager or MCM driver should have negotiated with the network to establish call parameters for the VC, and the underlying miniport driver should have completed activation of the VC.

In the call to **Ndis(M)CmMakeCallComplete**, the call manager or MCM driver passes the call parameters for the VC as a pointer to a structure of type CO_CALL_PARAMETERS. If the call manager has modified the call parameters as originally specified by the client, it can notify the client by setting the CALL_PARAMETERS_CHANGED flag in the CO_CALL_PARAMETERS structure.

A call to **Ndis(M)CmMakeCallComplete** causes NDIS to call the **ProtocolClMakeCallComplete** function of the client that originated the outgoing call. A call to *ProtocolClMakeCallComplete* indicates that the call manager has completed processing the client's request to establish a virtual connection with **NdisClMakeCall**.

If the client's attempt to establish an outgoing call was successful, *ProtocolClMakeCallComplete* should check the CALL_PARAMETERS_CHANGED flag to determine whether the call parameters originally specified by the client were modified. If the flag is set, indicating that the call parameters were changed, *ProtocolClMakeCallComplete* should examine the returned call parameters to determine whether they are acceptable for this connection.

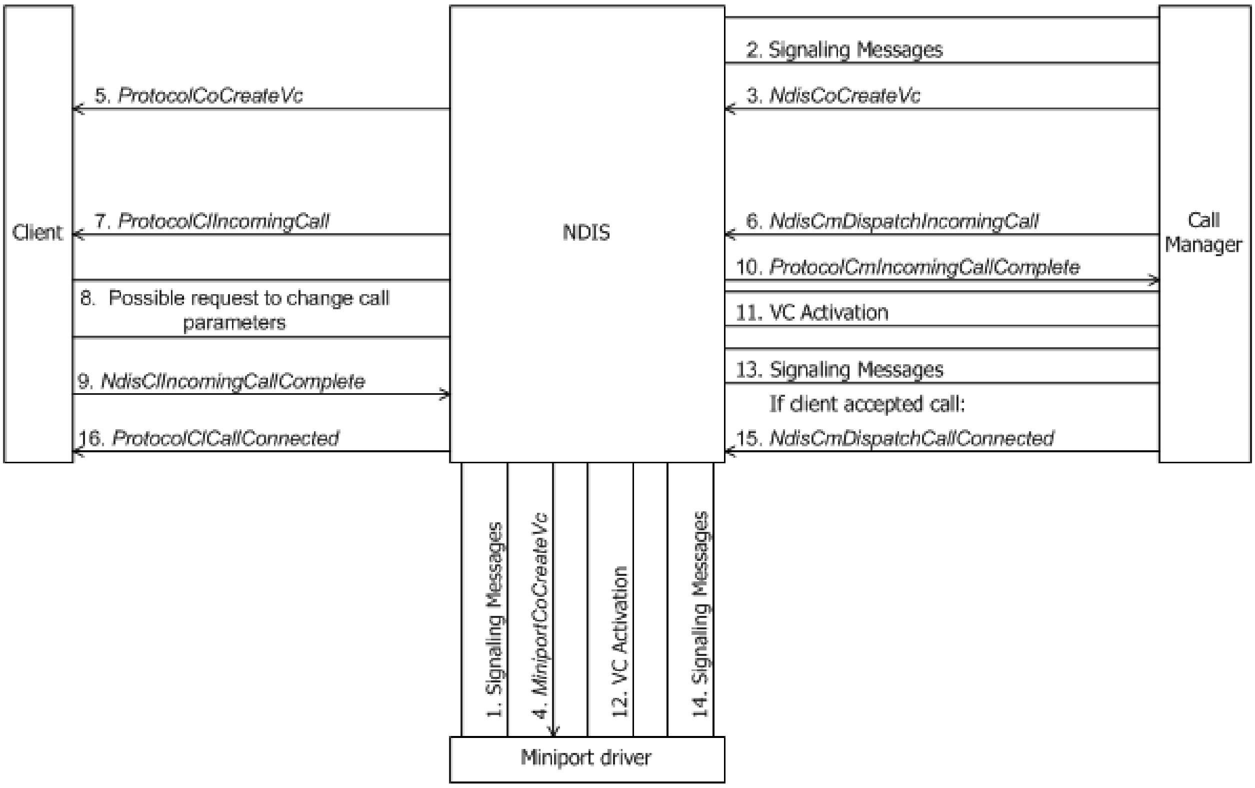
If the call parameters are acceptable, *ProtocolClMakeCallComplete* simply returns control. If the call parameters are not acceptable and if the signaling protocol allows renegotiation at this point, the client can call **NdisClModifyCallQoS** to request a change in call parameters (see Client-Initiated Request to Close a Call). If the signaling protocol does not allow renegotiation of unacceptable call parameters, *ProtocolClMakeCallComplete* must tear down the call with **NdisClCloseCall**(see Client-Initiated Request to Close a Call).

Indicating an Incoming Call

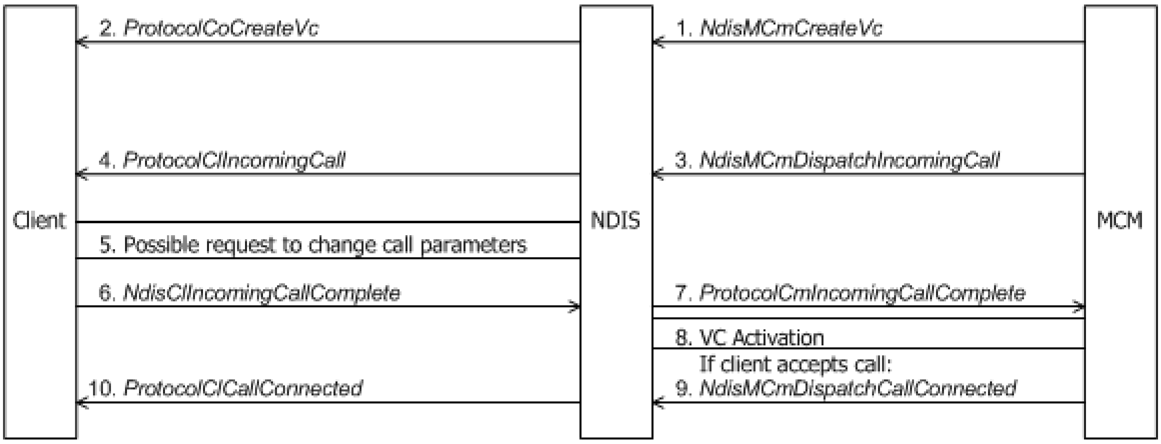
Article • 12/15/2021

A call manager or MCM driver is alerted to an incoming call by signaling messages from the network. From these signaling messages, the call manager or MCM driver extracts the call parameters for the call, including the SAP to which the incoming call is addressed.

The following figure shows an MCM driver indicating an incoming call.



The following figure shows a call manager indicating an incoming call.


Client
NDIS
MCM
1. NdisMCmCreateVc
2. ProtocolCoCreateVc
3. NdisMCmDispatchIncomingCall
4. ProtocolClIncomingCall
5. Possible request to change call parameters
6. NdisClIncomingCallComplete
7. ProtocolCmIncomingCallComplete
8. VC Activation
If client accepts call:
9. NdisMCmDispatchCallConnected
10. ProtocolClCallConnected

If the incoming call parameters are unacceptable to the call manager or MCM driver, it can attempt to negotiate a change in these parameters with the remote party if such negotiation is allowed by the signaling protocol. Alternatively, the client to which the incoming call is directed could attempt to negotiate the call parameters after receiving the call indication from the call manager or MCM driver (see Client-Initiated Request to Change Call Parameters). If the call manager or MCM driver cannot negotiate acceptable call parameters for the call with the remote party, it might refuse the call. The signaling protocol determines what is possible in such cases.

Before indicating an incoming call to a client, the call manager or MCM driver must identify the SAP to which the call is directed. The SAP must have been previously registered by a client. The call manager or MCM driver must also initiate the creation of a VC and initiate the activation of this VC.

The call manager or MCM driver then indicates the incoming call to the client that registered the SAP to which the incoming call is directed. A call manager indicates an incoming call with **NdisCmDispatchIncomingCall**. An MCM driver indicates an incoming call with **NdisMCmDispatchIncomingCall**.

In the call to **Ndis(M)CmDispatchIncomingCall**, the call manager or MCM driver passes the following:

- An *NdisSapHandle* that identifies the SAP to which the incoming call is addressed.

- An *NdisVcHandle* that identifies the virtual circuit for the incoming call.

- A pointer to a structure of type **CO_CALL_PARAMETERS**, which contains the call parameters for the call.

The call to **Ndis(M)CmDispatchIncomingCall** causes NDIS to call the client's **ProtocolClIncomingCall** function, within which the client either accepts or rejects the requested connection. *ProtocolClIncomingCall* should validate the SAP, VC, and call parameters.

ProtocolClIncomingCall can complete synchronously or it can return NDIS_STATUS_PENDING and complete asynchronously with **NdisClIncomingCallComplete**. A call to **NdisClIncomingCallComplete** causes NDIS to call the call manager's or MCM driver's **ProtocolCmIncomingCallComplete** function.

The NDIS_STATUS code that is returned by a synchronous completion of *ProtocolClIncomingCall* or supplied to **NdisClIncomingCallComplete** indicates the client's acceptance or rejection of the incoming call. The client also returns the call parameters for the call in a buffered CO_CALL_PARAMETERS structure. If the client finds the call parameters unacceptable, it can, if allowed by the signaling protocol, request a change in the call parameters by setting the **Flags** member in the CO_CALL_PARAMETERS structure with CALL_PARAMETERS_CHANGED and by supplying the revised call parameters in a buffered CO_CALL_PARAMETERS structure.

If the client accepts the incoming call, the call manager or MCM driver should send signaling messages to indicate to the calling entity that the call has been accepted. Otherwise, the call manager or MCM driver should send signaling messages to indicate that the call has been rejected. If the client is requesting a change in call parameters, the call manager or MCM driver sends signaling messages to request a change in call parameters.

If the client accepted the call, or if the client's requested change in call parameters was accepted by the remote party, a call manager calls **NdisCmDispatchCallConnected**, and an MCM driver calls **NdisMCmDispatchCallConnected**. The call to **Ndis(M)CmDispatchCallConnected** causes NDIS to call the client's *ProtocolClCallConnected* function.

If the client rejected the call and the call manager or MCM driver has already activated a VC for the incoming call, the call manager or MCM driver calls **Ndis(M)CmDeactivateVc** to deactivate the VC if the VC is activated. The call manager or MCM driver can then initiate deletion of the VC by calling **NdisCoDeleteVc** in the case of the call manager or **NdisMCmDeleteVc** in the case of the MCM driver.

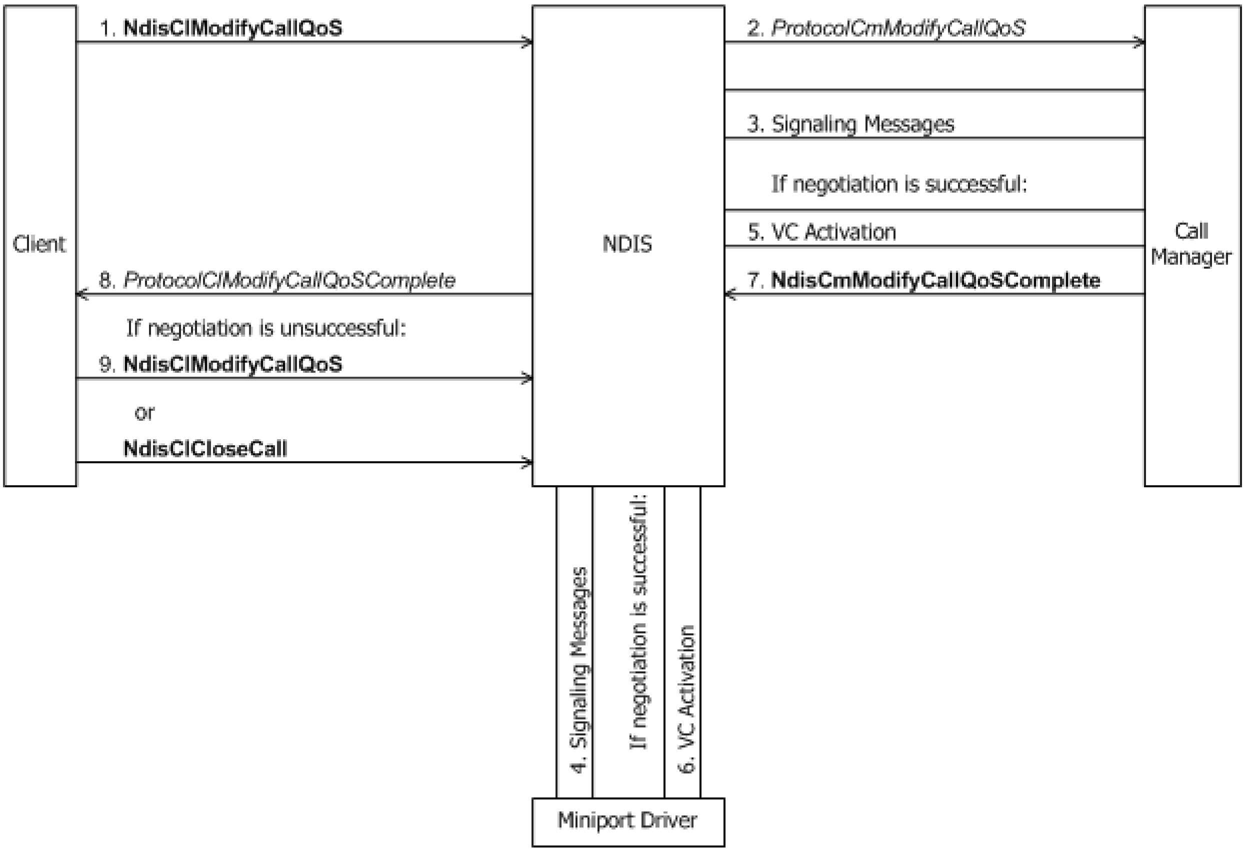
If the client accepted the call but the end-to-end connection was not successfully established (because, for example, the remote party tore down the call), the call manager or MCM driver will not call **Ndis(M)CmDispatchCallConnected**. Instead, it will call **Ndis(M)CmDispatchIncomingCloseCall**, which causes NDIS to call the client's *ProtocolClIncomingCloseCall* function. The client must then call **NdisClCloseCall** to complete the teardown of the call. The call manager or MCM driver then calls **Ndis(M)CmDeactivateVC** to deactivate the VC that it created for the incoming call. The call manager or MCM driver can then initiate deletion of the VC by calling **NdisCoDeleteVc** in the case of the call manager or **NdisMCmDeleteVc** in the case of the MCM driver.

Client-Initiated Request to Change Call Parameters

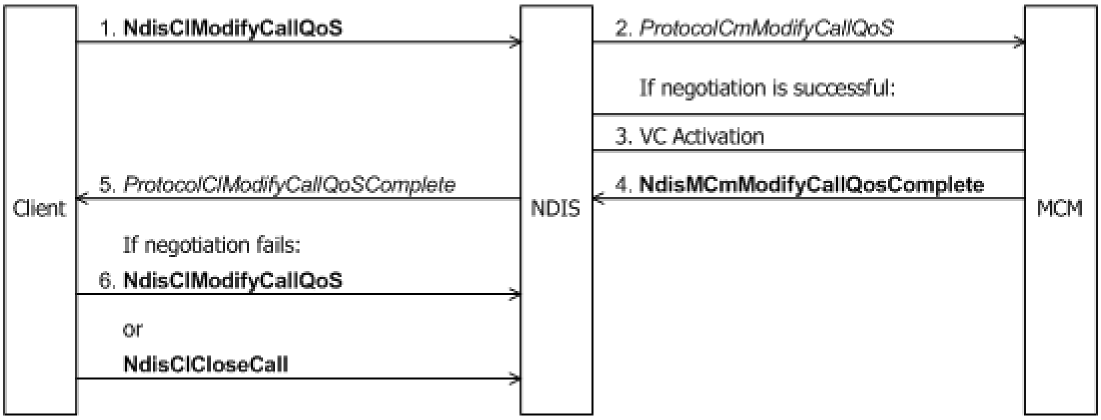
Article • 12/15/2021

A client requests a change in quality of service (QoS) on an active virtual connection (VC) with NdisClModifyCallQoS.

The following figure shows the client of a call manager requesting a change in quality of service.


Client
NDIS
Call Manager
Miniport Driver
1. NdisClModifyCallQoS
2. ProtocolCmModifyCallQoS
3. Signaling Messages
If negotiation is successful:
5. VC Activation
7. NdisCmModifyCallQoSComplete
8. ProtocolClModifyCallQoSComplete
If negotiation is unsuccessful:
9. NdisClModifyCallQoS
or
NdisClCloseCall
4. Signaling Messages
If negotiation is successful:
6. VC Activation

The following figure shows the client of an MCM driver requesting a change in quality of service.


Client
NDIS
MCM
1. NdisClModifyCallQoS
2. ProtocolCmModifyCallQoS
If negotiation is successful:
3. VC Activation
4. NdisMCmModifyCallQosComplete
5. ProtocolClModifyCallQoSComplete
If negotiation fails:
6. NdisClModifyCallQoS
or
NdisClCloseCall

In the call to **NdisClModifyCallQoS**, the client supplies:

- An *NdisVcHandle* parameter that identifies the VC.

- A pointer to a CO_CALL_PARAMETERS structure that contains the call parameters the client is requesting.

The circumstances under which a client can request a change in QoS are determined by the signaling protocol.

The call to **NdisClModifyCallQoS** causes NDIS to call the call manager's or MCM driver's ProtocolCmModifyCallQoS function, which inputs the *NdisVcHandle* and buffered CO_CALL_PARAMETERS structure that the client passes to **NdisClModifyCallQoS**. **ProtocolCmModifyQoS** communicates with network control devices or other media-specific agents, as necessitated by its media, to modify the media-specific call parameters for an established virtual connection.

After communicating with the network and determining that the changes were successful, a call manager must call NdisCmActivateVc(and an MCM driver must call NdisMCmActivateVc) to activate the specified VC with the new call parameters.

If the network does not accept the new call parameters or if the underlying miniport driver cannot accept the parameters, the call manager or MCM driver must restore the VC to the state that existed before any modifications were attempted, and return NDIS_STATUS_FAILURE.

To indicate the status of the client's request to change QoS, a call manager calls NdisCmModifyCallQoSComplete, and an MCM driver calls NdisMCmModifyCallQoSComplete. In this call, the call manager or MCM driver passes:

- An NDIS_STATUS that indicates the status of the request.

- An *NdisVcHandle* that identifies the VC.

- A pointer to a CO_CALL_PARAMETERS structure that contains the call parameters for the VC.

If allowed by the signaling protocol, the call manager or MCM driver can pass modified call parameters back to the client. These modifications can be the product of negotiation with the network or they can be supplied by the call manager or MCM driver itself. A call manager or MCM driver should indicate that the call parameters have been modified by setting the CALL_PARAMETERS_CHANGED flag in the CO_CALL_PARAMETERS structure.

The call to **Ndis(M)CmModifyCallQoSComplete** causes NDIS to call the client's ProtocolClModifyCallQoSComplete function. NDIS passes the following to *ProtocolClModifyCallQoSComplete*:

- An NDIS_STATUS that indicates the status of the client's request to change the QoS.
- A *ProtocolVcContext* handle that identifies the VC.
- A pointer to a CO_CALL_PARAMETERS structure that contains the call parameters that are passed by the call manager or MCM driver to **Ndis(M)CmModifyCallQoSComplete**.

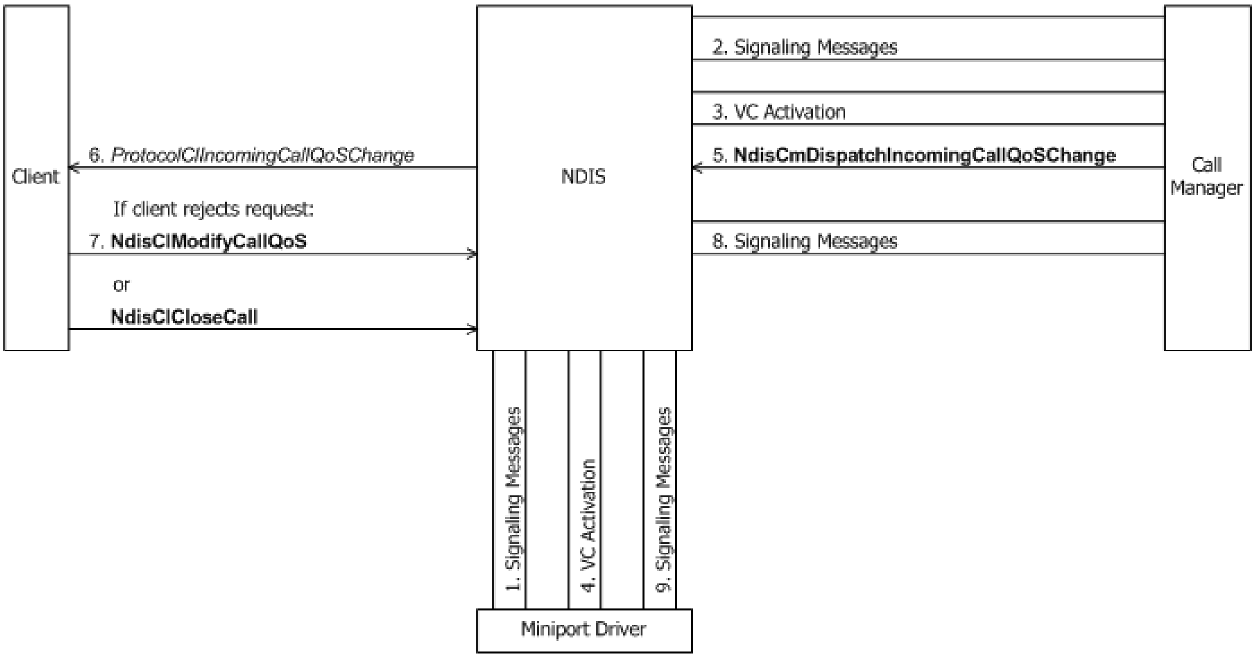
If the CALL_PARAMETERS_CHANGED flag is set in the CO_CALL_PARAMETERS structure, the client must examine the returned call parameters and determine whether the modifications are acceptable. If the client's call to **NdisClModifyCallQoS** succeeds, *ProtocolClModifyCallQoSComplete* can accept the QoS change by simply returning control. Otherwise, *ProtocolClModifyCallQoSComplete* can engage in further negotiation with the call manager if allowed by the signaling protocol and as long as the client's developer places some reasonable limit on the number of possible renegotiations. Alternatively, *ProtocolClModifyCallQoSComplete* can simply tear down the call with **NdisClCloseCall**(see Client-Initiated Request to Close a Call) whenever the call manager rejects a request to change the QoS and the previously established QoS has become unacceptable to the client.

Incoming Request to Change Call Parameters

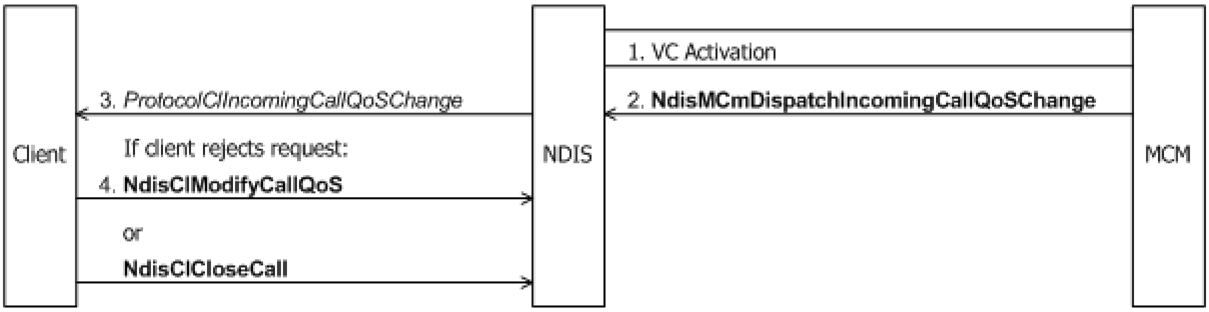
Article • 12/15/2021

A call manager or MCM driver is alerted to an incoming request from a remote party to change the call parameters on an active VC by signaling messages from the network. Whether a call manager or MCM driver supports dynamic QoS changes on active calls depends on the signaling protocol.

The following figure shows an incoming request through a call manager to change call parameters.



The following figure shows an incoming request through an MCM driver to change call parameters.


Client
NDIS
MCM
1. VC Activation
2. NdisMCmDispatchIncomingCallQoSChange
3. ProtocolClIncomingCallQoSChange
If client rejects request:
4. NdisClModifyCallQoS
or
NdisClCloseCall

After receiving an incoming request to change call parameters, a call manager passes appropriately modified call parameters to NdisCmActivateVc to notify the underlying miniport driver of the proposed QoS change. An MCM driver passes modified call parameters to NdisMCmActivateVc(see Activating a VC). If the underlying miniport driver accepts the changed call parameters, a call manager then calls

NdisCmDispatchIncomingCallQosChange(see Incoming Request to Change Call Parameters). An MCM driver calls **NdisMCmDispatchIncomingCallQosChange**(see Incoming Request to Change Call Parameters). The call manager or MCM driver passes an *NdisVcHandle* and a buffered **CO_CALL_PARAMETERS** structure to **Ndis(M)CmDispatchIncomingCallQoSChange**.

A call to **Ndis(M)CmDispatchIncomingCallQoSChange** causes NDIS to call the client's *ProtocolClIncomingCallQoSChange* function. NDIS passes a *ProtocolVcContext* handle that identifies the VC and the modified call parameters in a buffered CO_CALL_PARAMETERS structure to *ProtocolClIncomingCallQoSChange*.

The client accepts the proposed modifications to the call parameters for the VC by doing nothing, except possibly updating any state it maintains about the QoS for the VC, and returning control. If the proposed modifications are unacceptable, the client can attempt to renegotiate the call parameters with **NdisClModifyCallQoS** if allowed by the signaling protocol (see Client-Initiated Request to Change Call Parameters). Otherwise, the client rejects the proposed QoS change by tearing down the call with **NdisClCloseCall**(see Client-Initiated Request to Close a Call).

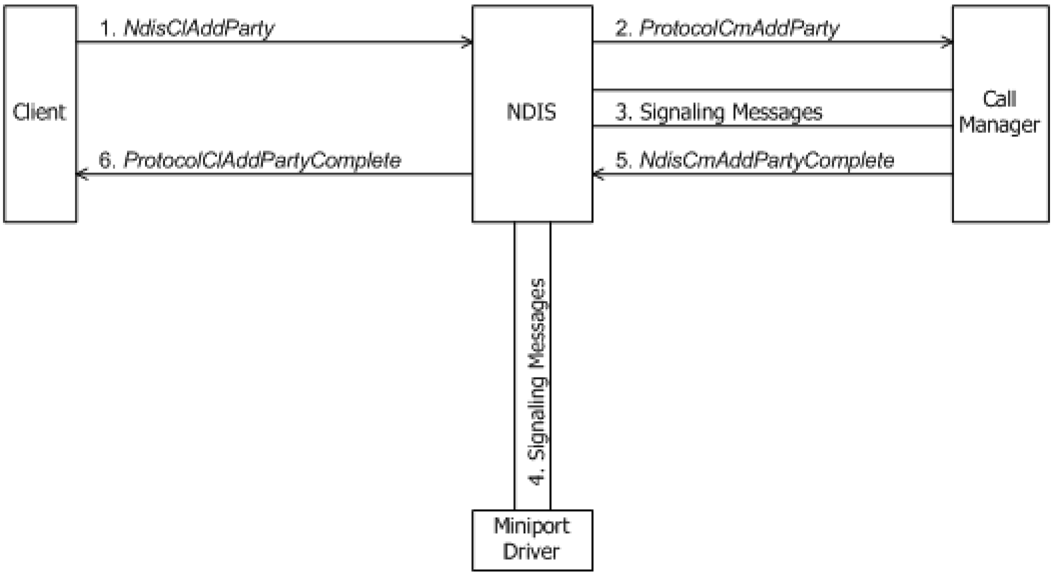
After **ProtocolClIncomingCallQoS** returns, the call manager or MCM driver communicates the client's acceptance or rejection of the proposed change to the remote party that originated the request.

Adding a Party to a Multipoint Call

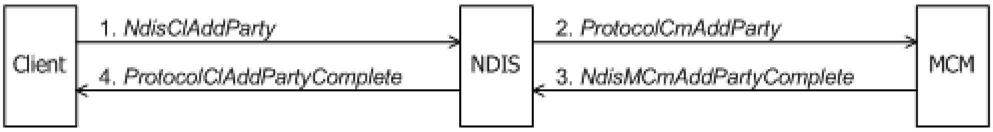
Article • 12/15/2021

A client requests to add a party to a multipoint call with **NdisClAddParty**. A client can add a party only to an existing multipoint call--that is, a call for which the client supplied a *ProtocolPartyContext* to **NdisClMakeCall**(see Making a Call).

The following figure shows a client of a call manager requesting to add a party to multipoint call.



The following figure shows a client of an MCM driver requesting to add a party to multipoint call.



Before it calls **NdisClAddParty**, a client must allocate and initialize its context area for the party to be added. Clients commonly pass a pointer to such a context area as the *ProtocolPartyContext* and a pointer to a variable within that context area as the *NdisPartyHandle* parameters when they call **NdisClAddParty**.

In addition to an *NdisVcHandle* and a *ProtocolPartyContext*, the client passes call parameters (a buffered **CO_CALL_PARAMETERS** structure) to **NdisClAddParty**. The underlying network medium determines whether a client can specify per-party traffic parameters on a multipoint VC.

The call to **NdisClAddParty** causes NDIS to forward this request to the **ProtocolCmAddParty** function of the call manager or MCM driver with which the client shares the given *NdisVcHandle* . NDIS passes the following to the *ProtocolCmAddParty*:

- A *CallMgrVcContext* that indicates the VC for the call.
- A pointer to a CO_CALL_PARAMETERS structure that contains the call parameters that the client passes to **NdisClAddParty**.
- An *NdisPartyHandle* that identifies the party to be added.

ProtocolCmAddParty allocates and initializes any dynamic resources needed for the party being added to the call. From *ProtocolCmAddParty*, a call manager or MCM driver communicates with network control devices or other media-specific agents, as necessary, to add the specified party to the multipoint call.

If the client passed in call parameters that did not match those already established for the multipoint VC, the call manager or MCM driver can, for example:

- Set up the per-party traffic parameters if the underlying network medium supports this feature on multipoint VCs.
- Reset the client-supplied traffic parameters to those originally established for the VC.
- Change the call parameters for the VC and for every party currently connected on it.
- Fail the client's attempt to add a party.

ProtocolCmAddParty can complete synchronously or, more probably, asynchronously with **NdisCmAddPartyComplete**, in the case of a call manager, or **NdisMCmAddPartyComplete**, in the case of an MCM driver. Whether the call manager or MCM driver completes the operation synchronously or asynchronously, it passes the buffered call parameters to NDIS.

The call to **Ndis(M)CmAddPartyComplete** causes NDIS to call the client's **ProtocolClAddPartyComplete** function. If the client's request to add the party succeeded and if the signaling protocol allows the call manager or MCM driver to modify the call parameters, *ProtocolClAddPartyComplete* should test the CALL_PARAMETERS_CHANGED flag in the buffered CO_CALL_PARAMETERS structure to determine whether the call parameters were modified. The signaling protocol determines what the client can do if it finds the modifications to CO_CALL_PARAMETERS unacceptable. Usually, a client calls **NdisClDropParty** in this case (see Dropping a Party from a Multipoint Call).

Dropping a Party from a Multipoint Call

Article • 12/15/2021

A connection-oriented client that serves as the root of a multipoint call must eventually drop each party from that call with **NdisClDropParty** or **NdisClCloseCall**.

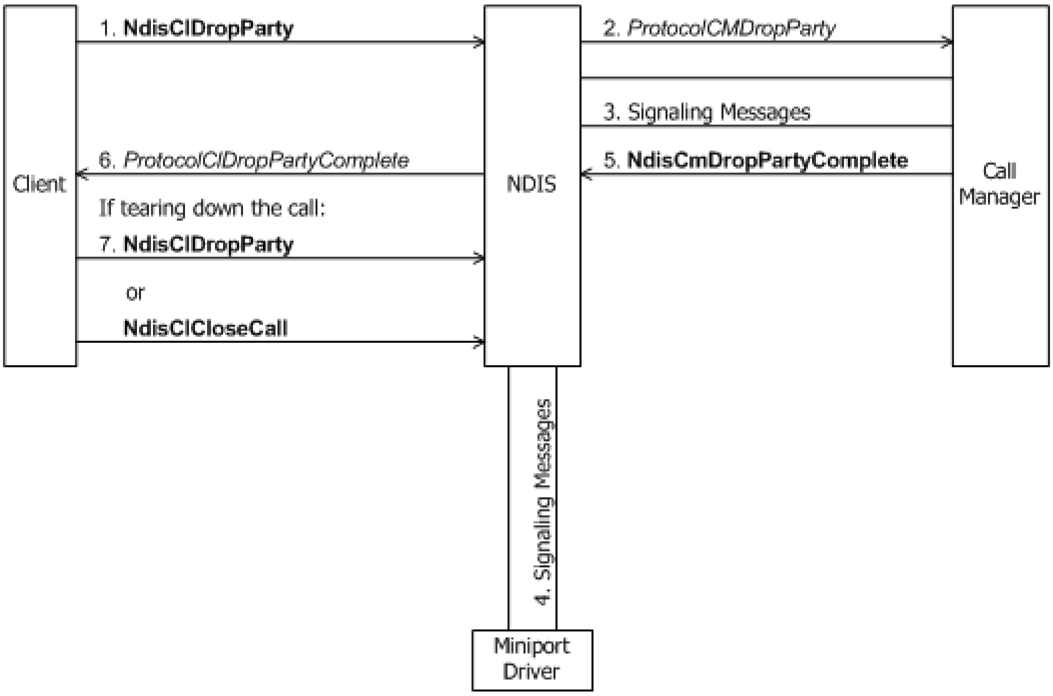
A client drops a party from a call in the following situations:

- Before initiating the tear down of a multipoint call with **NdisClCloseCall**(see Client-Initiated Request to Close a Call), a client must drop all but the last party with successive calls to **NdisClDropParty**. The client specifies the last party to drop from the call with **NdisClCloseCall**.

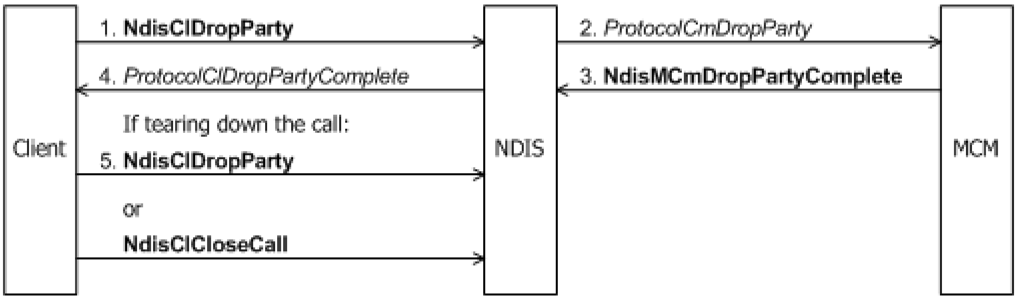
- In response to a remote party's request to be dropped from a multipoint call (see Incoming Request to Drop a Party from a Multipoint Call), a client, from its **ProtocolClIncomingDropParty** function, calls **NdisClDropParty**.

A client's call to **NdisClDropParty** causes NDIS to call the **ProtocolCmDropParty** function of the call manager or MCM driver that shares the same *NdisVcHandle* to the multipoint VC.

The following figure shows the client of a call manager requesting to drop a party from a multipoint call.



The next figure shows the client of an MCM driver requesting to drop a party from a multipoint call.



ProtocolCmDropParty communicates with network control devices to drop a party from an existing multipoint call. NDIS can pass to *ProtocolCmDropParty* a pointer to a buffer that contains data (supplied to the client in the call to **NdisClDropParty**). *ProtocolCmDropParty* must send any such data across the network before the connection is dropped.

ProtocolCmDropParty can complete synchronously, or more probably, asynchronously with **NdisCmDropPartyComplete**, in the case of a call manager, or **NdisMCmDropPartyComplete**, in the case of an MCM driver.

The call to **Ndis(M)CmDropPartyComplete** causes NDIS to call the client's **ProtocolClDropPartyComplete** function. If the client is in the process of tearing down a multipoint VC that it created, *ProtocolClDropPartyComplete* can call **NdisClDropParty** with any valid *NdisPartyHandle* to one of the remaining parties on the client's active multipoint VC. If only one party remains on its multipoint VC, the client should drop that party by passing its *NdisPartyHandle* to **NdisClCloseCall**(see Client-Initiated Request to Close a Call).

Incoming Request to Drop a Party from a Multipoint Call

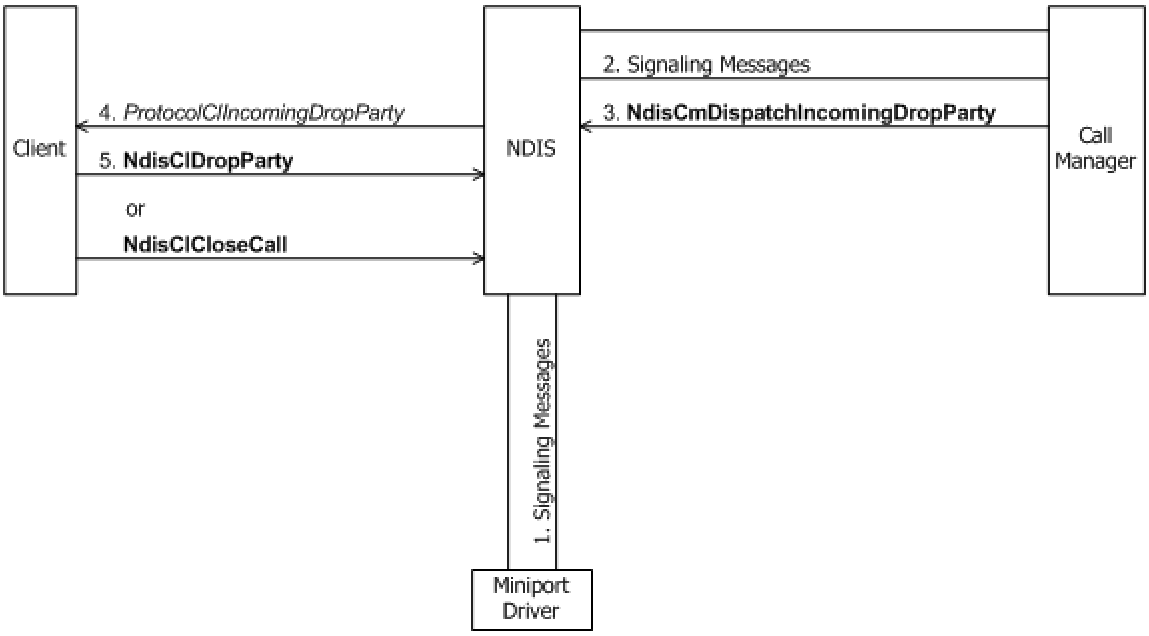
Article • 12/15/2021

A call manager or MCM driver is alerted to an incoming request from a remote party to drop that party from a multipoint call by signaling messages from the network. A call manager or MCM driver can also signal an incoming request to drop a party if it detects network problems that prevent further data transfers on the VC.

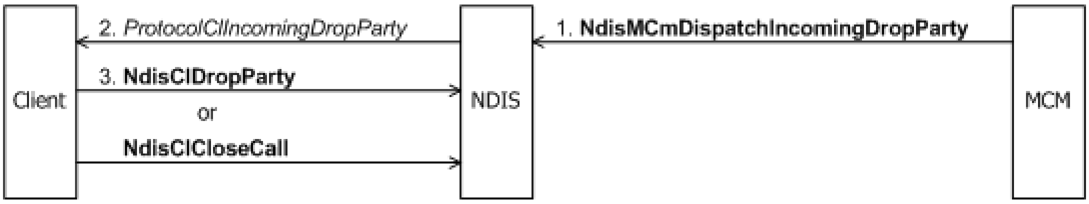
If the party that is being dropped from the call is not the last party on the VC, a call manager calls **NdisCmDispatchIncomingDropParty**. An MCM driver calls **NdisMCmDispatchIncomingDropParty**. If the party that is being dropped is the last party on the VC, a call manager calls **NdisCmDispatchIncomingCloseCall**, and an MCM driver calls **NdisMCmDispatchIncomingCloseCall**(see Incoming Request to Close a Call).

A call to **Ndis(M)CmDispatchIncomingDropParty** causes NDIS to call the client's **ProtocolClIncomingDropParty** function.

The following shows an incoming request through a call manager to drop a party through a multipoint call.



The next figure shows an incoming request through an MCM driver to drop a party through a multipoint call.



Client
2. *ProtocolClIncomingDropParty*
3. **NdisClDropParty**
or
NdisClCloseCall
NDIS
1. **NdisMCmDispatchIncomingDropParty**
MCM

ProtocolClIncomingDropParty should carry out any protocol-determined operations for dropping the party from the client's multipoint VC. If the party that is being dropped is not the last party on the VC, *ProtocolClIncomingDropParty* must call **NdisClDropParty**(see Dropping a Party from a Multipoint Call). If the party being dropped is the last party on the VC, *ProtocolClIncomingDropParty* must call **NdisClCloseCall**(see Client-Initiated Request to Close a Call).

Sending and receiving data in CoNDIS

Article • 03/14/2023

Transferring data involves sending or receiving packets over an established and activated VC.

Note Protocol drivers must not call **NdisCoSendNetBufferLists** to send data to a VC after calling **NdisClCloseCall** for that VC.

The CoNDIS send and receive functions are similar to connectionless send and receive functions. The primary difference between the CoNDIS and connectionless interfaces is the management of virtual connections (VCs). For more information about connectionless send and receive operations, see Send and Receive Operations.

In a single function call, CoNDIS drivers can send multiple **NET_BUFFER_LIST** structures with multiple **NET_BUFFER** structures on each NET_BUFFER_LIST structure. Also, CoNDIS drivers can indicate completed send operations for multiple NET_BUFFER_LIST structures with multiple NET_BUFFER structures on each NET_BUFFER_LIST structure.

In the receive path, CoNDIS miniport drivers can provide a list of NET_BUFFER_LIST structures to indicate receives. Each NET_BUFFER_LIST that a miniport driver provides contains one NET_BUFFER structure. Because a different protocol binding can process each NET_BUFFER_LIST structure, NDIS can independently return each NET_BUFFER_LIST structure to the miniport driver.

To support NDIS 5.*x* and earlier drivers, CoNDIS provides a translation layer between legacy **NDIS_PACKET** structures and the NET_BUFFER-based structures. CoNDIS performs the necessary conversion between NET_BUFFER structures and NDIS_PACKET structures. To avoid degrading performance because of the translation, CoNDIS drivers must be updated to support NET_BUFFER structures and should support multiple NET_BUFFER_LIST structures in all data paths.

This section includes the following topics:

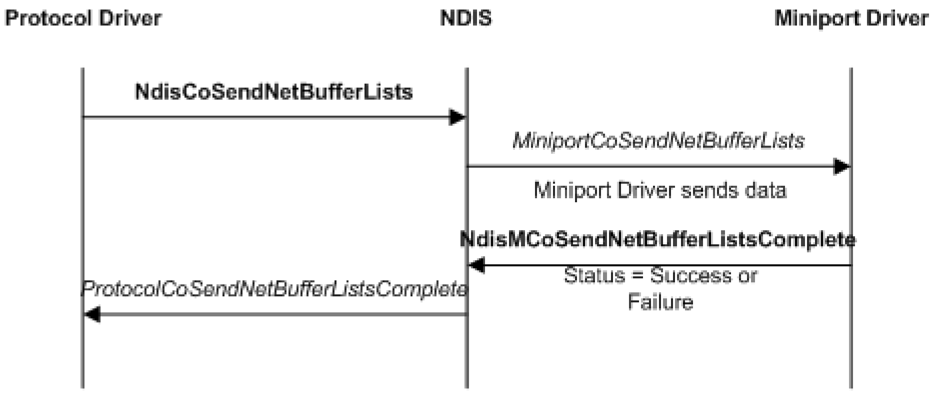
Sending NET_BUFFER Structures from CoNDIS Drivers

Receiving NET_BUFFER Structures in CoNDIS Drivers

Sending NET_BUFFER Structures from CoNDIS Drivers

Article • 03/14/2023

The following figure illustrates a basic CoNDIS send operation, which involves a protocol driver, NDIS, and a miniport driver.


Protocol Driver
NDIS
Miniport Driver
NdisCoSendNetBufferLists
MiniportCoSendNetBufferLists
Miniport Driver sends data
NdisMCoSendNetBufferListsComplete
Status = Success or Failure
ProtocolCoSendNetBufferListsComplete

As the preceding figure shows, protocol drivers call the NdisCoSendNetBufferLists function to send NET_BUFFER_LIST structures on a virtual connection (VC). NDIS then calls the miniport driver's MiniportCoSendNetBufferLists function to forward the NET_BUFFER_LIST structures to an underlying miniport driver.

All NET_BUFFER-based send operations are asynchronous. Therefore, the miniport driver always calls the NdisMCoSendNetBufferListsComplete function and provides an appropriate status code when it is done sending the data. The miniport driver can complete the send operation for each NET_BUFFER_LIST structure independent of other NET_BUFFER_LIST structures. NDIS calls the protocol driver's ProtocolCoSendNetBufferListsComplete function each time the miniport driver calls **NdisMCoSendNetBufferListsComplete**.

Protocol drivers can reclaim the ownership of the NET_BUFFER_LIST structures and all associated structures and data as soon as NDIS calls the protocol driver's *ProtocolCoSendNetBufferListsComplete* function.

The miniport driver or NDIS can return the NET_BUFFER_LIST structures in any order. But protocol drivers are guaranteed that the list of NET_BUFFER structures that are attached to each NET_BUFFER_LIST structure has not been modified.

Protocols drivers set the **SourceHandle** member in the NET_BUFFER_LIST structure to the same value as the *NdisVcHandle* parameter of **NdisCoSendNetBufferLists**. NDIS uses the **SourceHandle** member to return the NET_BUFFER_LIST structures to the protocol driver that sent the NET_BUFFER_LIST structures.

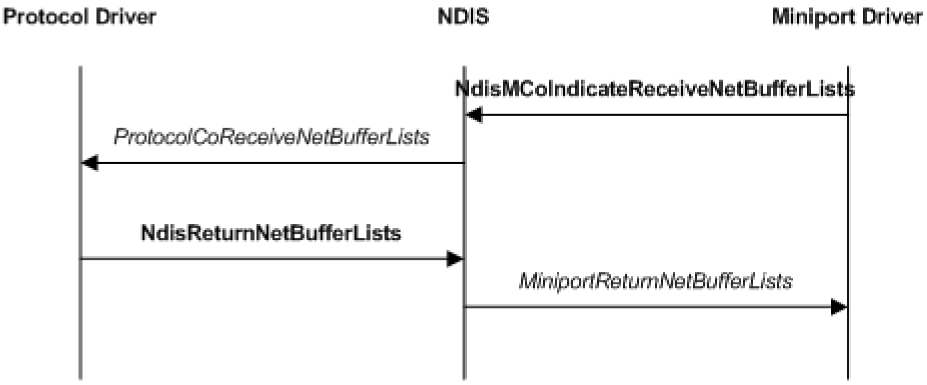
Intermediate drivers also set the **SourceHandle** member in the NET_BUFFER_LIST structure to the *NdisVcHandle* value. If an intermediate driver forwards a send request, the driver must save the **SourceHandle** value that the overlying driver provided before it writes to the **SourceHandle** member. When NDIS returns a forwarded NET_BUFFER_LIST structure to the intermediate driver, the intermediate driver must restore the **SourceHandle** that it saved.

Protocol drivers can cancel send requests by using the same mechanisms as connectionless drivers. For more information about canceling send requests, see Canceling a Send Operation.

Receiving NET_BUFFER Structures in CoNDIS Drivers

Article • 03/14/2023

The following figure illustrates a basic CoNDIS receive operation, which involves a protocol driver, NDIS, and a miniport driver.


Protocol Driver
NDIS
Miniport Driver
NdisMCoIndicateReceiveNetBufferLists
ProtocolCoReceiveNetBufferLists
NdisReturnNetBufferLists
MiniportReturnNetBufferLists

As the preceding figure shows, miniport drivers call the **NdisMCoIndicateReceiveNetBufferLists** function to indicate **NET_BUFFER** structures to overlying drivers. In most miniport drivers, each NET_BUFFER structure is attached to a separate **NET_BUFFER_LIST** structure, so protocol drivers can create a subset of the original list of NET_BUFFER_LIST structures and forward them to different clients. However, the number of NET_BUFFER structures that are attached to a NET_BUFFER_LIST depends on the driver.

After the miniport driver links all the NET_BUFFER_LIST structures, the miniport driver passes a pointer to the first NET_BUFFER_LIST structure in the list to the **NdisMCoIndicateReceiveNetBufferLists** function. NDIS examines the NET_BUFFER_LIST structures and calls the **ProtocolCoReceiveNetBufferLists** function of the protocol driver that is associated with the specified virtual connection (VC). NDIS passes a subset of the list that includes only the NET_BUFFER_LIST structures that are associated with the correct binding to each protocol driver.

If the NDIS_RECEIVE_FLAGS_STATUS_RESOURCES flag is set in the *CoReceiveFlags* parameter for a protocol driver's *ProtocolCoReceiveNetBufferLists* function, NDIS regains ownership of the **NET_BUFFER_LIST** structures immediately after *ProtocolCoReceiveNetBufferLists* returns.

If the NDIS_RECEIVE_FLAGS_STATUS_RESOURCES flag is not set in the *CoReceiveFlags* parameter for a protocol driver's **ProtocolCoReceiveNetBufferLists** function, the protocol driver can retain ownership of the NET_BUFFER_LIST structures. In this case, the protocol driver must return the NET_BUFFER_LIST structures by calling the NdisReturnNetBufferLists function.

If a miniport driver runs low on receive resources, it can set the NDIS_RECEIVE_FLAGS_STATUS_RESOURCES flag in the *CoReceiveFlags* parameter for the NdisMCoIndicateReceiveNetBufferLists function. In that case, the driver can reclaim ownership of all of the indicated NET_BUFFER_LIST structures and embedded NET_BUFFER structures as soon as **NdisMCoIndicateReceiveNetBufferLists** returns. If a miniport driver indicates NET_BUFFER structures with the NDIS_RECEIVE_FLAGS_RESOURCES flag set, the protocol drivers must copy the data, so you should avoid using NDIS_RECEIVE_FLAGS_RESOURCES in this way. A miniport driver should detect when it has low receive resources and should complete any steps that are necessary to avoid this situation.

NDIS calls a miniport driver's *MiniportReturnNetBufferLists* function after the protocol driver calls **NdisReturnNetBufferLists**.

Note If a miniport driver indicates a NET_BUFFER_LIST structure with a given status, NDIS is not required to indicate the NET_BUFFER_LIST structure to the overlying drivers with the same status. For example, NDIS could copy a NET_BUFFER_LIST structure with the NDIS_RECEIVE_FLAGS_RESOURCES flag set and indicate the copy to the overlying drivers with this flag cleared.

NDIS can return NET_BUFFER_LIST structures to the miniport driver in any arbitrary order and in any combination. That is, the linked list of NET_BUFFER_LIST structures that NDIS returns to a miniport driver by calling *MiniportReturnNetBufferLists* can have NET_BUFFER_LIST structures from different previous calls to NdisMCoIndicateReceiveNetBufferLists.

Miniport drivers must set the **SourceHandle** member in the NET_BUFFER_LIST structures to the same value as the *NdisVcHandle* parameter of **NdisMCoIndicateReceiveNetBufferLists**. so that NDIS can return the NET_BUFFER_LIST structures to the correct miniport driver.

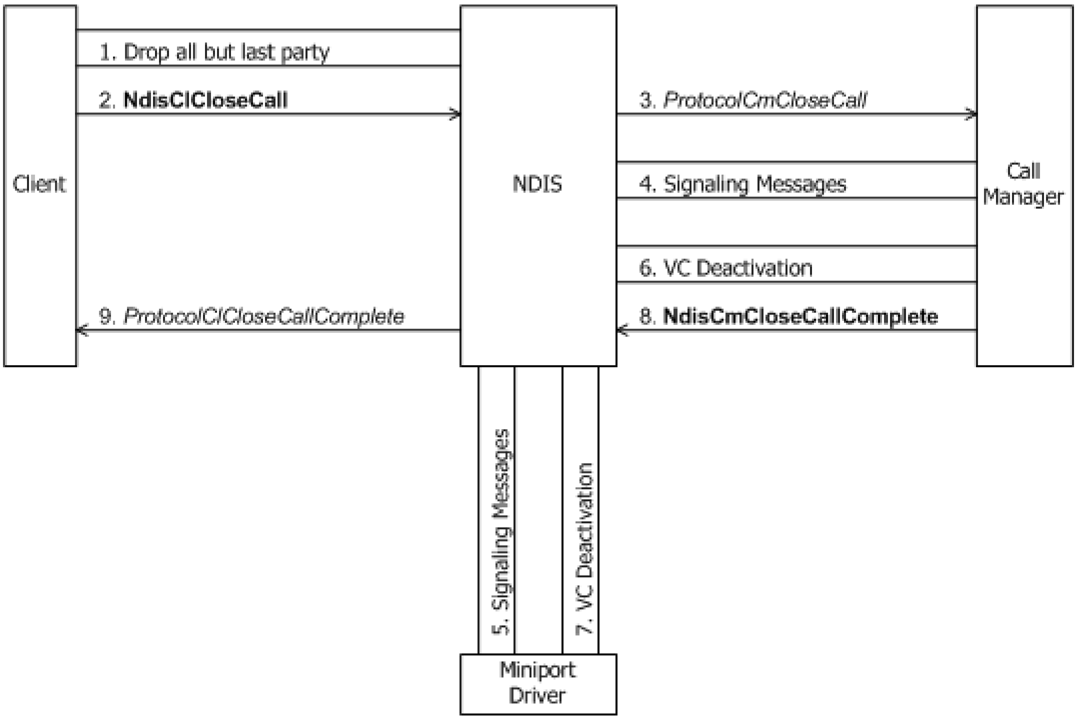
Intermediate drivers also set the **SourceHandle** member in the NET_BUFFER_LIST structure to the *NdisVcHandle* value. If an intermediate driver forwards a receive indication, the driver must save the **SourceHandle** value that the underlying driver provided before it writes to the **SourceHandle** member. When NDIS returns a forwarded NET_BUFFER_LIST structure to the intermediate driver, the intermediate driver must restore the **SourceHandle** that it saved.

Client-Initiated Request to Close a Call

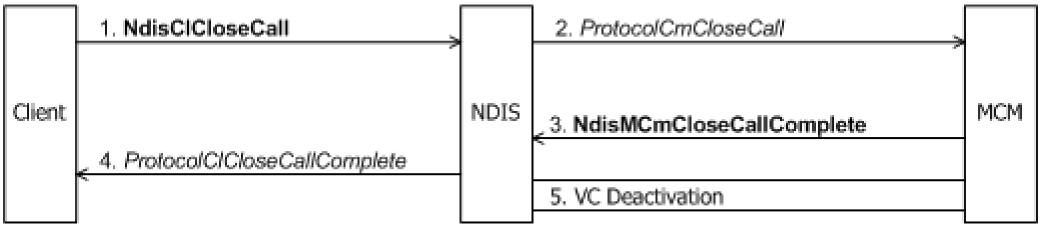
Article • 12/15/2021

If a client is closing a multipoint call to which more than one party is still connected, it must first call **NdisClDropParty** as many times as necessary to drop all but the last party from the call (see Dropping a Party from a Multipoint Call).

A client initiates the closing of a call with **NdisClCloseCall**. The following figure shows a client initiating the closing of a call through a call manager.



The next figure shows a client initiating the closing of a call through an MCM driver.



A connection-oriented client typically calls **NdisClCloseCall** in any one of the following circumstances:

- To close an established outgoing or incoming call.
- From the **ProtocolClIncomingCloseCall** function to tear down an established call (see Incoming Request to Close a Call).
- From the *ProtocolClIncomingCallQoSChange* function to tear down an established call if a QoS change that the remote party proposes is unacceptable (see Incoming

Request to Change Call Parameters).

- From the **ProtocolClModifyCallQoSComplete** function to tear down an established call if a QoS change that the client proposes is unacceptable to the remote party (see Client-Initiated Request to Change Call Parameters).

A client's call to **NdisClCloseCall** causes NDIS to call the call manager's or MCM driver's **ProtocolCmCloseCall** function. *ProtocolCmCloseCall* must communicate with network control devices to terminate a connection between the local node and a remote node.

If *ProtocolCmCloseCall* is passed an explicit *CallMgrPartyContext*, the call that is being terminated is a multipoint call. The call manager or MCM driver must perform any necessary network communication with its networking hardware, as appropriate to its media type, to terminate the call as a multipoint call.

NDIS can pass *ProtocolCmCloseCall* a pointer to a buffer containing data supplied by the client in the call to **NdisClClose**. This data can be diagnostic data that indicates why the call was closed or any other data that is required by the signaling protocol. *ProtocolCmCloseCall* must send any such data across the network before completing the call termination. If sending data concurrent with a connection being terminated is not supported, a call manager or MCM driver should return NDIS_STATUS_INVALID_DATA.

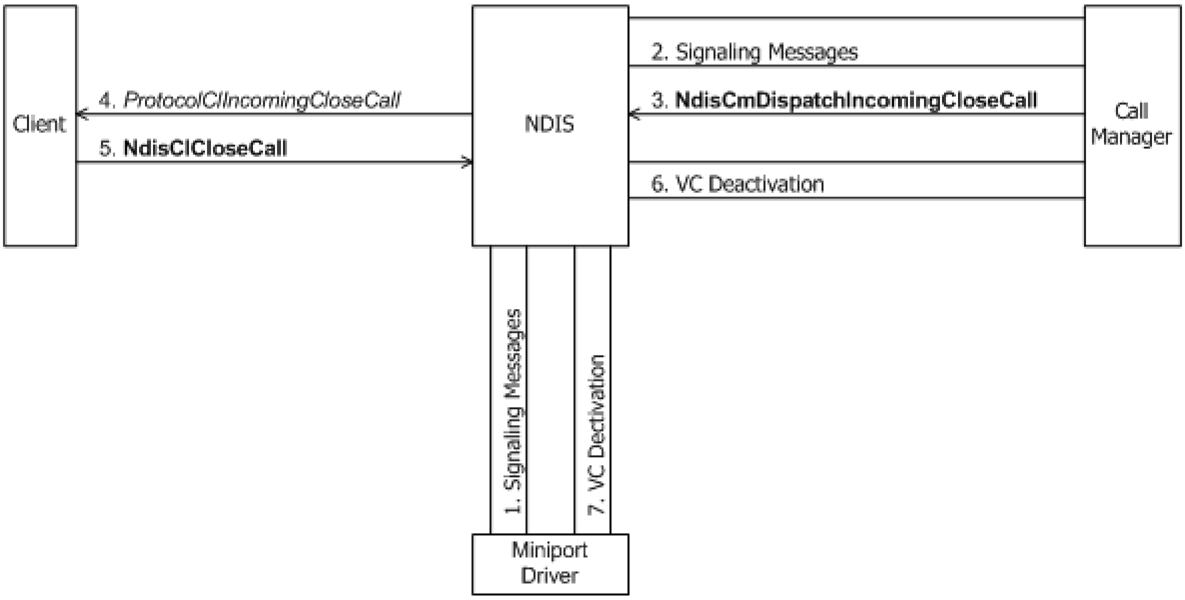
ProtocolCmCloseCall can complete synchronously or, more probably, asynchronously with **NdisCmCloseCallComplete**(in the case of a call manager) or **NdisMCmCloseCallComplete**(in the case of an MCM driver). A call to **Ndis(M)CmCloseCallComplete** causes NDIS to call the client's **ProtocolClCloseCallComplete** function.

The call manager or MCM driver must then initiate deactivation of the VC used for the call by respectively calling **NdisCmDeactivateVc** or **NdisMCmDeactivateVc**(see Deactivating a VC). The creator of the VC (client, call manager, or MCM driver) can then optionally initiate deletion of the VC (see Deleting a VC).

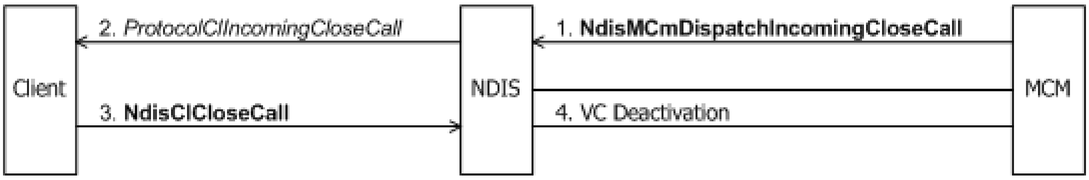
Incoming Request to Close a Call

Article • 12/15/2021

When the remote client closes a call, the local call manager or MCM driver must indicate this incoming request to the local client. To indicate such a request, a call manager calls **NdisCmDispatchIncomingCloseCall** with the *CloseStatus* set to NDIS_STATUS_SUCCESS (see the following figure).



An MCM driver calls **NdisMCmDispatchIncomingCloseCall** to indicate an incoming request to close a call (see the following figure).



Client
2. ProtocolClIncomingCloseCall
3. NdisClCloseCall
NDIS
1. NdisMCmDispatchIncomingCloseCall
4. VC Deactivation
MCM

A call manager or MCM driver also can call **Ndis(M)CmDispatchIncomingCloseCall**:

- From its **ProtocolCmIncomingCallComplete** function if it determines that the connection-oriented client is requesting an unacceptable change in call parameters in response to an incoming call previously that is indicated by the call manager or MCM driver (see Incoming Request to Change Call Parameters).

- If abnormal network conditions force the call manager to tear down active calls.

The call to **Ndis(M)CmDispatchIncomingCloseCall** causes NDIS to call the **ProtocolClIncomingCloseCall** function of the connection-oriented client on that connection. *ProtocolClIncomingCloseCall* should carry out any protocol-determined operations, such as notifying its own client or clients that the connection is being broken. If the call to be closed is a multipoint VC created by the client, *ProtocolClIncomingCloseCall* must call **NdisClDropParty** one or more times until only a single party remains on the VC (see Dropping a Party from a Multipoint Call).

ProtocolClIncomingCloseCall must then call **NdisClCloseCall**(with the handle to the last party on the VC if the VC is a multipoint VC created by the client) to acknowledge that the client will no longer attempt to send or expect to receive data on this particular VC. If the call manager or MCM driver created this VC, *ProtocolClIncomingCloseCall* should return control after it calls **NdisClCloseCall**. The call manager or MCM driver must also deactivate the VC (see Deactivating a VC).

If the client originally created this VC for an outgoing call and *CloseStatus* is NDIS_STATUS_SUCCESS, *ProtocolClIncomingCloseCall* can optionally tear down the VC with **NdisCoDeleteVc**(see Deleting a VC) or reuse the VC for another call. If *CloseStatus* is not NDIS_STATUS_SUCCESS, *ProtocolClIncomingCloseCall* must call **NdisCoDeleteVc**.

If the call manager or MCM driver originally created this VC for an incoming call, the call manager or MCM driver can optionally delete the VC by respectively calling **NdisCoDeleteVc** or **NdisMCmDeleteVc**.

Querying or Setting Information

Article • 12/15/2021

CoNDIS protocol drivers and NDIS can send OID requests to underlying drivers. CoNDIS protocol drivers and miniport call managers (MCMs) can also send OID requests to other protocol drivers.

A connection-oriented client or call manager calls **NdisCoOidRequest** to query or set information that is maintained by another protocol driver on a binding or by the underlying miniport driver.

Before it calls **NdisCoOidRequest**, a client or call manager allocates a buffer for its request and initializes an **NDIS_OID_REQUEST** structure. This structure specifies the type of request (query or set), identifies the information (OID) that is being queried or set, and points to buffers that are used for passing OID data.

If the connection-oriented client or call manager passes a valid *NdisAfHandle* (see Address Families), NDIS calls the **ProtocolCoOidRequest** function of each protocol driver on the binding.

NDIS defines object identifier (OID) values to identify adapter parameters, including operating parameters such as device characteristics, configurable settings, and statistics. For more information about OIDs, see NDIS OIDs.

This section includes the following topics:

CoNDIS Miniport Driver OID Requests

CoNDIS Protocol Driver OID Requests

CoNDIS MCM OID Requests

CoNDIS Miniport Driver OID Requests

Article • 03/14/2023

NDIS calls a CoNDIS miniport driver's **MiniportCoOidRequest** function to submit an OID request to query or set information in the driver. NDIS calls *MiniportCoOidRequest* either on its own behalf or on behalf of an overlying driver that called the **NdisCoOidRequest** function.

NDIS passes *MiniportCoOidRequest* a pointer to an **NDIS_OID_REQUEST** structure that contains the request information. The request structure contains an OID_*Xxx* identifier that indicates the type of request and other members to define the request data.

The **Timeout** member specifies a time-out, in seconds, for the request. NDIS can reset the driver or cancel the request if the time-out expires before the driver completes the request.

The **RequestId** member specifies an optional identifier for the request. Miniport drivers can set the **RequestId** member of a status indication to the value that the driver obtained from the **RequestId** member of an associated OID request. Typically, miniport drivers can ignore this member. If a driver must set this member, the driver must use one of the required values, which are specified in the reference page for the particular OID. For more information about status indications, see CoNDIS Miniport Driver Status Indications.

A miniport driver can complete an OID request synchronously by returning a success or failure status. The driver can complete an OID request asynchronously by returning NDIS_STATUS_PENDING. In this case, the driver must call the **NdisMCoOidRequestComplete** function to complete the operation.

If the *MiniportCoOidRequest* function returns NDIS_STATUS_PENDING, NDIS can call *MiniportCoOidRequest* with another request for the adapter before the pending request is completed. You should note that this is different from the connectionless NDIS interface where all OID requests are serialized.

NDIS can call a miniport driver's *MiniportCancelOidRequest* function to cancel a CoNDIS OID request.

CoNDIS Protocol Driver OID Requests

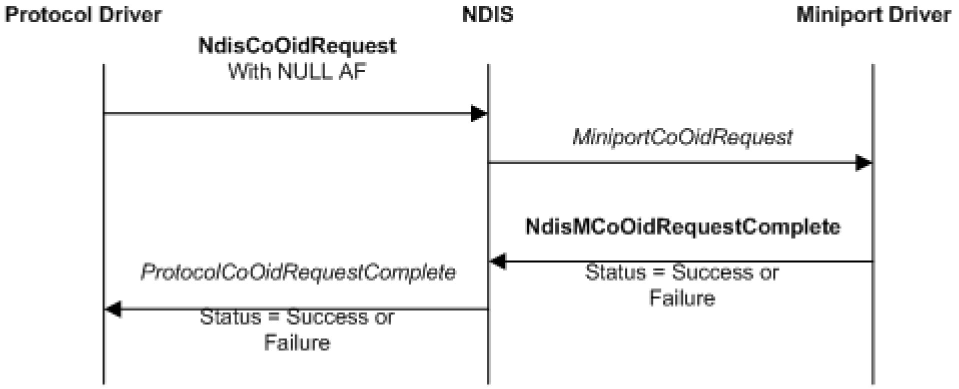
Article • 03/14/2023

CoNDIS protocol drivers, either clients or call managers, can query or set the operating parameters of miniport drivers and other protocol drivers. CoNDIS protocol drivers can also query or set information in miniport call managers (MCMs). For more information about OID requests and MCMs, see CoNDIS MCM OID Requests.

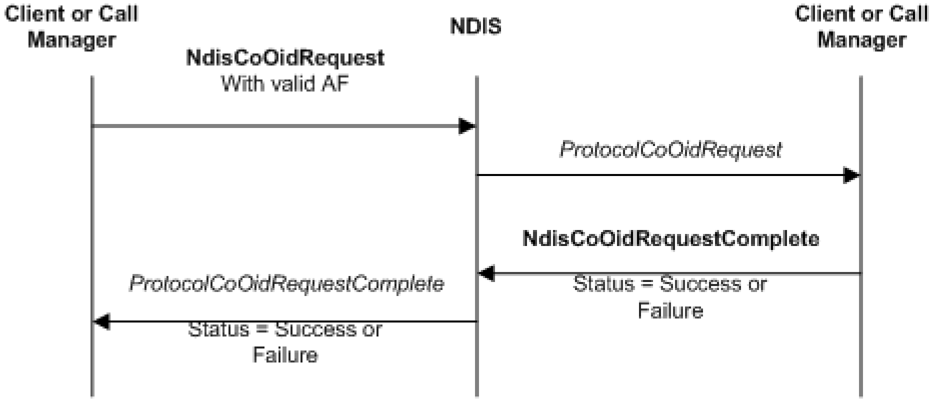
To originate an OID request to an underlying driver, a protocol driver calls the **NdisCoOidRequest** function and sets the address family (AF) handle, at the *NdisAfHandle* parameter, to **NULL**. To originate an OID request to another CoNDIS protocol driver, a protocol driver calls **NdisCoOidRequest** and provides a valid AF handle.

After a protocol driver calls the **NdisCoOidRequest** function, NDIS calls the OID request function of the other driver (an underlying driver or another CoNDIS protocol driver). For miniport drivers, NDIS calls the **MiniportCoOidRequest** function. For protocol drivers, NDIS calls the **ProtocolCoOidRequest** function.

The following figure illustrates an OID request that is directed to a miniport driver.


Protocol Driver
NDIS
Miniport Driver
NdisCoOidRequest
With NULL AF
MiniportCoOidRequest
NdisMCoOidRequestComplete
Status = Success or Failure
ProtocolCoOidRequestComplete
Status = Success or Failure

The following figure illustrates an OID request that is directed to a protocol driver.



To complete synchronously, NdisCoOidRequest returns NDIS_STATUS_SUCCESS or an error status. To complete asynchronously, **NdisCoOidRequest** returns NDIS_STATUS_PENDING.

If **NdisCoOidRequest** returns NDIS_STATUS_PENDING, NDIS calls the ProtocolCoOidRequestComplete function after the other driver completes the OID request by calling the NdisMCoOidRequestComplete function or the NdisCoOidRequestComplete function. In this case, NDIS passes the results of the request at the *OidRequest* parameter of *ProtocolCoOidRequestComplete*. NDIS passes the final status of the request at the *Status* parameter of *ProtocolCoOidRequestComplete*.

If NdisCoOidRequest returns NDIS_STATUS_SUCCESS, it returns the results of a query request in the NDIS_OID_REQUEST structure at the *OidRequest* parameter points. In this case, NDIS does not call the *ProtocolCoOidRequestComplete* function.

If an underlying driver should associate the OID request with a subsequent status indication, the protocol driver should set the **RequestId** and **RequestHandle** members in the NDIS_OID_REQUEST structure. If the underlying driver makes a status indication, the driver sets the **RequestId** member in the NDIS_STATUS_INDICATION structure to the value from the **RequestId** member of the NDIS_OID_REQUEST structure and the **DestinationHandle** member in the NDIS_STATUS_INDICATION structure to the value from the **RequestHandle** member of the NDIS_OID_REQUEST structure.

A driver can call NdisCoOidRequest when a binding is in the *Restarting*, *Running*, *Pausing*, or *Paused* state.

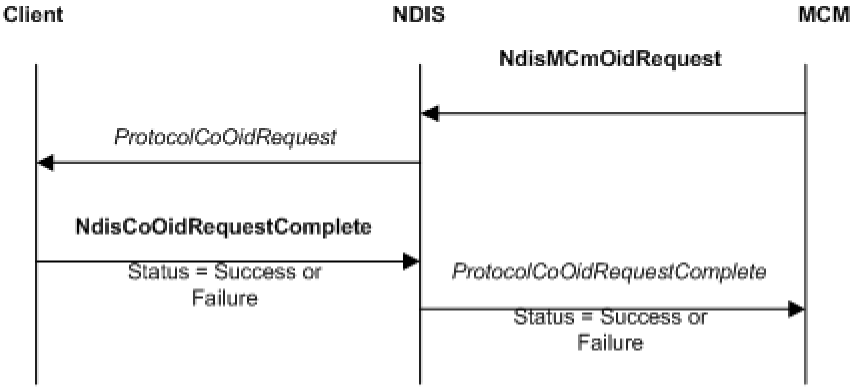
CoNDIS MCM OID Requests

Article • 03/14/2023

Like other CoNDIS call managers, miniport call managers (MCMs) can query or set the operating parameters of CoNDIS client drivers. CoNDIS client drivers can query or set the call manager parameters or the miniport driver parameters of an MCM.

To originate an OID request to a CoNDIS client driver, an MCM calls the NdisMCmOidRequest function.

The following figure illustrates an OID request that an MCM originated.


Client
NDIS
MCM
NdisMCmOidRequest
ProtocolCoOidRequest
NdisCoOidRequestComplete
Status = Success or Failure
ProtocolCoOidRequestComplete
Status = Success or Failure

After an MCM driver calls the NdisMCmOidRequest function, NDIS calls the ProtocolCoOidRequest function of the client driver.

To complete synchronously, **NdisMCmOidRequest** returns NDIS_STATUS_SUCCESS or an error status. To complete asynchronously, **NdisMCmOidRequest** returns NDIS_STATUS_PENDING.

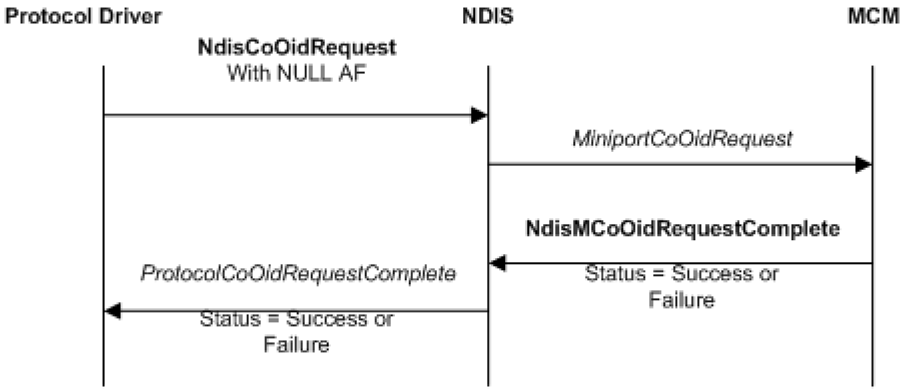
If NdisMCmOidRequest returns NDIS_STATUS_PENDING, NDIS calls the ProtocolCoOidRequestComplete function of the MCM after the client drivers complete the OID request by calling the NdisCoOidRequestComplete function. In this case, NDIS passes the results of the request at the *OidRequest* parameter of *ProtocolCoOidRequestComplete*. NDIS passes the final status of the request at the *Status* parameter of *ProtocolCoOidRequestComplete*.

If **NdisMCmOidRequest** returns NDIS_STATUS_SUCCESS, it returns the results of a query request in the NDIS_OID_REQUEST structure at the *OidRequest* parameter. In this case, NDIS does not call the *ProtocolCoOidRequestComplete* function of the MCM.

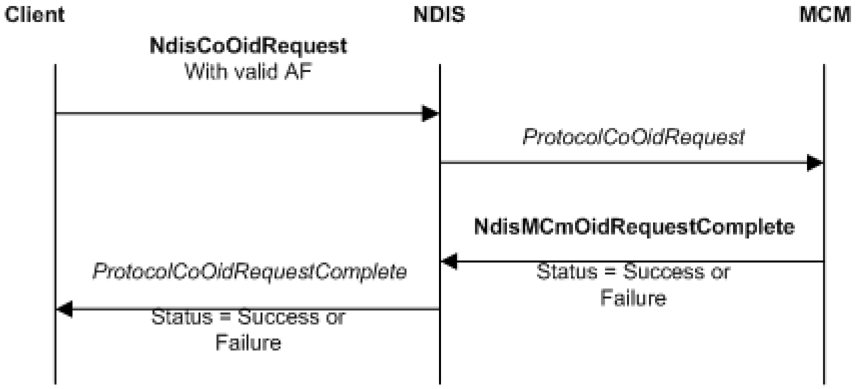
CoNDIS client drivers can query or set the call manager operating parameters or miniport operating parameters of MCMs. To originate an OID request for MCM call manager parameters, a client calls the NdisCoOidRequest function and provides a valid address family (AF) handle at the *NdisAfHandle* parameter. To originate an OID request for MCM miniport parameters, a client calls the **NdisCoOidRequest** function and sets the AF handle to **NULL**.

After a client calls the **NdisCoOidRequest** function, NDIS calls either the MiniportCoOidRequest function or the ProtocolCoOidRequest function of the MCM driver.

The following figure illustrates an OID request for the miniport parameters of the MCM.



The following figure illustrates an OID request for the call manager parameters of the MCM.



To complete synchronously, NdisCoOidRequest returns NDIS_STATUS_SUCCESS or an error status. To complete asynchronously, ProtocolCoOidRequest or MiniportCoOidRequest returns NDIS_STATUS_PENDING.

If *ProtocolCoOidRequest* or **MininportCoOidRequest** returns NDIS_STATUS_PENDING, NDIS calls the ProtocolCoOidRequestComplete function of the client after the MCM completes the OID request by calling the NdisMCoOidRequestComplete or NdisMCmOidRequestComplete function. In this case, NDIS passes the results of the request at the *OidRequest* parameter of *ProtocolCoOidRequestComplete*. NDIS passes the final status of the request at the *Status* parameter of *ProtocolCoOidRequestComplete*.

If NdisCoOidRequest returns NDIS_STATUS_SUCCESS, it returns the results of a query request in the NDIS_OID_REQUEST structure at the *OidRequest* parameter. In this case, NDIS does not call the client's *ProtocolCoOidRequestComplete* function.

Indicating Miniport Driver Status

Article • 12/15/2021

Miniport drivers provide status indications to overlying drivers. The CoNDIS status indication functions are similar to the connectionless status indication functions.

To report a change in the status of a connection-oriented NIC or a change in the status of a particular VC active on the NIC, a connection-oriented miniport driver calls **NdisMCoIndicateStatusEx**. If the miniport driver is reporting a change in the status of a particular VC, it supplies an *NdisVcHandle* that identifies the VC.

This section includes the following topics:

CoNDIS Miniport Driver Status Indications

Handling Status Indications in a CoNDIS Protocol Driver

CoNDIS Miniport Driver Status Indications

Article • 03/14/2023

Miniport drivers call the NdisMCoIndicateStatusEx function to report a change in the status of a miniport adapter. The miniport driver passes **NdisMCoIndicateStatusEx** a pointer to an NDIS_STATUS_INDICATION structure that contains the status information.

The status indication includes information to identify the type of status and a reason for the status change.

The miniport driver should set the **SourceHandle** member of the NDIS_STATUS_INDICATION structure to the handle that NDIS passed to the *MiniportAdapterHandle* parameter of the *MiniportInitializeEx* function. If the status indication is associated with an OID request, the miniport driver can set the **DestinationHandle** and **RequestId** members of NDIS_STATUS_INDICATION so that NDIS can provide the status indication to a specific protocol binding. For more information about OID requests, see CoNDIS Miniport Driver OID Requests.

Handling Status Indications in a CoNDIS Protocol Driver

Article • 03/14/2023

Protocol drivers must supply a **ProtocolCoStatusEx** function that NDIS calls when an underlying driver reports status.

NDIS calls a protocol driver's *ProtocolCoStatusEx* function after an underlying driver calls a status indication function (for example, **NdisMCoIndicateStatusEx**). For more information about indicating status from a miniport driver, see CoNDIS Miniport Driver Status Indications.

If the status indication is associated with an OID request, the underlying driver can set the **DestinationHandle** and **RequestId** members of the **NDIS_STATUS_INDICATION** structure that contains the status information so that NDIS can provide the status indication to a specific protocol binding. For more information about OID requests, see CoNDIS Protocol Driver OID Requests.

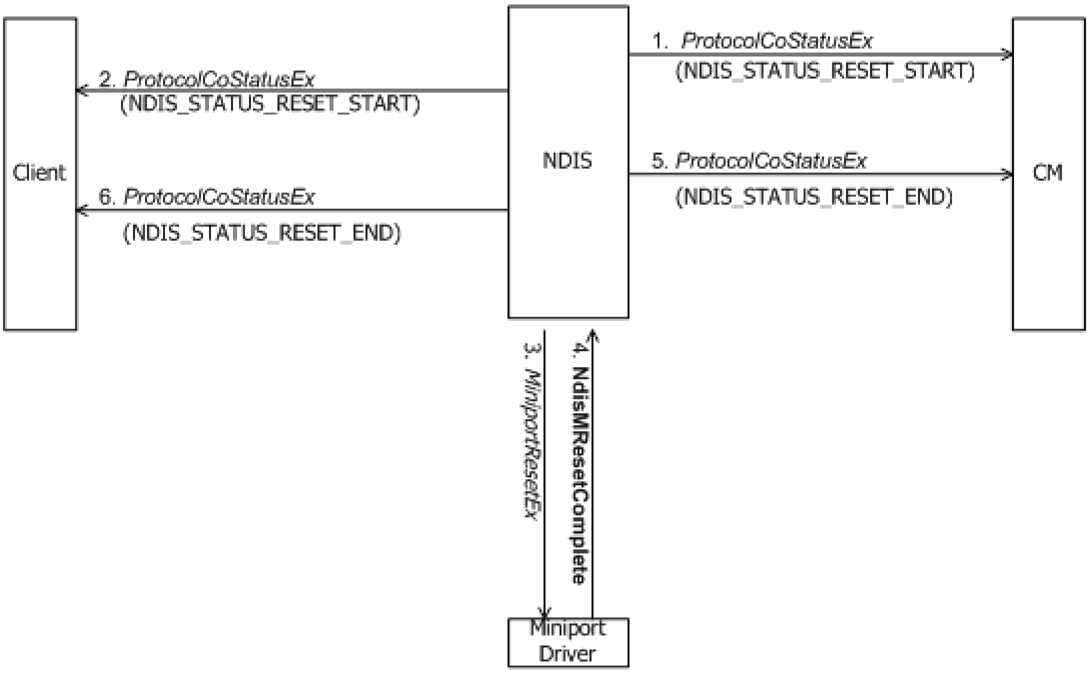
Reset

Article • 12/15/2021

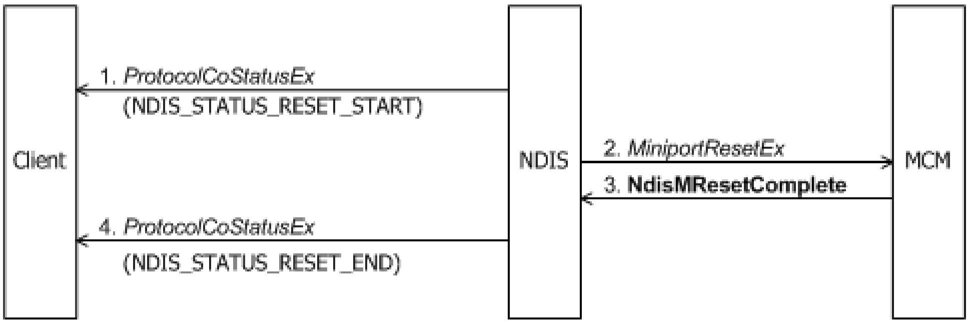
NDIS might call a miniport driver's or MCM driver's *MiniportResetEx* function to reset a NIC.

Note AF, SAP, and VC handles that are active and valid before a reset are active and valid after the reset.

The following figure shows a client issuing a reset request to a miniport driver.



The next figure shows a client issuing a reset request to an MCM driver.



When an underlying connection-oriented driver is resetting a NIC, NDIS notifies each bound protocol by calling the protocol's **ProtocolCoStatusEx** function with NDIS_STATUS_RESET_START.

NDIS will not accept protocol-initiated sends and requests to a miniport driver or MCM driver while the miniport driver's or MCM driver's NIC is being reset. While a reset is in progress, a protocol driver must not attempt to send packets to the miniport driver with

NdisCoSendNetBufferLists or request information from the miniport driver with NdisCoOidRequest.

MiniportResetEx performs any device-dependent actions that are required to reset the NIC. *MiniportResetEx* can complete synchronously, or it can complete asynchronously with a call to NdisMResetComplete:

- If the reset completes synchronously, NDIS calls each bound protocol's *ProtocolCoStatusEx* function with NDIS_STATUS_RESET_END.

- If the reset completes asynchronously, NDIS calls each bound protocol's *ProtocolCoStatusEx* function with NDIS_STATUS_RESET_END.

Introduction to NDIS Network Interfaces

Article • 03/14/2023

To support the management information base (MIB), NDIS manages a collection of network interface information for the local computer. NDIS interface providers provide information about some network interfaces to NDIS. NDIS provides a proxy interface provider that registers interfaces and handles interface provider requests for miniport adapters and filter modules. Therefore, no NDIS drivers are required to be network interface providers.

However, all NDIS network driver types can register as interface providers. Such drivers register network interfaces and provide callback functions to respond to interface OID requests. NDIS interface providers typically provide information about interfaces that are not directly accessible to NDIS and are not supported by the NDIS proxy interface provider. For example, a MUX intermediate driver can have internal interfaces between its virtual miniports and underlying adapters.

This section includes:

Overview of NDIS Network Interfaces

Registering as an Interface Provider

Managing NDIS Network Interfaces

Handling OID Query and Set Requests in an NDIS Interface Provider

Mapping of NDIS Network Interfaces to NDIS OIDs

Overview of NDIS Network Interfaces

Article • 03/14/2023

NDIS network interfaces provide a consistent representation for all of the various network interfaces that Microsoft Windows supports. Without the NDIS network interface services, all network interfaces are not visible to the computer administrator, and the interfaces that are visible do not necessarily support the management information base (MIB). Also, NDIS network interface services enable the layering relationships between interfaces to be visible to the administrator.

This section includes:

NDIS Network Interface Services

NDIS Network Interface Architecture

NDIS Interface Provider Operations

NDIS Interface Types

ⓘ **Note**

The NDIS 6.0 Network Interfaces section refers to many Request for Comments (RFCs) from the Internet Engineering Task Force (IETF). To view an IETF RFC, visit the **IETF Request for Comments** Web site, and search for the RFC in the **RFC number** box under **IETF repository retrieval**.

NDIS Network Interface Services

Article • 03/14/2023

The NDIS network interfaces programming interface provides services to:

- Generate a locally unique identifier (NET_LUID) for each interface. NET_LUID values:
  - Must persist when the computer restarts. Interface providers must make NET_LUIDs persistent even if the associated interface is not persistent. For example, this persistence allows the interface provider to free the NET_LUID index if there is a computer power failure.
  - Must be associated with an interface type (*IfType* in RFC 2863).
  - Must be unique on a local computer.
  - Can be converted to a text representation because a NET_LUID is equivalent to the interface name (*ifName* in RFC 2863).

- Generate a locally unique interface index (a 24-bit value that is also referred to as *IfIndex*) for each interface. *IfIndex* values have the following properties:
  - Low numbers are preferred. For example, NDIS reuses the lowest available interface index.
  - *IfIndex* values do not persist when the computer restarts.
  - There is a one-to-one correspondence between a NET_LUID value and an *IfIndex* value.

- Map between interface indexes, NET_LUID values, and "friendly names" (For example, a friendly name as displayed in the network connections folder).

- Define the layering order of interfaces in a driver stack.

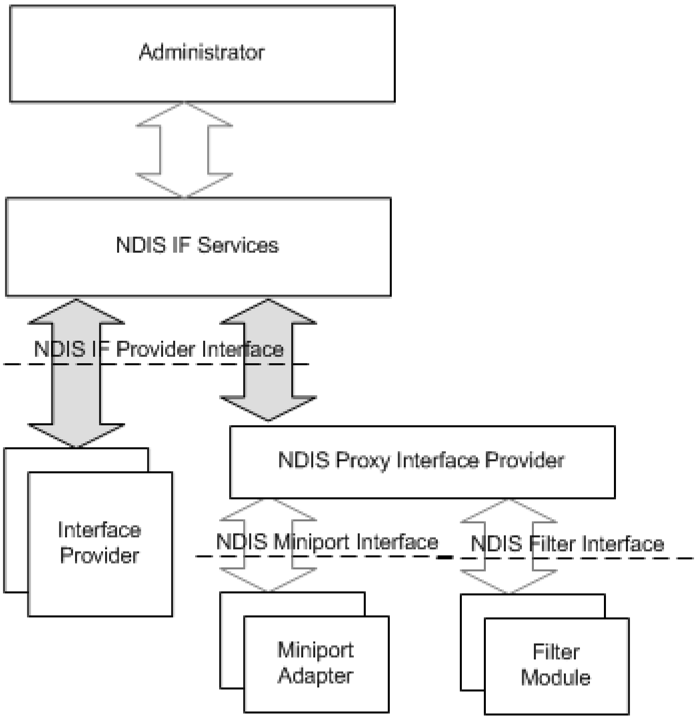
- Query and set interface properties and tables that NDIS drivers manage and that RFCs 2863 and 2864 specify.

NDIS Network Interface Architecture

Article • 03/14/2023

NDIS provides a set of services to support network interfaces and interface stacks. In the WDK, this set of services is referred to as *NDIS network interface (NDISIF)* services.

The following figure shows the NDISIF architecture for NDIS 6.0 and later.



The NDISIF components of the architecture include:

- *NDIS IF Services*
  An NDIS component that handles registration of interface providers and interfaces, implements OID query and set services for interface providers, and supplies other NDISIF services.
- *NDIS IF provider interface*
  An interface that the *NDIS IF Services* component supplies to enable NDIS drivers to implement interface providers.
- *NDIS proxy interface provider*
  An NDIS component that implements the NDISIF provider services on behalf of NDIS miniport drivers (for each miniport adapter) and filter drivers (for each filter module).
- *Interface provider*
  An NDIS driver that provides the NDISIF provider services for interfaces that the *NDIS proxy interface provider* component cannot serve. For example, a MUX intermediate driver can have internal interfaces between its virtual miniports and underlying adapters.

The NDIS proxy interface provider uses the standard NDIS miniport driver and NDIS filter driver interfaces to provide NDISIF services for miniport adapters and filter modules. Therefore, miniport drivers and filter drivers are not required to register as interface providers.

NDIS Interface Provider Operations

Article • 03/14/2023

All NDIS drivers can register as interface providers. Whenever a driver (or the NDIS proxy interface provider) detects a new interface that is being introduced to the computer, it allocates a **NET_LUID** index, registers the interface, and retains the associated NET_LUID value in persistent storage (such as the registry). The following list describes several examples of how a new interface can be introduced to a computer:

- Installing a network adapter, either a virtual adapter for an intermediate driver or a physical adapter. In this case, the NDIS proxy interface provider manages the interface.

- Attaching a filter module. In this case, the NDIS proxy interface provider manages the interface.

- MUX intermediate driver internal bindings. The MUX intermediate driver should implement NDIS provider services to handle this case because the internal interfaces are not visible to NDIS.

When the computer subsequently restarts, the interface provider should not allocate a new **NET_LUID** for the same interface if the interface is persistent; instead, the interface provider should use the previously stored NET_LUID value to register the same interface. Also, even if the interface is not persistent, the interface provider must free the NET_LUID index if there is a computer power failure. Therefore, the interface provider should store the NET_LUID in persistent storage (for example, the registry).

If an interface provider detects that an interface is being shut down, it should deregister the interface.

Note The NDIS proxy provider deregisters interfaces for miniport adapters when they are uninstalled and filter modules when they are detached.

If an interface provider detects that an interface is being removed completely (for example, the NDIS proxy provider is notified that a miniport adapter is being uninstalled), the interface provider deregisters the interface and releases the NET_LUID index. The NDIS proxy provider also releases the NET_LUID index when a filter module is detached.

During run time, interface providers handle OID requests for the interfaces that they registered. The NDIS proxy interface provider might issue OID requests to underlying drivers to obtain interface information.

NDIS Interface Types

Article • 03/14/2023

NDIS interface types correspond to the *IfType* object that is defined in the management information base (MIB). These interface types are used in the **IfType** members and *IfType* parameters for many NDIS structures and functions.

NDIS interface types are registered with the Internet Assigned Numbers Authority (IANA), which publishes a list of interface types periodically in the Assigned Numbers RFC, or in a derivative of it that is specific to Internet network management number assignments. For more information about the IANA IfType *definitions*, see IANA ifType MIB Definitions. For more information about the IANA, see the IANA Web site.

The following table describes IfType values.

Name	Value	Comment
IF_TYPE_OTHER	1	Use this value if none of the other IF_TYPE_*Xxx* types applies.
IF_TYPE_REGULAR_1822	2	
IF_TYPE_HDH_1822	3	
IF_TYPE_DDN_X25	4	
IF_TYPE_RFC877_X25	5	
IF_TYPE_ETHERNET_CSMACD	6	
IF_TYPE_IS088023_CSMACD	7	
IF_TYPE_ISO88024_TOKENBUS	8	
IF_TYPE_ISO88025_TOKENRING	9	
IF_TYPE_ISO88026_MAN	10	
IF_TYPE_STARLAN	11	
IF_TYPE_PROTEON_10MBIT	12	
IF_TYPE_PROTEON_80MBIT	13	
IF_TYPE_HYPERCHANNEL	14	
IF_TYPE_FDDI	15	
IF_TYPE_LAP_B	16	

IF_TYPE_SDLC	17	
IF_TYPE_DS1	18	DS1-MIB
IF_TYPE_E1	19	Obsolete. See DS1-MIB.
IF_TYPE_BASIC_ISDN	20	
IF_TYPE_PRIMARY_ISDN	21	
IF_TYPE_PROP_POINT2POINT_SERIAL	22	Proprietary serial
IF_TYPE_PPP	23	
IF_TYPE_SOFTWARE_LOOPBACK	24	
IF_TYPE_EON	25	CLNP over IP
IF_TYPE_ETHERNET_3MBIT	26	
IF_TYPE_NSIP	27	XNS over IP
IF_TYPE_SLIP	28	Generic Slip
IF_TYPE_ULTRA	29	ULTRA Technologies
IF_TYPE_DS3	30	DS3-MIB
IF_TYPE_SIP	31	SMDS and coffee
IF_TYPE_FRAMERELAY	32	DTE only
IF_TYPE_RS232	33	
IF_TYPE_PARA	34	Parallel port
IF_TYPE_ARCNET	35	
IF_TYPE_ARCNET_PLUS	36	
IF_TYPE_ATM	37	ATM cells
IF_TYPE_MIO_X25	38	
IF_TYPE_SONET	39	SONET or SDH
IF_TYPE_X25_PLE	40	
IF_TYPE_ISO88022_LLC	41	
IF_TYPE_LOCALTALK	42	
IF_TYPE_SMDS_DXI	43	

IF_TYPE_FRAMERELAY_SERVICE	44	FRNETSERV-MIB
IF_TYPE_V35	45	
IF_TYPE_HSSI	46	
IF_TYPE_HIPPI	47	
IF_TYPE_MODEM	48	Generic modem
IF_TYPE_AAL5	49	AAL5 over ATM
IF_TYPE_SONET_PATH	50	
IF_TYPE_SONET_VT	51	
IF_TYPE_SMDS_ICIP	52	SMDS InterCarrier interface
IF_TYPE_PROP_VIRTUAL	53	Proprietary virtual/internal
IF_TYPE_PROP_MULTIPLEXOR	54	Proprietary multiplexing
IF_TYPE_IEEE80212	55	100BaseVG
IF_TYPE_FIBRECHANNEL	56	
IF_TYPE_HIPPIINTERFACE	57	
IF_TYPE_FRAMERELAY_INTERCONNECT	58	Obsolete. Use 32 or 44 instead.
IF_TYPE_AFLANE_8023	59	ATM-emulated LAN-for 802.3
IF_TYPE_AFLANE_8025	60	ATM-emulated LAN for 802.5
IF_TYPE_CCTEMUL	61	ATM-emulated circuit
IF_TYPE_FASTETHER	62	Fast Ethernet (100BaseT)
IF_TYPE_ISDN	63	ISDN and X.25
IF_TYPE_V11	64	CCITT V.11/X.21
IF_TYPE_V36	65	CCITT V.36
IF_TYPE_G703_64K	66	CCITT G703 at 64Kbps
IF_TYPE_G703_2MB	67	Obsolete. See DS1-MIB.
IF_TYPE_QLLC	68	SNA QLLC
IF_TYPE_FASTETHER_FX	69	Fast Ethernet (100BaseFX)
IF_TYPE_CHANNEL	70	
IF_TYPE_IEEE80211	71	Radio spread spectrum

IF_TYPE_IBM370PARCHAN	72	IBM System 360/370 OEMI channel
IF_TYPE_ESCON	73	IBM Enterprise Systems connection
IF_TYPE_DLSW	74	Data link switching
IF_TYPE_ISDN_S	75	ISDN S/T interface
IF_TYPE_ISDN_U	76	ISDN U interface
IF_TYPE_LAP_D	77	Link access protocol D
IF_TYPE_IPSWITCH	78	IP switching objects
IF_TYPE_RSRB	79	Remote source route bridging
IF_TYPE_ATM_LOGICAL	80	ATM logical port
IF_TYPE_DS0	81	Digital signal level 0
IF_TYPE_DS0_BUNDLE	82	Group of ds0s on the same ds1
IF_TYPE_BSC	83	Bisynchronous protocol
IF_TYPE_ASYNC	84	Asynchronous protocol
IF_TYPE_CNR	85	Combat net radio
IF_TYPE_ISO88025R_DTR	86	ISO 802.5r DTR
IF_TYPE_EPLRS	87	Ext Pos Loc Report Sys
IF_TYPE_ARAP	88	Appletalk remote access protocol
IF_TYPE_PROP_CNLS	89	Proprietary connectionless protocol
IF_TYPE_HOSTPAD	90	CCITT-ITU X.29 PAD protocol
IF_TYPE_TERMPAD	91	CCITT-ITU X.3 PAD facility
IF_TYPE_FRAMERELAY_MPI	92	Multiproto interconnect over FR
IF_TYPE_X213	93	CCITT-ITU X213
IF_TYPE_ADSL	94	Asymmetric digital subscriber loop
IF_TYPE_RADSL	95	Rate-adapt digital subscriber loop
IF_TYPE_SDSL	96	Symmetric digital subscriber loop
IF_TYPE_VDSL	97	Very H-Speed digital subscriber loop
IF_TYPE_ISO88025_CRFPRINT	98	ISO 802.5 CRFP

IF_TYPE_MYRINET	99	Myricom Myrinet
IF_TYPE_VOICE_EM	100	Voice recEive and transMit
IF_TYPE_VOICE_FXO	101	Voice foreign exchange office
IF_TYPE_VOICE_FXS	102	Voice foreign exchange station
IF_TYPE_VOICE_ENCAP	103	Voice encapsulation
IF_TYPE_VOICE_OVERIP	104	Voice over IP encapsulation
IF_TYPE_ATM_DXI	105	ATM DXI
IF_TYPE_ATM_FUNI	106	ATM FUNI
IF_TYPE_ATM_IMA	107	ATM IMA
IF_TYPE_PPPMULTILINKBUNDLE	108	PPP multilink bundle
IF_TYPE_IPOVER_CDLC	109	IBM ipOverCdlc
IF_TYPE_IPOVER_CLAW	110	IBM common link access to workstation
IF_TYPE_STACKTOSTACK	111	IBM stackToStack
IF_TYPE_VIRTUALIPADDRESS	112	IBM VIPA
IF_TYPE_MPC	113	IBM multi-proto channel support
IF_TYPE_IPOVER_ATM	114	IBM ipOverAtm
IF_TYPE_ISO88025_FIBER	115	ISO 802.5j Fiber Token Ring
IF_TYPE_TDLC	116	IBM twinaxial data link control
IF_TYPE_GIGABITETHERNET	117	
IF_TYPE_HDLC	118	
IF_TYPE_LAP_F	119	
IF_TYPE_V37	120	
IF_TYPE_X25_MLP	121	Multi-link protocol
IF_TYPE_X25_HUNTGROUP	122	X.25 hunt group
IF_TYPE_TRANSPHDLC	123	
IF_TYPE_INTERLEAVE	124	Interleave channel
IF_TYPE_FAST	125	Fast channel

IF_TYPE_IP	126	IP (for APPN HPR in IP networks)
IF_TYPE_DOCSCABLE_MACLAYER	127	CATV MAC layer
IF_TYPE_DOCSCABLE_DOWNSTREAM	128	CATV downstream interface
IF_TYPE_DOCSCABLE_UPSTREAM	129	CATV upstream interface
IF_TYPE_A12MPPSWITCH	130	Avalon parallel processor
IF_TYPE_TUNNEL	131	Encapsulation interface
IF_TYPE_COFFEE	132	Coffee pot
IF_TYPE_CES	133	Circuit emulation service
IF_TYPE_ATM_SUBINTERFACE	134	ATM sub-interface
IF_TYPE_L2_VLAN	135	Layer 2 virtual LAN using 802.1Q
IF_TYPE_L3_IPVLAN	136	Layer 3 virtual LAN using IP
IF_TYPE_L3_IPXVLAN	137	Layer 3 virtual LAN using IPX
IF_TYPE_DIGITALPOWERLINE	138	IP over power lines
IF_TYPE_MEDIAMAILOVERIP	139	Multimedia mail over IP
IF_TYPE_DTM	140	Dynamic synchronous transfer mode
IF_TYPE_DCN	141	Data communications network
IF_TYPE_IPFORWARD	142	IP forwarding interface
IF_TYPE_MSDSL	143	Multi-rate symmetric DSL
IF_TYPE_IEEE1394	144	IEEE 1394 high performance serial bus
IF_TYPE_IF_GSN	145	
IF_TYPE_DVBRCC_MACLAYER	146	
IF_TYPE_DVBRCC_DOWNSTREAM	147	
IF_TYPE_DVBRCC_UPSTREAM	148	
IF_TYPE_ATM_VIRTUAL	149	
IF_TYPE_MPLS_TUNNEL	150	
IF_TYPE_SRP	151	

IF_TYPE_VOICEOVERATM	152
IF_TYPE_VOICEOVERFRAMERELAY	153
IF_TYPE_IDSL	154
IF_TYPE_COMPOSITELINK	155
IF_TYPE_SS7_SIGLINK	156
IF_TYPE_PROP_WIRELESS_P2P	157
IF_TYPE_FR_FORWARD	158
IF_TYPE_RFC1483	159
IF_TYPE_USB	160
IF_TYPE_IEEE8023AD_LAG	161
IF_TYPE_BGP_POLICY_ACCOUNTING	162
IF_TYPE_FRF16_MFR_BUNDLE	163
IF_TYPE_H323_GATEKEEPER	164
IF_TYPE_H323_PROXY	165
IF_TYPE_MPLS	166
IF_TYPE_MF_SIGLINK	167
IF_TYPE_HDSL2	168
IF_TYPE_SHDSL	169
IF_TYPE_DS1_FDL	170
IF_TYPE_POS	171
IF_TYPE_DVB_ASI_IN	172
IF_TYPE_DVB_ASI_OUT	173
IF_TYPE_PLC	174
IF_TYPE_NFAS	175
IF_TYPE_TR008	176
IF_TYPE_GR303_RDT	177
IF_TYPE_GR303_IDT	178
IF_TYPE_ISUP	179

IF_TYPE_PROP_DOCS_WIRELESS_MACLAYER	180	
IF_TYPE_PROP_DOCS_WIRELESS_DOWNSTREAM	181	
IF_TYPE_PROP_DOCS_WIRELESS_UPSTREAM	182	
IF_TYPE_HIPERLAN2	183	
IF_TYPE_PROP_BWA_P2MP	184	
IF_TYPE_SONET_OVERHEAD_CHANNEL	185	
IF_TYPE_DIGITAL_WRAPPER_OVERHEAD_CHANNEL	186	
IF_TYPE_AAL2	187	
IF_TYPE_RADIO_MAC	188	
IF_TYPE_ATM_RADIO	189	
IF_TYPE_IMT	190	
IF_TYPE_MVL	191	
IF_TYPE_REACH_DSL	192	
IF_TYPE_FR_DLCI_ENDPT	193	
IF_TYPE_ATM_VCI_ENDPT	194	
IF_TYPE_OPTICAL_CHANNEL	195	
IF_TYPE_OPTICAL_TRANSPORT	196	
IF_TYPE_WWANPP	243	Mobile Broadband devices based on GSM technology
IF_TYPE_WWANPP2	244	Mobile Broadband devices based on CDMA technology

Registering as an Interface Provider

Article • 05/20/2024

An NDIS interface provider is a software component that provides and manages information for NDIS network interfaces. For example, protocol drivers, MUX intermediate drivers, and NDIS are interface providers. (NDIS provides a proxy interface provider for miniport drivers and filter drivers. However, miniport drivers and filter drivers can also be interface providers.) Each interface provider calls the **NdisIfRegisterProvider** function to register as a network interface provider.

If the call to **NdisIfRegisterProvider** succeeds, **NdisIfRegisterProvider** returns a handle at the address that the *pNdisProviderHandle* parameter specifies. The caller uses this handle in subsequent calls (for example, to register interfaces). The *ProviderCharacteristics* parameter points to an **NDIS_IF_PROVIDER_CHARACTERISTICS** structure that contains the provider's entry points to handle OID query and set requests. NDIS_IF_PROVIDER_CHARACTERISTICS includes the following query and set functions:

- **ProviderQueryObject**
- **ProviderSetObject**

For more information about interface provider query and set handlers, see Handling OID Query and Set Requests in an NDIS Interface Provider.

NDIS drivers can call the **NdisIfDeregisterProvider** function to deregister as a network interface provider. For example, NDIS drivers should deregister as an interface providers when they are unloaded. An interface provider must ensure that it does not have any interfaces registered before it calls **NdisIfDeregisterProvider**. The provider must not use the provider handle that it passed at the *NdisProviderHandle* parameter of **NdisIfDeregisterProvider** after it calls **NdisIfDeregisterProvider**.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Managing NDIS Network Interfaces

Article • 03/14/2023

NDIS network interface providers register network interfaces with NDIS. Before registering an interface, an interface provider obtains a **NET_LUID** value for that interface. NDIS assigns an interface index (*IfIndex* in RFC 2863) to an interface when it is registered.

NDIS also provides services that drivers can use to manage entries in the interface stack table (*ifStackTable* in RFC 2863).

This section includes:

NET_LUID Value

Using a NET_LUID Index

Registering a Network Interface

Deregistering a Network Interface

Mapping a NET_LUID Value to an Interface Index

NET_LUID Values for Miniport Adapters and Filter Modules

Maintaining a Network Interface Stack

NET_LUID Value

Article • 12/15/2021

A **NET_LUID** value is a 64-bit value that identifies an NDIS network interface. The NET_LUID data type is a union that can provide access to the NET_LUID value as a single 64-bit value or as a structure that contains a NET_LUID index and an interface type.

The **NetLuidIndex** member of the NET_LUID union is a 24-bit NET_LUID index that NDIS allocates when an interface provider calls the **NdisIfAllocateNetLuidIndex** function. NDIS and interface providers use this index to distinguish between multiple interfaces that have the same interface type. Therefore, this index is unique within a local computer.

The **IfType** member of the **NET_LUID** union is a 16-bit value that contains an Internet Assigned Numbers Authority (IANA)-defined interface type. For a list of valid interface types, see NDIS Interface Types.

The NET_LUID data type is equivalent to the *ifName* object in RFC 2863, because NDIS derives the *ifName* string from a NET_LUID value.

To create a NET_LUID value, an interface provider calls the **NdisIfAllocateNetLuidIndex** function to allocate a NET_LUID index and then calls the **NDIS_MAKE_NET_LUID** macro to build the NET_LUID value. For more information about creating NET_LUID values, see Using NET_LUID Indexes.

Using a NET_LUID Index

Article • 12/15/2021

NDIS provides functions to allocate and free the NET_LUID indexes that are required to create NET_LUID values. An NDIS interface provider must allocate a NET_LUID value to register an interface.

To allocate a NET_LUID index, an interface provider calls the NdisIfAllocateNetLuidIndex function. After allocating the index, the interface provider calls the NDIS_MAKE_NET_LUID macro to build the NET_LUID value. To free a NET_LUID index, an interface provider calls the NdisIfFreeNetLuidIndex function.

NdisIfAllocateNetLuidIndex attempts to allocate a 24-bit value that is associated with the interface type that the caller specified at the *IfType* parameter and that is unique to the local computer. If the index allocation succeeds, **NdisIfAllocateNetLuidIndex** returns NDIS_STATUS_SUCCESS and provides a NET_LUID index at the address that is provided in the *pNetLuidIndex* parameter. If NDIS is not able to find a free NET_LUID index, **NdisIfAllocateNetLuidIndex** returns NDIS_STATUS_RESOURCES. **NdisIfAllocateNetLuidIndex** can return other NDIS status values to indicate internal errors within NDIS. NDIS records the allocation of this index for when the computer subsequently restarts. NDIS will not use a particular index for future callers, even after the computer restarts, until the interface provider that allocated that index calls the **NdisIfFreeNetLuidIndex** function for that index.

NdisIfFreeNetLuidIndex frees a previously allocated NET_LUID index so that NDIS can possibly reallocate that index to another interface. The caller must pass in the same interface type at *IfType* that the caller used when it called NdisIfAllocateNetLuidIndex to allocate the NET_LUID index. If the free operation succeeds, **NdisIfFreeNetLuidIndex** returns NDIS_STATUS_SUCCESS. If the call to **NdisIfFreeNetLuidIndex** fails, the interface provider should remove any information that it saved in persistent storage that is related to the NET_LUID index. Removing the information will ensure that the provider does not keep trying to free an index that is already freed after every computer restart. After calling **NdisIfFreeNetLuidIndex**, the caller must not use the NET_LUID value again unless it calls **NdisIfAllocateNetLuidIndex** again for the same interface type and receives the same NET_LUID index that it freed.

To register a network interface, an interface provider must pass a valid NET_LUID value to the NdisIfRegisterInterface function. For more information about registering network interfaces, see Registering a Network Interface.

Registering a Network Interface

Article • 12/15/2021

Whenever a computer restarts, NDIS starts with an empty list of registered network interfaces. An interface provider calls the NdisIfRegisterInterface function whenever it starts or detects an interface and its NET_LUID value is known. The mechanism for starting or detecting an interface is application-specific.

NdisIfRegisterInterface returns NDIS_STATUS_SUCCESS only if NDIS successfully adds the specified interface to its list of known interfaces on the computer. In this case, **NdisIfRegisterInterface** returns an interface index at the *pIfIndex* parameter. However, a call to **NdisIfRegisterInterface** does not imply that the interface is active; this call guarantees only that the interface exists. **NdisIfRegisterInterface** returns NDIS_STATUS_RESOURCES if NDIS does not have sufficient resources available to register the interface. **NdisIfRegisterInterface** can also return other NDIS status values.

The *ProviderIfContext* parameter of **NdisIfRegisterInterface** contains a handle to the caller's context area for the interface--this handle is passed to the caller's OID query and set functions. The *pIfInfo* parameter contains a pointer to a NET_IF_INFORMATION structure that includes information about the interface.

The following topics provide more information about network interfaces that **NdisIfRegisterInterface** successfully registers:

Allocating an Interface Index

Network Interface Information

Allocating an Interface Index

Article • 12/15/2021

If an interface provider successfully registers an interface by calling the NdisIfRegisterInterface function, NDIS allocates an interface index for that interface and returns the index value at the location that the *pIfIndex* parameter specifies. The interface index is a 16-bit value that is unique on the local computer. NDIS does not guarantee that it will allocate the same interface index when an interface provider registers the same NET_LUID value after the computer restarts. The interface index value zero (NET_IFINDEX_UNSPECIFIED) is reserved, and NDIS does not assign it to any interface.

Network Interface Information

Article • 12/15/2021

An interface provider supplies information about each registered interface by using the following data structures.

- **NET_IF_INFORMATION**
- **NDIS_INTERFACE_INFORMATION**

To register an interface, a provider passes a pointer to an initialized NET_IF_INFORMATION structure to the **NdisIfRegisterInterface** function.

NDIS interface providers provide an **NDIS_INTERFACE_INFORMATION** structure in response to a query of the OID_GEN_INTERFACE_INFO OID.

NDIS can also query providers with other OIDs. For more information about NDIS provider OIDs, see NDIS Network Interface to OID Mapping. For more information about handling OID requests in interface providers, see Handling OID Query and Set Requests in an NDIS Interface Provider.

Deregistering a Network Interface

Article • 12/15/2021

An NDIS interface provider calls the NdisIfDeregisterInterface function to indicate that a specified interface should be removed from the list of known interfaces on the computer, for example, because the interface has been uninstalled. Other reasons for deregistering interfaces are application-specific. To promote good resource management, interface providers should always deregister interfaces that are no longer useful.

NdisIfDeregisterInterface releases the interface index that is associated with the specified interface. NDIS can reassign the index to an interface that is registered in the future. However the NET_LUID index that is associated with the corresponding NET_LUID value is not reclaimed--if necessary, the interface provider can release the NET_LUID index by calling the NdisIfFreeNetLuidIndex function.

Note The NDIS proxy provider deregisters interfaces for miniport adapters when they are uninstalled and filter modules when they are detached.

Mapping a NET_LUID Value to an Interface Index

Article • 12/15/2021

NDIS provides services to obtain the interface index for a given NET_LUID value, and vice versa. Note that the NET_LUID value is the persistent identification for an interface, and the interface index that corresponds to a particular NET_LUID value can change even if the computer does not restart (for example, when a filter module is attached and detached because the associated miniport adapter was disabled and reenabled).

NDIS provides the following mapping functions:

- NdisIfGetInterfaceIndexFromNetLuid
- NdisIfGetNetLuidFromInterfaceIndex

These functions return NDIS_STATUS_INTERFACE_NOT_FOUND if the given NET_LUID or interface index is not present in the list of registered interfaces.

NET_LUID Values for Miniport Adapters and Filter Modules

Article • 12/15/2021

NDIS registers interfaces on behalf of miniport drivers (for each miniport adapter) and filter drivers (for each filter module). A protocol driver can query NDIS for the interface index and NET_LUID value of a miniport adapter that the driver is bound to by using its binding handle. For example, the protocol-driver lower edge of a MUX intermediate driver might obtain the NET_LUID values to specify the layering order of its internal interfaces.

A protocol driver passes a binding handle at the *NdisBindingHandle* parameter to the NdisIfQueryBindingIfIndex function and receives interface indexes and NET_LUID values for the interfaces at the top and bottom of a filter stack. Alternatively, the protocol driver can retrieve these values in the NDIS_BIND_PARAMETERS structure.

A miniport driver can also query NDIS for the interface index of a miniport adapter by using the NDIS miniport adapter handle. A miniport driver receives an interface index and a NET_LUID value in the NDIS_MINIPORT_INIT_PARAMETERS structure.

A filter driver gets an interface index and a NET_LUID value for a filter module in the NDIS_FILTER_ATTACH_PARAMETERS structure.

Maintaining a Network Interface Stack

Article • 12/15/2021

NDIS provides services to maintain the interface stack table (*ifStackTable* in RFC 2863). NDIS maintains the stack table for NDIS miniport adapters, NDIS 5.*x* filter intermediate drivers, and NDIS filter modules. NDIS also provides services to enable NDIS drivers to add and delete entries in this table. For MUX intermediate drivers, NDIS does not have access to the relationship between the virtual miniport interface and the protocol lower interface. Therefore, NDIS 6.0 MUX intermediate drivers must specify these internal interface relationships.

To define a stack relationship between two interfaces, any NDIS driver can pass *HigherLayerIfIndex* and *LowerLayerIfIndex* parameters to the **NdisIfAddIfStackEntry** function. These parameters specify one network interface that should be higher in the network interface stack and one network interface that should be lower in the stack.

A driver that has stack order information about an interface that is related to another interface (for example, internal bindings in a MUX intermediate driver that are not visible to NDIS) calls **NdisIfAddIfStackEntry** to populate the interface stack table. This function returns NDIS_STATUS_SUCCESS if the stack entry was successfully made. Typically, the component that owns or is the interface provider for the higher layer interface (which *HigherLayerIfIndex* identifies) calls **NdisIfAddIfStackEntry**.

To remove a stack table entry, a driver passes *HigherLayerIfIndex* and *LowerLayerIfIndex* parameters to the **NdisIfDeleteIfStackEntry** function.

For an example of maintaining the interface stack, see the MUX 6.0 sample driver.

Handling OID Query and Set Requests in an NDIS Interface Provider

Article • 03/14/2023

The NDISIF interface defines several interface parameters (including statistical counters) that can be queried or set which correspond to information in RFC 2863. NDIS accesses these interface parameters through entry points that the interface provider defines when it calls the **NdisIfRegisterProvider** function. For more information about registering as an interface provider, see Registering as an Interface Provider.

Interface parameters are identified by object identifiers (OIDs). Some OIDs are specific to interface providers.

The following topics describe how to handle query and set requests for interface parameters:

Handling an Interface Object Query Request

Handling an Interface Object Set Request

Handling an Interface Object Query Request

Article • 12/15/2021

To obtain the current value that is associated with an interface object, NDIS calls an interface provider's **ProviderQueryObject** function. This function returns NDIS_STATUS_SUCCESS if it successfully processes the query request or an NDIS_STATUS_*Xxx* error code otherwise.

For a list of interface provider-specific OID requests, see NDIS Network Interface OIDs. For a list of OIDs that NDIS uses with providers, miniport adapters, and filter modules to support network interface objects, see NDIS Network Interface to OID Mapping.

The handle at the *ProviderIfContext* parameter of **ProviderQueryObject** identifies the context area that the interface provider passed to NDIS when it called the **NdisIfRegisterInterface** function to register the interface. The *ObjectId* parameter specifies the OID for the object that is being queried. The *pOutputBufferLength* and *pOutputBuffer* parameters provide a pointer to the resulting length of the output buffer and a pointer to the output buffer, respectively.

Handling an Interface Object Set Request

Article • 12/15/2021

To set the data that is associated with an interface object, NDIS calls an interface provider's **ProviderSetObject** function. This function returns NDIS_STATUS_SUCCESS if it successfully changed the data or an NDIS_STATUS_*Xxx* error code otherwise.

For a list of interface provider-specific OID requests, see NDIS Network Interface OIDs. For a list of OIDs that NDIS uses with providers, miniport adapters, and filter modules to support network interface objects, see NDIS Network Interface to OID Mapping.

The handle at the *ProviderIfContext* parameter of **ProviderSetObject** identifies the context area that the interface provider passed to NDIS when it called the **NdisIfRegisterInterface** function to register the interface. The *ObjectId* parameter specifies the OID for the object that is being set. The *InputBufferLength* and *pInputBuffer* parameters provide the length of the input buffer and a pointer to the input buffer, respectively.

Mapping of NDIS Network Interfaces to NDIS OIDs

Article • 03/14/2023

To respond to NDIS interface object requests, NDIS interface providers can cache information that they obtain from underlying drivers and can also issue OID requests to obtain information about underlying interfaces.

As a proxy interface provider, NDIS typically caches information that it receives about miniport adapters and filter modules. The NDIS proxy interface provider uses the cached information, if appropriate, to respond to interface requests. In some cases, the NDIS proxy interface provider issues OIDs to obtain information for interfaces. For example, the primary source of interface information for NDIS 5.*x* and earlier drivers is through OID requests. In NDIS 6.0 drivers, there are additional sources of interface information, such as the **NDIS_RESTART_ATTRIBUTES** and **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structures. For more information about alternate sources of information in the OIDs, see the reference page for each OID.

The NDIS proxy interface provider also generates some interface information on behalf of miniport adapters and filter modules. For example, NDIS generates an interface alias (*ifAlias* in RFC 2863) in response to the *ifAlias* request. NDIS defines additional OIDs to obtain such information from NDIS interface providers. For example, OID_GEN_ALIAS allows an interface provider to specify an *ifAlias* object. Such OIDs are specific to interface providers and are never used to obtain information from other NDIS drivers.

In addition to the OIDs that are specific to interface providers, interface providers must support the other NDIS OIDs that NDIS can use to obtain interface information. NDIS can issue these OIDs to the provider and the provider can issue these OIDs, if necessary, to collect information from underlying interfaces.

Note NDIS defines additional statistics that are not included in RFC 2863. For a list that maps all of the NDIS-supported interface statistics to OIDs, see the members of the **NDIS_INTERFACE_INFORMATION** structure. The table in this topic defines the mapping for statistics that are defined in the RFC 2863 specification for readers that are trying to relate the specification to the NDIS implementation.

The following table shows the mapping from the objects that are defined in the management information base (MIB) to NDIS 6.0 OIDs and to OIDs that NDIS might use to obtain information from NDIS 5.*x* and earlier drivers. The table also includes some additional interface objects that are not defined as MIB objects. The interface objects also correspond to members in the **NDIS_INTERFACE_INFORMATION** structure that is associated with the OID_GEN_INTERFACE_INFO OID.

Note The NDIS 6.0 OIDs in the table that are marked with a asterisk (*) prefix are specific to interface providers. The other NDIS 6.0 OIDs can be issued to interface providers and other NDIS drivers.

Interfaces MIB value	NDIS 6.0 OIDs	NDIS 5.x and earlier OIDs
ifAdminStatus	* OID_GEN_ADMIN_STATUS	
ifAlias	* OID_GEN_ALIAS	
ifCounterDiscontinuityTime	* OID_GEN_DISCONTINUITY_TIME	
ifHCInBroadcastPkts	OID_GEN_BROADCAST_FRAMES_RCV	OID_GEN_BROADCAST_FRAMES_RCV
ifHCInMulticastPkts	OID_GEN_MULTICAST_FRAMES_RCV	OID_GEN_MULTICAST_FRAMES_RCV
ifHCInOctets	OID_GEN_BYTES_RCV	NDIS adds the results from these OIDs to collect the *ifHCInOctets* value from NDIS 5.*x* drivers: OID_GEN_DIRECTED_BYTES_RCV+ OID_GEN_MULTICAST_BYTES_RCV+ OID_GEN_BROADCAST_BYTES_RCV NDIS 6.0 interface providers should also support these OIDs.
ifHCInUcastPkts	OID_GEN_DIRECTED_FRAMES_RCV	OID_GEN_DIRECTED_FRAMES_RCV
ifHCOutBroadcastPkts	OID_GEN_BROADCAST_FRAMES_XMIT	OID_GEN_BROADCAST_FRAMES_XMIT
ifHCOutMulticastPkts	OID_GEN_MULTICAST_FRAMES_XMIT	OID_GEN_MULTICAST_FRAMES_XMIT
ifHCOutOctets	OID_GEN_BYTES_XMIT	NDIS adds the results from these OIDs to collect the *ifHCInOctets* value from NDIS 5.*x* drivers: OID_GEN_DIRECTED_BYTES_XMIT+ OID_GEN_MULTICAST_BYTES_XMIT+ OID_GEN_BROADCAST_BYTES_XMIT NDIS 6.0 interface providers should also support these OIDs.
ifHCOutUCastPkts	OID_GEN_DIRECTED_FRAMES_XMIT	OID_GEN_DIRECTED_FRAMES_XMIT
ifHighSpeed	* OID_GEN_LINK_SPEED_EX, * OID_GEN_XMIT_LINK_SPEED, * OID_GEN_RCV_LINK_SPEED	OID_GEN_LINK_SPEED

Interfaces MIB value	NDIS 6.0 OIDs	NDIS 5.x and earlier OIDs
ifInDiscards	OID_GEN_RCV_DISCARDS	
ifInErrors	OID_GEN_RCV_ERROR	OID_GEN_RCV_ERROR
ifLastChange	* OID_GEN_LAST_CHANGE	
ifMtu	OID_GEN_MAXIMUM_FRAME_SIZE	OID_GEN_MAXIMUM_FRAME_SIZE
ifOperStatus	* OID_GEN_OPERATIONAL_STATUS	
ifOutDiscards	OID_GEN_XMIT_DISCARDS	OID_GEN_XMIT_DISCARDS
ifOutErrors	OID_GEN_XMIT_ERROR	OID_GEN_XMIT_ERROR
ifPhysAddress	OID_802_3_CURRENT_ADDRESS	OID_802_3_CURRENT_ADDRESS
ifPromiscuousMode	* OID_GEN_PROMISCUOUS_MODE	
Not applicable	OID_802_3_PERMANENT_ADDRESS	OID_802_3_PERMANENT_ADDRESS
Not applicable	* OID_GEN_INTERFACE_INFO	
Not applicable	* OID_GEN_MEDIA_CONNECT_STATUS_EX	
Not applicable	* OID_GEN_MEDIA_DUPLEX_STATE	
Not applicable	OID_GEN_STATISTICS	

NDIS_MDL_LINKAGE macro

Article • 03/14/2023

The **NDIS_MDL_LINKAGE** macro retrieves a pointer to the next MDL that is associated with the specified MDL.

Syntax

```
ManagedCPlusPlus

PVOID NDIS_MDL_LINKAGE(
   PMDL _Mdl
);
```

Parameters

_Mdl
A pointer to an MDL.

Return value

NDIS_MDL_LINKAGE returns a pointer to an MDL or **NULL** if there is no next MDL.

Remarks

The **NDIS_MDL_LINKAGE** macro provides an MDL-based version of the NDIS_BUFFER_LINKAGE function.

Requirements

Target platform	Desktop
Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)
IRQL	Any level

See also

NDIS_BUFFER_LINKAGE

NDIS_MDL_TO_SPAN_PAGES macro

Article • 03/14/2023

The **NDIS_MDL_TO_SPAN_PAGES** macro retrieves the number of physical pages of memory that are being used to back a given MDL.

Syntax

ManagedCPlusPlus

```
int NDIS_MDL_TO_SPAN_PAGES(
   PMDL _Mdl
);
```

Parameters

_Mdl
A pointer to an MDL.

Return value

NDIS_MDL_TO_SPAN_PAGES returns the number of pages that are backing the virtual range for the MDL.

Remarks

The **NDIS_MDL_TO_SPAN_PAGES** macro provides an MDL-based version of the NDIS_BUFFER_TO_SPAN_PAGES function.

Requirements

Target platform	Desktop
Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)
IRQL	Any level

See also

NDIS_BUFFER_TO_SPAN_PAGES

NdisGetMdlPhysicalArraySize macro

Article • 12/15/2021

The **NdisGetMdlPhysicalArraySize** macro retrieves the number of disconnected physical memory blocks that are associated with an MDL.

Syntax

```
ManagedCPlusPlus

VOID NdisGetMdlPhysicalArraySize(
    _Mdl,
    _ArraySize
);
```

Parameters

_Mdl
A pointer to an MDL.

_ArraySize
A pointer to a caller-supplied variable in which this macro returns the number of disconnected physical memory blocks that are associated with the specified MDL.

Return value

None

Remarks

The **NdisGetMdlPhysicalArraySize** macro provides an MDL-based version of the NdisGetBufferPhysicalArraySize function.

Requirements

Target platform	Desktop
Version	Supported in NDIS 6.0 and later.

Header	Ndis.h (include Ndis.h)
IRQL	<= DISPATCH_LEVEL
DDI compliance rules	Irql_NetBuffer_Function

See also

NdisGetBufferPhysicalArraySize

NdisGetNextMdl macro

Article • 12/15/2021

The **NdisGetNextMdl** macro retrieves the next MDL in an MDL chain, given a pointer to the current MDL.

Syntax

ManagedCPlusPlus

```
VOID NdisGetNextMdl(
    _CurrentMdl,
    _NextMdl
);
```

Parameters

_CurrentMdl
A pointer to the specified current MDL.

_NextMdl
A pointer to a caller-supplied variable in which this macro returns a pointer to the next MDL in the MDL chain, if any, that follows the MDL at *_CurrentMdl* .

Return value

None

Remarks

The **NdisGetNextMdl** macro provides an MDL-based version of the NdisGetNextBuffer function.

Requirements

Target platform	Desktop
Version	Supported in NDIS 6.0 and later.

Header	Ndis.h (include Ndis.h)
IRQL	Any level

See also

NdisGetNextBuffer

NdisQueryMdl macro

Article • 12/15/2021

The **NdisQueryMdl** macro retrieves the buffer length, and optionally the base virtual address, from an MDL.

Syntax

```
ManagedCPlusPlus

VOID NdisQueryMdl(
    _Mdl,
    _VirtualAddress,
    _Length,
    _Priority
);
```

Parameters

_Mdl
A pointer to an MDL.

_VirtualAddress
A pointer to a caller-supplied variable in which this macro returns the base virtual address of the virtual address range that is described by the MDL. The base virtual address can be **NULL** for either of the following reasons:

- System resources are low or exhausted and the *_Priority* parameter is set to **LowPagePriority** or **NormalPagePriority**.

- System resources are exhausted and the *_Priority* parameter is set to **HighPagePriority**.

_Length
A pointer to a caller-supplied variable in which this macro returns the length, in bytes, of the virtual address range that is described by the MDL.

_Priority
A page priority value. For a list of the possible values for this parameter, see the *Priority* parameter of the MmGetSystemAddressForMdlSafe macro.

Return value

None

Remarks

The **NdisQueryMdl** macro provides an MDL-based version of the NdisQueryBuffer function.

Requirements

Target platform	Desktop
Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)
IRQL	<= DISPATCH_LEVEL
DDI compliance rules	Irql_NetBuffer_Function

See also

MmGetSystemAddressForMdlSafe

NdisQueryBuffer

NdisQueryMdlOffset macro

Article • 12/15/2021

The **NdisQueryMdlOffset** macro retrieves the offset within a physical page at which a given MDL buffer begins and the length of the buffer.

Syntax

ManagedCPlusPlus

```
VOID NdisQueryMdlOffset(
    _Mdl,
    _Offset,
    _Length
);
```

Parameters

_Mdl
A pointer to an MDL.

_Offset
A pointer to a caller-supplied variable in which this macro returns the zero-based byte offset within the physical page that contains the MDL-specified buffer.

_Length
A pointer to a caller-supplied variable in which this macro returns the length, in bytes, of the virtual address range that is specified by the MDL.

Return value

None

Remarks

The **NdisQueryMdlOffset** macro provides an MDL-based version of the NdisQueryBufferOffset function.

Requirements

Target platform	Desktop
Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)
IRQL	<= DISPATCH_LEVEL
DDI compliance rules	Irql_NetBuffer_Function

See also

NdisQueryBufferOffset

Overview of NDIS packet timestamping

Article • 03/14/2023

The NDIS packet timestamping interface supports the hardware timestamping capability of a network interface card (NIC) for the Precision Time Protocol (PTP) version 2.

Many NICs can generate timestamps in their hardware when a packet is received or transmitted using their own hardware clock. Starting with NDIS 6.82, NDIS packet timestamping allows you to add hardware timestamping support to your NIC driver.

You may want to enable timestamping support to improve the accuracy of clock synchronization applications. The miniport driver should disable all types of timestamping support by default.

Specifically, NDIS packet timestamping makes hardware timestamps available to the operating system so that applications implementing the PTP protocol with UDP as the transport can use them. PTP is a protocol that can utilize hardware timestamps to achieve more accurate time synchronization between systems.

The closer timestamp generation is to when a packet is sent or received by the network adapter hardware, the more accurate the synchronization application. NDIS packet timestamping can help improve the accuracy of time synchronization applications by enabling them to use timestamps generated in the NIC hardware.

NDIS packet timestamping enables PTP version 2 applications (as defined by IEEE) operating in the two-step mode to use the NIC’s hardware timestamping capabilities. In two-step mode, timestamps in PTP packets are retrieved from the hardware and conveyed as separate messages rather than being generated on the fly in the hardware.

NDIS packet timestamping provides the ability to:

- Discover the NIC hardware’s timestamping capabilities.
- Associate the NIC hardware clock’s timestamps to PTP version 2 traffic running over UDP (using the standard UDP ports defined for PTP, for example 319 and 320).
- Use the NIC hardware’s clock as a free running clock. The ability to query the network hardware’s clock and establish a relation between the network hardware clock and a system clock makes this possible.
- Generate software timestamps.

The target of the NDIS packet timestamping interface is Ethernet hardware. The interface works with both NICs that specifically support hardware timestamp generation for PTP version 2 traffic as well as NICs that can generate hardware timestamps for all traffic, as these NICs work with PTP traffic as well.

In this section

Reporting timestamping capabilities and current configuration

Article • 12/15/2021

Miniport drivers need to indicate the NIC's hardware timestamping capabilities and the miniport driver's software timestamping capabilities to NDIS and overlying drivers. They also need to report which timestamping capabilities are currently enabled or disabled. Miniport drivers use status indications to report the timestamping capabilities and their current configuration to the operating system.

During initialization, the miniport driver should report the timestamping capabilities and their current configuration within the *MiniportInitializeEx* function. The driver should:

1. Generate an NDIS_STATUS_TIMESTAMP_CAPABILITY status indication to report the timestamping capabilities.

2. Generate an NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG status indication to report the current timestamping configuration.

> ⓘ Note
>
> Miniport drivers read the ***PtpHardwareTimestamp** and ***SoftwareTimestamp** keywords values in the INF file to determine which timestamping capabilities are enabled or disabled. For more information, see Standardized INF keywords for NDIS packet timestamping.

Any time that the miniport driver detects a change in underlying hardware capabilities it must generate the **NDIS_STATUS_TIMESTAMP_CAPABILITY** status indication. It must also report the corresponding change in the current configuration using the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication.

The miniport driver must also generate the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication whenever it detects a change in the current timestamping configuration.

Attaching timestamps to packets

Article • 12/15/2021

After the miniport driver reports which timestamping capabilities are present and currently enabled, the driver can attach the relevant timestamps to packets using the NET_BUFFER_LIST (NBL) structure.

For more information on reporting the NIC's hardware timestamping capabilities and the miniport driver's software timestamping capabilities to the operating system, see Reporting timestamping capabilities and current configuration.

Hardware timestamps

The `PtpV2OverUdpIPv4EventMsgReceiveHw`, `PtpV2OverUdpIPv4AllMsgReceiveHw`, `PtpV2OverUdpIPv4EventMsgTransmitHw`, `PtpV2OverUdpIPv4AllMsgTransmitHw`, `PtpV2OverUdpIPv6EventMsgReceiveHw`, `PtpV2OverUdpIPv6AllMsgReceiveHw`, `PtpV2OverUdpIPv6EventMsgTransmitHw`, `PtpV2OverUdpIPv6AllMsgTransmitHw`, `AllReceiveHw`, `AllTransmitHw` and `TaggedTransmitHw` flags in the NDIS_TIMESTAMP_CAPABILITY_FLAGS structure indicate which hardware timestamps the miniport driver supports.

The timestamp that the NIC hardware generates on reception or transmission of a packet is represented by a 64-bit integer value. This should be the raw value of the NIC hardware's clock at the point the timestamp is captured. The timestamp is stored in the NBL structure's **NetBufferListInfo** array.

Miniport drivers can use the NET_BUFFER_LIST_TIMESTAMP structure to set the timestamp in the NBL's **NetBufferListInfo** field. The driver fills the **Timestamp** field of the **NET_BUFFER_LIST_TIMESTAMP** structure with the timestamp generated by the hardware and calls the NdisSetNblTimestampInfo utility function, passing in the structure.

Miniport drivers can use NdisGetNblTimestampInfo and NdisCopyNblTimestampInfo to retrieve and copy timestamps.

If a particular hardware timestamp setting is enabled but a timestamp that corresponds to that capability isn't generated, the miniport should set the timestamp it attaches to the NBL to **zero**.

ⓘ Note

When recognizing PTP version 2 packets to generate hardware timestamps, the implementation should not restrict timestamp generation to packets that use the multicast addresses (both IPv4 and IPv6) that are specified by the PTP specification. The implementation should try to recognize PTP packets in other ways, for example using the UDP header or the PTP payload. This is so timestamps are still generated in scenarios where a PTP implementation might not use the multicast addresses specified in the PTP specification, for example where unicast addresses are used.

Receive side timestamping

The hardware should obtain the timestamp as close as possible to the point when the hardware receives the frame from the medium. This guideline is specified by the IEEE 1588 standard.

When a packet is received, the miniport driver must:

1. Correct the timestamp for any delays that exist between when the hardware captured the timestamp and when the hardware actually received the frame.

2. Attach the timestamp generated in hardware to the NBL. The timestamp corresponds to the frame (NET_BUFFER structure) contained in the NBL.

3. Call NdisMIndicateReceiveNetBufferLists to indicate the NBL to NDIS.

Note that in the receive direction, miniport drivers for Ethernet hardware are required to indicate only one **NET_BUFFER** per NBL.

Transmit side timestamping

The hardware should obtain the timestamp as close as possible to the point when the hardware transmits the frame to the medium. This guideline is specified by the IEEE 1588 standard.

When a packet is transmitted, the miniport driver must:

1. Correct the timestamp for any delays that exist between when the hardware captured the timestamp and when the hardware actually transmitted the frame.

2. Attach the timestamp generated in hardware to the NBL. If the NBL contains multiple **NET_BUFFER**s, the hardware timestamp corresponding to the first **NET_BUFFER** in the NBL should be attached to the NBL.

3. Call NdisMSendNetBufferListsComplete to send complete the NBL to NDIS.

Miniports and NIC hardware that report that the `TaggedTransmitHw` capability flag is supported and currently enabled should check if the `NDIS_NBL_FLAGS_CAPTURE_TIMESTAMP_ON_TRANSMIT` flag is set in the **NblFlags** field of an NBL that is given to the miniport for transmission. If this flag is set, this indicates that a transmit time timestamp is needed for that NBL and a transmit time hardware timestamp should be generated for the NBL.

Software timestamps

The `AllReceiveSw`, `AllTransmitSw` and `TaggedTransmitSw` flags in the NDIS_TIMESTAMP_CAPABILITY_FLAGS structure indicate if the miniport supports generating software timestamps.

Software timestamps are also represented as 64-bit integer values and are stored in the same slot in the **NetBufferListInfo** array of the NET_BUFFER (NBL) structure as the hardware timestamps.

If software timestamping capabilities are present and enabled then the miniport driver sets the timestamp in the NBL using the performance counter value (QPC). The miniport driver must:

1. Call KeQueryPerformanceCounter to obtain the QPC.

2. Fill the **Timestamp** field of the NET_BUFFER_LIST_TIMESTAMP structure with the QPC.

3. Set the timestamp in the NBL by calling NdisSetNblTimestampInfo and passing in the **NET_BUFFER_LIST_TIMESTAMP**.

On receive the miniport driver should capture the QPC as early as possible but no earlier than when the packet arrived.

On transmit the miniport driver should capture the QPC as late as possible before the packet is given to the hardware for transmission.

The `TaggedTransmitSw` flag is analogous to the `TaggedTransmitHw` flag but corresponds to software timestamps. If the capability is supported and enabled then the miniport should check the `NDIS_NBL_FLAGS_CAPTURE_TIMESTAMP_ON_TRANSMIT` flag in the **NblFlags** field of the NBL. If this flag is set, the miniport should generate a transmit time software timestamp for the NBL.

NDIS_STATUS_TIMESTAMP_CAPABILITY

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_TIMESTAMP_CAPABILITY** status indication to report the NIC's hardware timestamping capabilities and the miniport driver's software timestamping capabilities to NDIS and overlying drivers.

This status indication represents the timestamping capabilities of the hardware and miniport driver, not which capability is currently enabled or disabled. For more information on reporting the current timestamping configuration, see NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG.

Remarks

During initialization, the miniport driver should indicate its hardware and software timestamp capabilities from within its MiniportInitializeEx function. The driver should:

1. Initialize an NDIS_TIMESTAMP_CAPABILITIES structure with the NIC's hardware and software timestamp capabilities. The driver sets the members of the **NDIS_TIMESTAMP_CAPABILITIES** structure as follows:

   - The driver uses the **TimestampFlags** field to indicate the hardware and software timestamp capabilities.

   > ⓘ **Note**
   >
   > An implementation must support hardware timestamps and cross timestamps. Supporting software timestamps is optional.

   - The driver must set the **CrossTimestamp** field to **TRUE**.
   - The **HardwareClockFrequencyHz** field should contain the nominal operating frequency of the hardware clock used for timestamping by the NIC. This data may be used to display the nominal clock frequency to end users for informational purposes.
   - The **Type** field in the **Header** field should be set to **NDIS_OBJECT_TYPE_DEFAULT** and the **Revision** to **NDIS_TIMESTAMP_CAPABILITIES_REVISION_1**.

2. Generate an **NDIS_STATUS_TIMESTAMP_CAPABILITY** status indication by calling NdisMIndicateStatusEx to report the timestamping capabilities. The **StatusBuffer** field of the NDIS_STATUS_INDICATION structure should point to the initialized **NDIS_TIMESTAMP_CAPABILITIES** structure.

The miniport driver must also generate the NDIS_STATUS_TIMESTAMP_CAPABILITY status indication whenever it detects a change in underlying hardware capabilities.

Here's how a miniport driver might indicate its supported timestamping capabilities:

C++

```cpp
// From within its initialization routine, the miniport in this
// example indicates that it supports the following capabilities:
// - PtpV2OverUdpIPv4EventMsgReceiveHw
// - PtpV2OverUdpIPv6EventMsgReceiveHw
// - TaggedTransmitHw
// - CrossTimestamp

NDIS_STATUS MiniportInitializeEx(
    _In_ NDIS_HANDLE MiniportAdapterHandle,
    _In_ NDIS_HANDLE MiniportDriverContext,
    _In_ PNDIS_MINIPORT_INIT_PARAMETERS MiniportInitParameters
)
{
. . .
    NDIS_TIMESTAMP_CAPABILITIES timeStampCapabilities;
    NDIS_STATUS_INDICATION timeStampStatus;
. . .

    // Initialize an NDIS_TIMESTAMP_CAPABILITIES structure

    RtlZeroMemory(&timeStampCapabilities, sizeof(timeStampCapabilities));
    RtlZeroMemory(&timeStampStatus, sizeof(timeStampStatus));

    timeStampCapabilities.Header.Type = NDIS_OBJECT_TYPE_DEFAULT;
    timeStampCapabilities.Header.Size = sizeof(timeStampCapabilities);
    timeStampCapabilities.Header.Revision =
NDIS_TIMESTAMP_CAPABILITIES_REVISION_1;

    timeStampCapabilities.CrossTimestamp = TRUE;
    timeStampCapabilities.TimestampFlags.PtpV2OverUdpIPv4EventMsgReceiveHw =
TRUE;
    timeStampCapabilities.TimestampFlags.PtpV2OverUdpIPv6EventMsgReceiveHw =
TRUE;
    timeStampCapabilities.TimestampFlags.TaggedTransmitHw = TRUE;

    timeStampCapabilities.HardwareClockFrequencyHz = 150000;

    timeStampStatus.Header.Type = NDIS_OBJECT_TYPE_STATUS_INDICATION;
    timeStampStatus.Header.Revision = NDIS_STATUS_INDICATION_REVISION_1;
    timeStampStatus.Header.Size = NDIS_SIZEOF_STATUS_INDICATION_REVISION_1;
    timeStampStatus.SourceHandle = MiniportAdapterHandle;
    timeStampStatus.StatusBuffer = &timeStampCapabilities;
    timeStampStatus.StatusBufferSize = sizeof(timeStampCapabilities);
    timeStampStatus.StatusCode = NDIS_STATUS_TIMESTAMP_CAPABILITY;

    // Generate an NDIS_STATUS_TIMESTAMP_CAPABILITY status indication
    NdisMIndicateStatusEx(MiniportAdapterHandle, &timeStampStatus);
. . .
}
```

Requirements

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
NDIS Version	NDIS 6.82 and later
Header	Ntddndis.h (include Ndis.h)

See also

Reporting timestamping capabilities and current configuration

NDIS_TIMESTAMP_CAPABILITIES

NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG

MiniportInitializeEx

NdisMIndicateStatusEx

NDIS_STATUS_INDICATION

NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication to report the current timestamping configuration of the NIC hardware and miniport driver to NDIS and overlying drivers.

This status indication represents which timestamping capabilities are currently enabled or disabled. For information about the status indication driver use to report the timestamping capabilities, see NDIS_STATUS_TIMESTAMP_CAPABILITY.

Remarks

During initialization, the miniport driver should indicate the current timestamping configuration from within its MiniportInitializeEx function. The driver should:

1. Initialize an NDIS_TIMESTAMP_CAPABILITIES structure with the current timestamping configuration. The driver sets the members of the **NDIS_TIMESTAMP_CAPABILITIES** structure as follows:

   - The driver uses the **TimestampFlags** field to indicate its current timestamping configuration. Each flag in the NDIS_TIMESTAMP_CAPABILITY_FLAGS structure should be set to **TRUE** if the corresponding timestamping capability is currently enabled or **FALSE** if it is not.

   > ⓘ Note
   >
   > To determine which timestamping capabilities are currently enabled or disabled, the miniport reads the current values of the timestamping related keywords ***PtpHardwareTimestamp** and ***SoftwareTimestamp**. For more on using these keywords and determining which timestamping capabilities to enable, see Standardized INF keywords for NDIS packet timestamping.

   > ⓘ Note
   >
   > If an implementation finds both hardware and software timestamps enabled through the keywords, then the miniport should only enable hardware timestamps and should disable software timestamps.

- The driver should set the **CrossTimestamp** field to **TRUE** if hardware cross timestamps are enabled in the current configuration or **FALSE** if they are not.
- The **HardwareClockFrequencyHz** field must contain the current operating frequency of the NIC's hardware clock.
- The **Type** field in the **Header** field should be set to **NDIS_OBJECT_TYPE_DEFAULT** and the **Revision** to **NDIS_TIMESTAMP_CAPABILITIES_REVISION_1**.

2. Generate an NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG status indication by calling NdisMIndicateStatusEx to report the current configuration. The **StatusBuffer** field of the NDIS_STATUS_INDICATION structure should point to the initialized **NDIS_TIMESTAMP_CAPABILITIES** structure.

The miniport driver must generate an NDIS_STATUS_TIMESTAMP_CAPABILITY indication at least once before indicating **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG**. Otherwise NDIS will reject the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication and it will not be indicated to overlying drivers.

If the miniport driver indicates a change in the NIC's hardware timestamping *capability* using the NDIS_STATUS_TIMESTAMP_CAPABILITY status indication (for example, a change in the **HardwareClockFrequencyHz** field in the **NDIS_TIMESTAMP_CAPABILITIES** structure because of an underlying change in the NIC hardware), then it must also report the corresponding change in the current configuration using the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication.

The miniport driver must also generate the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication whenever it detects a change in current timestamping configuration.

Requirements

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
NDIS Version	NDIS 6.82 and later
Header	Ntddndis.h (include Ndis.h)

See also

Reporting timestamping capabilities and current configuration

Standardized INF keywords for NDIS packet timestamping

NDIS_STATUS_TIMESTAMP_CAPABILITY

NDIS_TIMESTAMP_CAPABILITIES

NDIS_TIMESTAMP_CAPABILITY_FLAGS

MiniportInitializeEx

NdisMIndicateStatusEx

NDIS_STATUS_INDICATION

Standardized INF keywords for NDIS packet timestamping

Article • 03/14/2023

An INF file can define the following standardized INF keywords to enable or disable the timestamping capabilities that the miniport driver and NIC hardware supports.

Miniport drivers can use these keywords to determine the current configuration of the timestamping capabilities. For example, the driver can read these keyword values during initialization to determine which timestamping capabilities are enabled and the driver can therefore use.

***PtpHardwareTimestamp** INF keyword

***SoftwareTimestamp** INF keyword

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

*PtpHardwareTimestamp INF keyword

The ***PtpHardwareTimestamp** keyword is defined to enable or disable support for hardware timestamping for Precision Time Protocol (PTP) version 2 packets using UDP as the transport.

The default setting for the ***PtpHardwareTimestamp** keyword is disabled and the miniport driver should disable all types of hardware timestamping support in the NIC hardware by default.

Miniport drivers read the ***PtpHardwareTimestamp** keyword value to determine if hardware timestamping is currently enabled or disabled.

If ***PtpHardwareTimestamp** is enabled, the miniport driver should:

1. Enable the relevant hardware timestamping capabilities in the NIC hardware.

2. Generate the NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG status indication to report the timestamping capabilities it enabled to NDIS. The driver uses the NDIS_TIMESTAMP_CAPABILITIES structure to specify which capabilities it enabled. The flags within the **TimestampFlags** field in the **NDIS_TIMESTAMP_CAPABILITIES** structure that correspond to hardware timestamping are `PtpV2OverUdpIPv4EventMsgReceiveHw`, `PtpV2OverUdpIPv4AllMsgReceiveHw`,

`PtpV2OverUdpIPv4EventMsgTransmitHw`, `PtpV2OverUdpIPv4AllMsgTransmitHw`, `PtpV2OverUdpIPv6EventMsgReceiveHw`, `PtpV2OverUdpIPv6AllMsgReceiveHw`, `PtpV2OverUdpIPv6EventMsgTransmitHw`, `PtpV2OverUdpIPv6AllMsgTransmitHw`, `AllReceiveHw`, `AllTransmitHw` and `TaggedTransmitHw`. The **CrossTimestamp** field in the **NDIS_TIMESTAMP_CAPABILITIES** structure for the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indicates if hardware cross timestamping is enabled.

When ***PtpHardwareTimestamp** is enabled the miniport should turn on some form of capability to generate hardware timestamps for both Rx and Tx for PTP version 2 over UDP. The miniport should also turn on the hardware cross timestamping capability if the hardware supports it.

The specific hardware timestamping capabilities that the miniport driver should enable in hardware depends on the capabilities of the NIC hardware. For example, if the NIC hardware only supports the `PtpV2OverUDPIPv4EventMsgReceiveHw`, `PtpV2OverUDPIPv6EventMsgReceiveHw` and `TaggedTransmitHw` capabilities, then the miniport can turn on these hardware timestamping capabilities if the ***PtpHardwareTimestamp** keyword is enabled.

If the NIC hardware supports multiple forms of hardware timestamping capabilities that can enable the PTP version 2 over UDP scenario, then the IHV should consider their hardware and issues such as performance impact to decide which capabilities the miniport should turn on. For example, the hardware may be capable of generating timestamps for `AllTransmitHw` and `TaggedTransmitHw`. If turning on `AllTransmitHw` is more expensive than turning on `TaggedTransmitHw`, then the IHV may choose to only turn on the `TaggedTransmitHw` capability for Tx.

In all cases, the miniport driver should accurately report which hardware timestamping capabilities it enabled or disabled using the **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication.

> ⓘ Note
>
> PTP over raw Ethernet is not supported. The IHV needs to determine what the most efficient way of handling PTP over raw Ethernet packets when supporting PTP over UDP is enabled.

> ⓘ Note

No support is needed for PTP version 1. If the NIC hardware also supports PTP version 1, then the IHV needs to determine the most efficient way of handling PTP version 1 packets when supporting PTP version 2.

INF entries for *PtpHardwareTimestamp

The ***PtpHardwareTimestamp** INF keyword is an enumeration keyword. Enumeration standardized INF keywords have the following attributes:

SubkeyName: The name of the keyword that you must specify in the INF file.

ParamDesc: The display text that is associated with SubkeyName.

Value: The enumeration integer value that is associated with each SubkeyName in the list.

EnumDesc: The display text that is associated with each value that appears in the menu.

Default: The default value for the menu.

The following table describes the possible INF entries for the ***PtpHardwareTimestamp** INF keyword.

SubkeyName	ParamDesc	Value	EnumDesc
***PtpHardwareTimestamp**	PTP Hardware Timestamp	0 (Default)	Disabled
		1	Enabled

ⓘ Note

If the miniport driver finds an unsupported value for the ***PtpHardwareTimestamp** keyword, then it should disable the hardware timestamping capability completely.

*SoftwareTimestamp INF keyword

The ***SoftwareTimestamp** keyword corresponds to the types of software timestamping the miniport driver is capable of. The miniport driver uses the configured value for this keyword to determine which of the supported software timestamping capabilities are currently enabled.

The default setting for the ***SoftwareTimestamp** keyword is disabled and all types of software timestamping support in the miniport should be disabled by default.

The miniport generates the NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG status indication to inform NDIS of the various timestamping capabilities that are currently enabled.

The flags within the **TimestampFlags** field in the NDIS_TIMESTAMP_CAPABILITIES structure that correspond to software timestamping are `AllReceiveSw`, `AllTransmitSw` and `TaggedTransmitSw`.

If the ***SoftwareTimestamp** keyword contains a value that indicates that some configuration of software timestamping is enabled, then the miniport should enable the configured software timestamping capabilities and generate a **NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG** status indication that accurately reports which software timestamping capabilities have been enabled.

If the miniport does not support any type of software timestamping then the ***SoftwareTimestamp** keyword should not be included in its INF file.

The ***SoftwareTimestamp** INF keyword is an enumeration keyword. Enumeration standardized INF keywords have the following attributes:

SubkeyName: The name of the keyword that you must specify in the INF file.

ParamDesc: The display text that is associated with SubkeyName.

Value: The enumeration integer value that is associated with each SubkeyName in the list.

EnumDesc: The display text that is associated with each value that appears in the menu.

Default: The default value for the menu.

The following table describes the possible INF entries for the ***SoftwareTimestamp** INF keyword.

SubkeyName	ParamDesc	Value	EnumDesc
***SoftwareTimestamp**	Software Timestamp	0 (Default)	Disabled
		1	**RxAll**: This enum value corresponds to the miniport driver capability to generate software timestamps for all packets during Rx.

SubkeyName	ParamDesc	Value	EnumDesc
		2	**TxAll**: This enum value corresponds to the miniport driver capability to generate software timestamps for all packets during Tx.
		3	**RxAll & TxAll**: This enum value corresponds to the miniport driver capability to generate software timestamps for all packets during Rx and Tx.
		4	**TaggedTx**: This enum value corresponds to the miniport driver capability to generate software timestamps for a specific Tx packet when indicated to do so by the operating system.
		5	**RxAll & TaggedTx**: This enum value corresponds to the miniport driver capability to generate software timestamps for all packets during Rx and for a specific Tx packet when indicated to do so by the operating system.

ⓘ **Note**

If the miniport driver finds an unsupported value for the ***SoftwareTimestamp** keyword, then it should disable the software timestamping capability completely.

Querying timestamping capabilities and configuration

Article • 12/15/2021

Once the miniport driver is initialized, overlying drivers and applications can issue the following OID query requests to obtain hardware and software timestamping information.

- OID_TIMESTAMP_CAPABILITY. An overlying driver issues an object identifier (OID) query request of OID_TIMESTAMP_CAPABILITY to obtain the hardware timestamping capabilities of the NIC and software timestamping capabilities of the miniport driver.

- OID_TIMESTAMP_CURRENT_CONFIG. An overlying driver issues an OID query request of OID_TIMESTAMP_CURRENT_CONFIG to obtain the current timestamping configuration of the NIC.

- OID_TIMESTAMP_GET_CROSSTIMESTAMP. An overlying driver issues an OID query request of OID_TIMESTAMP_GET_CROSSTIMESTAMP to obtain the cross timestamp from the NIC hardware. Precision Time Protocol (PTP) version 2 applications use the information provided in this OID to establish a relation between the NIC's hardware clock and a system clock.

NDIS handles the OID_TIMESTAMP_CAPABILITY and OID_TIMESTAMP_GET_CROSSTIMESTAMP OIDs based on the information that the miniport driver reported when it registered the timestamping capabilities and current configuration to the operating system.

The miniport driver completes the OID_TIMESTAMP_GET_CROSSTIMESTAMP OID. The miniport must support this OID if it sets the **CrossTimestamp** field to **TRUE** in the **NDIS_TIMESTAMP_CAPABILITIES** structure as part of the current configuration.

For more information on how the miniport driver reports the timestamping capabilities, see Reporting timestamping capabilities and current configuration.

Overview of NDIS Ports

Article • 03/14/2023

This section introduces NDIS ports, which are an NDIS 6.0 feature and which enable overlying networking layers to access subinterfaces. In NDIS, network interfaces are associated with miniport adapters, and subinterfaces of a miniport adapter are called *NDIS ports*.

The architecture of the driver stack is much simpler because every network interface is treated as a miniport adapter. For example, each miniport adapter has its own IP and MAC address. In most cases, the overlying drivers do not require information about the virtual or physical nature of the miniport adapter or information about the physical device at the bottom of the driver stack.

An NDIS miniport adapter can provide an interface for a physical device or a virtual device. NDIS intermediate drivers provide interfaces for virtual devices that are called *virtual miniports*. NDIS intermediate drivers can bind to underlying miniport adapters and expose virtual miniports that overlying protocol drivers bind to.

In many cases, there is no one-to-one relationship between the underlying physical devices and virtual miniports. For example, an intermediate driver that implements failover functionality can create one virtual miniport to support multiple physical devices, and a virtual LAN (VLAN) intermediate driver can create multiple virtual miniports that are associated with a single physical device. Also, a driver that combines both failover and VLAN functionality can create a set of virtual miniports (*N* number of VLANs) while the driver is bound to multiple physical devices (*M* number of physical devices). For more information about intermediate drivers and virtual miniports, see NDIS 6.0 Intermediate Drivers.

In some applications, the ability to address the subinterfaces that are below virtual miniports is either required or simplifies the design. For example, the Extensible Authentication Protocol (EAP) protocol must specify the physical device that an EAP packet is sent or received on. If multiple physical devices are associated with a single virtual device, the EAP protocol is bound to the virtual device. In that case, the NDIS interfaces prior to NDIS 6.0 hide the subinterfaces, and the EAP protocol cannot choose which underlying physical device should carry the EAP packets. The EAP protocol then does not have any access to the underlying physical miniport adapters. Exposing the underlying physical miniport adapters as NDIS ports allows the EAP protocol to target a particular physical device.

The following topics further describe NDIS ports:

Identifying an NDIS Port

Default NDIS Port

Types of NDIS Ports

NDIS Port States

Identifying an NDIS Port

Article • 03/14/2023

An NDIS port is identified by its port number. When a miniport driver calls the **NdisMAllocatePort** function to allocate a port, NDIS allocates and assigns the lowest available port number to the port. When a miniport driver calls the **NdisMFreePort** function to free a port, NDIS also frees the port number that is assigned to the freed port so that NDIS can reuse the port number.

If a driver maintains separate context areas for each port, the driver must provide an efficient algorithm for translating the port number to the corresponding context area.

Default NDIS Port

Article • 03/14/2023

Port zero is reserved as the default port for a miniport adapter. If the *PortNumber* parameter of any function or the **PortNumber** member of any structure is set to zero, either the miniport driver did not allocate any ports, or the current activity is not port-specific.

For a good example of the default NDIS port, consider a load balancing and failover (LBFO) MUX intermediate driver. The virtual miniport of such a driver can be port zero (the default port). The intermediate driver can assign ports to the underlying miniport adapters with the port numbers ranging from 1 through the number of ports (*N*). An overlying driver could send data to port zero to allow the LBFO driver to select one of the underlying ports, or the overlying driver could specify a port number from 1 through *N* to choose a specific port for the send operation.

Miniport drivers do not have to allocate any ports or support any port numbers other than the default port. Even if a miniport driver does not allocate ports, NDIS allocates the default port and activates it after the miniport driver calls the **NdisMSetMiniportAttributes** function to set the registration attributes in the **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure. Miniport drivers can start operations on the default port when **NdisMSetMiniportAttributes** successfully returns. In this case, NDIS frees the default port when the miniport driver returns from the *MiniportHaltEx* function.

Types of NDIS Ports

Article • 03/14/2023

NDIS ports can be one of the following types:

NdisPortTypeUndefined
The default port type. Use this type for general port applications that do not fit into one of the following types.

NdisPortTypeBridge
Reserved for system use.

NdisPortTypeRasConnection
A Remote Access Service (RAS) connection.

NdisPortType8021xSupplicant
A remote wireless station that is associated with an access point on this host computer.

NdisPortTypeNdisImPlatform
Reserved for system use.

Note This value is supported only in NDIS 6.30 and later.

The characteristics of an NDIS port vary from one port application to another. For example, for a bridge interface, the miniport driver upper edge of an intermediate driver creates an **NdisPortTypeBridge** port when the protocol edge of the intermediate driver binds to a physical miniport adapter that requires a bridge at layer three.

Related topics

Overview of NDIS Ports

NDIS Port States

Article • 03/14/2023

NDIS ports have operating states that include initialization states and states that are specified in the NDIS_PORT_STATE structure. Port states fit into the following categories:

Initialization states
NDIS port initialization states are associated with startup initialization and Plug and Play (PnP) events. When NDIS or a miniport driver first allocates a port, the port is in the *allocated state*. After NDIS or the miniport driver activates a port, the port is in the *activated state*. Inactive ports cannot send or receive data, make status indications, receive OID requests, or initiate PnP events.

Link states
NDIS port link states are similar to link states that are associated with a miniport adapter and that are specified in an NDIS_LINK_STATE structure. The port link states indicate the media link connection state and link speeds. The link state of a port can be different from the link state of the associated miniport adapter.

Authentication states
NDIS port authentication states indicate if a port is controlled (requires authorization), the direction of data transmission (send, receive, or both), and the authorization state of a port (authorized, or not authorized). If a port is not controlled, the authenticated and not authenticated states are ignored.

A miniport driver can activate a port or deactivate a port with a PnP event. For more information about activating and deactivating ports, see Activating NDIS Ports and Deactivating NDIS Ports.

Overlying drivers use the OID_GEN_PORT_STATE OID to get the current state of the port that is specified in the **PortNumber** member of the NDIS_OID_REQUEST structure. NDIS handles this OID, and miniport drivers do not receive this OID query.

Miniport drivers that support NDIS ports must use the NDIS_STATUS_PORT_STATE status indication to indicate changes in the state of an NDIS port. Miniport drivers must set the port number in the **PortNumber** member of the NDIS_STATUS_INDICATION structure.

NDIS and overlying drivers use the OID_GEN_PORT_AUTHENTICATION_PARAMETERS OID to set the current authentication states of an NDIS port. Miniport drivers that support NDIS ports must support this OID.

Allocating an NDIS Port

Article • 03/14/2023

To allocate an NDIS port for a miniport adapter, a miniport driver calls the NdisMAllocatePort function. **NdisMAllocatePort** is synchronous and returns after NDIS has successfully allocated the resources that are required for the port.

Before the miniport driver calls **NdisMAllocatePort**, the driver must call the NdisMSetMiniportAttributes function to set the attributes in the NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES structure. Miniport drivers can call **NdisMAllocatePort** for a miniport adapter after the call to **NdisMSetMiniportAttributes** returns successfully and before NDIS calls the *MiniportHaltEx* function for that miniport adapter.

NDIS always allocates the default port (port zero) so miniport drivers should not allocate a default port. NDIS frees the default port after the miniport driver returns form *MiniportHaltEx*.

NDIS assigns a port number to a port when the miniport driver calls NdisMAllocatePort. The driver specifies port characteristics in the NDIS_PORT_CHARACTERISTICS structure before the driver calls **NdisMAllocatePort**. When **NdisMAllocatePort** successfully returns, the **PortNumber** member of NDIS_PORT_CHARACTERISTICS that the *PortCharacteristics* parameter specifies is set to the port number that NDIS assigned to the port.

Before returning from *MiniportHaltEx*, a miniport driver must call the NdisMFreePort function to free all of the ports that are associated with a miniport adapter. If a miniport adapter fails initialization, the driver must call **NdisMFreePort** to free all of the ports that the driver allocated before it returns from the *MiniportInitializeEx* function. For more information about freeing NDIS ports, see Freeing NDIS Ports.

The maximum number of ports that a miniport driver can allocate is 0xffffff. However, in practice, drivers will set a maximum number that is based on the port type and the requirements of the driver application. For example, for a bridge application, the number of ports is unlikely to exceed 16. The number of ports would be higher for access points that use 802.1x supplicant ports and significantly higher for WAN drivers that use virtual private network (VPN) ports.

After a miniport driver allocates a port, the port is in the allocated state, and the port is not active. A port cannot be used to send and receive data, initiate a status indication, issue an OID request, or initiate a Plug and Play (PnP) event, until the port is activated. NDIS activates the default port automatically after the miniport driver sets the registration attributes in an **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure. To request that NDIS not activate the default port, a miniport driver can set NDIS_MINIPORT_ATTRIBUTES_CONTROLS_DEFAULT_PORT in the **AttributeFlags** member of NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES.

NDIS passes the authentication state of the default port to the *MiniportInitializeEx* function at the **DefaultPortAuthStates** member of the **NDIS_MINIPORT_INIT_PARAMETERS** structure. If a miniport driver controls the default port, when the miniport driver activates the default port, it can activate the default port by using the default authentication settings. For more information about activating the default port, see Activating NDIS Ports.

Miniport drivers can use the NDIS_PORT_CHAR_USE_DEFAULT_AUTH_SETTINGS flag in the **Flags** member of the **NDIS_PORT_CHARACTERISTICS** structure for the ports that the drivers allocate and activate. For the allocation case, NDIS assigns the default authentication states to the new ports and ignores the authentication states that are passed to the **NdisMAllocatePort** function.

For more information about NDIS port states, see NDIS Port States. For more information about activating ports, see Activating NDIS Ports.

Freeing an NDIS Port

Article • 03/14/2023

Your miniport driver must free all NDIS ports that it allocates for miniport adapters in its *MiniportInitializeEx* function. It can free a port any time by calling **NdisMFreePort**, except for the two cases noted below.

Your miniport driver must free all allocated ports in these cases:

- If your driver's *MiniportInitializeEx* function fails, it must free all allocated ports.
- If a miniport adapter is halted, your driver's *MiniportHaltEx* function must free all allocated ports.

Your miniport driver cannot simply call **NdisMFreePort** in these cases:

- If the port is the default port, NDIS frees it automatically, so your miniport driver must not free it. If you try to free the default port, **NdisMFreePort** returns an NDIS_STATUS_INVALID_PORT error.
- If the port is active, your miniport driver will need to deactivate it before calling **NdisMFreePort**.

Related topics

Allocating NDIS Ports

Deactivating NDIS Ports

Default NDIS Port

Activating an NDIS Port

Article • 03/14/2023

After a miniport driver successfully allocates an NDIS port, and before using the port number in NDIS functions, the driver must activate the port. To activate the port, the miniport driver sends a port activation Plug and Play (PnP) event to NDIS. To send the port activation PnP event, miniport drivers use the **NetEventPortActivation** PnP event code in the call to the NdisMNetPnPEvent function.

To activate ports, the miniport driver must set the members of the NET_PNP_EVENT_NOTIFICATION structure that the *NetPnPEvent* parameter of **NdisMNetPnPEvent** points to as follows:

PortNumber
The source port of the event notification. Set this member to zero because the port numbers are provided in the **Buffer** member of the structure that the **NetPnPEvent** member specifies.

NetPnPEvent
A NET_PNP_EVENT structure that describes the port activation event. Set the members of this structure as follows:

NetEvent
An event code that describes the event. Set this member to **NetEventPortActivation**.

Buffer
A pointer to a linked list of NDIS_PORT structures. The **Next** member of the NDIS_PORT structures points to the next NDIS_PORT structure in the list.

BufferLength
The number of bytes that are specified in **Buffer** . Set **BufferLength** to the size of the NDIS_PORT structures.

Other members
Set the remaining members of NET_PNP_EVENT to **NULL**.

The miniport driver lists the ports that have changed states from inactive to active in a linked list of NDIS_PORT structures. However, if the default port of a miniport adapter is the target of a **NetEventPortActivation** PnP event, the default port must be the only port in the list.

When the miniport driver notifies NDIS of the activation of a port (and possibly before this notification call returns), the miniport driver must be ready to handle send requests and OID requests that are associated with the port. Miniport drivers must not use the port number of a newly activated port in status or receive indications until after the call to **NdisMNetPnPEvent** returns.

NDIS does not notify overlying drivers about activated ports until after the default port is active. When NDIS calls the *ProtocolBindAdapterEx* function of a protocol driver, NDIS provides a list of all currently active ports in the **ActivePorts** member of the **NDIS_BIND_PARAMETERS** structure that the *BindParameters* parameter points to. When a miniport driver activates new ports, NDIS notifies all of the protocol drivers that are bound to the miniport driver with the **NetEventPortActivation** PnP event. For more information about handling these port activation events in a protocol driver, see Handling the Port Activation PnP Event.

Before a miniport driver allocates an NDIS port, the driver must call the **NdisMSetMiniportAttributes** function to set the registration attributes in the **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure. Miniport drivers can control the activation of the default port by setting the NDIS_MINIPORT_CONTROLS_DEFAULT_PORT attribute flag when they call **NdisMSetMiniportAttributes**. If a miniport driver assumes the responsibility for activating the default port, NDIS does not initiate the binding between the miniport adapter and the overlying drivers until the miniport driver activates the default port with the port activation PnP event.

All of the ports that are specified by the linked list of NDIS_PORT structures must be in the allocated state. A miniport driver should not attempt to activate a port that is already active; if the driver does attempt to activate an active port, NDIS treates the situation as a port activation failure.

If NDIS fails to activate any ports on the list, it fails the call to **NdisMNetPnPEvent**, and none of the ports on the list change state to the activated state. If NDIS fails to activate the ports because some of the ports do not exist, **NdisMNetPnPEvent** returns an NDIS_STATUS_INVALID_PORT return value. If NDIS fails to activate the ports because some of the ports are not in the allocated state, **NdisMNetPnPEvent** returns an NDIS_STATUS_INVALID_PORT_STATE return value.

After a port has been successfully activated, the port is in the activated state. Miniport drivers can indicate received data and status for a port in the activated state.

NDIS passes the authentication state of the default port to the *MiniportInitializeEx* function at the **DefaultPortAuthStates** member of the **NDIS_MINIPORT_INIT_PARAMETERS** structure. If a miniport driver controls the default port, when the miniport driver activates the default port, it can activate the default port by using the default authentication settings. To use the default authentication settings, set the NDIS_PORT_CHAR_USE_DEFAULT_AUTH_SETTINGS flag in the **Flags** member of **NDIS_PORT_CHARACTERISTICS** structure. Miniport drivers can use the NDIS_PORT_CHAR_USE_DEFAULT_AUTH_SETTINGS flag for the ports that they allocate and activate. For the activation case, NDIS assigns the default authentication states to the newly activated port and ignores the authentication states that are passed to **NdisMNetPnPEvent** for the **NetEventPortActivation** event.

For more information about controlling the default port and allocating ports, see Allocating NDIS Ports.

Managing an NDIS Port

Article • 03/14/2023

Interested NDIS drivers and user-mode applications can manage NDIS ports. NDIS provides services to help manage ports.

NDIS notifies the interested NDIS drivers and user mode applications of port state changes by issuing the associated status indications, and PnP events.

The port number that is passed to send and receive functions identifies the target port of a send operation or the source port of a receive indication. Similarly, the port number in the associated structures identifies the port for status indications, OID requests, and PnP events. For more information about port numbers, see NDIS Ports Introduction.

To help manage NDIS ports, the following structures include the port number:

NDIS_OID_REQUEST
Describes OID requests.

NDIS_STATUS_INDICATION
Describes NDIS status indications.

NET_PNP_EVENT_NOTIFICATION
Describes PnP event notifications.

This section includes:

NDIS Port Send and Receive Operations

NDIS Port OID Requests

Handling NDIS Ports Status Indications

Handling NDIS Ports PnP Event Notifications

NDIS Port Send and Receive Operations

Article • 03/14/2023

NDIS drivers can associate send and receive operations with NDIS ports.The port number in the *PortNumber* parameter of the NDIS send and receive functions identifies the target port of a send operation or the source port of a receive indication. If *PortNumber* is zero, the default port is used. When NDIS handles the default port and the miniport driver does not allocate any other ports, no NDIS drivers in the driver stack are required to do anything beyond always setting the *PortNumber* parameter to zero. For more information about allocating ports in a miniport driver, see Allocating NDIS Ports.

If the miniport driver allocates ports, overlying drivers can use the ports to send and receive data on the appropriate subinterfaces of the associated miniport adapter. However, the overlying driver must ensure that the ports are active before sending any data. Miniport drivers activate ports when the associated subinterfaces become available. For more information about activating a port in a miniport driver, see Activating NDIS Ports.

When NDIS calls the *ProtocolBindAdapterEx* function of a protocol driver, NDIS provides a list of all currently active ports in the **ActivePorts** member of the **NDIS_BIND_PARAMETERS** structure that the *BindParameters* parameter points to. NDIS also informs protocol drivers with a PnP event when ports are activated and deactivated. For more information about PnP port activation and deactivation notifications, see Handling NDIS Ports PnP Notifications. For more general information about send and receive operations, see Send and Receive Operations.

NDIS Port OID Requests

Article • 03/14/2023

NDIS drivers can associate OID requests with NDIS ports. In such an OID request, the **PortNumber** member of the **NDIS_OID_REQUEST** structure is set to the target port number. The port number is zero if the OID request is for the default port. The overlying driver must ensure that a port is active before making any OID requests that specify a specific port number.

When NDIS calls the *ProtocolBindAdapterEx* function of a protocol driver, NDIS provides a list of all currently active ports in the **ActivePorts** member of the **NDIS_BIND_PARAMETERS** structure that the *BindParameters* parameter points to. NDIS also informs protocol drivers with a PnP event when ports are activated and deactivated. For more information about PnP port activation and deactivation notifications, see Handling NDIS Ports PnP Notifications.

The following OIDs are specific to the NDIS ports interface:

OID_GEN_ENUMERATE_PORTS
Enumerates the active ports that are associated with a miniport adapter.

OID_GEN_PORT_STATE
Retrieves the current link and authentication port states.

OID_GEN_PORT_AUTHENTICATION_PARAMETERS
Sets the current authentication states of an NDIS port.

This section includes:

- Enumerating Ports
- Querying the Port State
- Setting Port Authentication Parameters

Enumerating Ports

Article • 12/15/2021

NDIS protocol drivers and filter drivers can use an OID_GEN_ENUMERATE_PORTS OID query request to determine the characteristics of the active NDIS ports that are associated with an underlying miniport adapter. NDIS handles this OID, and miniport drivers do not receive this OID query.

If the query succeeds, NDIS provides the results of the query in an **NDIS_PORT_ARRAY** structure. The **NumberOfPorts** member of NDIS_PORT_ARRAY contains the number of active ports that are associated with the miniport adapter. The **Ports** member of NDIS_PORT_ARRAY contains a list of pointers to **NDIS_PORT_CHARACTERISTICS** structures. Each NDIS_PORT_CHARACTERISTICS structure defines the characteristics of a single port.

Querying the Port State

Article • 12/15/2021

Overlying drivers can issue an OID_GEN_PORT_STATE OID query request to get the current state of the port that is specified in the **PortNumber** member of an **NDIS_OID_REQUEST** structure. NDIS handles this OID, and miniport drivers do not receive this OID query. NDIS receives port state information in the **NDIS_PORT_CHARACTERISTICS** structure.

The OID_GEN_PORT_STATE OID is supported in NDIS 6.0 and later versions.

Overlying drivers should avoid using OID_GEN_PORT_STATE when possible and should instead rely on the **NDIS_STATUS_PORT_STATE** status indication. For more information about port-related status indications, see Handling NDIS Ports Status Indications.

If the OID_GEN_PORT_STATE query succeeds, NDIS returns the port state information in an **NDIS_PORT_STATE** structure.

Setting Port Authentication Parameters

Article • 12/15/2021

NDIS and overlying drivers use an OID_GEN_PORT_AUTHENTICATION_PARAMETERS OID set request to set the current state of an NDIS port. Miniport drivers that support NDIS ports must support this OID.

If the set request is successful, the miniport driver uses the receive port direction, port control state, and authenticate state from an **NDIS_PORT_AUTHENTICATION_PARAMETERS** structure.

The miniport should generate an **NDIS_STATUS_PORT_STATE** status indication to notify overlying drivers of any state changes.

Handling NDIS Ports Status Indications

Article • 03/14/2023

If an NDIS port is the source of a status indication, a miniport driver should use the **PortNumber** member in the NDIS_STATUS_INDICATION structure to specify the source port. Miniport drivers never indicate status for inactive ports.

Miniport drivers should use the NDIS_STATUS_PORT_STATE status indication to indicate changes in the state of an NDIS port. For this status indication, miniport drivers must set the port number in the **PortNumber** member of the NDIS_STATUS_INDICATION structure. The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure contains a pointer to an NDIS_PORT_STATE structure.

NDIS Ports PnP Event Notifications

Article • 03/14/2023

NDIS forwards PnP events to notify overlying drivers when ports are activated or deactivated. NDIS and miniport drivers do not generate a PnP event when a port is allocated. Miniport drivers notify NDIS that ports have been activated with the **NetEventPortActivation** PnP event and miniport drivers generate a **NetEventPortDeactivation** PnP event to notify NDIS that some ports have been deactivated.

When NDIS calls the *ProtocolBindAdapterEx* function of a protocol driver, NDIS provides a list of all currently active ports in the **ActivePorts** member of the **NDIS_BIND_PARAMETERS** structure at the *BindParameters* parameter.

The following topics describe how to handle port PnP events:

Handling the Port Activation PnP Event

Handling the Port Deactivation PnP Event

Handling the Port Activation PnP Event

Article • 12/15/2021

Overlying drivers must handle the **NetEventPortActivation** PnP event when a miniport driver activates an NDIS port. NDIS does not initiate the binding between a protocol driver and miniport adapter until the default port has been activated. Therefore, protocol drivers should treat the call to their *ProtocolBindAdapterEx* function as a notification that the default port is active.

Protocol drivers must not use a port number in any NDIS requests unless the driver received notification that the port is active, either through the bind parameters or through the **NetEventPortActivation** PnP event.

NDIS generates a port activation PnP event after the miniport driver activates some ports. (Miniport drivers specify the **NetEventPortActivation** PnP event code in the **NET_PNP_EVENT_NOTIFICATION** structure that the *NetPnPEvent* parameter points to in the call to **NdisMNetPnPEvent** to activate NDIS ports.)

Miniport drivers can indicate the activation of multiple ports in one PnP notification by using the **Next** member in an **NDIS_PORT** structure to link multiple NDIS_PORT structures. For more information about the linked list of NDIS_PORT structures, see Activating NDIS Ports.

NDIS generates a **NetEventPortDeactivation** PnP event to the bound protocol drivers when a miniport deactivates some ports. For more information about the **NetEventPortDeactivation** PnP event, see Handling the Port Deactivation PnP Event.

Handling the Port Deactivation PnP Event

Article • 12/15/2021

Overlying drivers must handle the **NetEventPortDeactivation** PnP event when a miniport driver deactivates an NDIS port. To notify overlying drivers about port deactivation events, NDIS propagates the port deactivation PnP event from the underlying miniport driver.

Before a protocol driver completes the handling of a port deactivation PnP event, it must ensure that all outstanding OID requests and send requests that are associated with the port have completed. After the protocol driver completes the PnP event, the driver must ensure that it does not issue any OID requests or send requests for that port.

Miniport drivers specify the **NetEventPortDeactivation** PnP event code in the **NET_PNP_EVENT_NOTIFICATION** structure that the *NetPnPEvent* parameter points to in the call to the **NdisMNetPnPEvent** function to report that some ports have been deactivated. The miniport driver specifies an array of NDIS_PORT_NUMBER values to list the deactivated ports. For more information about the array of port numbers, see Deactivating NDIS Ports.

Deactivating an NDIS Port

Article • 03/14/2023

To deactivate NDIS ports, a miniport driver sends a port deactivation Plug and Play (PnP) event to NDIS. After a miniport driver successfully activates a port, the driver must deactivate the port before it can free the port. Also, the driver might deactivate a port for application-specific reasons. A port can be reactivated after it is deactivated, but a port cannot be reactivated if it is freed.

To send a port deactivation PnP event, miniport drivers use the **NetEventPortDeactivation** PnP event code in the call to the NdisMNetPnPEvent function. To deactivate ports, the miniport driver must set the members of the NET_PNP_EVENT_NOTIFICATION structure that the *NetPnPEvent* parameter of **NdisMNetPnPEvent** points to as follows:

PortNumber
The source port of the event notification. Set this member to zero because the port numbers are provided in the **Buffer** member of the structure that the **NetPnPEvent** member specifies.

NetPnPEvent
A NET_PNP_EVENT structure that describes the port deactivation event. Set the members of this structure as follows:

NetEvent
An event code that describes the event. Set this member to **NetEventPortDeactivation**.

Buffer
A pointer to an array of NDIS_PORT_NUMBER-typed elements. The array contains the port numbers of all of the ports that the miniport driver is deactivating.

BufferLength
The number of bytes that are specified in **Buffer** . Set **BufferLength** to the size of the array that **Buffer** points to. To obtain the number of elements in the array, divide the value in **BufferLength** by the size of the NDIS_PORT_NUMBER data type.

Other members
Set the remaining members of NET_PNP_EVENT to **NULL**.

A miniport driver can provide an array with a list of ports to deactivate. However, if the default port of a miniport adapter is the target of a **NetEventPortDeactivation** PnP event, the default port must be the only port that is specified in the array.

Miniport drivers can deactivate active ports at any time. However, before a miniport driver deactivates a port, it must ensure that there are no outstanding status indications or receive indications that are associated with that port. After the miniport driver sends the port deactivation PnP event, it must not initiate any status or receive indications that are associated with the deactivated ports.

A miniport driver can also reactivate a port. For more information about activating NDIS ports, see Activating NDIS Ports.

When a miniport driver deactivates ports, NDIS notifies all of the protocol drivers that are bound to the miniport driver with the **NetEventPortDeactivation** PnP event. This PnP event lists those ports that have changed to the allocated state and does not include any ports that are already deactivated. For more information about handling port deactivation events in a protocol driver, see Handling the Port Deactivation PnP Event.

Before a miniport driver allocates an NDIS port, the driver must call the **NdisMSetMiniportAttributes** function to set the registration attributes in the **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** structure. Miniport drivers can control the activation of the default port by setting the NDIS_MINIPORT_CONTROLS_DEFAULT_PORT attribute flag when they call **NdisMSetMiniportAttributes**. If a miniport driver assumes the responsibility for activating the default port and the miniport driver activated the default port, it must deactivate the default port before returning from the *MiniportHaltEx* function.

All of the ports that are specified by the array of NDIS_PORT_NUMBER elements must be in the activated state. A miniport driver should not attempt to deactivate a port that has already deactivated.

If NDIS fails to deactivate any ports in the port array, none of the ports in the port array will change state. If the deactivation fails because some of the specified ports do not exist, the **NdisMNetPnPEvent** function returns the NDIS_STATUS_INVALID_PORT return value. If the deactivation fails because some of the ports are not in the activated state, **NdisMNetPnPEvent** returns the NDIS_STATUS_INVALID_PORT_STATE return value.

Until the call to **NdisMNetPnPEvent** returns, a port is not deactivated, and miniport drivers must be able to handle OID requests and send requests that are associated with that port.

When a miniport driver deactivates the default port, NDIS closes all of the bindings between the overlying protocol drivers and the miniport adapter. If a miniport driver tries to deactivate the default port and default port is already deactivated, **NdisMNetPnPEvent** fails and returns the NDIS_STATUS_INVALID_PORT_STATE return value. If a miniport driver tries to deactivates the default port and the default port is not the only port that is specified in the array of NDIS_PORT_NUMBER elements, **NdisMNetPnPEvent** fails and returns the NDIS_STATUS_INVALID_PORT return value. If a miniport driver sets the **Buffer** member to **NULL** or **BufferLength** member to zero, NDIS fails the **NdisMNetPnPEvent** call and returns the NDIS_STATUS_INVALID_PARAMETER return value.

After a port is successfully deactivated, the port is in the allocated state. Miniport drivers cannot indicate received data or status for the port in the allocated state.

Creating NDIS Interfaces for NDIS Ports

Article • 03/14/2023

By default, NDIS does not create an NDIS network interface for an NDIS port. If necessary, NDIS drivers can call the **NdisIfRegisterProvider** function to register as an NDIS interface provider and call the **NdisIfRegisterInterface** function to register an interface for a port.

For more information about NDIS network interfaces, see NDIS 6.0 Network Interfaces.

Power Management (NDIS 6.30)

Article • 03/14/2023

This section describes the power management interface that was introduced with NDIS 6.30 in Windows 8.

This section includes the following topics:

NDIS Packet Coalescing

NDIS Selective Suspend

NDIS Wake Reason Status Indications

Note The NDIS 6.30 power management interface is an extension to the NDIS 6.20 power management interface. For more information about the NDIS 6.20 power management interface, see Power Management (NDIS 6.20).

Introduction to NDIS Packet Coalescing

Article • 03/14/2023

Starting with NDIS 6.30, network adapters can support NDIS packet coalescing. This feature reduces the processing overhead and power consumption on a host system due to the reception of random broadcast or multicast packets.

This section includes the following topics:

Overview of Packet Coalescing

Article • 12/15/2021

Certain IP version 4 (IPv4) and IP version 6 (IPv6) network protocols involve the transmission of packets to broadcast or multicast addresses. These packets are received by multiple hosts in the IPv4/IPv6 subnet. In most cases, the host that receives these packets does not do anything with these packets. Therefore, the reception of these unwanted multicast or broadcast packets causes unnecessary processing and power consumption to occur within the receiving host.

For example, host A sends a multicast Link-local Multicast Name Resolution (LLMNR) request on an IPv6 subnet to resolve host B's name. Except for host A, this LLMNR request is received by all hosts on the subnet. Except for host B, the TCP/IP protocol stack that runs on the other hosts inspects the packet and determines that the packet is not intended for it. Therefore, the protocol stack rejects the packet and calls **NdisReturnNetBufferLists** to return the packet to the miniport driver.

Starting with NDIS 6.30, network adapters can support NDIS packet coalescing. By reducing the number of receive interrupts through the coalescing of random broadcast or multicast packets, the processing overhead and power consumption is significantly reduced on the system.

Packet coalescing involves the following steps:

1. Overlying drivers, such as the TCP/IP protocol stack, define NDIS receive filters that are used to screen broadcast and multicast packets. The overlying drivers download these filters to the underlying miniport driver that supports packet coalescing. Once downloaded, the miniport driver configures the network adapter with the packet coalescing receive filters.

   For more information about these filters, see Packet Coalescing Receive Filters.

2. Received packets that match receive filters are cached, or *coalesced*, on the network adapter. The adapter does not generate a receive interrupt for coalesced packets. Instead, the adapter interrupts the host when another hardware event occurs.

   When this interrupt is generated, the adapter must indicate a receive event with the interrupt. This allows the network adapter to process coalesced packets that were received by the network adapter.

   For example, the network adapter that supports packet coalescing can generate a receive interrupt when one of the following events occur:

- The expiration of a hardware timer whose expiration time is set to a maximum coalescing delay value of the matching receive filter.
- The available space within the hardware coalescing buffer reaches an adapter-specified low-water mark.
- A packet is received that does not match a coalescing filter.
- Another interrupt event, such as a send completion event, has occurred.

For more information about this process, see Handling Packet Coalescing Receive Filters.

The following points apply to the support of packet coalescing by NDIS:

- NDIS supports packet coalescing for packets received on the default NDIS port (port 0) assigned to the physical network adapter. NDIS does not support packet coalescing on NDIS ports that are assigned to virtual network adapters. For more information, see NDIS Ports.

- NDIS supports packet coalescing for packets received on the default receive queue of the network adapter. This receive queue has an identifier of NDIS_DEFAULT_RECEIVE_QUEUE_ID.

Packet Coalescing Receive Filters

Article • 12/15/2021

Starting with NDIS 6.30, NDIS receive filters have been extended to support packet coalescing. Each receive filter for packet coalescing defines the following:

- A set of fields within the various protocol headers of a packet, such as the destination address of a media access control (MAC) header or destination port of a User Datagram Protocol (UDP) header.

- The maximum time that a packet that matches a coalescing receive filter is coalesced by the network adapter. The adapter uses this value to set an expiration value on a hardware timer on the adapter. As soon as the timer expires, the adapter must interrupt the host so the miniport driver can process the coalesced packets.

  Note As soon as the first packet that matches a receive filter is coalesced and the timer is started, the network adapter must coalesce additional packets that match receive filters without resetting and restarting the timer.

Overlying drivers, such as protocol and filter drivers, download the packet coalescing receive filters to the miniport driver by issuing object identifier (OID) set requests of OID_RECEIVE_FILTER_SET_FILTER. For more information, see Setting Packet Coalescing Receive Filters.

Overlying drivers can also query the packet coalescing receive filters downloaded to the miniport driver. Overlying drivers do this by issuing OID method requests of OID_RECEIVE_FILTER_ENUM_FILTERS to the miniport driver. For more information, see Querying Packet Coalescing Receive Filters.

Reporting Packet Coalescing Capabilities

Article • 12/15/2021

Miniport drivers register the following capabilities with NDIS during network adapter initialization:

- The packet coalescing capabilities that the network adapter supports.
- The packet coalescing capabilities that are currently enabled on the network adapter.
- The packet coalescing receive filtering capabilities that are currently enabled on the network adapter.

Note A miniport driver's support for packet coalescing can be enabled or disabled through the ***PacketCoalescing** INF keyword setting. This setting is displayed in the **Advanced** property page for the network adapter. For more information about the packet coalescing INF file setting, see Standardized INF Keywords for Packet Coalescing.

The miniport driver reports the packet coalescing and filtering capabilities of the underlying network adapter through an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure. If the ***PacketCoalescing** keyword setting in the registry has a value of one, packet coalescing is enabled and the miniport driver initializes the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure in the following way:

1. The miniport driver initializes the **Header** member. The driver sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT.

   If the driver supports packet coalescing, it sets the **Revision** member of **Header** to the NDIS_RECEIVE_FILTER_CAPABILITIES_REVISION_2 and the **Size** member to NDIS_SIZEOF_RECEIVE_FILTER_CAPABILITIES_REVISION_2.

2. The miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_SUPPORTED_ON_DEFAULT_QUEUE flag in the **SupportedQueueProperties** member.

   If this flag is set, the network adapter must support the filtering of received multicast packets in hardware. This filtering is based on the multicast addresses that NDIS offloaded to the network adapter by sending it OID_802_3_MULTICAST_LIST OID set requests.

> **Note** Protocol drivers can also change the contents of the multicast address list by sending OID_802_3_ADD_MULTICAST_ADDRESS and OID_802_3_DELETE_MULTICAST_ADDRESS requests. NDIS combines these requests into OID_802_3_MULTICAST_LIST OID set requests.

Note The adapter is required to reject any incoming multicast packet whose destination media access control (MAC) address does not match any of the multicast addresses specified by these OID set requests.

3. The miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag in the **EnabledFilterTypes** member.

   Note If the driver sets this flag, it must also set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_SUPPORTED_ON_DEFAULT_QUEUE flag in the **SupportedQueueProperties** member. Otherwise, NDIS will fail the call to **NdisMSetMiniportAttributes** by returning NDIS_STATUS_BAD_CHARACTERISTICS.

4. If the miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must support all receive filter test criteria. The driver advertises this support by setting the following flags in the **SupportedFilterTests** member:

   - NDIS_RECEIVE_FILTER_TEST_HEADER_FIELD_EQUAL_SUPPORTED

   - NDIS_RECEIVE_FILTER_TEST_HEADER_FIELD_MASK_EQUAL_SUPPORTED

   - NDIS_RECEIVE_FILTER_TEST_HEADER_FIELD_NOT_EQUAL_SUPPORTED

   Note If the miniport driver does not set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must set the **SupportedFilterTests** member to zero.

5. If the miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the miniport driver must support the filtering of data within various fields of the media access control (MAC), IP version 4 (IPv4), and IP version 6 (IPv6) headers. The driver advertises this support by setting the following flags in the **SupportedHeaders** member:

   - NDIS_RECEIVE_FILTER_MAC_HEADER_SUPPORTED

   - NDIS_RECEIVE_FILTER_ARP_HEADER_SUPPORTED

- NDIS_RECEIVE_FILTER_IPV4_HEADER_SUPPORTED
- NDIS_RECEIVE_FILTER_IPV6_HEADER_SUPPORTED
- NDIS_RECEIVE_FILTER_UDP_HEADER_SUPPORTED

Note If the miniport driver does not set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must set the **SupportedHeaders** member to zero.

6. If the miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the miniport driver must support the filtering of data within the media access control (MAC) header of the received packet. The driver advertises this support by setting the following flags in the **SupportedMacHeaderFields** member:

   - NDIS_RECEIVE_FILTER_MAC_HEADER_DEST_ADDR_SUPPORTED
   - NDIS_RECEIVE_FILTER_MAC_HEADER_PROTOCOL_SUPPORTED
   - NDIS_RECEIVE_FILTER_MAC_HEADER_PACKET_TYPE_SUPPORTED

   Note If the miniport driver does not set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must set the **SupportedMacHeaderFields** member to zero.

7. If the miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the miniport driver must support the filtering of data within the header of a received Address Resolution Protocol (ARP) packet. The driver advertises this support by setting the following flags in the **SupportedARPHeaderFields** member:

   - NDIS_RECEIVE_FILTER_ARP_HEADER_OPERATION_SUPPORTED
   - NDIS_RECEIVE_FILTER_ARP_HEADER_SPA_SUPPORTED
   - NDIS_RECEIVE_FILTER_ARP_HEADER_TPA_SUPPORTED

   Note If the miniport driver does not set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must set the **SupportedARPHeaderFields** member to zero.

8. If the miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the miniport driver must support the filtering of data within the Open Systems Interconnection (OSI) layer 3 (L3) header of a received IP version 4 (IPv4) packet. The driver

advertises this support by setting the following flags in the **SupportedIPv4HeaderFields** member:

- NDIS_RECEIVE_FILTER_IPV4_HEADER_PROTOCOL_SUPPORTED

Note If the miniport driver does not set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must set the **SupportedIPv4HeaderFields** member to zero.

9. If the miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the miniport driver must support the filtering of data within the L3 header of a received IP version 6 (IPv6) packet. The driver advertises this support by setting the following flags in the **SupportedIPv6HeaderFields** member:

   - NDIS_RECEIVE_FILTER_IPV6_HEADER_PROTOCOL_SUPPORTED

   Note If the miniport driver does not set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must set the **SupportedIPv6HeaderFields** member to zero.

10. If the miniport driver sets the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the miniport driver must support the filtering of data within the OSI layer 4 (L4) header of a received User Datagram Protocol (UDP) packet. The driver advertises this support by setting the following flags in the **SupportedIUdpHeaderFields** member:

    - NDIS_RECEIVE_FILTER_UDP_HEADER_DEST_PORT_SUPPORTED

    Note If the received UDP packet contains IPv4 options or IPv6 extension headers, the network adapter can handle the packet as if it failed the UDP filter test. In this way, the adapter can automatically drop the received packet.

Note If the miniport driver does not set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED flag, the driver must set the **SupportedIUdpHeaderFields** member to zero.

11. The miniport driver must report the maximum number of tests on packet header fields that can be specified for a single packet coalescing filter. The driver specifies this value in the **MaxFieldTestsPerPacketCoalescingFilter** member.

    Note Network adapters that support packet coalescing must support five or more packet header fields that can be specified for a single packet coalescing filter. If the adapter does not support packet coalescing, the miniport driver must set this value to zero.

12. The miniport driver must report the maximum number of packet coalescing filters that are supported by the network adapter. The driver specifies this value in the **MaxPacketCoalescingFilters** member.

    Note Network adapters that support packet coalescing must support ten or more packet coalescing filters. If the adapter does not support packet coalescing, the miniport driver must set this value to zero.

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver reports the packet coalescing and filtering capabilities of the underlying network adapter by following these steps:

- The miniport driver initializes an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

  If the ***PacketCoalescing** keyword setting in the registry has a value of one, the miniport driver sets the **HardwareReceiveFilterCapabilities** member to a pointer to the previously initialized **NDIS_RECEIVE_FILTER_CAPABILITIES** structure.

  If the ***PacketCoalescing** keyword setting in the registry has a value of zero, the miniport driver does not advertise support for packet coalescing. It must set the **HardwareReceiveFilterCapabilities** member to NULL.

- The driver calls **NdisMSetMiniportAttributes** and sets the *MiniportAttributes* parameter to a pointer to the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

The method that is used by miniport drivers to report the packet coalescing and filtering capabilities of the underlying network adapter is based on the NDIS 6.20 method for reporting power management capabilities. For more information about this method, see Reporting Power Management Capabilities.

For more information about the adapter initialization process, see Initializing a Miniport Adapter.

Querying Packet Coalescing Capabilities

Article • 12/15/2021

Once the miniport driver is initialized, overlying drivers and applications can issue the following OID query requests to obtain the packet coalescing capabilities of the network adapter:

- OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES
- OID_RECEIVE_FILTER_CURRENT_CAPABILITIES
- OID_RECEIVE_FILTER_GLOBAL_PARAMETERS

NDIS handles these OID query requests for miniport drivers and returns the packet coalescing capabilities that the miniport driver registered when NDIS called the driver's *MiniportInitializeEx* function. Therefore, these OID query requests are not handled by miniport drivers.

For more information about how the miniport driver registers its packet coalescing capabilities, see Determining Receive Filtering Capabilities.

Guidelines for Managing Packet Coalescing Receive Filters

Article • 12/15/2021

If the miniport driver supports NDIS packet coalescing, it must follow these guidelines for managing packet coalescing receive filters:

- The miniport driver and underlying network adapter must be able to handle the setting and clearing of receive filters dynamically. Individual receive filters may be set or cleared at any time.

- The miniport driver must maintain a coalesced packet counter. This 64-bit counter contains a value for the number of received packets that have matched a packet coalescing filter. NDIS queries this counter through an OID query request of OID_PACKET_COALESCING_FILTER_MATCH_COUNT.

  Note The miniport driver clears this counter when it transitions to a full-power state by handling an OID set request of OID_PNP_SET_POWER. The miniport driver also clears the counter when its *MiniportResetEx* function is called.

- The miniport driver must not discard the packet coalescing receive filters when it transitions to a low-power state. However, while the network adapter is in a low-power state, it must only filter received packets based on wake-up patterns that have been offloaded to the adapter through OID set requests of OID_PNP_ENABLE_WAKE_UP.

  The miniport driver must configure the network adapter with the packet coalescing receive filters when the adapter transitions to a full-power state.

- The miniport driver must not discard the packet coalescing receive filters when NDIS calls the driver's *MiniportResetEx* function. After the driver resets the network adapter, it must configure the adapter with the packet coalescing filters. Also, the driver *must clear* the coalesced packet counter.

  Note The miniport driver must perform this operation regardless of whether the driver sets the *AddressingReset* parameter to TRUE.

- If the miniport driver is operating in the Native 802.11 extensible station (ExtSTA) mode, it must not discard the packet coalescing receive filters when it handles an OID method request of OID_DOT11_RESET_REQUEST. After the miniport driver performs the 802.11 reset operation, it must configure the network adapter with the packet coalescing receive filters. Also, the driver *must not clear* the coalesced packet counter.

For more information about the Native 802.11 extensible station mode, see Extensible Station Operation Mode.

Note NDIS does not support packet coalescing for native 802.11 miniport drivers that operate in extensible access point (ExtAP) mode. For more information about the ExtAP operation mode, see Extensible Access Point Operation Mode.

Specifying a Packet Coalescing Receive Filter

Article • 12/15/2021

An overlying driver can set one or more receive filters on a miniport driver that support NDIS packet coalescing. The overlying driver can specify up to the maximum number of receive filters that the miniport driver specified in the **MaxPacketCoalescingFilters** member of the NDIS_RECEIVE_FILTER_CAPABILITIES structure.

Note The overlying protocol driver obtains the NDIS_RECEIVE_FILTER_CAPABILITIES structure within the NDIS_BIND_PARAMETERS structure. The overlying filter driver obtains the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure within the NDIS_FILTER_ATTACH_PARAMETERS structure.

The overlying driver downloads receive filters to the miniport driver by issuing OID method requests of OID_RECEIVE_FILTER_SET_FILTER. The **InformationBuffer** member of the NDIS_OID_REQUEST structure for this OID request contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An NDIS_RECEIVE_FILTER_PARAMETERS structure that specifies the parameters for an NDIS receive filter.

  For more information about how to initialize this structure, see Specifying a Receive Filter.

- An array of NDIS_RECEIVE_FILTER_FIELD_PARAMETERS structures that specifies the filter test criterion for a field in a network packet header.

  For more information about how to initialize these structures, see Specifying Header Field Tests.

Specifying a Receive Filter

An overlying driver specifies a packet coalescing receive filter by initializing an NDIS_RECEIVE_FILTER_PARAMETERS structure with the configuration parameters for the filter. When it initializes the **NDIS_RECEIVE_FILTER_PARAMETERS** structure, the overlying driver must follow these rules:

- The **FilterType** member must be set to the NDIS_RECEIVE_FILTER_TYPE enumeration value of **NdisReceiveFilterTypePacketCoalescing**.

- The **QueueId** member must be set to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

  Note Starting with NDIS 6.30, packet coalescing receive filter are only supported on the default receive queue of the network adapter. This receive queue has an identifier of NDIS_DEFAULT_RECEIVE_QUEUE_ID.

- If the overlying driver is creating a new receive filter, it must set the **FilterId** member to NDIS_DEFAULT_RECEIVE_FILTER_ID.

  Note NDIS will generate a unique filter identifier (ID) for the receive filter before it forwards the OID method request of OID_RECEIVE_FILTER_SET_FILTER to the miniport driver.

- If the overlying driver is modifying an existing receive filter, it must set the **FilterId** member to the nonzero filter ID of the receive filter. The overlying driver obtains the filter ID for the receive filter when it issues an OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS. For more information about how to modify a receive filter, see Modifying Packet Coalescing Receive Filters.

- The **FieldParametersArrayOffset**, **FieldParametersArrayNumElements**, and **FieldParametersArrayElementSize** members of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure must be set to define a field parameter's array. Each element in the array is an **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure that specifies the parameters for a header field test of a receive filter.

- The **RequestedFilterIdBitCount** member must be set to zero.

- The **MaxCoalescingDelay** must be set to the maximum time, in units of milliseconds, that the first packet that matches the receive filter is saved and coalesced on the network adapter. As soon as the first packet that matches the filter is received, the network adapter coalesces the packet and starts a hardware timer whose expiration time is set to the value of the **MaxCoalescingDelay** member.

The overlying driver must order the header field tests in the field parameters array to be in the same order that the associated MAC and protocol headers would exist in a packet.

For example, before the overlying driver specifies the filter parameters for an IP version 4 (IPv4) protocol field, it must first specify the filter parameters for a MAC header protocol field (NdisMacHeaderFieldProtocol). In this manner, the driver specifies a header field test that verifies the field is set to the correct EtherType value (0x0800) for

IPv4 packets. If the test fails, the adapter does not have to perform the test of the IPV4 protocol field.

Specifying Header Field Tests

Each receive filter can specify one or more test criteria (*header field tests*). The network adapter performs these tests to determine whether a received packet should be coalesced in a hardware coalescing buffer on the adapter. Also, the overlying driver can specify separate filter tests for various media access control (MAC), IP version 4 (IPv4), and IP version 6 (IPv6) header fields.

To optimize filtering on the network adapter, header field tests are based on standardized header field names instead of byte offset/length specifications within the packet data. By using header/field names, the network adapter's hardware or firmware can optimize how multiple header field tests are performed on a received packet.

Each receive filter can contain one or more header field tests specified by an **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure. Each **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure is an element of the field parameters array that is referenced by the **FieldParametersArrayOffset**, **FieldParametersArrayNumElements**, and **FieldParametersArrayElementSize** members of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure.

The miniport driver must follow these guidelines when it handles an OID method request of OID_RECEIVE_FILTER_SET_FILTER:

- If the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is set in the **Flags** member of the **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure, the network adapter must only indicate received packets with a matching MAC address and untagged packets or packets with a VLAN identifier of zero. That is, the network adapter must not indicate packets with a matching MAC address and a nonzero VLAN identifier.

- If the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is not set and there is no VLAN identifier filter configured by an OID set request of OID_RECEIVE_FILTER_SET_FILTER, the miniport driver must do one of the following:

  - If the miniport driver supports NDIS 6.20, it must return a failed status for the OID request of OID_RECEIVE_FILTER_SET_FILTER.

  - If the miniport driver supports NDIS 6.30 or later versions of NDIS, it must configure the network adapter to inspect and filter the specified MAC address fields. If a VLAN tag is present in the received packet, the network adapter must remove it from the packet data. The miniport driver must put the VLAN tag in an NDIS_NET_BUFFER_LIST_8021Q_INFO that is associated with the packet's NET_BUFFER_LIST structure.

- If the overlying driver sets a MAC address filter and a VLAN identifier filter in the NDIS_RECEIVE_FILTER_PARAMETERS structure, it does not set the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag in either of the filter fields. In this case, the miniport driver should indicate packets that match both the specified MAC address and the VLAN identifier. That is, the miniport driver should not indicate packets with a matching MAC address that have a zero VLAN identifier or are untagged packets.

Setting a Packet Coalescing Receive Filter

Article • 12/15/2021

To download and set a receive filter on a miniport driver that supports packet coalescing, an overlying driver issues an OID method request of OID_RECEIVE_FILTER_SET_FILTER. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for the OID request contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An **NDIS_RECEIVE_FILTER_PARAMETERS** structure that specifies the parameters for an NDIS receive filter.

- An array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures that specifies the filter test criterion for a field in a network packet header.

For more information about how an overlying driver specifies the parameters for a packet coalescing receive filter, see Specifying a Packet Coalescing Receive Filter.

When NDIS receives an OID request to set a receive filter on the underlying network adapter, it verifies the receive filter parameters. If the overlying driver is specifying a new receive filter, NDIS will also generate a unique filter identifier (ID) for the receive filter.

After NDIS allocates the necessary resources and the filter ID, it forwards the OID request to the miniport driver. If the miniport driver can successfully allocate the necessary software and hardware resources for the receive filter, the miniport driver completes the OID request with a status of NDIS_STATUS_SUCCESS.

After a successful return from the OID method request of OID_RECEIVE_FILTER_SET_FILTER, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. This structure is updated by NDIS with the new filter ID.

Using the Filter ID

Article • 12/15/2021

Overlying drivers download receive filters to the miniport driver by issuing OID method requests of OID_RECEIVE_FILTER_SET_FILTER. Each receive filter that is downloaded to the miniport driver has a unique filter identifier (ID) that is generated by NDIS. The overlying driver must use this filter ID in later OID requests in order to do the following:

- Query the receive filter parameters. For more information about how to query the parameters of a receive filter, see Querying Packet Coalescing Receive Filters.
- Modify the receive filter parameters. For more information about how to modify the parameters of a receive filter, see Modifying Packet Coalescing Receive Filters.
- Free, or *clear*, a receive filter. For more information about how to clear receive filters, see Clearing Packet Coalescing Receive Filters.

The miniport driver must retain the filter IDs for the allocated receive filters. When it receives an OID request to modify, query, or free a receive filter, the miniport driver must verify that the specified filter ID in the OID request matches an allocated receive filter from a previous OID method request of OID_RECEIVE_FILTER_SET_FILTER. If the filter ID does not match any of the allocated receive filters, the miniport driver must complete the OID request with a failed status.

Querying Packet Coalescing Receive Filters

Article • 12/15/2021

Overlying drivers and applications can query packet coalescing receive filters that have been downloaded to a miniport driver by doing the following:

- Request an enumerated list of the receive filters on the miniport driver by issuing an OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS. For more information, see Enumerating the Receive Filters on a Miniport Driver.

- Request the test criterion parameters for a receive filter on the miniport driver by issuing an OID method request of OID_RECEIVE_FILTER_PARAMETERS. For more information, see Querying the Receive Filters on a Miniport Driver

NDIS handles the OID_RECEIVE_FILTER_ENUM_FILTERS and OID_RECEIVE_FILTER_PARAMETERS method OID requests for miniport drivers. NDIS obtained the information from an internal cache of the data that it received from the OID_RECEIVE_FILTER_SET_FILTER OID request.

Enumerating the Receive Filters on a Miniport Driver

To obtain a list of all the packet coalescing receive filters that have been downloaded to a miniport driver, overlying drivers and applications issue an OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure.

Note When the overlying driver or application initializes the **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure, it must set the **QueueId** member to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer is formatted to contain the following:

- An **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure that specifies a list of receive filters that are currently configured on a miniport driver.

- An array of **NDIS_RECEIVE_FILTER_INFO** structures about a receive filter that is currently configured on a miniport driver.

Querying the Parameters of a Receive Filters on a Miniport Driver

To obtain the parameters of a specific packet coalescing receive filter that was downloaded to the miniport driver, overlying drivers or applications issue an OID method request of OID_RECEIVE_FILTER_PARAMETERS. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. The overlying driver or application initializes the **NDIS_RECEIVE_FILTER_PARAMETERS** structure by setting the **FilterId** member to the nonzero ID value of the filter whose parameters are to be returned.

Note The overlying driver obtained the filter ID from an earlier OID method request of OID_RECEIVE_FILTER_SET_FILTER or OID_RECEIVE_FILTER_ENUM_FILTERS. The application can only obtain the filter ID from an earlier OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer is formatted to contain the following:

- An **NDIS_RECEIVE_FILTER_PARAMETERS** structure that specifies the parameters for an NDIS receive filter.

- An array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures that specifies the filter test criterion for one field in a network packet header.

Modifying Packet Coalescing Receive Filters

Article • 12/15/2021

To modify a receive filter on a miniport driver that supports packet coalescing, an overlying protocol or filter driver performs the following steps:

1. To obtain a list of all the packet coalescing receive filters that have been downloaded to a miniport driver, the overlying driver issues an OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure.

   Note When the overlying driver or application initializes the **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure, it must set the **QueueId** member to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

   After a successful return from the OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an updated **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure that is followed by one or more **NDIS_RECEIVE_FILTER_INFO** structures. Each **NDIS_RECEIVE_FILTER_INFO** structure specifies the identifier (ID) for a filter that is set on the network adapter.

2. To obtain the parameters of a specific packet coalescing receive filter that was downloaded to the miniport driver, the overlying driver issues OID method request of OID_RECEIVE_FILTER_PARAMETERS. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. The overlying driver or application initializes the **NDIS_RECEIVE_FILTER_PARAMETERS** structure by setting the **FilterId** member to the nonzero ID value of the filter whose parameters are to be returned.

   After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer is formatted to contain the following:

   - An **NDIS_RECEIVE_FILTER_PARAMETERS** structure that specifies the parameters for the NDIS receive filter.

   - An array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures that specifies the filter test criterion for one field in a network packet header.

3. The overlying driver modifies the receive filter to add, delete, or change the filter's set of test criterion. The driver does this by adding, deleting, or modifying individual **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures from the field parameter array specified by the **NDIS_RECEIVE_FILTER_PARAMETERS** structure.

   When the overlying driver has completed the modifications to the test criterion, it must update the members of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure to reflect the changes that were made to the receive filter. For example, the overlying driver must update the **FieldParametersArrayNumElements** member to contain the new number of elements in the array.

   For more information, see Specifying a Packet Coalescing Receive Filter.

4. The overlying driver issues an OID method request of OID_RECEIVE_FILTER_SET_FILTER to download the modified receive filter to the miniport driver.

   For more information, see Setting a Packet Coalescing Receive Filter.

Clearing Packet Coalescing Receive Filters

Article • 12/15/2021

To free, or *clear*, a receive filter on a miniport driver that supports packet coalescing, an overlying driver issues an OID set request of OID_RECEIVE_FILTER_CLEAR_FILTER. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS** structure.

The overlying driver, such as a protocol or filter driver, initializes the **NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS** structure in the following way:

- The **QueueId** member must be set to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

  Note Starting with NDIS 6.30, packet coalescing receive filters are only supported on the default receive queue of the network adapter. This receive queue has an identifier of NDIS_DEFAULT_RECEIVE_QUEUE_ID.

- The **FilterId** member must be set to the nonzero identifier (ID) value of the filter to be cleared on the miniport driver. The overlying driver obtained the filter ID from an earlier OID method request of OID_RECEIVE_FILTER_SET_FILTER or OID_RECEIVE_FILTER_ENUM_FILTERS.

  Note Only the overlying driver that set the packet coalescing receive filter can clear it.

Note Before it completes the unbind or detach operation, the overlying protocol or filter driver must clear all the packet coalescing receive filters that it set on the underlying miniport driver.

Handling Packet Coalescing Receive Filters

Article • 12/15/2021

Multiple receive filters are downloaded to a miniport driver through OID method requests of OID_RECEIVE_FILTER_SET_FILTER. Each filter can specify one or more tests (*header field tests*) that the network adapter uses to determine whether a received packet should be coalesced in a hardware coalescing buffer on the adapter.

Before the miniport driver configures the network adapter with the receive filters, the driver should optimize the receive filters based on the hardware capabilities of the adapter. For example, all receive filters require a header field test for the MAC header. Therefore, the driver could optimize filter rules based on the results of this test. This allows the adapter to determine which Open Systems Interconnection (OSI) layer 3 (L3) and layer 4 (L4) header field tests to perform next.

As soon as the network adapter has been configured with receive filters, it must do the following:

- All the header field test parameters for a particular filter must match on the received packet in order to coalesce the packet in the coalescing buffer.

  The network adapter combines the results from all header field tests of a receive filter with a logical AND operation. That is, if any header field test that is included in the array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures for a receive filter fails, the received packet does not meet the specified filter criterion and must not be coalesced.

- The network adapter only inspects packet data based on the specified header field test parameters. The adapter must ignore all header fields in the packet for which header field tests are not specified.

- If a received packet matches all the header field tests for any of the receive filters, the network adapter must coalesce the packet within the hardware coalescing buffer. As soon as the first packet is coalesced, the network adapter must start a hardware timer and must set the expiration time to the value of the **MaxCoalescingDelay** member of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure for the matching receive filter.

- As more packets are received that match a packet coalescing receive filter, the network adapter puts them into the coalescing buffer.

If the hardware timer is already running, the adapter must not stop or restart the timer for the matching receive filter. However, the adapter can configure the hardware timer with the smallest expiration value from matching receive filters. For example, when the driver receives a packet that matches receive filter *X*, the adapter starts the timer with the specified expiration value for that receive filter. If the adapter then receives a packet that matches receive filter *Y*, the adapter can reconfigure the hardware timer with the specified expiration value for that receive filter.

Note The network adapter must not reconfigure the hardware timer if the time that is remaining on the timer is less than a receive filter's expiration time.

- As soon as received packets are coalesced, the network adapter generates an interrupt if any of the following events occur:

  - If the available space within the hardware coalescing buffer reaches a hardware-specific low-water mark, the network adapter must generate a receive interrupt so that the miniport driver can process the coalesced receive packets.

  - If the hardware timer used for the hardware coalescing buffer expires, the network adapter must generate a receive interrupt so that the miniport driver can process the coalesced receive packets.

  - If a receive filter is cleared and packets have been coalesced that match that filter, the network adapter must generate a receive interrupt so that the miniport driver can process the coalesced receive packets.

  - If a received packet does not match any of the receive filters, the network adapter must generate a receive interrupt so that the miniport driver can process the received packet. If any packets have been coalesced, the miniport driver must also process those packets.

  - If the network adapter generates an interrupt for any other interrupt status other than a receive interrupt, the network adapter must also signal a receive interrupt status so that the miniport driver can process the coalesced received packets.

  As soon as the interrupt is generated, the network adapter must stop the hardware timer if it hasn't expired and must clear the hardware coalescing buffer.

The miniport driver must maintain a coalesced packet counter, which contains a value for the number of received packets that have matched a packet coalescing filter. NDIS queries this counter through an OID query request of OID_PACKET_COALESCING_FILTER_MATCH_COUNT.

The network adapter only performs packet coalescing while the hardware is operating in a full-power state. While the hardware is in a low-power state, the adapter must only filter received packets based on wake-up patterns that have been offloaded to the adapter through OID set requests of OID_PNP_ENABLE_WAKE_UP.

When the network adapter transitions to a full-power state, the miniport driver must follow these steps:

- The miniport driver must configure the network adapter to discard any coalesced packets within the hardware coalescing buffer. The network adapter may have coalesced these packets when it was transitioned to a low-power state.

- The miniport driver must configure the network adapter with the set of packet coalescing receive filters that were downloaded to the driver before the low-power transition.

- The miniport driver must clear the coalesced packet counter.

Standardized INF Keywords for Packet Coalescing

Article • 12/15/2021

A standardized INF keyword is defined to enable or disable support for packet coalescing on a miniport driver.

The INF file for the miniport driver of an adapter that supports packet coalescing must specify the ***PacketCoalescing** standardized INF keyword. Once the driver is installed, administrators can update the ***PacketCoalescing** keyword value in the **Advanced** property page for the adapter. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

Note The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter.

The ***PacketCoalescing** INF keyword is an enumeration keyword. The following table describes the possible INF entries for the ***PacketCoalescing** INF keyword. The columns in this table describe the following attributes for an enumeration keyword:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params\** key for the network adapter.

ParamDesc
The display text that is associated with SubkeyName.

Note The independent hardware vendor (IHV) can define any descriptive text for the SubkeyName.

Value
The enumeration integer value that is associated with each SubkeyName in the list.

EnumDesc
The display text that is associated with each value that appears in the menu.

SubkeyName	ParamDesc	Value	EnumDesc
*PacketCoalescing	Packet coalescing	0	Disabled
		1 (Default)	Enabled

The miniport driver must check the ***PacketCoalescing** keyword value in the registry before it advertises its support for packet coalescing. If the ***PacketCoalescing** keyword has a value of zero, the miniport must not advertise support for any packet coalescing capabilities. For more information, see Reporting Packet Coalescing Capabilities.

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

Introduction to NDIS Selective Suspend

Article • 03/14/2023

Starting with NDIS 6.30, the NDIS selective suspend interface allows NDIS to suspend an idle network adapter by transitioning the adapter to a low-power state. This enables the system to reduce the power overhead on the CPU and network adapter.

This section includes the following topics:

Overview of NDIS Selective Suspend

Reporting NDIS Selective Suspend Capabilities

Registering NDIS Selective Suspend Handler Functions

NDIS Selective Suspend Idle Notifications

Standardized INF Keywords for NDIS Selective Suspend

NDIS Selective Suspend Implementation Guidelines

Note Although the NDIS selective suspend interface is especially useful for USB network adapters, the interface is bus-independent. As a result, miniport drivers can use the interface for network adapters on other bus types in order to reduce CPU and power overhead.

Overview of NDIS Selective Suspend

Article • 03/14/2023

Starting with NDIS 6.30, the NDIS selective suspend interface enables NDIS to suspend an idle network adapter by transitioning the adapter to a low-power state. This enables the system to reduce the CPU and power overhead of the adapter.

NDIS selective suspend is especially useful for network adapters that are based on the USB v1.1 and v2.0 interface. These adapters are continuously polled for received packets regardless of whether they are active or idle. By suspending idle USB adapters, the CPU overhead can be reduced by as much as 10 percent.

NDIS selective suspend is based on the USB selective suspend technology. However, NDIS selective suspend is designed to be bus-independent. In this way, bus-independent I/O request packets (IRPs) for selective suspend are issued by NDIS. This makes the miniport driver responsible for issuing any IRPs that are required for selective suspend on a specific bus. For example, miniport drivers for USB network adapters issue the bus-specific USB idle request IRP (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) to the USB bus driver during a selective suspend operation.

NDIS and the miniport driver participate in NDIS selective suspend in the following way:

1. If a miniport driver has registered its support for NDIS selective suspend, NDIS monitors the I/O activity of the network adapter. I/O activity includes receive packet indications, send packet completions, and OID requests that are handled by the miniport driver.

2. NDIS considers the network adapter to be idle if it has been inactive for longer than a specified idle time-out period. When this happens, NDIS starts a selective suspend operation by issuing an idle notification to the miniport driver in order to transition the network adapter to a low-power state.

   > ⓘ **Note**
   >
   > The length of the idle time-out period is specified by the value of the ***SSIdleTimeout** standardized INF keyword. For more information about this keyword, see **Standardized INF Keywords for NDIS Selective Suspend**.

   For more information about how NDIS determines that a network adapter is idle, see How NDIS Detects Idle Network Adapters.

3. NDIS issues the idle notification to the miniport driver by calling the driver's *MiniportIdleNotification* handler function. When this function is called, the miniport driver determines whether the network adapter can transition to a low-power state. The miniport driver performs this determination in a bus-specific manner.

   For example, a USB miniport driver determines whether the network adapter can transition to a low-power state by issuing a USB idle request IRP (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) to the underlying USB bus driver. This informs the bus driver that the network adapter is idle and confirms whether the adapter can be transitioned to a low-power state.

   > ⓘ Note
   >
   > The miniport driver must specify a callback and completion routine for the USB idle request IRP.

   For more information about how a miniport driver handles an idle notification, see Handling the NDIS Selective Suspend Idle Notification.

4. After the miniport driver confirms that the network adapter can transition to a low-power state, it calls **NdisMIdleNotificationConfirm**. In this call, the miniport driver specifies the lowest power state that the network adapter can transition to.

5. When **NdisMIdleNotificationConfirm** is called, NDIS issues OID requests to the miniport driver to prepare the adapter for the transition to a low-power state. NDIS also issues IRPs to the underlying bus driver to set the adapter to a low-power state.

6. After the network adapter has been suspended, it remains in a low power state until the outstanding idle notification is canceled.

   NDIS cancels the outstanding idle notification by calling the miniport driver's *MiniportCancelIdleNotification* handler function. NDIS calls this handler function if one or more of the following conditions are true:

   - NDIS detects send packet requests or OID requests that are issued to the miniport driver from overlying protocol or filter drivers.

   - The network adapter signals a wake-up event. This might occur when the adapter receives a packet or detects a change in its media connection status.

   After the network adapter has been suspended, the miniport driver can also complete the idle notification in order to resume the adapter to a full-power state.

The reasons for doing this are specific to the design and requirements of the driver and adapter.

For more information about how NDIS cancels the idle notification, see Canceling the NDIS Selective Suspend Idle Notification.

For more information about how the miniport driver completes the idle notification, see Completing the NDIS Selective Suspend Idle Notification.

7. When the *MiniportCancelIdleNotification* handler function is called, the miniport driver determines whether the network adapter can resume to a full-power state. The driver also cancels any bus-specific IRPs that it may have previously issued for the idle notification.

   The determination that the network adapter can transition to a full-power state is bus-specific. For example, when *MiniportCancelIdleNotification* is called, the USB miniport must cancel the previously issued USB idle request IRP. As soon as the USB driver has canceled the IRP, it calls the IRP's completion routine to confirm that the IRP is canceled and the network adapter can resume to a full-power state. In the context of the completion routine, the miniport driver calls **NdisMIdleNotificationComplete**.

   When the miniport determines that the network adapter can resume to a full-power state, it calls **NdisMIdleNotificationComplete**. This call notifies NDIS that the idle notification has been completed. NDIS then proceeds with completing the selective suspend operation by transitioning the network adapter to a full-power state.

8. When **NdisMIdleNotificationComplete** is called, NDIS issues OID requests to the miniport driver to prepare the adapter for the transition to a full-power state. NDIS also issues IRPs to the underlying bus driver to set the adapter to a full-power state.

9. When the network adapter resumes to a full-power state, the selective suspend operation is completed. NDIS resumes monitoring the I/O activity of the network adapter. If the adapter becomes inactive after another idle time-out period, NDIS issues an idle notification to the miniport driver in order to suspend the network adapter.

Reporting NDIS Selective Suspend Capabilities

Article • 03/14/2023

Starting with NDIS 6.30, miniport drivers must report whether the driver has enabled the support for NDIS selective suspend. The support for NDIS selective suspend is enabled or disabled through the setting of the ***SelectiveSuspend** standardized INF keyword. For more information about this INF keyword, see Standardized INF Keywords for NDIS Selective Suspend.

When NDIS calls the driver's *MiniportInitializeEx* function, the miniport driver reports its support for NDIS selective suspend support by following these steps:

1. The driver initializes an **NDIS_PM_CAPABILITIES** structure with the power management capabilities of the underlying hardware.

   If the driver enables the support for NDIS selective suspend, it must set the members of the **NDIS_PM_CAPABILITIES** structure as follows:

   - The miniport driver must specify NDIS_PM_CAPABILITIES_REVISION_2 and NDIS_SIZEOF_NDIS_PM_CAPABILITIES_REVISION_2 for the revision and length of the **NDIS_PM_CAPABILITIES** structure within the structure's **Header** member.
   - If the ***SelectiveSuspend** keyword has a value of one, the miniport driver support for NDIS selective suspend is enabled. The miniport driver reports this by setting the NDIS_PM_SELECTIVE_SUSPEND_SUPPORTED flag within the **Flags** member of this structure.

2. Once it has initialized the **NDIS_PM_CAPABILITIES** structure, the miniport driver sets the **PowerManagementCapabilitiesEx** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure to point to the initialized **NDIS_PM_CAPABILITIES** structure. The miniport driver passes a pointer to an **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure in the *MiniportAttributes* parameter when the driver calls the **NdisMSetMiniportAttributes** function.

The method that is used by miniport drivers to report the support status of NDIS selective suspend is based on the NDIS 6.20 method for reporting power management capabilities. For more information about this method, see Reporting Power Management Capabilities.

For more information about the adapter initialization process, see Initializing a Miniport Adapter.

Registering NDIS Selective Suspend Handler Functions

Article • 03/14/2023

If a miniport driver supports NDIS selective suspend, NDIS notifies the driver that the underlying network adapter has become idle. The miniport driver must provide the following functions to handle these idle notifications:

MiniportIdleNotification
NDIS calls the *MiniportIdleNotification* handler function to notify the miniport driver that the network adapter has become idle. The miniport driver handles the idle notification by determining whether the network adapter can transition to a low-power state. The miniport driver performs this determination in a bus-specific manner.

For example, a USB miniport driver determines whether the network adapter can transition to a low-power state by issuing an I/O request packet (IRP) for a USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) to the underlying USB bus driver. Through the processing of this IRP, the miniport driver is notified that the adapter is idle and can be transitioned to a low-power state.

MiniportCancelIdleNotification
NDIS calls the *MiniportCancelIdleNotification* handler function to cancel the outstanding idle notification. When this function is called, the miniport driver cancels any bus-specific IRPs that it may have previously issued for the idle notification.

For example, when *MiniportCancelIdleNotification* is called, the USB miniport must cancel the previously-issued USB idle request IRP. When the IRP is canceled, the miniport driver is notified that the adapter can now be transitioned to a full-power state.

When the miniport driver's **DriverEntry** function is called, the driver registers its NDIS selective suspend handler functions by following these steps:

1. The miniport driver must set the **SetOptionsHandler** member of the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure to the entry point for the driver's *MiniportSetOptions* function. The driver calls **NdisMRegisterMiniportDriver** to register its **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure with NDIS.

2. NDIS calls the *MiniportSetOptions* function in the context of the call to **NdisMRegisterMiniportDriver**.

   When *MiniportSetOptions* is called, the miniport driver initializes an **NDIS_MINIPORT_SS_CHARACTERISTICS** structure with pointers to the handler functions. The miniport driver then calls **NdisSetOptionalHandlers** and sets the *OptionalHandlers* parameter to a pointer to the **NDIS_MINIPORT_SS_CHARACTERISTICS** structure.

For more information on how to handle idle notifications for NDIS selective suspend, see NDIS Selective Suspend Idle Notifications.

Overview of NDIS Selective Suspend Idle Notifications

Article • 03/14/2023

When the miniport driver has enabled and registered support for NDIS selective suspend, NDIS monitors the I/O activity of the underlying network adapter. If NDIS determines that the driver and adapter are idle, NDIS performs a selective suspend operation. This operation suspends the network adapter by transitioning the adapter to a low-power state.

NDIS starts the selective suspend operation by issuing an idle notification to the miniport driver. When the network adapter is suspended in a low-power state, the adapter can resume to a full-power state only when the idle notification is canceled. After the notification is canceled and the adapter is in a full-power state, the selective suspend operation is complete.

The following topics provide more information on the NDIS selective suspend operation and idle notifications:

How NDIS Detects Idle Network Adapters

Handling the NDIS Selective Suspend Idle Notification

Canceling the NDIS Selective Suspend Idle Notification

Completing the NDIS Selective Suspend Idle Notification

How NDIS Detects Idle Network Adapters

Article • 03/14/2023

After the miniport driver has enabled NDIS selective suspend and registered its handler functions, NDIS monitors the I/O activity of the network adapter in the following way:

- NDIS monitors the calls to the I/O handler functions that the miniport driver registers through the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** and **NDIS_MINIPORT_PNP_CHARACTERISTICS** structures. For example, NDIS monitors calls to the miniport driver's *MiniportSendNetBufferLists* or *MiniportReturnNetBufferLists* to determine whether the driver is involved in any packet I/O activity.

- NDIS also monitors the calls of **NdisOidRequest** and **NdisDirectOidRequest** made by overlying protocol drivers.

  Note NDIS monitors only those object identifier (OID) requests to the underlying miniport driver that are not handled directly by NDIS.

NDIS determines that the network adapter is idle if it does not detect any activity on the adapter for an idle time-out period. The duration of this time-out period is specified by the value of the ***SSIdleTimeout** standardized INF keyword. For more information about this keyword, see Standardized INF Keywords for NDIS Selective Suspend.

After the network adapter has become idle, NDIS starts the selective suspend operation. Through this operation, the network adapter is suspended by transitioning it to a low-power state.

NDIS begins this selective suspend operation by issuing an idle notification to the miniport driver. NDIS does this by calling the driver's *MiniportIdleNotification* handler function. For more information about how the miniport driver handles this notification, see Handling the NDIS Selective Suspend Idle Notification.

If NDIS detects that I/O requests to the network adapter are issued from overlaying drivers or if the adapter signals a wake-up event, NDIS cancels the idle notification. NDIS does this by calling the miniport driver's *MiniportCancelIdleNotification* handler function.

For more information about how NDIS cancels the idle notification, see Canceling the NDIS Selective Suspend Idle Notification.

For more information about how the miniport driver completes the idle notification, see Completing the NDIS Selective Suspend Idle Notification.

Handling the NDIS Selective Suspend Idle Notification

Article • 03/14/2023

NDIS starts a selective suspend operation if one of the following events occurs:

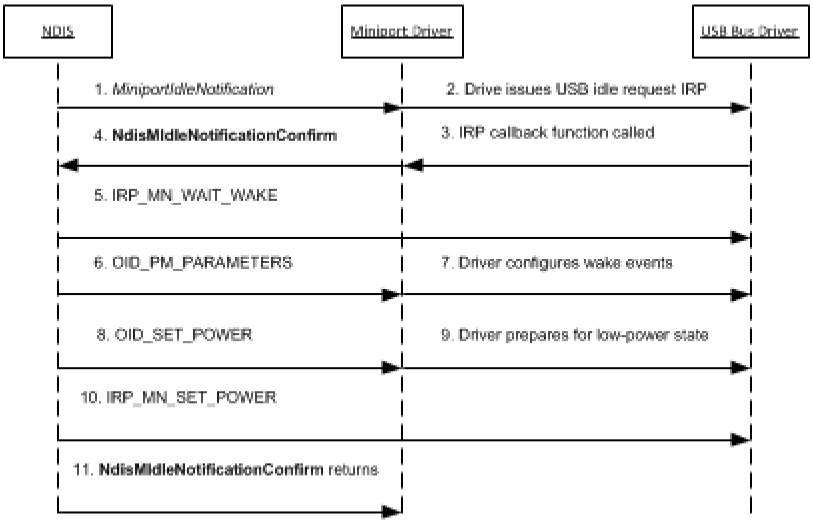
- The network adapter has been inactive for longer than an idle time-out period. The duration of this time-out period is specified by the value of the ***SSIdleTimeout** standardized INF keyword. For more information about this keyword, see Standardized INF Keywords for NDIS Selective Suspend.

  For more information about how NDIS determines that a network adapter is idle, see How NDIS Detects Idle Network Adapters.

- The system that is compliant with the Always On Always Connected (AOAC) technology is being transitioned to a Connected Standby state.

Through the selective suspend operation, the network adapter is transitioned to a low-power state. NDIS begins this operation by calling the *MiniportIdleNotification* handler function to issue an idle notification to the miniport driver.

The miniport driver may need to perform bus-dependent actions when it handles the idle notification. The following figure shows the steps that are involved with handling an idle notification by a miniport driver for a USB network adapter.


NDIS
Miniport Driver
USB Bus Driver
1. *MiniportIdleNotification*
2. Drive issues USB idle request IRP
3. IRP callback function called
4. **NdisMIdleNotificationConfirm**
5. IRP_MN_WAIT_WAKE
6. OID_PM_PARAMETERS
7. Driver configures wake events
8. OID_SET_POWER
9. Driver prepares for low-power state
10. IRP_MN_SET_POWER
11. **NdisMIdleNotificationConfirm** returns

This topic includes the following information about how to handle an NDIS selective suspend idle notification:

Guidelines for Handling the Call to *MiniportIdleNotification*

Guidelines for the Call to **NdisMIdleNotificationConfirm**

Canceling and Completing an NDIS Selective Suspend Idle Notification

Guidelines for Handling the Call to *MiniportIdleNotification*

NDIS and the miniport driver follow these steps when NDIS calls *MiniportIdleNotification*:

1. NDIS calls the *MiniportIdleNotification* handler function to notify the driver that the underlying network adapter seems to be idle. NDIS sets the *ForceIdle* parameter of the *MiniportIdleNotification* handler function to one of the following values:

   - NDIS sets the *ForceIdle* parameter to **FALSE** when the network adapter has been inactive for longer than the idle time-out period.

   - NDIS sets the *ForceIdle* parameter to **TRUE** when a system that is compliant with the Always On Always Connected (AOAC) technology is transitioning to a Connected Standby state.

2. When *MiniportIdleNotification* is called, the miniport driver can veto the idle notification and the selective suspend operation by returning NDIS_STATUS_BUSY. For example, the driver could veto the idle notification if the driver detects activity on the network adapter.

   If the miniport driver vetoes the idle notification, NDIS restarts the monitor of activity on the network adapter. If the adapter becomes inactive again within the idle time-out period, NDIS calls *MiniportIdleNotification*.

   Note The miniport driver must not veto the idle notification if the *ForceIdle* parameter is set to **TRUE**. In this case, the driver must continue with the selective suspend operation.

3. If the miniport driver does not veto the idle notification, it must perform any bus-specific operations to prepare the network adapter for a selective suspend operation. For example, the miniport driver for a USB network adapter performs the following steps to determine whether the network adapter can transition to a low-power state:

   a. The miniport driver calls **IoCallDriver** to issue an I/O request packet (IRP) for a USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) to the underlying USB bus driver. In this IRP, the miniport driver must specify a callback and completion routine.

The USB bus driver does not immediately complete the IRP. The IRP is left in a pending state through the low-power transition. The bus driver completes the IRP later when any of the following events occur:

    - The miniport driver cancels the IRP.
    - A system power state change is required.
    - The device is removed from the USB hub.

   b. After the USB bus driver determines that it can put the network adapter in a low-power state, it calls the miniport driver's IRP callback routine. This call confirms that the network adapter can transition to a low-power state.

      For guidelines on how to write a callback routine for the USB idle request IRP, see Implementing a USB Idle Request IRP Callback Routine.

4. After the miniport driver completes the preparation of the network adapter for a selective suspend operation, it calls **NdisMIdleNotificationConfirm**. In this call, the miniport driver specifies the lowest power state that the network adapter can transition to.

   Depending on the bus requirements for selective suspend operations, the miniport driver calls **NdisMIdleNotificationConfirm** either synchronously in the context of the call to *MiniportIdleNotification* or asynchronously after *MiniportIdleNotification* returns. For example, the miniport driver for a USB network adapter calls **NdisMIdleNotificationConfirm** within the context of the callback routine for the USB idle request. The USB bus driver calls the callback routine either synchronously in the context of the call to **IoCallDriver** or asynchronously after *MiniportIdleNotification* returns.

5. If the network adapter can be transitioned to a low-power state, the miniport driver returns NDIS_STATUS_PENDING from the call to *MiniportIdleNotification*.

   Note The miniport driver returns NDIS_STATUS_PENDING because the idle notification is not completed until the driver calls **NdisMIdleNotificationComplete**. The miniport driver must not return NDIS_STATUS_SUCCESS from *MiniportIdleNotification*.

The miniport driver should perform the following operations until the network adapter is suspended and transitioned to a low-power state:

- The miniport driver should process received packets and indicate them to NDIS by calling **NdisMIndicateReceiveNetBufferLists**.

- The miniport driver should process completed send packets and indicate them to NDIS by calling **NdisMSendNetBufferListsComplete**.

  Note NDIS will not call the driver's *MiniportSendNetBufferLists* function to send packets if *MiniportIdleNotification* returns NDIS_STATUS_PENDING.

Guidelines for the Call to NdisMIdleNotificationConfirm

NDIS and the miniport driver follow these steps when the miniport driver calls **NdisMIdleNotificationConfirm**:

1. NDIS issues **IRP_MN_WAIT_WAKE** to the underlying bus driver. This IRP enables the bus driver to wake the network adapter in response to an external wake-up signal.

2. NDIS issues an object identifier (OID) set request of OID_PM_PARAMETERS to the miniport driver. This OID request is associated with an **NDIS_PM_PARAMETERS** structure that specifies the settings under which the network adapter generates a wake-up event.

   The miniport driver must follow these guidelines when it processes the members of the **NDIS_PM_PARAMETERS** structure:

   - If the *ForceIdle* parameter of the *MiniportIdleNotification* handler function was set to FALSE, NDIS only sets the NDIS_PM_SELECTIVE_SUSPEND_ENABLED flag in the **WakeUpFlags** member of the **NDIS_PM_PARAMETERS** structure. In this case, the network adapter can signal a wake-up event when one of the following events occur:

     - The network adapter receives a packet that matches a receive packet filter. The adapter is configured to use these filters through OID set requests of OID_GEN_CURRENT_PACKET_FILTER.

     - The network adapter detects other external events that require processing by the networking driver stack, such as when the link state changes to either media disconnect or media connected.

   - If the *ForceIdle* parameter of the *MiniportIdleNotification* handler function was set to **TRUE**, NDIS does not set the NDIS_PM_SELECTIVE_SUSPEND_ENABLED flag in the **WakeUpFlags** member of the **NDIS_PM_PARAMETERS** structure. In this case, NDIS sets other members in the **NDIS_PM_PARAMETERS** structure for wake-up events not related to NDIS selective suspend.

Note NDIS sets the *ForceIdle* parameter to **TRUE** only when a system that is compliant with the Always On Always Connected (AOAC) technology is transitioning to a Connected Standby state.

The driver completes the OID request with NDIS_STATUS_SUCCESS.

Note If NDIS sets the NDIS_PM_SELECTIVE_SUSPEND_ENABLED flag in the **WakeUpFlags** member of **NDIS_PM_PARAMETERS** structure, it issues the OID set request of OID_PM_PARAMETERS directly to the miniport driver. This allows NDIS to bypass the processing by filter drivers in the networking driver stack.

3. After the OID set request of OID_PM_PARAMETERS is completed successfully, NDIS issues an OID set request OID_PNP_SET_POWER to the miniport driver.

   When it handles this OID set request, the driver prepares the network adapter to transition to the low-power state that is specified in the OID request. The driver must complete all pending operations in the following way:

   - The miniport driver waits for all previously indicated receive packets to be returned through calls to *MiniportReturnNetBufferLists*.

   - The miniport driver waits for send requests processed by the hardware to finish. After the requests are completed, the miniport driver must call **NdisMSendNetBufferListsComplete**.

   - The miniport driver completes all pending send requests by calling **NdisMSendNetBufferListsComplete**.

   - The miniport driver must cancel all pending NDIS timers and work items. After these are canceled, the driver must wait for the completion of these timers and work items.

   - The miniport driver must put the network adapter in a quiescent state. For example, the driver must cancel all hardware timers.

   The miniport driver configures the underlying network adapter to enable the specified wake-up events that were previously specified in the OID set request of OID_PM_PARAMETERS. After the network adapter is prepared for the low-power transition, the miniport driver completes the OID set request of OID_PNP_SET_POWER with NDIS_STATUS_SUCCESS.

4. NDIS issues an **IRP_MN_SET_POWER** to the underlying bus driver. This IRP requests that the network adapter be transitioned to a low-power state.

> **Note** During a selective suspend operation, the network adapter will be transitioned to the device power state that was specified in the call to **NdisMIdleNotificationConfirm**. The miniport driver specifies this device power state in the *IdlePowerState* parameter of this function.

After the IRP is completed, NDIS returns from the call to **NdisMIdleNotificationConfirm**.

Canceling and Completing an NDIS Selective Suspend Idle Notification

After the idle notification is issued, it can be canceled and completed in the following ways:

- NDIS can cancel the outstanding idle notification if the following conditions are true:

  - An overlying protocol or filter driver issues either a send packet request or an OID request to the miniport driver.

  - The underlying adapter signals a wake-up event, such as receiving a packet that matches a wake-on-LAN (WOL) pattern or detecting a change in its media connection status.

  NDIS cancels the idle notification by calling *MiniportCancelIdleNotification*. When this handler function is called, the miniport driver cancels any bus-specific IRPs that it may have previously issued for the idle notification. Finally, the miniport driver calls **NdisMIdleNotificationComplete** to complete the idle notification.

  For more information about how NDIS cancels the idle notification, see Canceling the NDIS Selective Suspend Idle Notification.

- After the network adapter is in a low-power state, the miniport driver can complete the idle notification itself to resume the adapter to a full-power state. The reasons for doing this are specific to the design and requirements of the driver and adapter. The miniport driver completes the idle notification by calling **NdisMIdleNotificationComplete**.

  For more information about how the miniport driver completes the idle notification, see Completing the NDIS Selective Suspend Idle Notification.

Canceling the NDIS Selective Suspend Idle Notification

Article • 03/14/2023

If the network adapter becomes inactive for an idle time-out period, NDIS starts the selective suspend operation. Through this operation, the network adapter is transitioned to a low-power state. NDIS begins this operation by issuing an idle notification to the miniport driver. For more information about this operation, see Handling the NDIS Selective Suspend Idle Notification.

NDIS calls the *MiniportIdleNotification* handler function to notify the driver that the underlying network adapter seems to be idle. After the idle notification is issued, NDIS cancels a pending idle notification if one or more of the following conditions are true:

- An overlying protocol or filter driver issues either a send packet request or an object identifier (OID) request to the miniport driver.

  For more information about how NDIS cancels the idle notification for this scenario, see Canceling the Idle Notification because of Overlying Driver Activity.

- The underlying adapter signals a wake-up event, such as receiving a packet or detecting a change in its media connection status.

  For more information about how NDIS cancels the idle notification for this scenario, see Canceling the Idle Notification because of Wake-up Events.

NDIS cancels the idle notification by calling the *MiniportCancelIdleNotification* handler function of the underlying miniport driver. When this function is called, the miniport driver must complete the idle notification to resume the adapter to a full-power state. For guidelines on this process, see Completing the NDIS Selective Suspend Idle Notification.

For more information about how to implement the *MiniportCancelIdleNotification* handler function, see Implementing a *MiniportCancelIdleNotification* Handler Function.

Canceling the Idle Notification Because of Overlying Driver Activity

NDIS monitors send requests and OID requests that are issued to a miniport driver whose network adapter has been suspended and is in a low-power state. When this happens, NDIS cancels the outstanding idle notification so that the network adapter can resume to a full-power state.

NDIS and the miniport driver follow these steps when an idle notification is canceled:

1. NDIS calls the *MiniportCancelIdleNotification* handler function to cancel an outstanding idle notification. When this handler function is called, the miniport driver must cancel any bus-specific I/O request packets (IRPs) that it may have previously issued for the idle notification.

   For example, when *MiniportCancelIdleNotification* is called, the miniport for a USB network adapter performs the following steps:

   a. The miniport driver cancels the pending USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) IRP. The miniport driver previously issued this IRP to the underlying USB bus driver when NDIS called the driver's *MiniportIdleNotification* function. The miniport driver cancels this IRP by calling **IoCancelIrp**.

   b. When the bus driver cancels the USB idle request IRP, it calls the miniport driver's completion routine for the IRP. This call notifies the driver that the IRP is completed and the network adapter can transition to a full-power state. From the context of the completion routine, the driver calls **NdisMIdleNotificationComplete** to notify NDIS that the network adapter can be transitioned to a full-power state.

   Note Depending on the dependencies for canceling bus-specific idle requests, the miniport driver calls **NdisMIdleNotificationComplete** either synchronously in the context of the call to *MiniportCancelIdleNotification* or asynchronously after *MiniportCancelIdleNotification* returns.

   For more information about how to implement a USB idle request IRP completion routine, see Implementing a USB Idle Request IRP Completion Routine.

2. After the miniport driver cancels any bus-specific IRPs for the idle notification, it calls **NdisMIdleNotificationComplete**. This call notifies NDIS that the idle notification has been completed. NDIS then completes the selective suspend operation by transitioning the network adapter to a full-power state.

   When **NdisMIdleNotificationComplete** is called, NDIS performs the following steps:

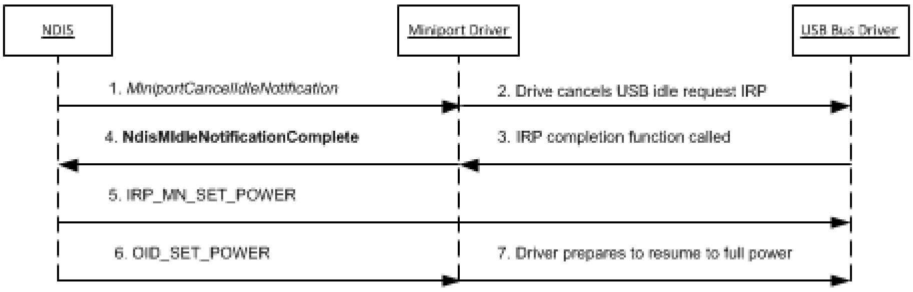
   a. NDIS issues **IRP_MN_SET_POWER** to the underlying bus driver. This IRP requests the bus driver to set the power state of the network adapter to

PowerDeviceD0.

b. NDIS issues an OID set request of OID_PNP_SET_POWER to the miniport driver. In this OID request, NDIS specifies that the network adapter is now transitioning to a full-power state of NdisDeviceStateD0.

   When it handles this OID set request, the driver prepares the adapter for full-power operation. This includes restoring the receive and send engines to the same state they were in before the transition to the low-power state. The driver then completes the OID request with NDIS_STATUS_SUCCESS.

The following figure shows the steps that are involved when NDIS cancels an idle notification that was issued to a miniport driver for a USB network adapter.


NDIS
Miniport Driver
USB Bus Driver
1. MiniportCancelIdleNotification
2. Drive cancels USB idle request IRP
3. IRP completion function called
4. NdisMIdleNotificationComplete
5. IRP_MN_SET_POWER
6. OID_SET_POWER
7. Driver prepares to resume to full power

Canceling the Idle Notification Because of Wake-up Events

Before the network adapter is transitioned to a low-power state, NDIS issues an OID set request of OID_PM_PARAMETERS to the network adapter. This OID request specifies the types of wake-up events that the adapter can signal to resume to a full-power state. For NDIS selective suspend, the adapter is configured to signal any of the following wake-up events:

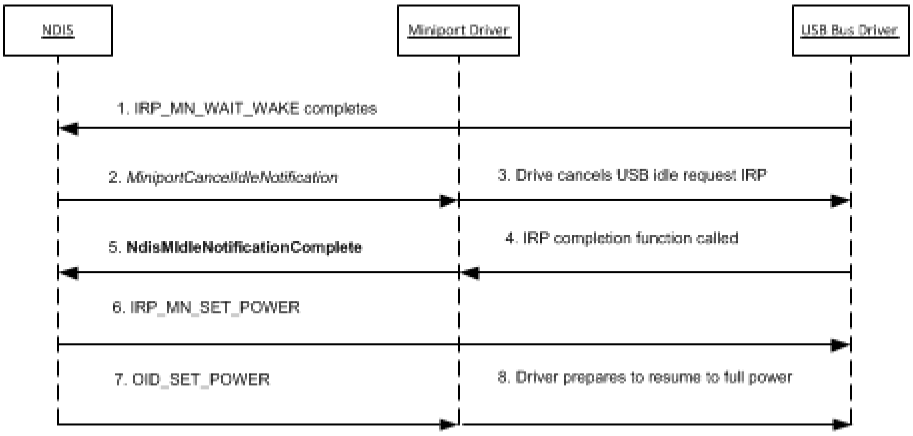
- The reception of a packet that matches a filter that was previously configured through an OID set request of OID_PM_ADD_WOL_PATTERN or OID_GEN_CURRENT_PACKET_FILTER.
- A change in the media connection status on the adapter.

NDIS and the miniport driver follow these steps when NDIS cancels an idle notification because of a wake-up signal generated by the network adapter:

1. The bus driver completes the **IRP_MN_WAIT_WAKE** that was issued by NDIS before transitioning the adapter to a low-power state. By completing the IRP, the bus driver notifies NDIS that the network adapter has generated a wake-up signal.

2. NDIS calls the *MiniportCancelIdleNotification* handler function to start the operation of canceling the idle notification. The steps that are involved in this operation are the same as described in Canceling the Idle Notification because of Overlying Driver Activity.

For example, the following figure shows the steps that are involved when NDIS cancels an idle notification because of a wake-up event signaled by a USB network adapter.


NDIS
Miniport Driver
USB Bus Driver
1. IRP_MN_WAIT_WAKE completes
2. *MiniportCancelIdleNotification*
3. Drive cancels USB idle request IRP
4. IRP completion function called
5. **NdisMIdleNotificationComplete**
6. IRP_MN_SET_POWER
7. OID_SET_POWER
8. Driver prepares to resume to full power

Completing the NDIS Selective Suspend Idle Notification

Article • 03/14/2023

NDIS calls the *MiniportIdleNotification* handler function to notify the driver that the underlying network adapter seems to be idle. For more information about this operation, see Handling the NDIS Selective Suspend Idle Notification.

After the idle notification is issued, the miniport driver completes the NDIS selective suspend idle notification under the following conditions:

- NDIS cancels the idle notification by calling the *MiniportCancelIdleNotification* handler function of the underlying miniport driver.
- The miniport driver completes the idle notification itself. The reasons for doing this are specific to the design and requirements of the driver and adapter. For example, the driver could complete the idle notification if it detects receive activity on the network adapter.

Note The miniport driver cannot explicitly cancel the idle notification. When NDIS cancels the idle notification, the miniport driver must complete the notification as described in this topic. For more information, see Canceling the NDIS Selective Suspend Idle Notification.

In either case, the miniport driver must complete the idle notification to resume the adapter to a full-power state. To complete the idle notification, the miniport driver must cancel any bus-specific I/O request packets (IRPs) that it may have previously issued for the idle notification. Finally, the driver calls **NdisMIdleNotificationComplete** to notify NDIS that the network adapter can be transitioned to a full-power state.

For example, the miniport driver for a USB network adapter completes an idle notification by following these steps:

1. The miniport driver cancels the pending USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) IRP. The miniport driver previously issued this IRP to the underlying USB bus driver when NDIS called the driver's *MiniportIdleNotification* function. The miniport driver cancels this IRP by calling **IoCancelIrp**.
2. When the bus driver cancels the USB idle request IRP, it calls the miniport driver's completion routine for the IRP. This call notifies the driver that the IRP is completed and the network adapter can transition to a full-power state. From the

context of the completion routine, the driver calls **NdisMIdleNotificationComplete** to notify NDIS that the network adapter can be transitioned to a full-power state.

For more information about how to implement a USB idle request IRP completion routine, see Implementing a USB Idle Request IRP Completion Routine.

Note Depending on the dependencies for canceling bus-specific idle requests, the miniport driver calls **NdisMIdleNotificationComplete** either synchronously in the context of the call to *MiniportCancelIdleNotification* or asynchronously after *MiniportCancelIdleNotification* returns.

After the miniport driver cancels any bus-specific IRPs for the idle notification, it calls **NdisMIdleNotificationComplete**. This call notifies NDIS that the idle notification has been completed. NDIS then completes the selective suspend operation by transitioning the network adapter to a full-power state.

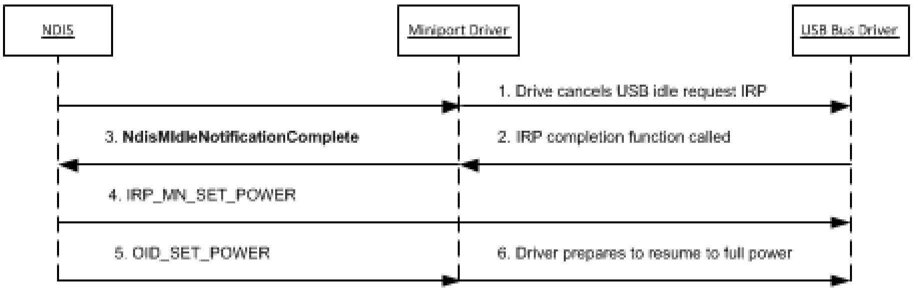
When **NdisMIdleNotificationComplete** is called, NDIS performs the following steps:

1. NDIS issues **IRP_MN_SET_POWER** to the underlying bus driver. This IRP requests the bus driver to set the power state of the network adapter to PowerDeviceD0.

2. NDIS issues an object identifier (OID) set request of OID_PNP_SET_POWER to the miniport driver. In this OID request, NDIS specifies that the network adapter is now transitioning to a full-power state of NdisDeviceStateD0.

   When it handles this OID set request, the driver prepares the adapter for full-power operation. This includes restoring the receive and send engines to the same state they were in before the transition to the low-power state. The driver then completes the OID request with NDIS_STATUS_SUCCESS.

The following figure shows the steps that are involved when the miniport driver completes an idle notification for a USB network adapter.


NDIS
Miniport Driver
USB Bus Driver
1. Drive cancels USB idle request IRP
3. NdisMIdleNotificationComplete
2. IRP completion function called
4. IRP_MN_SET_POWER
5. OID_SET_POWER
6. Driver prepares to resume to full power

Note When the miniport driver completes an idle notification, it must not call **NdisMIdleNotificationConfirm** for an idle notification that was previously completed through a call to **NdisMIdleNotificationComplete**.

Standardized INF Keywords for NDIS Selective Suspend

Article • 03/14/2023

> ⓘ **Note**
>
> Selective Suspend related keywords are for traditional NDIS miniport driver use only. They are deprecated in **Network Adapter WDF Class Extension (NetAdapterCx)** and must not be used by its client drivers.

The following standardized INF keywords are defined to enable, disable, and configure parameters for NDIS selective suspend on a miniport driver:

***SelectiveSuspend** INF Keyword

***SSIdleTimeout** INF Keyword

***SSIdleTimeoutScreenOff** INF Keyword

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

*SelectiveSuspend INF Keyword

The INF file for the miniport driver that supports NDIS selective suspend must specify the ***SelectiveSuspend** standardized INF keyword. After the driver is installed, administrators can update the ***SelectiveSuspend** keyword value in the **Advanced** property page for the network adapter. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

Note The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter.

The ***SelectiveSuspend** INF keyword is an enumeration keyword. The following table describes the possible INF entries for the ***SelectiveSuspend** INF keyword. The columns in this table describe the following attributes for an enumeration keyword:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params\** key for the network adapter.

ParamDesc
The display text that is associated with SubkeyName.

Note The independent hardware vendor (IHV) can define any descriptive text for the SubkeyName.

Value
The enumeration integer value that is associated with each SubkeyName in the list.

EnumDesc
The display text that is associated with each value that appears in the **Advanced** property page.

SubkeyName	ParamDesc	Value	EnumDesc
***SelectiveSuspend**	Selective suspend	0	Disabled
		1 (Default)	Enabled

The miniport driver must check the ***SelectiveSuspend** keyword value in the registry before it advertises its support for NDIS selective suspend. If the ***SelectiveSuspend** keyword has a value of zero, the miniport must not advertise support for any selective suspend capabilities. For more information, see Reporting NDIS Selective Suspend Capabilities.

*SSIdleTimeout INF Keyword

The INF file for the miniport driver that supports NDIS selective suspend should specify the optional ***SSIdleTimeout** standardized INF keyword. This keyword specifies the idle time-out period in units of seconds. If NDIS does not detect any activity on the network adapter for a period that exceeds the ***SSIdleTimeout** value, NDIS starts a selective suspend operation by calling the miniport driver's *MiniportIdleNotification* handler function.

After the driver is installed, administrators can update the ***SSIdleTimeout** keyword value in the **Advanced** property page for the network adapter. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

Note The miniport driver is automatically restarted after a change is made in the advanced property page for the adapter.

The ***SSIdleTimeout** INF keyword is a numeric (**Int**) keyword. The following table describes the possible INF entries for the ***SSIdleTimeout** INF keyword. The columns in the table describe the following attributes for an **Int** keyword:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params\** key for the network adapter.

ParamDesc
The display text that is associated with SubkeyName.

Note The independent hardware vendor (IHV) can define any descriptive text for the SubkeyName.

Default value
The default value for the integer.

Minimum value
The minimum value that is allowed for an integer.

Maximum value
The maximum value that is allowed for an integer.

SubkeyName	ParamDesc	Default value	Minimum value	Maximum value
***SSIdleTimeout**	Selective suspend idle time-out in units of seconds	5	1	60

Note NDIS reads the value of the ***SSIdleTimeout** standardized INF keyword for every instance of the network adapter whose driver supports NDIS selective suspend. Miniport drivers should not read this keyword.

NDIS measures the idle time-out by using timers that are precise to within 30 percent of the ***SSIdleTimeout** value. For example, if the ***SSIdleTimeout** value is 10, the adapter is suspended between 10 to 13 seconds after NDIS first detects the adapter is idle.

*SSIdleTimeoutScreenOff INF Keyword

The INF file for the miniport driver that supports NDIS selective suspend should specify the optional ***SSIdleTimeoutScreenOff** standardized INF keyword. This keyword specifies the idle time-out period in units of seconds and is only applicable when the screen is off. If NDIS does not detect any activity on the network adapter for a period that exceeds the ***SSIdleTimeoutScreenOff** value after the screen is off, NDIS starts a selective suspend operation by calling the miniport driver's *MiniportIdleNotification* handler function.

After the driver is installed, administrators can update the ***SSIdleTimeoutScreenOff** keyword value in the **Advanced** property page for the network adapter. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

Note The miniport driver is automatically restarted after a change is made in the advanced property page for the adapter.

The ***SSIdleTimeoutScreenOff** INF keyword is a numeric (**Int**) keyword. The following table describes the possible INF entries for the ***SSIdleTimeoutScreenOff** INF keyword. The columns in the table describe the following attributes for an **Int** keyword:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params\** key for the network adapter.

ParamDesc
The display text that is associated with SubkeyName.

Note The independent hardware vendor (IHV) can define any descriptive text for the SubkeyName.

Default value
The default value for the integer.

Minimum value
The minimum value that is allowed for an integer.

Maximum value
The maximum value that is allowed for an integer.

SubkeyName	ParamDesc	Default value	Minimum value	Maximum value
*SSIdleTimeoutScreenOff	Selective suspend idle time-out in units of seconds	3	1	60

Note NDIS reads the value of the ***SSIdleTimeoutScreenOff** standardized INF keyword for every instance of the network adapter whose driver supports NDIS selective suspend.

Miniport drivers should not read this keyword.

Note The maximum value is only for testing purposes. The HLK certification test will explicitly check and fail if the value is more than 5.

NDIS measures the idle time-out by using timers that are precise to within 30 percent of the ***SSIdleTimeoutScreenOff** value. For example, if the ***SSIdleTimeoutScreenOff** value is 5, the adapter is suspended between 5 to 6.5 seconds after NDIS first detects the adapter is idle.

Managing IRP Resources for NDIS Selective Suspend

Article • 03/14/2023

If a miniport driver supports and enables NDIS selective suspend, NDIS calls *MiniportIdleNotification* to issue an idle notification to the driver if the network adapter becomes inactive. When the miniport driver handles this notification, it may need to issue I/O request packets (IRPs) to the underlying bus driver. These IRPs notify the bus driver about the adapter's idle state and request confirmation that the adapter can transition to a low-power state.

IRPs that are issued by the miniport driver are bus-specific. For example, when NDIS calls *MiniportIdleNotification*, the USB miniport issues an USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) IRP to the underlying USB bus driver.

NDIS may issue the idle notification to the miniport driver many times after the driver has been initialized. Therefore, we recommend that the driver allocate the resources for the USB idle request IRP in the context of the call to the driver's *MiniportInitializeEx* function.

The following example shows how the miniport driver allocates the IRP resources.

C++

```
//
// MiniportInitializeEx()
//
// In the miniport's initialization routine, the miniport should allocate
// an IRP.  It can also set up the USB_IDLE_CALLBACK_INFO structure that
// will be used with each successive USB idle request.
//
NDIS_STATUS MiniportInitializeEx(
    _In_ NDIS_HANDLE MiniportAdapterHandle,
    _In_ NDIS_HANDLE MiniportDriverContext,
    _In_ PNDIS_MINIPORT_INIT_PARAMETERS MiniportInitParameters
    )
    {
    PIRP UsbSsIrp;
    USB_IDLE_CALLBACK_INFO UsbSsCallback;
    ...

    UsbSsIrp = IoAllocateIrp(Adapter->Fdo->StackSize, FALSE);
    if (!UsbSsIrp)
       {
       // Handle failure
       return NDIS_STATUS_RESOURCES;
       }

    UsbSsCallback.IdleCallback = MiniportUsbIdleRequestCallback;
    UsbSsCallback.IdleContext = Adapter;

    // Save these in the adapter structure for later use
    Adapter->UsbSsIrp = UsbSsIrp;
    Adapter->UsbSsCallback = UsbSsCallback;
    ...
    }
```

If the miniport driver allocates the IRP resources during the call to *MiniportInitializeEx*, the driver must free those resources during the call to *MiniportHaltEx*.

The following example shows how the miniport driver frees the IRP resources.

```C++
//
// MiniportHaltEx
//
// During halt (or when the miniport performs its cleanup from
// MiniportInitializeEx) the miniport should free the IRP allocated
// earlier.
//
VOID MiniportHaltEx(
     _In_  NDIS_HANDLE MiniportAdapterContext,
     _In_  NDIS_HALT_ACTION HaltAction
    )
    {
    ...
    if (Adapter->UsbSsIrp)
        {
        IoFreeIrp(Adapter->UsbSsIrp);
        Adapter->UsbSsIrp = NULL;
        }
    ...
    }
```

Implementing a MiniportIdleNotification Handler Function

Article • 12/15/2021

NDIS calls the miniport driver's *MiniportIdleNotification* handler function in order to selectively suspend the network adapter. The adapter is suspended when NDIS transitions the adapter to a low-power state.

The miniport driver can veto the idle notification if the network adapter is still being used. The driver does this by returning NDIS_STATUS_BUSY from the *MiniportIdleNotification* handler function.

Note The miniport driver cannot veto the idle notification if the *ForceIdle* parameter of the *MiniportIdleNotification* handler function is set to **TRUE**.

If the miniport driver does not veto the idle notification, it may have to issue bus-specific I/O request packets (IRPs) to the underlying bus driver. These IRPs notify the bus driver about the adapter's idle state and request confirmation that the adapter can transition to a low-power state.

For example, when *MiniportIdleNotification* is called, the USB miniport driver prepares an I/O request packet (IRP) for a USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**). When the miniport driver prepares the IRP, it must specify a callback function. The driver must also call either **IoSetCompletionRoutine** or **IoSetCompletionRoutineEx** to specify a completion routine for the IRP. The miniport driver then calls **IoCallDriver** to issue the IRP to the USB bus driver.

Note The USB bus driver does not immediately complete the IRP. The IRP is left in a pending state through the low-power transition. The bus driver completes the IRP only when it is canceled by the miniport driver or a hardware event occurs, such as the surprise removal of the network adapter from the USB hub.

The following is an example of a *MiniportIdleNotification* handler function for a USB miniport driver. This example shows the steps that are involved with issuing a USB idle request IRP to the underlying USB driver. This example also shows how the IRP resources, which were previously allocated in *MiniportInitializeEx*, can be reused for the IRP.

C++

```
//
// MiniportIdleNotification()
//
// This routine is invoked by NDIS when it has detected that the miniport
// is idle.  The miniport must prepare to issue its selective suspend IRP
// to the USB stack.  The driver can return NDIS_STATUS_BUSY if it is
// unwilling to become idle at this moment; NDIS will then retry later.
// Otherwise, the miniport should return NDIS_STATUS_PENDING.
//
NDIS_STATUS MiniportIdleNotification(
    _In_ NDIS_HANDLE MiniportAdapterContext,
    _In_ BOOLEAN ForceIdle
    )
{
    PIO_STACK_LOCATION IoSp;

    IoReuseIrp(Adapter->UsbSsIrp, STATUS_NOT_SUPPORTED);

    IoSp = IoGetNextIrpStackLocation(Adapter->UsbSsIrp);
    IoSp->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
    IoSp->Parameters.DeviceIoControl.IoControlCode
            = IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION;
    IoSp->Parameters.DeviceIoControl.InputBufferLength
            = sizeof(Adapter->UsbSsCallback);
    IoSp->Parameters.DeviceIoControl.Type3InputBuffer
            = Adapter->UsbSsCallback;

    IoSetCompletionRoutine(
            Adapter->UsbSsIrp,
            MiniportUsbIdleRequestCompletion,
            Adapter,
            TRUE,
            TRUE,
            TRUE);

    NtStatus = IoCallDriver(Adapter->Fdo, Adapter->UsbSsIrp);
    if (!NT_SUCCESS(NtStatus))
    {
       return NDIS_STATUS_FAILURE;
    }

    return NDIS_STATUS_PENDING;
}
```

For guidelines on implementing a callback routine for a USB idle request IRP, see Implementing a USB Idle Request IRP Callback Routine.

Implementing a USB Idle Request IRP Callback Routine

Article • 12/15/2021

When *MiniportIdleNotification* is called, the USB miniport driver calls **IoCallDriver** to issue an I/O request packet (IRP) for a USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) to the underlying USB bus driver. The miniport driver issues this IRP to inform the USB bus driver that the network adapter is idle and must be suspended.

The USB miniport driver must provide an IRP callback routine for the USB idle request IRP. The USB bus driver calls this routine when it determines that the network adapter can be suspended and transitioned to a low-power state.

Note After the USB bus driver handles the USB idle request IRP, it calls the callback routine either synchronously in the context of the call to **IoCallDriver** or asynchronously after *MiniportIdleNotification* returns.

The callback routine only has to call **NdisMIdleNotificationConfirm** in order to notify NDIS that it can continue with the low-power state transition of the network adapter. When the driver calls **NdisMIdleNotificationConfirm**, it must also specify the lowest device power state that the network adapter can transition to.

Within the context of the call to **NdisMIdleNotificationConfirm**, NDIS performs the steps that are required to transition the network adapter to a low-power state. For more information, see Handling the NDIS Selective Suspend Idle Notification.

The following is an example of a callback routine for a USB idle request IRP.

C++

```
//
// MiniportUsbIdleRequestCallback()
//
// This is the USB selective suspend idle notification.  All that is
// needed is to inform NDIS that the USB stack is ready to go to a
// low-power state.  Be aware that USB devices will always be requested
// to transition to a power state of NdisDeviceStateD2.
//
VOID MiniportUsbIdleRequestCallback(PVOID AdapterContext)
{
    NdisMIdleNotificationConfirm(
        AdapterContext->MiniportAdapterHandle,
        NdisDeviceStateD2
        );
    return;
}
```

For more information about the USB idle request callback routine, see USB Idle Request IRP Callback Routine.

Implementing a MiniportCancelIdleNotification Handler Function

Article • 12/15/2021

NDIS calls the miniport driver's *MiniportCancelIdleNotification* handler function in order to cancel the idle notification process and transition the network adapter to a full-power state. When this function is called, the miniport driver must follow these steps:

1. The miniport driver must cancel any bus-specific IRPs that it may have previously issued for the idle notification.

2. The miniport driver calls **NdisMIdleNotificationComplete**. This call notifies NDIS that the idle notification has been completed. NDIS then comples the selective suspend operation by transitioning the network adapter to a full-power state.

For example, when *MiniportCancelIdleNotification* is called, the USB miniport driver calls **IoCancelIrp** to cancel the I/O request packet (IRP) for a USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**). The USB miniport driver previously issued this IRP in its *MiniportIdleNotification* handler function. As soon as the USB bus driver has canceled the IRP, it calls the IRP's completion routine. When the USB bus driver calls the completion routine, it confirms that the IRP is canceled and the device can resume to a full-power state. In the context of the completion routine, the miniport driver calls **NdisMIdleNotificationComplete**.

Note The USB bus driver can call the completion routine either synchronously in the context of the call to **IoCancelIrp** or asynchronously after *MiniportCancelIdleNotification* returns.

The following is an example of a *MiniportCancelIdleNotification* handler function for a USB miniport driver. This example shows the steps that are involved with canceling a USB idle request IRP.

```C++
//
// MiniportCancelIdleNotification()
//
// This routine is called if NDIS has to cancel an idle notification.
// All that is needed is to cancel the selective suspend IRP.
//
VOID MiniportCancelIdleNotification(
    _In_ NDIS_HANDLE MiniportAdapterContext
    )
{
    IoCancelIrp(Adapter->UsbSsIrp);
}
```

For guidelines on implementing a completion routine for a USB idle request IRP, see Implementing a USB Idle Request IRP Completion Routine.

Implementing a USB Idle Request IRP Completion Routine

Article • 12/15/2021

When *MiniportIdleNotification* is called, the USB miniport driver calls **IoCallDriver** to issue an I/O request packet (IRP) for a USB idle request (**IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION**) to the underlying USB bus driver. The miniport driver issues this IRP to inform the USB bus driver that the network adapter is idle and must be suspended.

The USB miniport driver must also call **IoSetCompletionRoutineEx** in order to register a completion routine for the USB idle request IRP. The USB bus driver calls the completion routine when it completes the IRP after it is canceled by the USB miniport driver. The USB miniport driver cancels the IRP when NDIS cancels the idle notification by calling *MiniportCancelIdleNotification*.

The completion routine only has to call **NdisMIdleNotificationComplete** in order to notify NDIS that it can continue with the full-power state transition of the network adapter.

Note The completion routine must return STATUS_MORE_PROCESSING_REQUIRED if the USB miniport driver will reuse the IRP resources during another idle notification from NDIS.

The following is an example of a completion routine for the USB idle request IRP.

C++

```cpp
//
// MiniportUsbIdleRequestCompletion()
//
// This is the IO_COMPLETION_ROUTINE for the selective suspend IOCTL.
// All that is needed is to inform NDIS that the IdleNotification
// operation is complete.
//
VOID MiniportUsbIdleRequestCompletion(PVOID AdapterContext)
{
    NdisMIdleNotificationComplete(Adapter->MiniportAdapterHandle);

    // We will be reusing the IRP later, so do not let the IO manager delete
it.
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

For more information about the USB idle request callback routine, see USB Idle Request IRP Completion Routine.

Overview of NDIS Wake Reason Status Indications

Article • 03/14/2023

Starting with NDIS 6.30, miniport drivers issue an NDIS wake reason status indication (**NDIS_STATUS_PM_WAKE_REASON**) to notify NDIS and overlying drivers about the reason for a system wake-up event. If the network adapter generates a wake-up event, the miniport driver immediately issues an NDIS status indication of **NDIS_STATUS_PM_WAKE_REASON** when the network adapter resumes to a full-power state.

Note Support for NDIS wake reason status indications is optional for Mobile Broadband (MB) miniport drivers.

The miniport driver is configured with power management (PM) parameters through an object identifier (OID) set request of OID_PM_PARAMETERS. This OID request specifies the PM parameters through an **NDIS_PM_PARAMETERS** structure.

The **NDIS_PM_PARAMETERS** structure specifies the parameters for the following types of wake-up events.

Received Packet Wake-up Events
The network adapter generates a wake-up event if it receives a packet that matched a wake-on-LAN (WOL) pattern. WOL patterns include the following:

- Media-independent WOL patterns, such as magic packets or TCP/IP data patterns within the packet payload. For example, the **NDIS_PM_PARAMETERS** structure could specify a WOL pattern for a TCP SYN frame.

- Media-specific WOL patterns, such as an EAPOL request identifier packet or mobile broadband (MB) Short Message Service (SMS) message.

- Wildcard patterns that match a receive filter specified through an OID set request of OID_GEN_CURRENT_PACKET_FILTER.

Note For this type of wake reason status indication, the network adapter must be able to save the received packet. The driver must return the received packet within the status indication.

WOL patterns are specified through the **EnabledWoLPacketPatterns** member of the **NDIS_PM_PARAMETERS** structure.

Media-Specific Wake-up Events
The network adapter generates a wake-up event because of a media-specific reason, such as a disassociation from an 802.11 access point (AP) or the receipt of a mobile broadband (MB) Short Message Service (SMS) message.

Wake-up events of this type are specified through the **MediaSpecificWakeUpEvents** member of the **NDIS_PM_PARAMETERS** structure.

Media-Independent Wake-up Events
The network adapter generates a wake-up event because of a media-independent reason, such as media connection or disconnection.

Wake-up events of this type are specified through the **WakeUpFlags** member of the **NDIS_PM_PARAMETERS** structure.

The miniport driver must follow these guidelines for NDIS wake reason status indications:

- If the miniport driver supports the ability to issue wake packet indications, it must report this ability when NDIS calls the driver's *MiniportInitializeEx* function. For more information, see Reporting Wake Reason Status Indication Capabilities.

  Note The miniport driver does not have to report its ability to issue NDIS wake reason status indications for events that are not related to the receipt of a WOL packet.

- When the miniport driver issues a wake packet indication for a WOL packet, it must include the packet that caused the wake-up event. For more information, see Issuing NDIS Wake Reason Status Indications.

- If the network adapter generated a wake-up signal, the miniport driver must issue an **NDIS_STATUS_PM_WAKE_REASON** status indication. The driver does this while it is handling the OID set request of OID_PNP_SET_POWER for the transition to a full-power state.

- The miniport driver must issue an **NDIS_STATUS_PM_WAKE_REASON** status indication before it issues a status indication that is related to the wake-up event. For example, if the wake-up event was due to a change in the media connectivity state, the miniport driver must issue an **NDIS_STATUS_LINK_STATE** status indication after it has issued the **NDIS_STATUS_PM_WAKE_REASON** status indication.

- The miniport driver must ssue an **NDIS_STATUS_PM_WAKE_REASON** status indication only for power management events that were previously enabled through an OID set request of OID_PM_PARAMETERS.

- The miniport driver must issue an **NDIS_STATUS_PM_WAKE_REASON** status indication only for wake-up events that were generated by the underlying network adapter.

Reporting Wake Reason Status Indication Capabilities

Article • 12/15/2021

Starting with NDIS 6.30, the miniport driver must report whether it can issue an NDIS wake reason status indication (**NDIS_STATUS_PM_WAKE_REASON**) to report wake-up events caused by one of the following:

- The network adapter received a packet that matched a wake-on-LAN (WOL) pattern. This includes the receipt of a packet that matches a receive filter specified through an object identifier (OID) set request of OID_GEN_CURRENT_PACKET_FILTER.

  Note For this type of wake reason status indication, the network adapter must be able to save the received packet. The driver must return the received packet within the status indication.

- The network adapter detected a media-specific event, such as a disassociation from an 802.11 access point (AP) or the receipt of a mobile broadband (MB) Short Message Service (SMS) message.

- The network adapter detected another enabled event that is not specific to a WOL pattern or media type (*media-independent event*). For example, the miniport driver issues the **NDIS_STATUS_PM_WAKE_REASON** status indication if it enabled the network adapter to detect media connection or disconnection.

Note Support for NDIS wake reason status indications is optional for Mobile Broadband (MB) miniport drivers.

When NDIS calls the driver's *MiniportInitializeEx* function, the miniport driver reports its wake reason status indication capabilities by following these steps:

1. The miniport driver initializes an **NDIS_PM_CAPABILITIES** structure with the power management capabilities of the underlying hardware.

   To enable the support for wake reason status indications, the miniport driver must set the members of the **NDIS_PM_CAPABILITIES** structure as follows:

   - The miniport driver must specify NDIS_PM_CAPABILITIES_REVISION_2 and NDIS_SIZEOF_NDIS_PM_CAPABILITIES_REVISION_2 for the revision and length of the **NDIS_PM_CAPABILITIES** structure within the structure's **Header** member.

- If the network adapter can store the received packet that caused a system wake-up event, the miniport driver sets the NDIS_PM_WAKE_PACKET_INDICATION_SUPPORTED flag within the **Flags** member of this structure.

  If this flag is set, the network adapter must be able to save the received packet that caused the adapter to generate a wake-up event. In addition, the miniport driver must be able to do the following with this packet after the network adapter transitions to a full-power state:

  - The miniport driver must be able to indicate the packet by calling **NdisMIndicateReceiveNetBufferLists**.

  - The miniport driver must be able to issue an **NDIS_STATUS_PM_WAKE_REASON** status indication and must pass the packet with indication.

- The miniport driver sets the **MaxWoLPacketSaveBuffer** member to the maximum size, in units of bytes, of the buffer that contains the WOL packet that caused a system wake-up event.

  The value of the **MaxWoLPacketSaveBuffer** member must be less than or equal to the size, in bytes, of the maximum transmission unit (MTU) and media access control (MAC) header for the network media. The driver reports the MTU size through OID query requests of OID_GEN_MAXIMUM_FRAME_SIZE.

- The miniport driver sets the **SupportedWakeUpEvents** to the media-independent wake-up events that the network adapter supports, such as generating a wake-up event when the adapter becomes connected to the networking interface.

- The miniport driver sets the **MediaSpecificWakeUpEvents** to the media-specific wake-up events that the network adapter supports. These events include generating a wake-up event when the 802.11 adapter becomes disassociated with the AP.

2. The miniport driver initializes an NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure and sets the**PowerManagementCapabilitiesEx** member to the address of the initialized **NDIS_PM_CAPABILITIES** structure.

3. The miniport driver calls the **NdisMSetMiniportAttributes** function to register its power management capabilities. When the miniport driver calls this function, it

sets the *MiniportAttributes* parameter to the address of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure.

The method that is used by miniport drivers to report the wake reason status indication capabilities is based on the NDIS 6.20 method for reporting power management capabilities. For more information about this method, see Reporting Power Management Capabilities.

For more information about the adapter initialization process, see Initializing a Miniport Adapter.

For more information about how to report power management capabilities, see Reporting Power Management Capabilities.

Issuing NDIS Wake Reason Status Indications

Article • 03/14/2023

If a miniport driver supports NDIS wake reason status indications (**NDIS_STATUS_PM_WAKE_REASON**), it must generate this status indication immediately after the network adapter generates a wake-up event and the adapter resumes to a full-power state.

Note Support for NDIS wake reason status indications is optional for Mobile Broadband (MB) miniport drivers.

The miniport driver is configured with power management (PM) parameters through an object identifier (OID) set request of OID_PM_PARAMETERS. This OID request specifies the PM parameters through an **NDIS_PM_PARAMETERS** structure.

The **NDIS_PM_PARAMETERS** structure specifies the parameters for the following types of wake-up events.

Received Packet Wake-up Events
The network adapter generates a wake-up event if it receives a packet that matched a wake-on-LAN (WOL) pattern. WOL patterns include the following:

- Media-independent WOL patterns, such as magic packets or TCP/IP data patterns within the packet payload. For example, the **NDIS_PM_PARAMETERS** structure could specify a WOL pattern for a TCP SYN frame.

- Media-specific WOL patterns, such as an EAPOL request identifier packet or mobile broadband (MB) Short Message Service (SMS) message.

- Wildcard patterns that match a receive filter specified through an OID set request of OID_GEN_CURRENT_PACKET_FILTER.

Note For this type of wake reason status indication, the network adapter must be able to save the received packet. The driver must return the received packet within the status indication.

WOL patterns are specified through the **EnabledWoLPacketPatterns** member of the **NDIS_PM_PARAMETERS** structure.

Media-Specific Wake-up Events
The network adapter generates a wake-up event because of a media-specific reason,

such as a disassociation from an 802.11 access point (AP) or the receipt of a mobile broadband (MB) Short Message Service (SMS) message.

Wake-up events of this type are specified through the **MediaSpecificWakeUpEvents** member of the NDIS_PM_PARAMETERS structure.

Media-Independent Wake-up Events
The network adapter generates a wake-up event because of a media-independent reason, such as media connection or disconnection.

Wake-up events of this type are specified through the **WakeUpFlags** member of the NDIS_PM_PARAMETERS structure.

If the network adapter generated a wake-up signal, the miniport driver must issue an NDIS_STATUS_PM_WAKE_REASON status indication. The driver does this while it is handling the OID set request of OID_PNP_SET_POWER for the transition of the adapter to a full-power state.

Note The miniport driver must issue an NDIS_STATUS_PM_WAKE_REASON status indication before it issues a status indication that is related to the wake-up event. For example, if the wake-up event was due to a change in the media connectivity state, the miniport driver must issue an NDIS_STATUS_LINK_STATE status indication after it has issued the **NDIS_STATUS_PM_WAKE_REASON** status indication.

When the miniport driver issues the NDIS_STATUS_PM_WAKE_REASON status indication, it must follow these steps:

1. The miniport driver must allocate a buffer that is large enough to contain the following:

    - An NDIS_PM_WAKE_REASON structure.

    - An NDIS_PM_WAKE_PACKET structure along with the received packet (*wake packet*) that caused the network adapter to generate the wake-up event.

      Note The miniport driver does not need to allocate this buffer space if it indicates media-specific or media-independent wake-up events.

2. The miniport driver initializes an NDIS_PM_WAKE_REASON structure at the start of the buffer. The driver sets the **WakeReason** member to an NDIS_PM_WAKE_REASON_TYPE enumeration value that defines the type of the wake-up event.

   For example, if the miniport driver is indicating a received packet wake-up event, it must set the **WakeReason** member to **NdisWakeReasonPacket**. Otherwise, the driver sets the **WakeReason** member to the enumeration value that best describes the media-specific or media-independent wake-up event.

3. If the miniportdriver is issuing an NDIS_STATUS_PM_WAKE_REASON status indication for a received packet wake-up event, it must follow these steps:

   a. The miniport driver sets the **InfoBufferOffset** member to the offset of an NDIS_PM_WAKE_PACKET structure that follows the NDIS_PM_WAKE_REASON structure in the buffer.

   Note The miniport driver must align the start of the NDIS_PM_WAKE_PACKET structure on a 64-bit boundary.

   b. The miniport driver sets the **InfoBufferSize** member to the size of the NDIS_PM_WAKE_PACKET structure plus the size of the packet that caused the wake-up event.

   c. The miniport driver initializes an NDIS_PM_WAKE_PACKET structure following the NDIS_PM_WAKE_REASON structure in the buffer.

   The miniport driver sets the members of the NDIS_PM_WAKE_PACKET structure as follows:

   - The **PatternId** member is set to the identifier of the WOL pattern that matches the wake packet. This identifier is specified by the **PatternId** member of the NDIS_PM_WOL_PATTERN structure that is passed to the driver during an OID set request of OID_PM_ADD_WOL_PATTERN.

   - The **PatternFriendlyName** member is set to the user-readable description of the wake pattern that is specified by the **PatternId** member. This value is specified by the **FriendlyName** member of the NDIS_PM_WOL_PATTERN structure.

     Note The miniport driver does not need to initialize this member. NDIS sets the **PatternFriendlyName** member to the correct value before it passes the NDIS_PM_WAKE_PACKET structure to overlying drivers.

   - The **OriginalPacketSize** member is set to the length of the packet as received by the network adapter.

   - The **SavedPacketSize** member must be set to the length of the packet that is being reported through the NDIS_STATUS_PM_WAKE_REASON status indication.

Note The value of this member must not be greater than the value that the miniport driver set in the **MaxWoLPacketSaveBuffer** member of the **NDIS_PM_CAPABILITIES** structure. The driver returns this structure when it reports its wake packet indication capabilities. For more information, see Reporting Wake Reason Status Indication Capabilities.

- The **SavedPacketOffset** member must be set to the offset, in units of bytes, to the wake packet that follows the **NDIS_PM_WAKE_PACKET** structure.

  Note The miniport driver must align the start of the wake packet on a 64-bit boundary in the buffer.

d. The miniport copies the wake packet into the buffer at the offset specified by the **SavedPacketOffset** member.

4. If the miniport driver is issuing an **NDIS_STATUS_PM_WAKE_REASON** status indication for a media-specific or media-independent wake-up event, it sets the **InfoBufferOffset** and **InfoBufferSize** members of the **NDIS_PM_WAKE_REASON** structure to zero.

5. The miniport driver initializes an **NDIS_STATUS_INDICATION** structure. The driver sets the **StatusCode** member to NDIS_STATUS_PM_WAKE_REASON. The driver also sets the **StatusBuffer** member to point to the buffer, and sets the **StatusBufferLength** to the length, in bytes, of the buffer.

6. The miniport driver calls **NdisMIndicateStatusEx** and passes a pointer to the **NDIS_STATUS_INDICATION** structure in the *StatusIndication* parameter.

Note After the miniport driver issues the **NDIS_STATUS_PM_WAKE_REASON** status indication for a received packet wake-up event, it must indicate this received packet by calling **NdisMIndicateReceiveNetBufferLists**.

NDIS Power Management Overview

Article • 03/14/2023

This section provides an overview of the features that are provided with the power management interface that is introduced in Windows 7 for NDIS 6.20 drivers.

Miniport drivers and protocol drivers that support NDIS 6.20 and later versions of NDIS must support the NDIS 6.20 power management interface. However, NDIS provides translation to the previous interface for older network adapters and NDIS 6.1 or earlier miniport drivers that do not support the NDIS 6.20 power management features. For more information about NDIS 6.20 backward compatibility issues, see NDIS 6.20 Backward Compatibility.

The NDIS 6.20 power management interface supports:

- Wake-on-LAN (WOL) patterns that are based on the packet type in addition to the NDIS 6.1 and earlier methods. Therefore, NDIS 6.20 WOL patterns can be more specific to avoid unnecessary wake-up events. For example, a network adapter can identify TCP synchronize (SYN) packets. For more information about WOL methods, see WOL Methods in NDIS 6.20.

- Protocol offloads to network adapters for some of the most common protocols. Because the protocols are offloaded to the network adapter, it can respond on behalf of the computer to avoid unwanted wake-up events. For example, a network adapter can handle IPv4 Address Resolution Protocol (ARP) and IPv6 Neighbor Solicitation (NS) protocol packets without waking the computer. For more information about power management protocol offloads, see Protocol Offloads for NDIS Power Management.

For information about the WOL event sequences that NDIS uses to set a low-power state and restore full power, see Low Power for Wake on LAN.

The NDIS 6.20 power management also supports:

- NDIS 6.20 can return the network adapter to a full-power state when the media connects. The operating system puts the network adapter in a low-power state when the media is disconnected. For more information about setting a low-power state when media disconnects, see Low Power on Media Disconnect.

This section includes the following topics:

WOL Methods in NDIS 6.20

WOL Patterns for NDIS Power Management

Protocol Offloads for NDIS Power Management

WOL Methods in NDIS 6.20

Article • 03/14/2023

The power management capabilities that are supported in NDIS 6.20 and later versions of NDIS consist of the following wake-on-LAN (WOL) methods:

- Wake on magic packet
- Wake on pattern match
- Wake device on media connect

For more information about the power management capabilities in previous versions of Windows, see Power Management (NDIS 6.0 and Later).

The *wake on magic packet* method wakes the computer when the network adapter receives a *magic packet*. A *magic packet* contains 16 contiguous copies of the receiving network adapter's Ethernet address.

The *wake on magic packet* method is separate from the *wake on pattern match* method. WOL patterns include other packet types or a bitmap. For more information about WOL patterns, see WOL Patterns for NDIS Power Management.

Although some network adapters report support for the *wake device on media connect* method, previous versions of Windows did not. Windows 7 fully supports the *wake device on media connect* method if an NDIS 6.20 miniport driver reports support. NDIS sets the network adapter to a low power state if the media is disconnected.

For more information about the *wake device on media connect* method, see Low Power on Media Disconnect.

WOL Patterns for NDIS Power Management

Article • 03/14/2023

Starting with NDIS 6.20, Wake-on-LAN (WOL) patterns are supported for the *wake on pattern match* method. This WOL method minimizes spurious wake-up events and ensures that the computer is brought back to running state when expected. The interface for WOL patterns identifies specific patterns that are based on the packet type (for example, TCP SYN packets on IPv4). Specific patterns provide reliable pattern matches.

There are two types of WOL patterns:

WOL packet
A packet in which the wake-up pattern defines a specific packet type (such as TCP SYN on IPv4).

WOL bitmap
A WOL pattern that is specified with an offset and bitmap.

Note NDIS 6.20 and later versions of NDIS also support the *wake on magic packet* method. This method is separate from the *wake on pattern match* method.

Starting with NDIS 6.20, multiple protocol drivers can set WOL patterns on a network adapter. To ensure that the correct set of WOL patterns is set when the number of requested WOL patterns is higher than the number that the network adapter can support, protocol drivers assign a priority to each WOL pattern. When NDIS cannot add a new high-priority WOL pattern because the network adapter is out of resources, NDIS can delete the lower priority patterns.

For more information about managing WOL patterns, see Adding and Deleting Wake on LAN Patterns.

For more information about WOL methods supported in NDIS 6.20 and later versions, see WOL Methods in NDIS 6.20.

Protocol Offloads for NDIS Power Management

Article • 03/14/2023

NDIS 6.20 and later versions of NDIS support protocol offloads for NDIS power management. For example, NDIS can offload the handling of Address Resolution Protocol (ARP) requests to a network adapter. Some applications use periodic ARP request packets to discover and ensure the presence of a host on the network. These applications send the ARP requests even when there is no current need to send data to the host. Such ARP requests wake up the host and waste power when there is nothing for the host to do.

Note In Windows 7, the power management offload functionality is enabled only when all protocol and filter drivers that are bound to the miniport adapter support NDIS 6.20 and later versions. In Windows 8, the power management offload functionality is enabled if the miniport adapter supports it, regardless of the protocol and filter driver versions.

Note If an incoming packet matches both an offloaded protocol and a pattern (for example, because of a configuration error), the network adapter responds to the packet and wakes up the computer.

To minimize spurious wake ups, NDIS protocol drivers attempt to offload the response to commonly used network requests to the hardware. Some network protocols require the host to periodically advertise certain information. When a network adapter responds to ARP requests, or takes over protocol specific periodic advertisements without waking up the system for processing these requests, many spurious wake-up events can be avoided.

There are three types of low power protocol offloads:

- IPv4 ARP
- IPv6 Neighbor Solicitation (NS)
- IEEE 802.11 robust secure network (RSN) 4-way and 2-way handshake

NDIS allows multiple protocol drivers to offload different protocols to a network adapter. To ensure that the correct set of protocols is offloaded when the number of requested protocol offloads is higher than the number that the network adapter can support, protocol drivers assign a priority to each protocol offload. When NDIS cannot add a new high-priority protocol offload because the network adapter is out of resources, NDIS might delete the lower priority offloads.

For more information about managing protocol offloads, see Adding and Deleting Low Power Protocol Offloads.

Standardized INF Keywords for Power Management

Article • 12/15/2021

The power management standardized keywords are defined in the device driver INF file. The operating system reads these standardized keywords and adjusts the current power management capabilities of the device.

Both Network Adapter WDF Class Extension (NetAdapterCx) client drivers and traditional NDIS miniport device drivers use these power management keywords. However, some keywords are used exclusively by NetAdapterCx drivers while others are used exclusively by NDIS drivers as the following sections describe:

- Power management keywords for NetAdapterCx and NDIS
- Power management keywords exclusive to NetAdapterCx
- Power management keywords exclusive to NDIS

The traditional NDIS miniport device driver should always indicate the device's hardware power management capabilities to NDIS in the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure.

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

Power management keywords for NetAdapterCx and NDIS

The following standardized INF keywords are defined to enable or disable support for power management features of network adapters. They are used by both NetAdapterCx client drivers and traditional NDIS miniport device drivers.

***WakeOnPattern**
A value that describes whether the device should be enabled to wake the computer when a network packet matches a specified pattern.

***WakeOnMagicPacket**
A value that describes whether the device should be enabled to wake the computer when the device receives a *magic packet*. (A *magic packet* is a packet that contains 16 contiguous copies of the receiving network adapter's Ethernet address)

***PMARPOffload**
A value that describes whether the device should be enabled to offload the Address Resolution Protocol (ARP) when the system enters a sleep state.

***PMNSOffload**
A value that describes whether the device should be enabled to offload neighbor solicitation (NS) when the system enters a sleep state.

***PMWiFiRekeyOffload**
A value that describes whether the device should be enabled to offload group temporal key (GTK) rekeying for wake-on-wireless-LAN (WOL) when the computer enters a sleep state.

***EEE**
A value that describes whether the device should enable IEEE 802.3az Energy-Efficient Ethernet.

The columns in the table at the end of this topic describe the following attributes for enumeration keywords:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with SubkeyName.

Value
The enumeration integer value that is associated with each option in the list. This value is stored in **NDI\params\**_SubkeyName\Value_.

EnumDesc
The display text that is associated with each value that appears in the menu.

The following table describes the possible INF entries for the power management keywords used by NDIS and NetAdapterCx drivers.

SubkeyName	ParamDesc	Value	EnumDesc
***WakeOnPattern**	Wake on pattern match	0	Disabled
		1 (Default)	Enabled
***WakeOnMagicPacket**	Wake on magic packet	0	Disabled
		1 (Default)	Enabled

SubkeyName	ParamDesc	Value	EnumDesc
***PMARPOffload**	ARP offload	0	Disabled
		1 (Default)	Enabled
***PMNSOffload**	NS offload	0	Disabled
		1 (Default)	Enabled
***PMWiFiRekeyOffload**	WiFi rekeying offload	0	Disabled
		1 (Default)	Enabled
***EEE**	Energy-Efficient Ethernet	0	Disabled
		1 (Default)	Enabled

Power management keywords exclusive to NetAdapterCx

The following power management keywords are for NetAdapterCx client driver use only.

In addition to the standard WDF process for giving user control over the device idle and wake behavior as described in User Control of Device Idle and Wake Behavior, NetAdapterCx also defines a network device specific standardized INF keyword for allowing more control.

***IdleRestriction**
If a network device has both idle power down and wake on packet filter capabilities, this setting allows the user to decide when the device idle power down can happen.

***IdleRestriction** is an enumeration standardized INF keyword and has the following attributes:

The following table describes the possible INF entries for the ***IdleRestriction** keyword.

SubkeyName	ParamDesc	Value	EnumDesc
***IdleRestriction**	idle power down restriction	0 (Default)	No Restriction
		1	Only idle when user is not present

Power management keywords exclusive to NDIS

The following power management keywords are for traditional NDIS miniport driver use only. They must not be used by NetAdapterCx client drivers.

***ModernStandbyWoLMagicPacket**
A value that describes whether the device should be enabled to wake the computer when the device receives a *magic paket* and the system is in the *S0ix* power state. This does not apply when the system is in the *S4* power state.

> ⓘ Note
>
> ***ModernStandbyWoLMagicPacket** is supported in NDIS 6.60 and later, or Windows 10, version 1607 and later.

***DeviceSleepOnDisconnect**
A value that describes whether the device should be enabled to put the device into a low-power state (sleep state) when media is disconnected and return to a full-power state (wake state) when media is connected again.

The following table describes the possible INF entries for the power management keywords used by NDIS miniport drivers.

SubkeyName	ParamDesc	Value	EnumDesc
***ModernStandbyWoLMagicPacket**	Wake on magic packet when system is in the *S0ix* power state	0 (Default)	Disabled
		1	Enabled
***DeviceSleepOnDisconnect**	Device sleep on disconnect	0	Disabled
		1 (Default)	Enabled

Reporting Power Management Capabilities

Article • 12/15/2021

Miniport drivers that support NDIS 6.20 and later versions of NDIS report their hardware power management capabilities during initialization. NDIS reports the current capabilities to overlying NDIS protocol drivers during the bind operation. However, NDIS can hide some capabilities from the protocol driver. For example, NDIS might report different capabilities when a user disables some or all of the power management capabilities.

Note that the current power management capabilities that NDIS reports to a protocol driver are not necessarily the same as the hardware capabilities that the miniport driver reported to NDIS.

If an NDIS 6.1 or earlier miniport driver is bound to an NDIS 6.20 protocol driver, NDIS translates the power management capabilities to a format that is supported by the NDIS 6.20 protocol driver. NDIS also translates power management capabilities that an NDIS 6.20 miniport driver reports into a format that is supported by the NDIS 6.1 and earlier overlying drivers.

The hardware capabilities that a miniport driver reports can be enabled or disabled in INF file settings. For more information about power management INF file settings, see Standardized INF Keywords for Power Management.

During miniport initialization, a miniport driver initializes an **NDIS_PM_CAPABILITIES** structure with the power management capabilities of the underlying hardware. The miniport driver sets the **PowerManagementCapabilitiesEx** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure to point to the **NDIS_PM_CAPABILITIES** structure.

The **NDIS_PM_CAPABILITIES** structure includes the following information:

Flags
For NDIS 6.20, this member is reserved for NDIS.

Starting with NDIS 6.30, the following flags are defined:

NDIS_PM_WAKE_PACKET_INDICATION_SUPPORTED
If this flag is set, the network adapter can save the received packet that caused the adapter to generate a wake-up event.

For more information about this power management capability, see NDIS Wake Reason Status Indications.

NDIS_PM_SELECTIVE_SUSPEND_SUPPORTED
If this flag is set, the miniport driver supports NDIS selective suspend for network adapters.

For more information about this power management capability, see NDIS Selective Suspend.

SupportedWoLPacketPatterns
Contains flags that specify the wake-on-LAN (WOL) packet patterns that a network adapter supports. For example, the network adapter can generate a wake-up event when it receives a bitmap, a WOL magic packet, or an EAP over LAN (EAPOL) request identifier message. For a complete list of the patterns that are supported in the current operating system, see the **NDIS_PM_CAPABILITIES** reference page.

NumTotalWoLPatterns
A **ULONG** value that contains the total number of WOL patterns that a network adapter supports. This is the sum of "number of supported WOL protocol patterns" and "number of supported WOL bitmap patterns."

For example, if your driver supports 8 flexible bitmap patterns, IPv4 TCP SYN (via preset filter), and magic packet, then you would report 9 in NumTotalWoLPatterns. (8 bitmaps + 1 IPv4 TCP SYN = 9)

Note The total number of WOL patterns does not include the magic packet wake-up pattern.

For more information about WOL protocol patterns, see **NDIS_PM_WOL_PATTERN**.

MaxWoLPatternSize
Contains the maximum number of bytes that can be compared with a pattern.

MaxWoLPatternOffset
Contains the number of bytes in a packet that can be examined, which starts from the beginning of the MAC header.

MaxWoLPacketSaveBuffer
Contains the number of bytes of a WOL protocol pattern that a miniport driver can save to a buffer and indicate up the driver stack.

SupportedProtocolOffloads
Contains flags that specify the power management protocol offload features that a network adapter supports. Miniport drivers use these flags to report the low power

protocol offload capabilities of a network adapter. For example, the network adapter can support IPv4 ARP offload, IPv6 Neighbor Solicitation (NS), or IEEE 802.11 robust secure network (RSN) 4-way and 2-way handshake. For a complete list of the protocol offloads that are supported in the current operating system, see the NDIS_PM_CAPABILITIES reference page.

NumArpOffloadIPv4Addresses
Contains the number of ARP offload IPv4 addresses.

NumNSOffloadIPv6Addresses
Contains the number of network solicitation (NS) offload IPv6 requests that the network adapter supports.

MinMagicPacketWakeUp
Specifies the lowest device power state from which a network adapter can signal a wake-up event on receipt of a *magic packet*. (A *magic packet* is a packet that contains 16 contiguous copies of the receiving network adapter's Ethernet address.)

MinPatternWakeUp
Specifies the lowest device power state from which a network adapter can signal a wake-up event on receipt of a network frame that contains a pattern that is specified by the protocol driver.

MinLinkChangeWakeUp
Specifies the lowest device power state from which a network adapter can signal a wake-up event when there is a link change (media connect or disconnect).

SupportedWakeUpEvents
Specifies the media-independent wake-up events that a network adapter supports. These events are not specific to media type. For example, these wake-up events include link change events.

MediaSpecificWakeUpEvents
Specifies the media-specific wake-up events that a network adapter supports. For example, these events include following:

- The 802.11 network adapter disassociates with the access point (AP).

- The mobile broadband (MB) network adapter detects a change in its registration state to the MB Service.

If a miniport driver supports offloading protocols to a network adapter in a low power state, it must support the same low power state for the protocol offload that it supports for a pattern match WOL event; that is, the value that is specified in the **MinPatternWakeUp** or **MinMagicPacketWakeUp** member.

NDIS initializes an **NDIS_PM_CAPABILITIES** structure with the currently available power management capabilities of the underlying network adapter and passes it the protocol overlying protocol drivers during the bind operation. NDIS sets the **PowerManagementCapabilitiesEx** member of the **NDIS_BIND_PARAMETERS** structure to point to the NDIS_PM_CAPABILITIES structure.

Overlying drivers can use the OID_PM_HARDWARE_CAPABILITIES OID query to obtain the hardware power management capabilities of the network adapter. NDIS handles this OID request on behalf of the miniport driver. NDIS miniport drivers are not required to support the OID_PM_HARDWARE_CAPABILITIES OID request.

Overlying drivers can use the OID_PM_CURRENT_CAPABILITIES OID to query the currently available power management capabilities of a network adapter. NDIS handles this OID request on behalf of the miniport driver. NDIS miniport drivers are not required to support the OID_PM_CURRENT_CAPABILITIES OID request.

Obtaining and Updating Power Management Parameters

Article • 12/15/2021

Protocol drivers can use the OID_PM_PARAMETERS OID to query the hardware capabilities of a network adapter that is currently enabled. After a successful return from the query, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_PM_PARAMETERS** structure.

Protocol drivers can also use OID_PM_PARAMETERS as a set request to enable or disable the current hardware capabilities of a network adapter. The protocol driver provides a pointer to an NDIS_PM_PARAMETERS structure in the **InformationBuffer** member of the NDIS_OID_REQUEST structure.

Note Protocol drives cannot disable capabilities that were enabled by other protocol drivers. If none of the protocol drivers enable a capability, that capability is unused.

Note NDIS enables magic packet and low power on disconnect capabilities based on the user settings, and these capabilities cannot be disabled by protocol drivers.

NDIS_PM_PARAMETERS includes the following information:

EnabledWoLPacketPatterns
Contains flags that correspond to capabilities that the miniport driver reported in the **SupportedWoLPacketPatterns** member of the **NDIS_PM_CAPABILITIES** structure. For example, the network adapter is enabled to generate a wake-up event when it receives a bitmap, a WOL magic packet, or an EAP over LAN (EAPOL) request identifier message. For a complete list of the patterns that are possible in the current operating system, see the **NDIS_PM_PARAMETERS** reference page.

EnabledProtocolOffloads
Contains flags that correspond to capabilities that the miniport driver reported in the **SupportedProtocolOffloads** member of the **NDIS_PM_CAPABILITIES** structure. NDIS uses these flags to enable or disable the low power protocol offload capabilities on a network adapter. For example, the network adapter offload for IPv4 ARP, IPv6 Neighbor Solicitation (NS), or IEEE 802.11 robust secure network (RSN) 4-way and 2-way handshake is enabled. For a complete list of the protocol offloads that are supported in the current operating system, see the **NDIS_PM_PARAMETERS** reference page.

WakeUpFlags
Contains flags that NDIS uses to enable or disable wake-up capabilities on a network

adapter.

For NDIS 6.20, the NDIS_PM_WAKE_ON_LINK_CHANGE_ENABLED flag enables the capability to wake on a link change (media connect). For more information about this flag, see Low Power on Media Disconnect.

Starting with NDIS 6.30, the NDIS_PM_SELECTIVE_SUSPEND_ENABLED flag enables the support for NDIS selective suspend on underlying USB network adapters. For more information, see NDIS Selective Suspend.

When a driver sets the OID_PM_PARAMETERS OID, NDIS completes the request without forwarding it to the miniport driver. NDIS stores the requested settings and combines them with the settings from other such requests.

Before NDIS transitions the network adapter to the low power state, NDIS sends a set request to the miniport driver that contains the combined settings from all of the requests that NDIS stored. For more information about setting a low power state, see Low Power for Wake on LAN.

The capabilities that are currently enabled can be a subset of the capabilities that the hardware supports. For more information about the capabilities that the hardware supports, see Reporting Power Management Capabilities.

Adding and Deleting Wake on LAN Patterns

Article • 12/15/2021

To add a wake-on-LAN (WOL) pattern, NDIS protocol drivers issue an OID set request of OID_PM_ADD_WOL_PATTERN. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_PM_WOL_PATTERN** structure. Protocol drivers should specify a WOL packet if that WOL packet is supported by a network adapter. When the network adapter does not support the WOL packet, the protocol driver should use the WOL bitmap wake method.

NDIS_PM_WOL_PATTERN includes the following information:

Priority
Contains the priority of the WOL pattern. If an overlying driver adds a higher priority WOL pattern when there are no resources available for more WOL patterns, NDIS might remove a lower priority WOL pattern to free resources. Miniport drivers should ignore this member. A protocol driver can specify any priority that is within the pre-defined range from NDIS_PM_WOL_PRIORITY_LOWEST to NDIS_PM_WOL_PRIORITY_HIGHEST.

WoLPacketType
Contains an **NDIS_PM_WOL_PACKET** enumeration value that specifies the type of the WOL packet.

FriendlyName
Contains an **NDIS_PM_COUNTED_STRING** structure that contains the user-readable description of the WOL packet.

PatternId
Contains an NDIS-provided value that identifies the WOL pattern. Before NDIS sends the OID_PM_ADD_WOL_PATTERN OID request down to the underlying NDIS drivers or completes the request to the overlying driver, NDIS sets **PatternId** to a value that is unique among the WOL patterns on a network adapter.

NextWoLPatternOffset
Contains the offset (from the beginning of the OID request **InformationBuffer**) of one **NDIS_PM_WOL_PATTERN** structure to the next NDIS_PM_WOL_PATTERN structure in a list for the OID_PM_WOL_PATTERN_LIST OID. For more information about OID_PM_WOL_PATTERN_LIST, see Obtaining the Current Settings of WOL Patterns.

WoLPattern
Contains one of the **IPv4TcpSynParameters**, **IPv6TcpSynParameters**, **EapolRequestIdMessageParameters**, or **WoLBitMapPattern** structures in a union.

IPv4TcpSynParameters
Contains IPv4 TCP synchronize (SYN) information.

IPv6TcpSynParameters
Contains IPv6 TCP SYN information.

EapolRequestIdMessageParameters
Contains 802.1X EAP over LAN (EAPOL) request identity message parameters.

WoLBitMapPattern
Contains a WOL bitmap pattern specification.

NDIS assigns an identifier that is unique for network adapter to every WOL pattern. The pattern identifier is a unique value for each of the patterns that are set on a network adapter. However, the pattern identifier is not globally unique across all network adapters. NDIS passes the identifier to the underlying network adapter when NDIS sends the OID_PM_ADD_WOL_PATTERN OID request to the miniport driver. If adding the WOL pattern is successful, NDIS returns the identifier to the overlying driver that added the WOL pattern. The overlying driver uses the identifier to remove a previously added WOL pattern. The pattern identifier is also used in status indications to the overlying drivers when a WOL pattern is removed from a network adapter.

Protocol drivers must issue the OID set request of OID_PM_REMOVE_WOL_PATTERN to remove all of the patterns that they added to a network adapter before they close a binding to that network adapter. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a pattern identifier.

User-mode applications use the GUID_PM_REMOVE_WOL_PATTERN WMI GUID to remove a previously added WOL pattern from a network adapter. NDIS translates this WMI request to the OID set request of OID_PM_REMOVE_WOL_PATTERN for the network adapter. NDIS deletes all of the WOL patterns that an application added from the network adapter before it halts the network adapter.

NDIS allows multiple NDIS protocol drivers to add WOL patterns to the same network adapter. To ensure that the right set of WOL patterns have been set when the number of requested WOL patterns is higher than what a network adapter can support, protocol drivers assign a priority to each requested WOL pattern in the **Priority** member of the NDIS_PM_WOL_PATTERN structure. When NDIS cannot add a new high priority WOL pattern because the network adapter is out of resources, NDIS deletes one of the lower priority patterns (if any) and attempts to add the high priority pattern again.

Note A miniport driver should fail a pattern add request and return the STATUS_NDIS_PM_WOL_PATTERN_LIST_FULL status code to allow NDIS to re-prioritize the patterns.

If NDIS deletes one of the lower priority patterns, it notifies the overlying driver that set the deleted pattern with an NDIS_STATUS_PM_WOL_PATTERN_REJECTED status indication. The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure contains a ULONG for the WOL pattern identifier of the rejected WOL pattern. NDIS provided the WOL pattern identifier in the **PatternId** member of the NDIS_PM_WOL_PATTERN structure.

For wireless network adapter's that might use an infrastructure element to offload the patterns as it roams across the infrastructure, a new infrastructure element might not support the same capabilities and the miniport driver can send an NDIS_STATUS_PM_WOL_PATTERN_REJECTED status indication with an appropriate status code.

Obtaining the Current Settings of WOL Patterns

Article • 12/15/2021

Overlying drivers can use the OID_PM_WOL_PATTERN_LIST OID query request to enumerate the wake-on-LAN (WOL) patterns that are set on an underlying network adapter. After a successful return from the query, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a list of **NDIS_PM_WOL_PATTERN** structures that describe the currently added WOL patterns. For information about the contents of the **NDIS_PM_WOL_PATTERN** structure, see Adding and Deleting Wake on LAN Patterns.

NDIS handles OID_PM_WOL_PATTERN_LIST OID requests on behalf of the miniport driver. Therefore, NDIS miniport drivers are not required to support OID_PM_WOL_PATTERN_LIST OID request.

Adding and Deleting Low Power Protocol Offloads

Article • 12/15/2021

To add a low power protocol offload, NDIS protocol drivers issue an OID set request of OID_PM_ADD_PROTOCOL_OFFLOAD. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_PM_PROTOCOL_OFFLOAD** structure.

Note If an incoming packet matches both an offloaded protocol and a pattern (for example, because of a configuration error), the network adapter should respond to the packet and wake up the computer.

The **NDIS_PM_PROTOCOL_OFFLOAD** structure includes the following information:

Member	Description
Priority	Contains the priority of the protocol offload. If an overlying driver adds a higher priority protocol offload when there are no resources available for more protocol offloads, NDIS might remove a lower priority protocol offload to free resources. Miniport drivers should ignore this member. Protocol drivers can provide any value within the predefined range from NDIS_PM_PROTOCOL_OFFLOAD_PRIORITY_LOWEST to NDIS_PM_PROTOCOL_OFFLOAD_PRIORITY_HIGHEST.
ProtocolOffloadType	Contains an **NDIS_PM_PROTOCOL_OFFLOAD_TYPE** value that specifies the type of protocol offload.
FriendlyName	Contains an **NDIS_PM_COUNTED_STRING** structure that contains the user-readable description of the low power protocol offload.
ProtocolOffloadId	Contains an NDIS-provided value that identifies the offloaded protocol. Before NDIS sends the OID request of OID_PM_ADD_PROTOCOL_OFFLOAD down to the underlying NDIS drivers or completes the request to the overlying driver, NDIS sets **ProtocolOffloadId** to a value that is unique among the protocol offloads on a network adapter.

<table>
<tr><th>Member</th><th>Description</th></tr>
<tr><td><b>NextProtocolOffloadOffset</b></td><td>Contains the offset, the beginning of the OID request <i>InformationBuffer</i>, to the next <b>NDIS_PM_PROTOCOL_OFFLOAD</b> structure in a list for the OID_PM_PROTOCOL_OFFLOAD_LIST OID. For more information about OID_PM_PROTOCOL_OFFLOAD_LIST, see Obtaining the Current Parameter Settings of Low Power Protocol Offloads.</td></tr>
<tr><td><b>ProtocolOffloadParameters</b></td><td>Contains one of the <b>IPv4ARPParameters</b>, <b>IPv6NSParameters</b>, or <b>Dot11RSNRekeyParameters</b> structures in a union.
<table>
<tr><th>Term</th><th>Description</th></tr>
<tr><td>IPv4ARPParameters</td><td>Contains IPv4 ARP parameters.</td></tr>
<tr><td>IPv6NSParameters</td><td>Contains IPv6 Neighbor Solicitation (NS) parameters.</td></tr>
<tr><td>Dot11RSNRekeyParameters</td><td>Contains IEEE 802.11 robust secure network (RSN) handshake parameters</td></tr>
</table>
</td></tr>
</table>

NDIS assigns an identifier that is unique for a network adapter to every offloaded protocol. The protocol offload identifier is a unique value for each of the protocols that are offloaded on a network adapter. However, the protocol offload identifier is not globally unique across all network adapters. NDIS passes this identifier to the underlying miniport driver when NDIS sends the OID_PM_ADD_PROTOCOL_OFFLOAD OID request to the miniport driver. If offloading the protocol is successful, NDIS returns the identifier to the overlying driver that offloaded the protocol. The overlying driver uses the identifier to remove a previously offloaded protocol. The protocol offload identifier is also used in status indications to the upper layer drivers when an offloaded protocol is removed from a network adapter.

Protocol drivers must remove all of the offloaded protocols from a network adapter before closing the binding to that network adapter. To remove a low power protocol offload, a protocol driver sends an OID set request of

OID_PM_REMOVE_PROTOCOL_OFFLOAD. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a protocol offload identifier.

NDIS allows multiple NDIS protocol drivers to add protocol offloads to the same network adapter. To ensure that the right set of protocols have been offloaded to a network adapter when the number of requested offloaded protocols is higher than what a network adapter can support, protocol drivers assign a priority to each offloaded protocol. When NDIS cannot offload a new high priority protocol because the network adapter is out of resources, NDIS deletes one of the lower priority offloaded protocols (if any) and attempts to offload the high priority protocol again.

Note A miniport driver should fail a low power protocol offload add request and return the STATUS_NDIS_PM_PROTOCOL_OFFLOAD_LIST_FULL status code to allow NDIS to re-prioritize the protocol offloads.

If as a result of offloading a high priority protocol, one of the lower priority offloaded protocols is deleted, NDIS sends an **NDIS_STATUS_PM_OFFLOAD_REJECTED** status indication to notify the overlying driver that set the deleted protocol offload. The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains a protocol offload identifier of the rejected protocol offload. NDIS provided the protocol offload identifier in the **ProtocolOffloadId** member of the **NDIS_PM_PROTOCOL_OFFLOAD** structure.

Implementing IPv6 NS Offload

Article • 12/15/2021

An NDIS protocol driver sends an IPv6 neighbor solicitation (NS) offload request as an OID_PM_ADD_PROTOCOL_OFFLOAD OID request. To support these NS offload requests, miniports should do the following.

Indicating How Many Offload Requests the Miniport Adapter Supports

A miniport driver sets the **NumNSOffloadIPv6Addresses** member of the **NDIS_PM_CAPABILITIES** structure to indicate how many NS offload requests the miniport adapter supports.

Note Despite its name, the **NumNSOffloadIPv6Addresses** member contains the number of supported requests, not the number of addresses.

Note Some Windows Hardware Certification requirements, such as **Device.Network.LAN.PM.PowMgmtNDIS** and **Device.Network.WLAN.WoWLAN.ImplementWakeOnWLAN**, specify that the miniport adapter must support at least 2 NS offload requests. (In other words, to meet these requirements, the value of **NumNSOffloadIPv6Addresses** must be at least 2.) For more information, see the Windows 8 Hardware Certification Requirements.

Each NS offload request can contain 1 or 2 target addresses.

In addition, there are 2 types of NS messages: unicast and multicast. Miniport drivers must be prepared to match both types of NS message for each target address.

Example

If a miniport driver sets the **NDIS_PM_CAPABILITIES** member of the **NumNSOffloadIPv6Addresses** structure to 3, then NDIS may send up to 3 OID_PM_ADD_PROTOCOL_OFFLOAD requests of type **NdisPMProtocolOffloadIdIPv6NS**. Each OID_PM_ADD_PROTOCOL_OFFLOAD request may have exactly 1 or 2 addresses in the **TargetIPv6Addresses** member of the **NDIS_PM_PROTOCOL_OFFLOAD** structure. Therefore, the miniport must support a 3 x 2 = 6 target addresses.

Because the miniport must match both unicast and multicast NS messages for each target address, the miniport should be able to match a total of 6 x 2 = 12 NS message patterns.

Matching the NS Message

The NS message format is specified in RFC 4861 section 4.3, "Neighbor Solicitation Message Format". The miniport should match the fields in the following table.

Field	Match value	Notes
Ethernet.EtherType	0x86dd (IPv6)	Adjust as needed for non-Ethernet media types.
IPv6.Version	6	
IPv6.NextHeader	58 (ICMPv6)	
IPv6.HopLimit	255	
IPv6.Destination	**OID.TargetIPv6Addresses[x]** or **OID.SolicitedNodeIPv6Address**	The miniport must match both options for this field: **OID.TargetIPv6Addresses[x]** and **OID.SolicitedNodeIPv6Address**. If this field is **OID.TargetIPv6Addresses[x]**, the NS message is a unicast message. If this field is **OID.SolicitedNodeIPv6Address**, the NS message is a multicast message. **OID.TargetIPv6Addresses** is an array that can contain 1 or 2 addresses. If it contains 2 addresses, the miniport must match both of them. If the second address is "0::0", it must be ignored, and a second match pattern must not be created.
IPv6.ICMPv6.Type	135 (NS)	
IPv6.ICMPv6.Code	0	
IPv6.ICMPv6.TargetAddress	**OID.TargetIPv6Addresses[x]**	**OID.TargetIPv6Addresses[x]** is an array that can contain 1 or 2 addresses.

Field	Match value	Notes
IPv6.Source	OID.RemoteIPv6Address	If **OID.RemoteIPv6Address** is "0::0", this field should be ignored.

Sending the NA Message

Upon receiving the NS message, device firmware should perform the validation steps called out in RFC 4861 section 7.1, "Message Validation", including validating checksums. If the incoming NS message passes all validation, then an NA message must be generated and sent as a reply. Its format is specified in RFC 4861 section 4.4, "Neighbor Advertisement Message Format". The miniport should set the fields in the following table.

Field	Value	Notes
Ethernet.Destination	**Ethernet.Source**	Copy this value from the NS frame. Adjust as needed for non-Ethernet media types.
Ethernet.Source	The miniport's current MAC address	
IPv6.HopLimit	255	
IPv6.Source	**IPv6.ICMPv6.TargetAddress**	Copy this value from the NS frame.
IPv6.Destination	**IPv6.Source**	Copy this value from the NS frame, unless the value of **IPv6.Source** was "0::0". If the value of **IPv6.Source** was "0::0" set this field to "FF02::1".
IPv6.ICMPv6.Type	136 (NA)	
IPv6.ICMPv6.Code	0	
IPv6.ICMPv6.RouterFlag	0	

Field	Value	Notes
IPv6.ICMPv6.SolicitedFlag	0	If the value of **IPv6.Source** in the NS frame was "0::0", set this field to 1.
IPv6.ICMPv6.OverrideFlag	1	
IPv6.ICMPv6.TargetAddress	**IPv6.ICMPv6.TargetAddress**	Copy this value from the NS frame.
IPv6.ICMPv6.TLLAOption.Type	2 (Target Link-layer Address)	
IPv6.ICMPv6.TLLAOption.Length	1	
IPv6.ICMPv6.TLLAOption.LinkLayerAddress	**OID.MacAddress**	

Obtaining the Current Parameter Settings of Low Power Protocol Offloads

Article • 12/15/2021

A protocol driver can use OID_PM_PROTOCOL_OFFLOAD_LIST OID query to get a list of all the protocols that have been offloaded by that protocol on a network adapter. After a successful return from the query, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a list of **NDIS_PM_PROTOCOL_OFFLOAD** structures that describe the currently active protocol offloads. For information about the contents of the **NDIS_PM_PROTOCOL_OFFLOAD** structure, see Adding and Deleting Low Power Protocol Offloads.

NDIS handles OID_PM_PROTOCOL_OFFLOAD_LIST OID and GUID_PM_PROTOCOL_OFFLOAD_LIST WMI requests on behalf of the miniport driver. Therefore, NDIS miniport drivers are not required to support OID_PM_PROTOCOL_OFFLOAD_LIST OID request.

Overlying drivers can use the OID_PM_GET_PROTOCOL_OFFLOAD method OID to get parameter settings for a low power protocol offload from a miniport driver. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to a protocol offload identifier. After a successful return from the method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_PM_PROTOCOL_OFFLOAD** structure.

Low Power for Wake on LAN

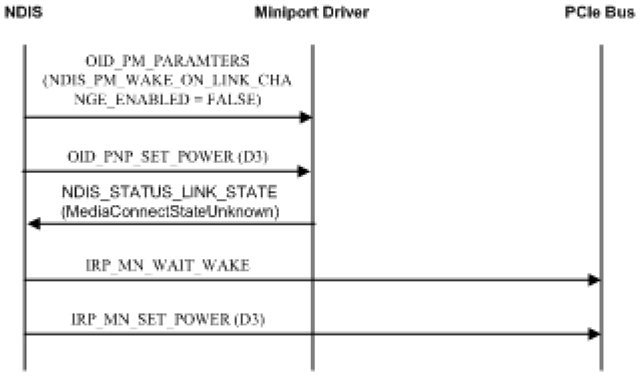
Article • 12/15/2021

The wake on LAN (WOL) feature wakes the computer from a low power state when a network adapter detects a WOL event.

A miniport driver reports network adapter WOL capabilities during initialization. For more information about reporting WOL capabilities, see Reporting Power Management Capabilities.

Note that the lower power on the media disconnect (D3 on disconnect) feature is canceled when the computer enters a sleep state in order to prevent waking the computer when the link state is externally cycled; that is, when a switch is turned off and on. For more information about D3 on disconnect, see Low Power on Media Disconnect.

The following figure illustrates the sequence of events that occurs to set a network adapter to a low power state.


NDIS
Miniport Driver
PCIe Bus
OID_PM_PARAMTERS (NDIS_PM_WAKE_ON_LINK_CHANGE_ENABLED = FALSE)
OID_PNP_SET_POWER (D3)
NDIS_STATUS_LINK_STATE (MediaConnectStateUnknown)
IRP_MN_WAIT_WAKE
IRP_MN_SET_POWER (D3)

When NDIS puts a network adapter in a low power state, the following sequence occurs:

1. NDIS uses OID_PM_PARAMETERS to enable wake on LAN and to disable wake on media connect. NDIS_PM_WAKE_ON_LINK_CHANGE_ENABLED is cleared in the **WakeUpFlags** member.

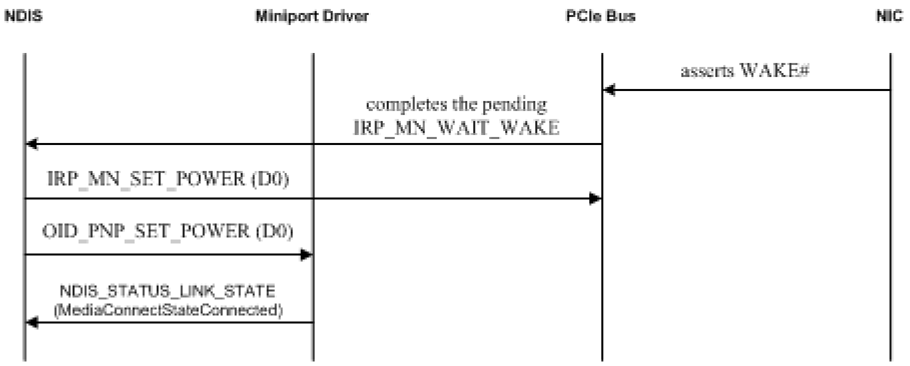
2. NDIS uses OID_PNP_SET_POWER to notify the miniport driver of the new power state (D3).

3. The miniport driver may indicate an unknown media connect state using the **NDIS_STATUS_LINK_STATE** status indication. The **MediaConnectStateUnknown** value is set in the **MediaConnectState** member of the **NDIS_LINK_STATE** structure. For more information, see the **NDIS_STATUS_LINK_STATE** documentation.

4. NDIS sends the PCI Express (PCIe) bus an **IRP_MN_WAIT_WAKE** IRP to wait for a WOL event.

5. NDIS sends the PCIe bus an **IRP_MN_SET_POWER** IRP to set the bus to the D3 state.

The following figure illustrates the sequence of events that occurs to restore full power to a network adapter after a WOL event.


NDIS
Miniport Driver
PCIe Bus
NIC
asserts WAKE#
completes the pending IRP_MN_WAIT_WAKE
IRP_MN_SET_POWER (D0)
OID_PNP_SET_POWER (D0)
NDIS_STATUS_LINK_STATE (MediaConnectStateConnected)

When the network adapter is waking the computer the following sequence occurs:

1. The network adapter wakes the system by asserting WAKE# on the PCIe bus or PME# on the PCI bus.
2. The bus completes the pending **IRP_MN_WAIT_WAKE** IRP. The IRP is pending completion from the last step in the power down sequence.
3. NDIS sets the bus to full power (D0) with the **IRP_MN_SET_POWER** IRP.
4. NDIS notifies the miniport driver that the network adapter is at full power (D0) with the OID set request of OID_PNP_SET_POWER.
5. The network adapter notifies NDIS of a media connect event with the **NDIS_STATUS_LINK_STATE** status indication. The **MediaConnectStateConnected** value is set in the **MediaConnectState** member of the **NDIS_LINK_STATE** structure.

Starting with NDIS 6.30, if the miniport driver supports **NDIS_STATUS_PM_WAKE_REASON** status indications, it must issue this status notification if the network adapter wakes the system. The driver issues this status notification while it is handling the OID set request of OID_PNP_SET_POWER for the transition to a full-power (D0) state.

For more information, see NDIS Wake Reason Status Indications.

Low Power on Media Disconnect

Article • 12/15/2021

The low power on media disconnect (D3 on disconnect) feature saves power by placing a network adapter in a low-power state (D3) when the media is disconnected. When the media is reconnected, the network adapter is brought back up to the full-power state (D0).

NDIS uses the D3 on disconnect feature under these conditions:

- The network adapter hardware must be able to generate a wake event on media connect.
- The miniport driver must report the wake event capability of the network adapter in the **MinLinkChangeWakeUp** member of the NDIS_PM_CAPABILITIES structure.
- The value of **MinLinkChangeWakeUp** must correspond to the value of the **DeviceWake** member of the DEVICE_CAPABILITIES structure that is reported by the IRP_MN_QUERY_CAPABILITIES IRP.
- The miniport driver must register as an NDIS 6.20 driver or later version.
- The network adapter must be an Ethernet PCI adapter.
- The wake event capability must be enabled by the ***DeviceSleepOnDisconnect** standard INF file keyword.
- The computer chipset must be able to correctly propagate the wake event while the computer is fully powered. NDIS validates this by querying the DEVPKEY_PciDevice_S0WakeupSupported PCI property.

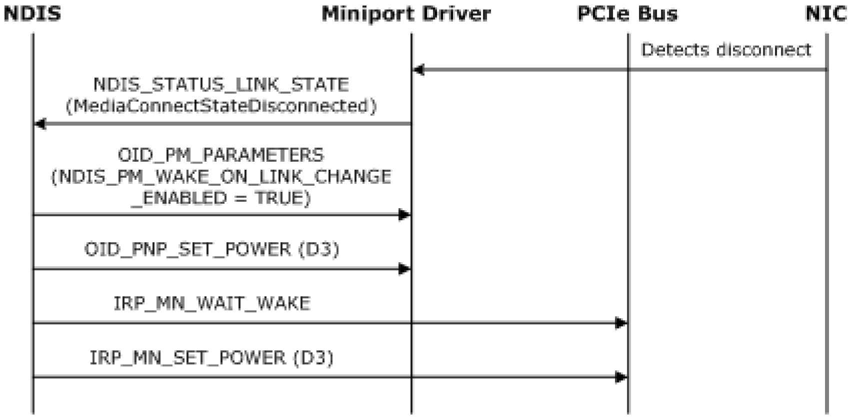
Note that D3 on disconnect is only available while the computer is fully powered in the working state (S0). This feature is canceled when the computer enters a sleep state to prevent waking the computer when the link state is externally cycled; that is, when a switch is turned off and on. For more information about the setting the low-power state when a computer enters a sleep state, see Low Power for Wake on LAN.

A miniport driver reports D3 on disconnect capabilities during initialization. For more information about reporting D3 on disconnect capabilities, see Reporting Power Management Capabilities.

The ***DeviceSleepOnDisconnect** standard INF file keyword specifies whether the device has enabled or disabled support for D3 on disconnect. For more information about this INF keyword, see Standardized INF Keywords for Power Management.

During initialization, a miniport drivers that supports D3 on disconnect must report the lowest power level where it can support the ability to notify the operating system of media connect event. The miniport driver reports the power level in the **MinLinkChangeWakeUp** member of the NDIS_PM_CAPABILITIES structure. For example, the miniport driver can report **NdisDeviceStateD3**.

The following figure illustrates the sequence of events to set a network adapter to a low-power state after a media disconnect event.


NDIS
Miniport Driver
PCIe Bus
NIC
Detects disconnect
NDIS_STATUS_LINK_STATE (MediaConnectStateDisconnected)
OID_PM_PARAMETERS (NDIS_PM_WAKE_ON_LINK_CHANGE _ENABLED = TRUE)
OID_PNP_SET_POWER (D3)
IRP_MN_WAIT_WAKE
IRP_MN_SET_POWER (D3)

When the adapter detects a media disconnect, the following sequence occurs:

1. The network adapter hardware detects a media disconnect event and passes the information to the miniport driver.

2. The miniport driver notifies NDIS of a media disconnect event using the NDIS_STATUS_LINK_STATE status indication. The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure contains an NDIS_LINK_STATE structure. The MediaConnectStateDisconnected value is set in the **MediaConnectState** member of the **NDIS_LINK_STATE** structure.

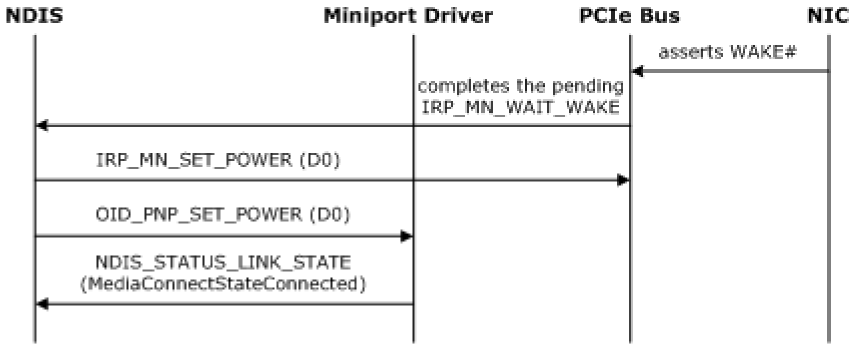
3. NDIS uses OID_PM_PARAMETERS to disable Wake-on-LAN and to enable wake on media connect (NDIS_PM_WAKE_ON_LINK_CHANGE_ENABLED is set in the **WakeUpFlags** member).

4. NDIS uses the OID_PNP_SET_POWER OID to notify the miniport driver of the new power state (D3).

5. NDIS sends the PCIe bus an IRP_MN_WAIT_WAKE IRP to wait for a reconnect event.

6. NDIS sets the PCIe bus to the D3 state with the IRP_MN_SET_POWER IRP.

The following figure illustrates the sequence of events to restore full power to a network adapter after a media connect event.


NDIS
Miniport Driver
PCIe Bus
NIC
asserts WAKE#
completes the pending IRP_MN_WAIT_WAKE
IRP_MN_SET_POWER (D0)
OID_PNP_SET_POWER (D0)
NDIS_STATUS_LINK_STATE (MediaConnectStateConnected)

When the media is reconnected the following sequence occurs:

1. The network adapter wakes the system by asserting WAKE# on the PCIe bus or PME# on the PCI bus.

2. The bus completes the pending **IRP_MN_WAIT_WAKE** IRP. The IRP is pending completion from the last step in the disconnect sequence.

3. NDIS sets the bus to full power (D0) with the **IRP_MN_SET_POWER** IRP.

4. NDIS notifies the miniport driver that the network adapter is in the full power (D0) state with the OID set request of OID_PNP_SET_POWER.

5. The network adapter notifies NDIS of a media connect event with the **NDIS_STATUS_LINK_STATE** status indication. The **MediaConnectStateConnected** value is set in the **MediaConnectState** member of the **NDIS_LINK_STATE** structure.

Starting with NDIS 6.30, if the miniport driver supports **NDIS_STATUS_PM_WAKE_REASON** status indications, it must issue this status notification if the network adapter wakes the system. The driver issues this status notification while it is handling the OID set request of OID_PNP_SET_POWER for the transition to a full-power (D0) state.

For more information, see NDIS Wake Reason Status Indications.

Note If the miniport driver issues an **NDIS_STATUS_PM_WAKE_REASON** status indication, it must do this before it issues the **NDIS_STATUS_LINK_STATE** status indication.

NDIS_STATUS_PM_CAPABILITIES_CHANGE

Article • 03/14/2023

The NDIS_STATUS_PM_CAPABILITIES_CHANGE status indicates a change in the power management capabilities of a network adapter to overlying drivers.

Remarks

NDIS generates an NDIS_STATUS_PM_CAPABILITIES_CHANGE status indication when an update to the previously reported power management capabilities is required.

The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure contains a pointer to an NDIS_PM_CAPABILITIES structure with the updated power management capabilities.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_PM_CAPABILITIES

NDIS_STATUS_INDICATION

NDIS_STATUS_PM_HARDWARE_CAPABILITIES

Article • 03/14/2023

The **NDIS_STATUS_PM_HARDWARE_CAPABILITIES** status indicates to overlying drivers that a change in the power management (PM) hardware capabilities of a network adapter has occurred.

Remarks

The miniport driver generates an **NDIS_STATUS_PM_HARDWARE_CAPABILITIES** status indication when an update to the previously reported power management capabilities is required.

The miniport driver for an 802.11 network adapter can generate this status indication.

A MUX intermediate driver that provides load balancing failover (LBFO) support can also generate this status indication. The MUX driver aggregates the PM capabilities of the underlying network adapters that are part of the LBFO team. If the PM capabilities change because an adapter has been either added or removed from the team, the MUX driver must generate this status indication. For more information on LBFO MUX intermediate drivers, see NDIS MUX Intermediate Drivers.

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains a pointer to an **NDIS_PM_CAPABILITIES** structure with the updated power management capabilities.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_PM_CAPABILITIES

NDIS_STATUS_INDICATION

NDIS_STATUS_PM_OFFLOAD_REJECTED

Article • 03/14/2023

The NDIS_STATUS_PM_OFFLOAD_REJECTED status indicates to overlying drivers that a power management protocol offload was rejected.

Remarks

NDIS or miniport drivers can generate the NDIS_STATUS_PM_OFFLOAD_REJECTED status indication when either of them removes an offloaded protocol. The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains a ULONG for the protocol offload identifier of the rejected protocol offload. NDIS provided the protocol offload identifier in the **ProtocolOffloadId** member of the **NDIS_PM_PROTOCOL_OFFLOAD** structure.

NDIS generates an NDIS_STATUS_PM_OFFLOAD_REJECTED status indication when it has to remove a previously offloaded protocol from a network adapter. For example, NDIS might remove the protocol offload to free resources for a higher priority protocol offload. NDIS sends the status indication to the binding that offloaded the rejected protocol offload, but does not send it to other bindings.

Miniport drivers report this status indication to reject a previously accepted protocol offload. For example, for a WiFi WOL case, the miniport driver must make an NDIS_STATUS_PM_OFFLOAD_REJECTED status indication when PTK/GTK rotation is not required to support WOL (due to vendor specific infrastructure support).

For wireless network adapters that use infrastructure elements to offload protocols and roam across the infrastructure, it is possible that a new infrastructure element might not support the same capabilities as the previous one. In this case, the miniport driver can issue a status indication to NDIS, and NDIS will issue NDIS_STATUS_PM_OFFLOAD_REJECTED with a specific error code.

A WiFi driver might cache protocol offload requests locally. When the driver processes an OID for adding or deleting a protocol offload, the driver can choose to only update its local cache. The driver can defer the update of the infrastructure until it receives the OID_PM_PARAMETERS OID.

The infrastructure might not have enough resources to accommodate all of the protocol offloads. In this case, the infrastructure can accept a partial list of the protocol offloads. When the miniport driver completes the OID_PM_PARAMETERS set request, the miniport driver must make NDIS_STATUS_PM_OFFLOAD_REJECTED status indications for each of the protocol offloads that the AP rejects.

For example, a network adapter can use the AP's proxy ARP to support ARP offload.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_PM_PROTOCOL_OFFLOAD

NDIS_STATUS_INDICATION

OID_PM_PARAMETERS

NDIS_STATUS_PM_WOL_PATTERN_REJECTED

Article • 03/14/2023

The NDIS_STATUS_PM_WOL_PATTERN_REJECTED status indicates to overlying drivers that a power management wake on LAN (WOL) pattern was rejected.

Remarks

NDIS or miniport drivers can generate the NDIS_STATUS_PM_WOL_PATTERN_REJECTED status indication when either of them removes a WOL pattern. The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains a ULONG for the WOL pattern identifier of the rejected WOL pattern. NDIS provided the WOL pattern identifier in the **PatternId** member of the **NDIS_PM_WOL_PATTERN** structure.

NDIS generates an NDIS_STATUS_PM_WOL_PATTERN_REJECTED status indication when it must remove a previously added WOL pattern from a network adapter. For example, NDIS might remove the WOL pattern to free resources for a higher priority WOL pattern. The notification event will only be sent to the binding that added the removed pattern.

For wireless network adapters that use infrastructure elements to offload the patterns and roam across the infrastructure, it is possible that a new infrastructure element might not support the same capabilities as the previous one. In this case, the miniport driver can issue a status indication to NDIS, and NDIS will issue NDIS_STATUS_PM_WOL_PATTERN_REJECTED with a specific error code.

A WiFi driver might cache wake-up patterns locally. When the driver processes an OID for adding or deleting a wake-up pattern, the driver can choose to only update its local cache. The driver can defer the update of the infrastructure until it receives the OID_PM_PARAMETERS OID.

The infrastructure might not have enough resources to accommodate all of the wake-up patterns. In this case, the infrastructure can accept a partial list of the wake-up patterns. When the miniport driver completes the OID_PM_PARAMETERS set request, the driver must make NDIS_STATUS_PM_WOL_PATTERN_REJECTED status indications for each of the WOL patterns that the access point (AP) rejects.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_PM_WOL_PATTERN

NDIS_STATUS_INDICATION

OID_PM_PARAMETERS

Required and Optional OIDs for Power Management

Article • 12/15/2021

For a miniport driver, supporting power management involves supporting power management object identifiers (OIDs). For a detailed description of how miniport drivers process queries and sets to OIDs, see Obtaining and SettingMiniport Driver Information and NDIS Support for WMI.

There are two levels of power management support for miniport drivers:

1. A miniport driver can support a network adapter making a transition between power states. This support is the minimum level of power management support. For a description of device power states for network adapters, see Device Power States for Network Adapters.

2. A miniport driver can also support one or more network wake-up events.

Miniport drivers report power management capabilities during initialization. For more information about power management capabilities that are reported during initialization, see **NDIS_MINIPORT_ADAPTER_ATTRIBUTES** and the related attributes structures.

A miniport driver must support the following OIDs directly or in attributes for a network adapter to make a transition between power states:

- OID_PNP_CAPABILITIES

  Intermediate drivers must respond to this OID query. NDIS responds to OID_PNP_CAPABILITIES requests on behalf of physical network adapters. For more information about responding to this OID in an intermediate driver, see Handling PnP Events and Power Management Events in an Intermediate Driver.

- OID_PNP_QUERY_POWER

  This OID specifies a device power state to which the network adapter should prepare to transition. A miniport driver must always return NDIS_STATUS_SUCCESS in response to a query of OID_PNP_QUERY_POWER. By returning NDIS_STATUS_SUCCESS in response to this OID request, the miniport driver guarantees that it will transition the network adapter to the specified device power state on receipt of a subsequent OID_PNP_SET_POWER request. The miniport driver, in this case, must do nothing to jeopardize the transition.

- OID_PNP_SET_POWER

  This OID indicates that the network adapter must transition to the indicated device power state. A miniport driver must set the network adapter to the specified state before the driver returns NDIS_STATUS_SUCCESS. A miniport driver must always return NDIS_STATUS_SUCCESS in response to this OID. If OID_PNP_SET_POWER sets a network adapter to working power state and the miniport driver fails this OID, NDIS assumes that the device is in a unrecoverable state.

To support network wake-up events, a miniport driver must also support the OID_PNP_ENABLE_WAKE_UP OID. Both protocol drivers and NDIS use this OID to enable a network adapter's wake-up capabilities. For more information, see Enabling Wake-Up Events.

To support network wake-up frames (see Network Wake-Up Events), a miniport driver must also support the following OIDs that are related to wake-up events:

- OID_PNP_ADD_WAKE_UP_PATTERN

  A protocol driver uses this OID to add a wake-up pattern to a list that either the network adapter or miniport driver or both maintain.

- OID_PNP_REMOVE_WAKE_UP_PATTERN

  A protocol driver uses this OID to delete a wake-up pattern that it previously specified with OID_PNP_ADD_WAKE_UP_PATTERN.

NDIS miniport drivers that support network wake-up events can optionally support the following statistical OIDs that are related to wake-up events:

- OID_PNP_WAKE_UP_ERROR

  Protocol drivers query this OID to determine the number of false wake-ups signaled by the miniport driver's network adapter.

- OID_PNP_WAKE_UP_OK

  Protocol drivers query this OID to determine the number of valid wake-ups that are signaled by the miniport driver's network adapter.

Device Power States for Network Adapters

Article • 12/15/2021

A device power state for a network adapter describes a network adapter's level of power consumption and computing activity.

There are four device power states: D0, D1, D2, and D3. D0 is the highest-powered state. D1, D2, and D3 are the sleeping states. D3 is subdivided into D3hot and D3cold.

The state number is inversely related to power consumption: higher-numbered states use less power. Power might be fully removed from the network adapter in the D3 state.

For a thorough description of device states, see the following topics:

- Device Power States
- Device Working State D0
- Device Low-Power States
- Required Support for Device Power States

Note NDIS processes power management IRPs, but NDIS drivers do not.

The device power states for network adapters are defined as follows:

Device Working State D0

This power state is described for all devices in Device Working State D0. For network adapters and miniport drivers:

Power consumption
The network adapter is fully powered and delivering full functionality and performance.

Device context
The hardware device context is maintained by either the network adapter or miniport driver or both.

Miniport driver and network adapter behavior
The network adapter is fully compliant with the requirements of the attached network. The operation of the miniport driver and network adapter is not restricted because of low-power requirements.

Restore time
Not applicable.

Device Power State D1

This power state is described for all devices in Device Low-Power States. For network adapters and miniport drivers:

Power consumption
This state is the highest-powered sleeping state. Power consumption is less than that in the D0 state and greater than or equal to that in the D2 state.

Device context
The miniport driver should preserve any hardware device context that might be lost. The miniport driver should restore such context when the device returns to the D0 state.

Miniport driver and network adapter behavior
The miniport driver does not receive transmission requests from protocol drivers. NDIS either notifies a bound protocol driver of the network adapter's transition to the sleeping state or, if the protocol driver is an old driver that is not power management-aware, NDIS disables transmission requests from the protocol driver. However, the miniport driver should be able to handle the case in which it does receive transmission requests when it is in this low-power state. In this case, the miniport driver should fail all transmission requests.

The miniport driver does not indicate up any packets that the network adapter might receive while it is in this state.

The network adapter does not generate interrupts. However, the miniport driver must be able to handle interrupts, because a shared interrupt could be generated on the bus.

Restore time
The time to restore the network adapter to the D0 state is less than that required when the network adapter is in the D2 state.

Device Power State D2

This power state is described for all devices in Device Low-Power States. For network adapters and miniport drivers:

Power consumption
An intermediate sleeping state. Power consumption is less than that in the D1 state and greater than or equal to that in the D3 state.

Device context
Same as for D1.

Miniport driver and network adapter behavior
Same as for D1.

Restore time
The time to restore the network adapter to the D0 state is greater than that required when the network adapter is in the D1 state and less than that required when the network adapter is in the D3 state.

Device Power State D3

This power state is described for all devices in Device Low-Power States. For network adapters and miniport drivers:

Power consumption
The sleeping state with the least amount of power. The amount of power may be nonzero (D3hot) or it may be exactly zero (D3cold). For more information about D3hot and D3cold, see Device Low-Power States.

Device context
Same as for D1.

Miniport driver and network adapter behavior
Same as for D1.

Restore time
The time to restore the network adapter to the D0 state is greater than that required when the network adapter is in the D2 state.

Before a network adapter can transition to a sleeping state, its miniport driver must disable everything under the miniport driver's control: interrupts must be disabled, timers must be canceled, and so on. A miniport driver cannot access the network adapter hardware after the bus driver sets the network adapter to the D3 state.

Transitions Allowed Between Device Power States

The only transitions allowed between device power states are from the highest-powered state (D0) to a sleeping state (D1, D2, D3), or from a sleeping state to the highest-powered state. NDIS never commands a network adapter to transition directly from one sleeping state to another.

About Network Wake-Up Events

Article • 08/23/2022

A *network wake-up event* is an external event that causes a network adapter to wake the system. A network adapter wakes the system by asserting a bus-specific wake-up signal that eventually results in the system making a transition from a sleeping state to the working state.

NDIS defines the following two network wake-up events:

- Receipt of a network wake-up frame that contains a pattern that was specified by a bound protocol driver.

- Receipt of a Magic Packet.

A network adapter can support any combination of network wake-up events, including none at all. NDIS treats the miniport driver as not power management-aware if the miniport driver sets the **PowerManagementCapabilities** member of NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES to **NULL**.

Depending on the capabilities of the network adapter, a network wake-up event can occur from any device power state, including the highest-powered state (D0).

Network Wake-Up Frames

If, during initialization, a miniport driver indicates that a network adapter can signal a wake-up on the receipt of a packet that contains a specified pattern, a bound protocol can enable the pattern-based wake up method on the network adapter and specify wake-up patterns. To enable this type of wake-up, a protocol driver sets the NDIS_PNP_WAKE_UP_PATTERN_MATCH flag in OID_PNP_ENABLE_WAKE_UP.

A protocol driver uses OID_PNP_ADD_WAKE_UP_PATTERN to specify a wake-up pattern, along with a mask that indicates which bytes of an incoming packet should be compared with the pattern. A protocol driver can remove a wake-up pattern with OID_PNP_REMOVE_WAKE_UP_PATTERN.

Magic-Packet Wake-Up

A *Magic Packet* is a packet that contains 16 contiguous copies of the receiving network adapter's MAC address.

This section includes:

Enabling Wake-Up Events

Handling Wake-Up Events

Enabling Wake-Up Events

Article • 12/15/2021

A protocol driver can send an OID_PNP_ENABLE_WAKE_UP request to enable one or more of the network adapter's wake-up capabilities. NDIS does not immediately enable these wake-up capabilities. Instead, NDIS keeps track of the wake-up capabilities that are enabled by the protocol driver and, just before the miniport driver transitions to a sleeping state, sends an OID_PNP_ENABLE_WAKE_UP to the miniport driver to enable the appropriate wake-up events. After the miniport driver initializes the network adapter, or when it resumes from a low-power state, the miniport driver must disable any wake up methods that are set on the network adapter.

Before the miniport driver transitions to a low-power state (that is, before NDIS sends the miniport driver an OID_PNP_SET_POWER request), NDIS sends the miniport driver an OID_PNP_ENABLE_WAKE_UP request to enable the network adapter's wake-up capabilities.

Handling Wake-Up Events

Article • 12/15/2021

A miniport driver does not handle a wake-up event detected by a NIC. When a NIC detects an enabled wake-up event, it asserts a bus-specific wake-up line. The power manager then sends a power IRP to NDIS, which, in response, sends the miniport driver an OID_PNP_SET_POWER OID that requests the miniport driver to put a NIC in the highest-powered (D0) state.

Handling an OID_PNP_QUERY_POWER OID

Article • 12/15/2021

The OID_PNP_QUERY_POWER OID requests a miniport driver to indicate whether it can transition a network adapter to a low-power state. A miniport driver must always return NDIS_STATUS_SUCCESS in response to a query of OID_PNP_QUERY_POWER. By returning NDIS_STATUS_SUCCESS to this OID request, the miniport driver guarantees that it will transition the network adapter to the specified device power state on receipt of a subsequent OID_PNP_SET_POWER request. The miniport driver, in this case, must do nothing to jeopardize the transition.

An OID_PNP_QUERY_POWER request is always followed by an OID_PNP_SET_POWER request. The OID_PNP_SET_POWER request can immediately follow the OID_PNP_QUERY_POWER request or can arrive at an unspecified interval after the OID_PNP_QUERY_POWER request. A device state of D0, which is specified in the OID_PNP_SET_POWER request, effectively cancels a preceding OID_PNP_QUERY_POWER request.

Transitioning to a Sleeping State

Article • 12/15/2021

If a miniport driver supports wake-up events, NDIS sends the driver an OID_PNP_ENABLE_WAKE_UP request before sending an OID_PNP_SET_POWER request. For more information, see Enabling Wake-Up Events. A miniport driver must not fail an OID_PNP_SET_POWER request.

Before returning NDIS_STATUS_SUCCESS in response to an OID_PNP_SET_POWER request, the miniport driver must:

- Perform the device-dependent operations that are needed to prepare the network adapter for the sleeping state.

- Save any packet filters, multicast addresses, the current MAC address, wake-up patterns, and any other hardware context that the network adapter cannot preserve in a sleeping state.

- Disable interrupts and the network adapter's DMA engine. A miniport driver cannot access the network adapter hardware after the network adapter has been set to the D3 state by the bus driver.

Transitioning to the Working State

Article • 12/15/2021

NDIS initiates the transition to the working power state (D0) by sending the miniport driver an OID_PNP_SET_POWER request that specifies state D0. The miniport driver must then perform any device-dependent operations needed to restore the network adapter to a working state. The miniport driver must also restore any hardware context--packet filters, multicast addresses, the current media access control (MAC) address, or wake-up patterns--that the network adapter might have lost.

Note Starting with NDIS 6.30, the miniport driver that support NDIS packet coalescing must clear its coalesced packet counter. The driver must also configure the network adapter to flush any packets that it coalesced before the low-power transition. For more information, see Handling Packet Coalescing Receive Filters.

Before the miniport driver returns NDIS_STATUS_SUCCESS in response to the OID_PNP_SET_POWER request, the miniport driver and a network adapter must be ready for normal operation.

Power Management Considerations for Gigabit Ethernet Network Adapters

Article • 12/15/2021

When a gigabit Ethernet network adapter is operating at 1000 megabits per second (Mbps), it draws a lot of electrical power. Before such a network adapter transitions to a low-power state, its link speed is typically reduced so that the network adapter draws less power. The reduced link speed enables the network adapter to transition to a low-power state. While changing link speeds during the transition to a low-power state, the network adapter typically loses network connectivity for a short time.

Conversely, when a gigabit Ethernet network adapter transitions to the fully-on state from a low-power state, the network adapter's link speed is increased to its fully operational rate. During this transition, the network adapter might also lose connectivity for a short time.

While a miniport driver's underlying network adapter is transitioning to or from a low-power state, the miniport must not indicate either a change in link speed or a change in connection status. For more information about indicating a change in link speed, see **NDIS_STATUS_LINK_STATE**. For more information about indicating a change in connection status, see Indicating Connection Status.

Power Management for Old Miniport Drivers

Article • 12/15/2021

NDIS treats a miniport driver as an old miniport driver that is not power management-aware if:

- During initialization, the bus driver indicates that the system or the NIC is not power management-aware.
- The miniport driver returns NDIS_STATUS_UNSUPPORTED in response to the OID_PNP_CAPABILITIES query. Only NDIS 5.1 and earlier miniport drivers or intermediate drivers receive this OID query.
- The user disables power management in the user interface.

NDIS supports only two device power states for old miniport drivers that do not support power manegement: the highest-powered (D0) state and the D3 state.

During initialization, an old miniport driver can indicate that NDIS should not halt it before the system transitions to the sleeping (D3) state. A miniport driver makes such an indication by setting the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag in the *AttributeFlags* parameter that the miniport driver passes to the **NdisMSetMiniportAttributes** function. An old miniport driver should set this flag only if it can:

- Save all hardware context that it might require.
- Put a NIC in an appropriate state for the sleeping state (D3).
- Reactivate the NIC to the highest-powered state (D0).

If NDIS determines from the bus driver that the NIC is not power management-aware and if the miniport driver did not set the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag, NDIS will not query the miniport driver's power management capabilities. However, if the miniport driver set the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag, NDIS issues an OID_PNP_CAPABILITIES request to the miniport driver. In this case, the miniport driver should succeed the OID_PNP_CAPABILITIES request with NDIS_STATUS_SUCCESS. In the NDIS_PM_WAKE_UP_CAPABILITIES structure that the miniport driver returns in response to this request, the miniport driver must also specify a device power state of **NdisDeviceStateUnspecified** for each wake-up capability.

NDIS provides the following power management support for old miniport drivers:

- NDIS succeeds all **IRP_MN_QUERY_POWER** requests that the system power manager sends to the device object that represents the NIC. That is, NDIS guarantees that the miniport driver's NIC will transition to the D3 state in response to any IRP_MN_QUERY_POWER request from the system.

- If the miniport driver did not set the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag during initialization, NDIS calls the miniport driver's *MiniportHaltEx* function before the miniport driver's NIC transitions to state D3. The miniport driver's NIC loses all hardware context information.

- If the miniport driver set the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag during initialization, NDIS does not halt the miniport driver before the system transitions to state D3. Instead, NDIS issues the miniport driver an OID_PNP_SET_POWER request to D3 state. The miniport driver must save any hardware context that it uses and put a NIC in a state appropriate for the D3 state.

- While the system is transitioning to the S0 system power states, NDIS calls the miniport driver's *MiniportInitializeEx* function if NDIS halted the miniport driver. If NDIS did not halt the miniport driver, NDIS issues the miniport driver an OID_PNP_SET_POWER request to D0 state. The miniport driver must put a NIC in a state appropriate for the D0 state.

- If the miniport driver was halted and reinitialized, NDIS restores all the appropriate miniport driver settings, such as packet filters and multicast address lists, by issuing OID requests. If the miniport driver was not halted and then reinitialized, the miniport driver must restore such settings.

How NDIS Sets the Power Policy for a Network Adapter

Article • 03/14/2023

NDIS serves as the device power policy owner for each network device. As such, NDIS sets and administers the power policy for each network device. For more information about managing device power policy, see Managing Device Power Policy.

NDIS uses the following information to set the power policy for a NIC:

- The **DEVICE_CAPABILITIES** structure that the bus driver returns in response to an **IRP_MN_QUERY_CAPABILITIES** request that NDIS issued.
- The miniport driver's response to an OID_PNP_CAPABILITIES request issued by NDIS.
- User input from the user interface (UI).

Using the DEVICE_CAPABILITIES Structure

When a NIC is enumerated, NDIS queries the NIC's capabilities by issuing, in addition to other requests, an **IRP_MN_QUERY_CAPABILITIES** request. In response to this request, the bus driver returns a **DEVICE_CAPABILITIES** structure. NDIS copies this structure and uses the following information from this structure when setting the power policy for the NIC.

Member	Description
DeviceD1 and DeviceD2	TRUE if the device supports the D1 power state.
DeviceD1 and DeviceD2	TRUE if the device supports the D2 power state.
WakeFromD0, WakeFromD1, WakeFromD2, and WakeFromD3	TRUE if the device can respond to an external wake signal while in the D0 power state.
WakeFromD0, WakeFromD1, WakeFromD2, and WakeFromD3	TRUE if the device can respond to an external wake signal while in the D1 power state.
WakeFromD0, WakeFromD1, WakeFromD2, and WakeFromD3	TRUE if the device can respond to an external wake signal while in the D2 power state.
WakeFromD0, WakeFromD1, WakeFromD2, and WakeFromD3	TRUE if the device can respond to an external wake signal while in the D3 power state.

Member	Description
DeviceState[**PowerSystemMaximum**]	Specifies the highest-powered device state that this device can maintain for each system power state, from **PowerSystemUnspecified** to **PowerSystemShutdown**.
SystemWake	Specifies lowest-powered system power state (S0 through S4) from which the device can signal a wake event.
DeviceWake	Specifies lowest-powered device power state (D0 through D3) from which the device can signal a wake event.

NDIS uses the DEVICE_CAPABILITIES information to determine if:

- Both the system and the NIC support power management, and if so, which device power states the NIC can be in for each system power state.

- Both the system and the NIC support wake-on-LAN, and if so, from which device power states the NIC can wake the system.

WakeFromD0 through **WakeFromD3** indicate the device power states from which the NIC can wake the system.

The **DeviceState** array indicates, for each system power state, the highest-powered device power state in which the NIC can be and still support that system power state. For example, consider the following array values.

```C++
DeviceState[PowerSystemWorking] PowerDeviceD0
DeviceState[PowerSystemSleeping1] PowerDeviceD1
DeviceState[PowerSystemSleeping2] PowerDeviceD2
DeviceState[PowerSystemSleeping3] PowerDeviceD2
DeviceState[PowerSystemHibernate] PowerDeviceD3
DeviceState[PowerSystemShutdown] PowerDeviceD3
```

As indicated by the preceding array of sample values, when the system is in system power state S1, the NIC can be in device power state D1, D2, or D3. When the system is in system power state S2 or S3, the NIC can be in device power state D2 or D3.

To determine whether both the system and NIC support wake-on-LAN, NDIS examines both the **SystemWake** and **DeviceWake** members. If both **SystemWake** and **DeviceWake** are set to **PowerSystemUnspecified**, NDIS treats the NIC as capable of power management. In this case, or if the miniport driver set the

NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag during initialization, NDIS subsequently issues the miniport driver an OID_PNP_CAPABILITIES request to obtain more information about the NIC's wake-up capabilities.

Using OID_PNP_CAPABILITIES

After a miniport driver successfully returns from its *MiniportInitializeEx* function, NDIS sends an OID_PNP_CAPABILITIES request to the driver if either of the following is true:

- Both the **SystemWake** and **DeviceWake** members of the **DEVICE_CAPABILITIES** structure that is returned by the bus driver are *not* set to **PowerSystemUnspecified**.
- The miniport driver set the NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND flag when it called **NdisMSetMiniportAttributes** during initialization.

Note that NDIS issues an OID_PNP_CAPABILITIES request regardless of whether the user has enabled wake-on-LAN in the user interface.

If the miniport driver returns NDIS_STATUS_SUCCESS in response to a query of OID_PNP_CAPABILITIES, NDIS treats the miniport driver as power management-capable. If the miniport driver returns NDIS_STATUS_NOT_SUPPORTED, NDIS treats the miniport driver as an old miniport driver that is not power management-capable. For more information about power management for such drivers, see Power Management for Old Miniport Drivers.

A miniport driver that succeeds an OID_PNP_CAPABILITIES request returns the following information to NDIS in response to the request:

- The lowest device power state from which the NIC can wake the system on receipt of a Magic Packet.
- The lowest device power state from which the NIC can wake the system on receipt of a network frame that contains a pattern that the protocol driver specifies.

As soon as NDIS gets this information, it determines, for each system power state, the device power states to which it can set the NIC if wake-on-LAN is enabled by the user in the UI. If there are no allowable low-power device states from which the NIC can generate a wake-up signal (that is, if all the low-power device power states specified in the **DeviceState** array of DEVICE_CAPABILITIES structures are lower than lowest device power states from which the NIC can wake the system), NDIS makes the **Allow the device to bring the computer out of standby** option in the **Power Management** tab unavailable for the NIC. Then, the user cannot enable wake-on-LAN.

Note Wake-on-LAN is possible only if both the NIC and the system are power management-capable. If the system is not power management-capable, NDIS will not query a NIC's power management capabilities.

Using User Input

For a power management-capable NIC, Microsoft Windows 2000 and later versions provide the following options in the **Power Management** tab of a NIC's **Properties** dialog box:

Allow the computer to turn off this device to save power

Allow the device to bring the computer out of standby

The first option is selected by default to enable power management for the NIC. If the user clears the option, NDIS treats the NIC as an old NIC with regard to power management. For more information, see Power Management for Old Miniport Drivers.

The second option is clear by default. If NDIS determines that there is no allowable low-power state from which the NIC can generate a wake-up signal, NDIS makes the second option unavailable. For example, if the **DeviceState** array member of the DEVICE_CAPABILITIES structure indicates that the NIC must be in D3 for all low-power system states, and if **DeviceWake** indicates that the lowest-powered device state from which the NIC can wake the system is D2, then NDIS shades makes the second check box unavailable.

In addition to the preceding two options, Windows XP and Windows Vista provide a third option in the **Power Management** tab for a NIC:

Only allow management stations to bring the computer out of standby

This option, which is subordinate to the second option that is described earlier, is available only if:

- The user selected the second option to enable wake-on-LAN.
- The miniport driver, in responding to the OID_PNP_CAPABILITIES, indicated that the NIC could wake the system on receipt of a Magic Packet.

The **Only allow management stations to bring the computer out of standby** option is clear by default. The user can select this option to specify that only the receipt of a Magic Packet will cause the NIC to generate a wake-up signal to the system.

Whenever a user selects or clears a power management option for a NIC, the system notifies NDIS of the change. NDIS writes the new setting to the registry so that the changed setting persists across restarts.

Avoiding NDIS Power Management Problems

Article • 03/14/2023

The following rules will help you avoid power-management problems with your network adapter:

- A network adapter must always report its power management capabilities to the bus driver.
- Do not try to enable or disable power management for a network adapter based on registry settings. NDIS obtains power management information about the network adapter from the bus driver before the network adapter's miniport driver is initialized. If the information obtained from the bus driver indicates that the network adapter is not power management-capable, NDIS treats the network adapter as an old network adapter and does not issue an OID_PNP_CAPABILITIES request to the network adapter's miniport driver.
- Do not attempt to provide custom power-management controls in the user interface.

Introduction to NDIS QoS for Data Center Bridging

Article • 03/14/2023

This section describes NDIS Quality of Service (QoS) for the IEEE 802.1 Data Center Bridging (DCB) interface. Miniport drivers use NDIS QoS for traffic prioritization of transmit, or *egress*, packets over a DCB-compliant network adapter.

NDIS QoS for DCB was introduced with NDIS 6.30 in Windows Server 2012.

This section includes the following topics:

Overview of Data Center Bridging

Article • 05/31/2022

IEEE 802.1 Data Center Bridging (DCB) is a collection of standards that defines a unified 802.3 Ethernet media interface, or *fabric*, for local area network (LAN) and storage area network (SAN) technologies. DCB extends the current 802.1 bridging specification to support the coexistence of LAN-based and SAN-based applications over the same networking fabric within a data center. DCB also supports technologies, such as Fibre Channel over Ethernet (FCoE) and iSCSI, by defining link-level policies that prevent packet loss.

DCB consists of the following 802.1 draft standards that specify how networking devices can interoperate within a unified data center fabric:

Priority-based Flow Control (PFC)

PFC is specified in the IEEE 802.1Qbb draft standard. This standard is part of the framework for the DCB interface.

PFC supports the reliable delivery of data by substantially reducing packet loss due to congestion. This allows loss-sensitive protocols, such as FCoE, to coexist with traditional loss-insensitive protocols over the same unified fabric.

PFC specifies a link-level flow control mechanism between directly connected peers. PFC is similar to IEEE 802.3 PAUSE frames but operates on individual 802.1p priority levels instead. This allows a receiver to pause a transmitter on any priority level.

For more information on PFC, see Priority-based Flow Control (PFC).

Enhanced Transmission Selection (ETS)

ETS is a transmission selection algorithm (TSA) that is specified in the IEEE 802.1Qaz draft standard. This standard is part of the framework for the DCB interface.

ETS allocates bandwidth between traffic classes that are assigned to different IEEE 802.1p priority levels. Each traffic class is allocated a percentage of available bandwidth on the data link between directlyconnected peers. If a traffic class doesn't use its allocated bandwidth, ETS allows other traffic classes to use the available bandwidth that the traffic class is not using.

For more information on ETS, see Enhanced Transmission Selection (ETS) Algorithm.

Data Center Bridging Exchange (DCBX) Protocol

The Data Center Bridging Exchange (DCBX) protocol is also specified in the IEEE 802.1Qaz draft standard. DCBX allows DCB configuration parameters to be exchanged between two directlyconnected peers. This allows these peers to adapt and tune Quality of Service (QoS) parameters to optimize data transfer over the connection.

DCBX is also used to detect conflicting QoS parameter settings between the network adapter (*local peer*) and the remote peer. Based on the local and remote QoS parameter settings, the miniport driver resolves the conflicts and derives a set of operational QoS parameters. The network adapter uses these operational parameters for the prioritized transmission of packets to the remote peer. For more information about how the driver resolves its operational NDIS QoS parameter settings, see Resolving Operational NDIS QoS Parameters.

DCBX consists of DCB type-length-value (TLV) settings that are carried over the Link Layer Discovery Protocol (LLDP) packets. LLDP is specified in the IEEE 802.1AB-2005 standard.

> ⓘ **Note**
>
> DCBX specifies that the local peer maintain configuration parameters from only one remote peer at any given time. As a result, the network adapter maintains only one set of local, remote, and operational NDIS QoS parameters.

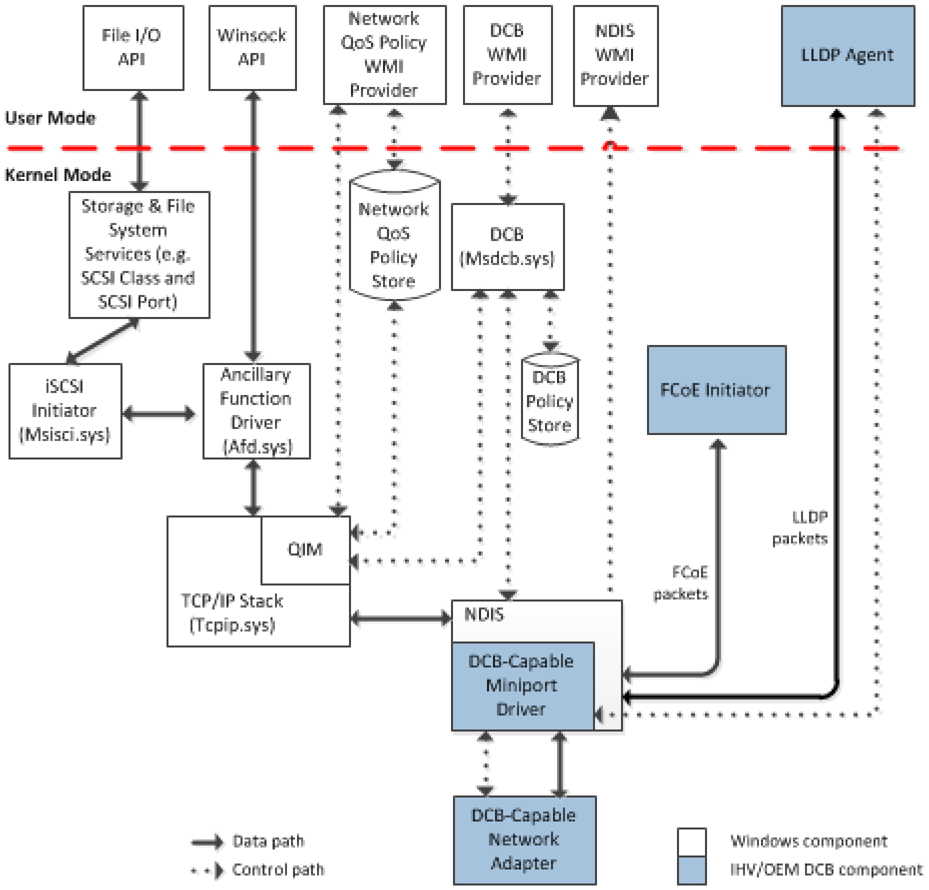
Each ETS traffic class and PFC configuration setting is associated with an IEEE 802.1p priority level. The priority level is specified as a 3-bit value within a packet's 802.1Q tag. For NDIS packets, the 802.1p priority level is specified by the **UserPriority** member of the **NDIS_NET_BUFFER_LIST_8021Q_INFO** structure that is associated with a packet's **NET_BUFFER_LIST** structure.

For more information about priority levels, see IEEE 802.1p Priority Levels.

NDIS QoS Architecture for Data Center Bridging

Article • 03/14/2023

This section describes the various components that are part of the NDIS Quality of Service (QoS) architecture for IEEE 802.1 Data Center Bridging (DCB). These components are shown in the following diagram.



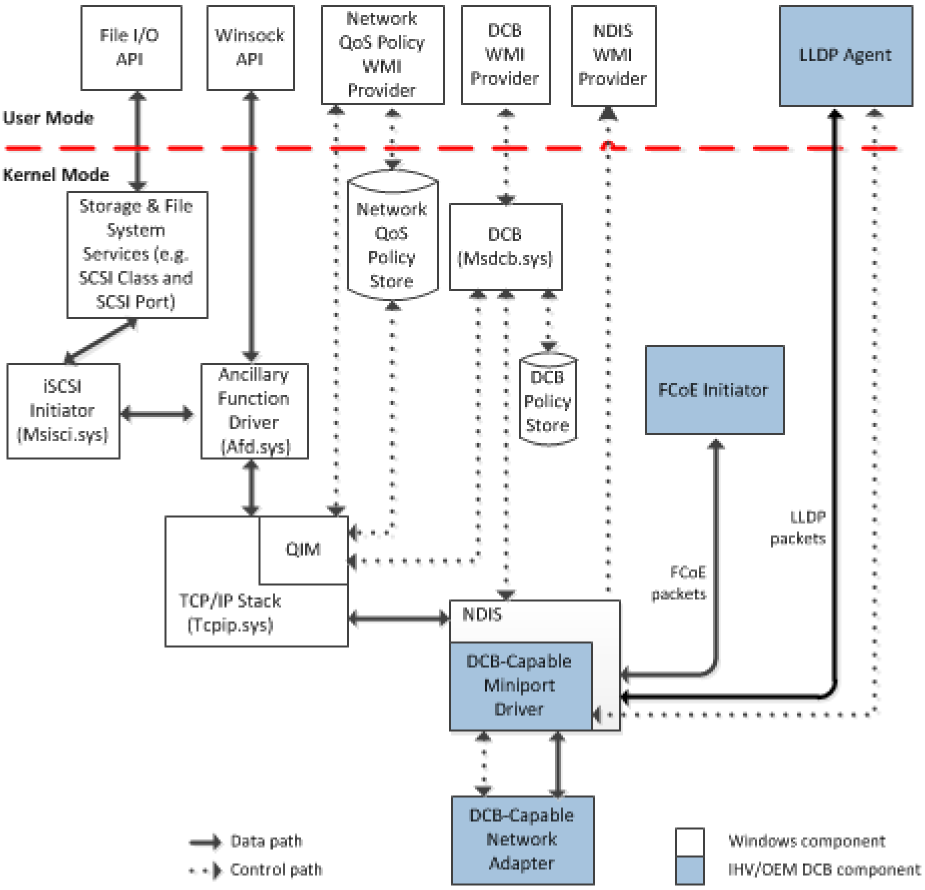
The unshaded boxes in the diagram represent components that the Windows operating system provides, including components that support DCB. For more information about these components, see System-Provided DCB Components.

The shaded boxes in the diagram represent DCB components that independent hardware vendors (IHVs) and original equipment manufacturers (OEMs) provide. For more information about these components, see Vendor-Provided DCB Components.

System-Provided DCB Components

Article • 12/15/2021

This section describes the various components that are part of the NDIS Quality of Service (QoS) architecture for IEEE 802.1 Data Center Bridging (DCB). These components are shown in the following diagram.



The unshaded boxes in the diagram represent modules that the Windows operating system provides. In particular, the operating system provides the following modules that support DCB:

Network QoS Policy WMI Provider
This module provides an interface for Windows Management Instrumentation (WMI) clients to query and set QoS-based network policies within the operating system's network stack. These policies allow specific types of network traffic to be assigned to DCB traffic classes for transmit, or *egress*, management and prioritized delivery.

A network policy defines a set of conditions and actions. An egress packet that matches a condition, such as a TCP or UDP port number, is assigned the action related to the condition. Starting with NDIS 6.30, policy actions specify an 802.1p priority level to which a DCB traffic class has been assigned.

Network QoS policies are a superset of NDIS QoS classifications. A policy defined by using the Network Policy WMI Provider may be automatically migrated to NDIS QoS as long as the policy conditions and actions match the restrictions of an NDIS QoS classification element. For more information about these elements, see NDIS QoS Traffic Classifications.

This WMI provider saves the network policies within a separate store in the system registry.

DCB WMI Provider
This component provides an interface for WMI clients to query and set NDIS QoS parameters on the underlying miniport driver. Through WMI-based PowerShell cmdlets and WMI methods, clients can configure DCB functionality, such as Priority-based Flow Control (PFC) and Enhanced Transmission Selection (ETS), on the miniport driver that supports DCB.

DCB
The DCB component (Msdcb.sys) configures the DCB-capable miniport driver with DCB parameter settings. The DCB component obtains these settings from the following sources:

- Persistent settings from the DCB policy store in the system registry.
- Dynamic settings from the DCB WMI user-mode provider. These settings are delivered over a private I/O control (IOCTL) interface between the DCB WMI provider and the DCB module.

The DCB component also relays QOS classification settings from the QIM component to miniport drivers that support NDIS QoS.

QoS Inspection Module (QIM)
The QIM component is part of the packet inspection layer in the core TCP/IP network stack (Tcpip.sys). Starting with Windows Server 2012, this component performs QoS-based packet classification for traffic prioritization.

The QIM component exposes a private Network Programming Interface (NPI). When the DCB component sets QoS parameters on the underlying miniport driver, it relays those settings to the QIM component over this NPI interface. This allows DCB to create QoS policies in QIM that are based on DCB application priority settings. For more information about the NPI interface, see Network Programming Interface.

The QIM component also processes networking QoS policies from the policy store in the registry. If those policies are compatible with NDIS QoS classification elements, the QIM component migrates the policies and issues them to the DCB component over the NPI interface.

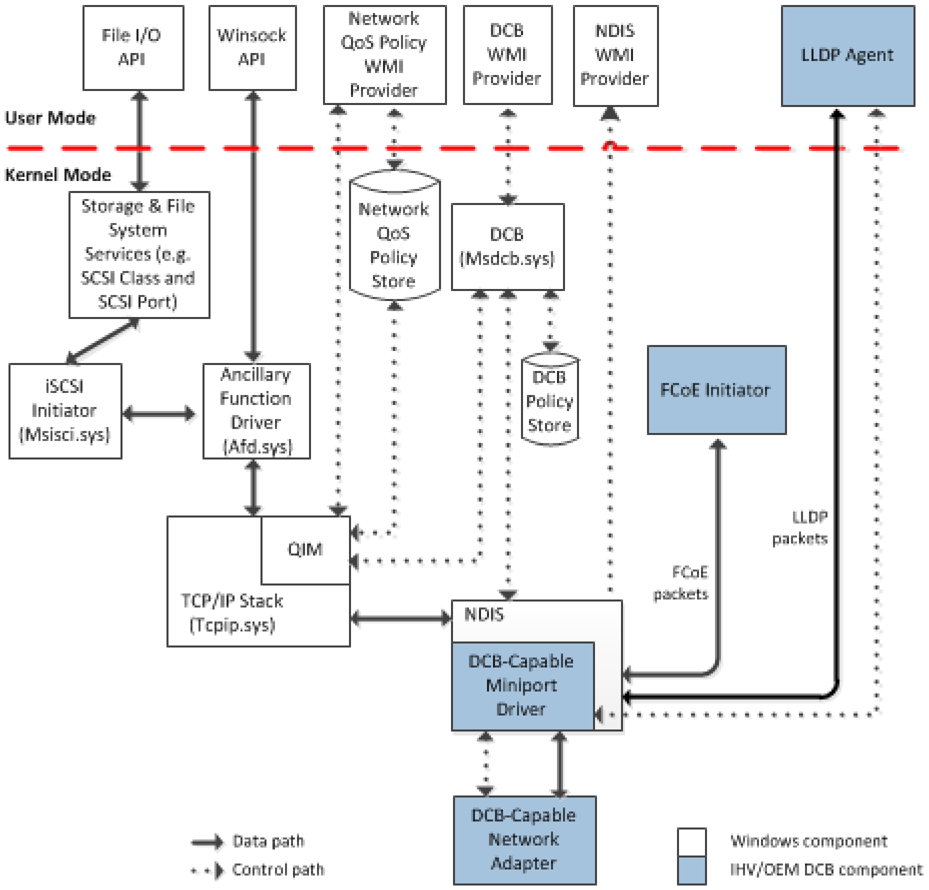
Note The policies that are created by the QIM component go into the active store and do not persist through a system restart.

Note Starting with Windows Server 2012, the DCB and DCB WMI provider components are not installed by default. These components are installed and enabled through the installation of the Microsoft DCB server feature. This feature is installed by using the Add Roles and Features wizard of the Server Manager.

Vendor-Provided DCB Components

Article • 12/15/2021

This section describes the various components that are part of the NDIS Quality of Service (QoS) architecture for IEEE 802.1 Data Center Bridging (DCB). These components are shown in the following diagram.



The shaded boxes in the diagram represent DCB components that independent hardware vendors (IHVs) and original equipment manufacturers (OEMs) provide. The following list describes these components:

Link Layer Discovery Protocol (LLDP) Agent
Starting with Windows Server 2012, vendors that support the IEEE 802.1Qaz DCB Exchange (DCBX) protocol can provide support for the IEEE 802.1Qab LLDP protocol over which DCBX is carried. This LLDP support can be provided through either the miniport driver or an LLDP agent.

Typically, the LLDP agent and the DCB-capable miniport driver communicate over a private control path, such as a private I/O control (IOCTL) interface.

If the vendor provides an LLDP agent, we highly recommend that the agent reside in user mode to mitigate the general stability risks associated with processing network packets.

Fibre Channel over Ethernet (FCoE) Initiator
The Windows operating system does not natively support FCoE. To support FCoE, the vendor must provide an FCoE initiator stack that is used to connect to remote storage devices.

DCB-Capable Miniport Driver and Network Adapter
To support NDIS QoS for DCB, the miniport driver and network adapter must support the requirements described in NDIS QoS Requirements for Data Center Bridging.

NDIS QoS Requirements for Data Center Bridging

Article • 03/14/2023

To support NDIS Quality of Service (QoS) for IEEE 802.1 Data Center Bridging (DCB), the miniport driver and network adapter must support the following:

- The miniport driver and network adapter must support Priority-based Flow Control (PFC) as specified by the IEEE 802.1Qbb draft standard.

- The miniport driver and network adapter must support the Enhanced Transmission Selection (ETS) algorithm as specified by the IEEE 802.1Qaz draft standard.

- The miniport driver and network adapter must support a minimum of three NDIS QoS traffic classes, and must support a minimum of two ETS-based traffic classes. Of these two, at least one ETS-based traffic class must support PFC.

  For more information about traffic classes, see NDIS QoS Traffic Classes.

- The miniport driver and network adapter must support the strict priority algorithm for transmission selection as specified by the IEEE 802.1Q-2005 standard.

For NDIS QoS, the miniport driver and network adapter can optionally support the Data Center Bridging Exchange (DCBX) protocol as specified by the IEEE 802.1Qaz draft standard. To support DCBX, the miniport driver and adapter must also support the Link Layer Discovery Protocol (LLDP) protocol as specified in the IEEE 802.1AB-2005 standard.

In addition, the miniport driver itself must support the following for NDIS QoS:

- The miniport driver must support NDIS 6.30 or later versions of NDIS.

- The miniport driver must support object identifier (OID) method requests of OID_QOS_PARAMETERS for setting NDIS QoS parameters. For more information, see Setting Local NDIS QoS Parameters.

  Note NDIS handles most of the NDIS QoS OID requests for the miniport driver with the exception of OID_QOS_PARAMETERS.

- The miniport driver must be able to resolve conflicting NDIS QoS parameter settings that were received over a DCBX frame that was sent from the remote peer. The driver resolves conflicts between its local and remote NDIS QoS parameters to determine its operational NDIS QoS parameters that the network adapter uses for prioritized packet transmission. For more information about this process, see Resolving Operational NDIS QoS Parameters.

- The miniport driver must be able to issue NDIS status indications when its operational NDIS QoS parameters change. For more information about this process, see Indicating Changes to the Operational NDIS QoS Parameters.
- The miniport driver must be able to issue NDIS status indications when it detects a change in the NDIS QoS parameters on the remote peer. For more information about this process, see Indicating Changes to the Remote NDIS QoS Parameters.

NDIS QoS Traffic Classes

Article • 03/14/2023

NDIS Quality of Service (QoS) traffic classes specify a set of policies that determine how the network adapter handles transmit, or *egress*, packets for prioritized delivery. Each traffic class specifies the following policies that are applied to egress packets:

Priority Level and Flow Control
This policy defines the IEEE 802.1p priority level and optional flow control algorithms for the egress traffic.

For more information, see Priority Levels and Flow Control.

Traffic Selection Algorithms (TSAs)
This policy specifies how the network adapter selects egress traffic for delivery from its transmit queues. For example, the adapter could select egress packets based on IEEE 802.1p priority or the percentage of the egress bandwidth that is allocated to each traffic class.

For more information, see Transmission Selection Algorithms (TSAs).

Note Bandwidth allocation is only supported for the Enhanced Transmission Selection (ETS) TSA. For more information, see Enhanced Transmission Selection (ETS) Algorithm.

Traffic classes are specified through object identifier (OID) method requests of OID_QOS_PARAMETERS. This OID request contains an **NDIS_QOS_PARAMETERS** structure that specifies the following NDIS QoS parameters:

- The number of traffic classes to be configured on the network adapter. Each traffic class is identified by a value in the range from zero to (**NumTrafficClasses**–1), where **NumTrafficClasses** is a member of the **NDIS_QOS_PARAMETERS** structure.

  Note Starting with NDIS 6.30, NDIS QoS supports a maximum of NDIS_QOS_MAXIMUM_TRAFFIC_CLASSES (8) traffic classes. The network adapter must support a minimum of three traffic classes.

- The 802.1p priority level associated with the traffic class.

- The TSA associated with the traffic class.

- The transmit bandwidth allocated to each traffic class that uses the ETS TSA.

OID method requests of OID_QOS_PARAMETERS also specify traffic classifications. These classifications define the relationship between egress packets and IEEE 802.1p priority levels. For more information, see NDIS QoS Traffic Classifications.

IEEE 802.1p Priority Levels

Article • 12/15/2021

IEEE 802.1p was specified by an IEEE 802.1 Task Group to address traffic prioritization for Quality of Service (QoS). 802.1p is not a separate IEEE 802.1 standard, but is defined in Annex G of the IEEE 802.1Q-2005 standard.

IEEE 802.1p defines a 3-bit field called the Priority Code Point (PCP) within an IEEE 802.1Q tag. For NDIS packets, the 802.1p PCP value is specified by the **UserPriority** member of the NDIS_NET_BUFFER_LIST_8021Q_INFO structure that is associated with a packet's NET_BUFFER_LIST structure.

The PCP value defines 8 priority levels, with 7 the highest priority and 1 the lowest priority. The priority level of 0 is the default. Each priority level defines a *class of service* that identifies separate traffic classes of transmitted packets.

Priority-based Flow Control (PFC)

Article • 12/15/2021

Priority-based Flow Control (PFC) is specified in the IEEE 802.1Qbb draft standard. This standard is part of the framework for the IEEE 802.1 Data Center Bridging (DCB) interface.

PFC enables flow control over a unified 802.3 Ethernet media interface, or *fabric*, for local area network (LAN) and storage area network (SAN) technologies. PFC is intended to eliminate packet loss due to congestion on a network link. This allows loss-sensitive protocols, such as Fibre Channel over Ethernet (FCoE), to coexist with traditional loss-insensitive protocols over the same unified fabric.

PFC specifies a link-layer flow control mechanism between directly connected peers. PFC is similar to IEEE 802.3 PAUSE frames but operates on individual 802.1p priority levels instead. This allows a receiver to pause a transmitter on any 802.1p priority level.

PFC uses the 802.3 PAUSE frame, and extends it with the following PFC fields:

- An 8-bit mask that specifies which 802.1p priority levels should be paused.
- A timer value for each priority specifying how long the traffic for that priority level should be paused.

When the receiver sends an 802.3 PAUSE frame with PFC data, the switch blocks the transmit of frames with the specified priority level to the port on which the receiver is connected. When the timer value expires, the switch resumes the transmit of paused frames on the port.

NDIS Quality of Service (QoS) parameters are specified through the **NDIS_QOS_PARAMETERS** structure. The **PfcEnable** member contains a bitmap, in which each bit specifies whether PFC is enabled for a 802.1p priority level.

For more information about priority levels, see IEEE 802.1p Priority Levels.

Transmission Selection Algorithms (TSAs)

Article • 12/15/2021

This section provides an overview of the traffic selection algorithms (TSAs) that are used by NDIS Quality of Service (QoS) for the IEEE 802.1 Data Center Bridging (DCB) interface.

This section includes the following topics:

Strict Priority Algorithm

Enhanced Transmission Selection (ETS) Algorithm

Strict Priority Algorithm

Article • 12/15/2021

Strict priority is a transmission selection algorithm (TSA) that is specified in the IEEE 802.1Q-2005 standard. This standard is part of the framework for the IEEE 802.1 Data Center Bridging (DCB) interface.

When the network adapter employs the strict priority TSA, it selects packets for transmission based solely on the packet's specified IEEE 802.1p priority level. As a result, packets with higher priority levels are always transmitted before packets with lower priority levels.

The miniport driver specifies its support for the strict priority TSA by setting NDIS_QOS_CAPABILITIES_STRICT_TSA_SUPPORTED in the **Flags** member of the **NDIS_QOS_CAPABILITIES** structure. The driver uses this structure to register its NDIS QoS and DCB capabilities in the call to **NdisMSetMiniportAttributes**.

For more information about priority levels, see IEEE 802.1p Priority Levels.

Note Starting with NDIS 6.30, the miniport driver that supports NDIS Quality of Service (QoS) for the DCB interface must advertise support for the strict priority TSA.

Enhanced Transmission Selection (ETS) Algorithm

Article • 12/15/2021

Enhanced Transmission Selection (ETS) is a transmission selection algorithm (TSA) that is specified by the IEEE 802.1Qaz draft standard. This standard is part of the framework for the IEEE 802.1 Data Center Bridging (DCB) interface.

Transmission selection based solely on IEEE 802.1p priority levels can lead to situations in which higher-priority traffic blocks lower-priority traffic. ETS ensures fairness by allowing a minimum amount of bandwidth to be allocated to traffic classes that are assigned to different 802.1p priority levels.

Each traffic class is allocated a percentage of the available bandwidth on the data link between directly connected peers. If a traffic class doesn't use its allocated bandwidth, ETS allows other traffic classes to use the available bandwidth that the traffic class is not using.

NDIS Quality of Service (QoS) traffic classes are defined through OID method requests of OID_QOS_PARAMETERS. This OID request contains an **NDIS_QOS_PARAMETERS** structure which specifies the following traffic class attributes:

- The number of traffic classes that is specified by the **NumTrafficClasses** member.

- The TSA used by the traffic class. This is specified by the **TsaAssignmentTable** member. If the table element for the traffic class is set to NDIS_QOS_TSA_ETS, the traffic class uses the ETS TSA.

- The 802.1p priority that is assigned to the traffic class. A traffic class can be assigned to one or more priority levels. However, each priority level can only be assigned to one traffic class.

  For more information, see Traffic Class Priority Assignment.

- The bandwidth allocated for the traffic class. This is specified by the **TcBandwidthAssignmentTable** member. This table is only valid for traffic classes that use the ETS TSA.

  For more information about ETS bandwidth allocation, see Bandwidth Allocation.

For more information about priority levels, see IEEE 802.1p Priority Levels.

Traffic Class Priority Assignment

Article • 12/15/2021

The Enhanced Transmission Selection (ETS) algorithm specifies a method by which a traffic class is assigned an 802.1p priority level. ETS is specified in the IEEE 802.1Qaz draft standard. This standard is part of the framework for the IEEE 802.1 Data Center Bridging (DCB) interface.

NDIS Quality of Service (QoS) traffic classes specify a set of policies that determine how the network adapter handles transmit, or *egress*, packets. Under ETS, each traffic class is assigned an IEEE 802.1p priority level in which to transmit packets. A traffic class can be assigned to one or more IEEE 802.1p priority levels. However, each IEEE 802.1p priority level can only be assigned to one traffic class.

NDIS QoS parameters are specified through the **NDIS_QOS_PARAMETERS** structure. The **PriorityAssignmentTable** member contains an array that specifies the priority assignments for each traffic class.

For more information about priority levels, see IEEE 802.1p Priority Levels.

Bandwidth Allocation

Article • 12/15/2021

Bandwidth allocation is a component of the Enhanced Transmission Selection (ETS) algorithm. ETS is specified in the IEEE 802.1Qaz draft standard. This standard is part of the framework for the IEEE 802.1 Data Center Bridging (DCB) interface.

Under ETS, each traffic class is assigned a percentage of the bandwidth that is available to transmit packets between two directly connected peers. If the bandwidth allocated to a traffic class is not completely used, ETS allows the unused bandwidth to be shared by traffic classes that have different IEEE 802.1p priority levels.

NDIS Quality of Service (QoS) parameters are specified through the **NDIS_QOS_PARAMETERS** structure. The **TcBandwidthAssignmentTable** member contains an array that specifies the bandwidth allocation for traffic classes that use the ETS algorithm.

For more information about priority levels, see IEEE 802.1p Priority Levels.

NDIS QoS Traffic Classifications

Article • 03/14/2023

NDIS Quality of Service (QoS) classifies transmit, or *egress*, packets for prioritized delivery by the network adapter. Each traffic classification specifies the following:

- A classification *condition* that is based on a data pattern within the egress packet data.

  Starting with NDIS 6.30, classification conditions are based on a 16-bit value, such as a UDP or TCP destination port or a media access control (MAC) EtherType.

- A classification *action* that defines the traffic class to be used to handle the egress packet.

  Starting with NDIS 6.30, classification actions specify an 802.1p priority level.

Note Traffic classifications are also known as "application priorities" in the IEEE 802.1 specifications.

NDIS QoS traffic classifications are intended for the following types of egress packet traffic:

- Packets based on traffic that is offloaded to the miniport driver, such as Fibre Channel over Ethernet (FCoE) or iSCSI packets.

- Packets based on connections that are managed and enforced by the miniport driver, such as RDMA.

Because NDIS QoS traffic classifications are not intended for TCP/IP traffic generated by the operating system, the miniport driver does not need to perform packet inspection. Instead, if a classification condition matches a packet type that has been offloaded or managed by the driver, it can simply apply the classification action to all packets that belong to that type. For example, if the miniport driver is enabled for FCoE offloads and the classification condition specifies the iSCSI TCP port number (860 or 3260), the driver prioritizes all egress iSCSI packets with the priority level defined for the classification action.

The DCB component (Msdcb.sys) specifies traffic classifications through OID method requests of OID_QOS_PARAMETERS. This OID request contains an **NDIS_QOS_PARAMETERS** structure that specifies an array of **NDIS_QOS_CLASSIFICATION_ELEMENT** structures. Each of these structures defines a traffic classification.

The DCB component specifies a *default* traffic classification that is applied to all egress packets that do not match other classification conditions. In this case, the network adapter assigns the IEEE 802.1p priority level that is associated with the default classification to these egress packets. The default traffic classification has the following attributes:

- It has a traffic classification condition of type NDIS_QOS_CONDITION_DEFAULT.
- It is the first traffic classification defined in the array of **NDIS_QOS_CLASSIFICATION_ELEMENT** structures.

For more information on the DCB component, see NDIS QoS Architecture for Data Center Bridging.

Registering NDIS QoS Capabilities

Article • 03/14/2023

Miniport drivers regsiter the following Quality of Service (QoS) capabilities with NDIS during network adapter initialization:

- The NDIS QoS hardware capabilities that the network adapter supports.

  Note Starting with NDIS 6.30, the miniport driver must register the NDIS QoS hardware capabilities that the adapter supports only if the***QOS** INF keyword setting is present in the registry. In this case, the driver must register its NDIS QoS hardware capabilities regardless of whether those capabilities are enabled or disabled on the adapter.

- The NDIS QoS hardware capabilities that are currently enabled on the network adapter.

  Note A miniport driver's NDIS QoS hardware capabilities can be enabled or disabled through the ***QOS** INF keyword setting in the registry. This setting is displayed on the **Advanced** property page for the network adapter.

For more information about the NDIS QoS INF keyword settings, see Standardized INF Keywords for NDIS QoS.

The miniport driver reports the hardware NDIS QoS capabilities of the underlying network adapter through an **NDIS_QOS_CAPABILITIES** structure that is initialized in the following way:

1. The miniport driver initializes the **Header** member. The driver sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_QOS_CAPABILITIES.

   Starting with NDIS 6.30, the miniport driver sets the **Revision** member of **Header** to NDIS_QOS_CAPABILITIES_REVISION_1 and the **Size** member to NDIS_SIZEOF_QOS_CAPABILITIES_REVISION_1.

2. If the network adapter supports the strict priority transmission selection algorithm (TSA), the miniport driver sets the NDIS_QOS_CAPABILITIES_STRICT_TSA_SUPPORTED flag in the **Flags** member. For more information on this algorithm, see Strict Priority Algorithm.

   Note Starting with NDIS 6.30, the miniport driver and network adapter that support NDIS QoS for IEEE Data Center Bridging (DCB) must support the strict priority TSA.

3. If the network adapter supports the ability to bypass media access control security (MACsec) processing, the miniport driver sets the NDIS_QOS_CAPABILITIES_MACSEC_BYPASS_SUPPORTED flag in the **Flags** member. For more information about MACsec, refer to the IEEE 802.1AE-2006 standard.

   Note Starting with NDIS 6.30, the network adapter does not need to support the bypass of MACsec processing.

4. The miniport driver sets the **MaxNumTrafficClasses** member to the maximum number of NDIS QoS traffic classes that the network adapter supports. A traffic class defines the transmit, or *egress* policies for QoS, such as IEEE 802.1p priority level and bandwidth allocation. For more information about traffic classes, see NDIS QoS Traffic Classes.

   Note Starting with NDIS 6.30, the network adapter must support a minimum of three traffic classes.

5. The miniport driver sets the **MaxNumEtsCapableTrafficClasses** member to the maximum number of NDIS QoS traffic classes that the network adapter can use with the Enhanced Transmission Selection (ETS) algorithm. This value must be less than or equal to the value of the **MaxNumTrafficClasses** member.

   For more information on ETS, see Enhanced Transmission Selection (ETS) Algorithm.

   Note For the network adapter to support NDIS QoS, it must support a minimum of two ETS-capable traffic classes.

6. The miniport driver sets the **MaxNumPfcEnabledTrafficClasses** member to the maximum number of NDIS QoS traffic classes that the network adapter can use with the Priority-based Flow Control (PFC) algorithm. This value must be less than or equal to the value of the **MaxNumTrafficClasses** member.

   For more information on PFC, see Priority-based Flow Control (PFC).

   Note For the network adapter to support NDIS QoS, it must support at least one PFC-capable traffic class.

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver registers the NDIS QoS attributes of the network adapter by following these steps:

1. The miniport driver initializes an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

The miniport driver sets the **HardwareQOSCapabilities** member to a pointer to the previouslyinitialized NDIS_QOS_CAPABILITIES structure.

If the registry setting for the ***QOS** INF keyword has a value of one, the NDIS QoS capabilities are enabled on the network adapter. The miniport driver sets the **CurrentQOSCapabilities** members to a pointer to the same NDIS_QOS_CAPABILITIES structure.

If the registry setting for the ***QOS** INF keyword has a value of zero, the NDIS QoS capabilities are disabled on the network adapter. The miniport driver must set the **CurrentQOSCapabilities** member to NULL.

2. The driver calls NdisMSetMiniportAttributes and sets the *MiniportAttributes* parameter to a pointer to the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

For more information about the adapter initialization process, see Initializing a Miniport Adapter.

Querying NDIS QoS Capabilities

Article • 03/14/2023

Overlying protocol and filter drivers can query the NDIS Quality of Service (QoS) capabilities of a network adapter in the following way:

- The overlying driver can query the hardware NDIS QoS capabilities supported by the network adapter through an object identifier (OID) query request of OID_QOS_HARDWARE_CAPABILITIES.

- The overlying driver can query the hardware NDIS QoS capabilities that are currently enabled on the network adapter through an OID query request of OID_QOS_CURRENT_CAPABILITIES.

NDIS handles these OID requests for the miniport driver. When the miniport driver registers the hardware and currently enabled NDIS QoS capabilities for the network adapter during network adapter initialization, NDIS caches this information. NDIS then returns this data when it handles the OID requests from an overlying driver.

For more information about how the miniport driver registers the NDIS QoS capabilities, see Registering NDIS QoS Capabilities.

Overview of NDIS QoS Parameters

Article • 03/14/2023

NDIS Quality of Service (QoS) parameters specify the policies and settings of traffic classes that the network adapter uses for transmit, or *egress*, packet delivery. NDIS QoS parameters contain the following settings:

- Priority level and flow control settings. These settings define the IEEE 802.1p priority level and optional flow control algorithms for the transmit, or *egress*, traffic.

  For more information, see Priority Levels and Flow Control.

- Traffic selection algorithm (TSA) settings. These settings define how the network adapter selects egress traffic from its transmit queues. For example, the adapter could use the strict priority TSA and select egress packets based only on IEEE 802.1p priority. The adapter could also use the Enhanced Transmission Selection (ETS) TSA that moderates egress traffic among traffic classes based on their bandwidth allocation.

  For more information, see Transmission Selection Algorithms (TSAs).

- Traffic classifications that specify the assignment of IEEE 802.1p priority levels to packets that contain data that matches a classification condition, such as an EtherType or destination TCP port. For more information, see NDIS QoS Traffic Classifications.

  Note Traffic classifications are also known as "application priorities" in the IEEE 802.1 specifications.

NDIS QoS defines the following types of parameters:

Local NDIS QoS Parameters
Local NDIS QoS parameters specify the core QoS settings for a miniport driver and its network adapter. These parameters persist in the system registry, and are administered locally to the miniport driver in the following way:

- Through an NDIS object identifier (OID) method request of OID_QOS_PARAMETERS that is issued by the DCB component. This OID request contains an **NDIS_QOS_PARAMETERS** structure that specifies the local NDIS QoS parameters.

  For more information on the DCB component, see NDIS QoS Architecture for Data Center Bridging.

- Through proprietary registry settings for the network adapter. The miniport driver reads these settings when its *MiniportInitializeEx* function is called by NDIS.

- Through settings issued to the miniport driver through a management application developed by the independent hardware vendor (IHV).

For more information about how the miniport driver obtains its local NDIS QoS parameters, see Setting Local NDIS QoS Parameters.

Remote NDIS QoS Parameters
Remote NDIS QoS parameters are those that are configured on a remote peer that the network adapter is connected to over the data link. The miniport driver discovers these parameters through the Data Center Bridging Exchange (DCBX) protocol that is specified by the IEEE 802.1Qaz draft standard.

DCBX requires that the miniport driver maintain only one set of remote QoS parameters that were received from a single data-link peer. The miniport driver must issue an NDIS status indication when its remote QoS parameters are either received from a peer for the first time or change later. For example, the driver may change its remote NDIS QoS parameters because it received a different set of QoS parameters from its remote peer. For more information on this process, see Indicating Changes to the Remote NDIS QoS Parameters.

For more information about how the miniport driver obtains its remote NDIS QoS parameters, see Receiving Remote NDIS QoS Parameters.

Operational NDIS QoS Parameters
Operational NDIS QoS parameters are those that the miniport driver resolves for traffic prioritization over the data-link connection to a remote peer. The miniport driver resolves its operational NDIS QoS parameters from its local or remote NDIS QoS parameters.

The miniport driver must issue an NDIS status indication when its operational QoS parameters are either resolved for the first time or change later. For example, the driver may change its operational NDIS QoS parameters because it received a different set of QoS parameters from its remote peer. For more information on how to generate this status indication, see Indicating Changes to the Operational NDIS QoS Parameters.

For more information about how the miniport driver resolves its operational NDIS QoS parameters, see Resolving Operational NDIS QoS Parameters.

Managing the Local DCBX Willing State

Article • 12/15/2021

The IEEE 802.1Qaz draft standard defines the Data Center Bridging Exchange (DCBX) protocol. This protocol allows DCB configuration parameters to be exchanged between the network adapter (local peer) and a directly connected remote peer. This allows these peers to adapt and tune Quality of Service (QoS) parameters to optimize data transfer over the connection.

Based on the local and remote QoS parameter settings, the miniport driver resolves the conflicts and derives a set of operational QoS parameters. The network adapter uses these operational parameters for the prioritized transmission of packets to the remote peer. For more information about how the driver resolves its operational NDIS QoS parameter settings, see Resolving Operational NDIS QoS Parameters.

DCBX consists of DCB type-length-value (TLV) settings that are carried over Link Layer Discovery Protocol (LLDP) packets. A separate TLV is defined for the following types of QoS parameters:

- Enhanced Transmission Selection (ETS)
- Priority-based Flow Control (PFC)

The TLVs for ETS and PFC define a bit known as the *Willing* bit. If the network adapter sends its TLV settings to the remote peer with the Willing bit set to one, it indicates that the adapter is willing to accept QoS parameters from the remote peer.

The ability to set individual Willing bits in these TLVs depends on the local DCBX Willing state that is managed by the miniport driver. The miniport driver must follow these guidelines for managing the local DCBX Willing state:

- If the local DCBX Willing state is disabled, the local Willing bit must be set to zero in the DCBX TLVs. In this case, the operational QoS parameters are always resolved from the local QoS parameters. For more information on these parameters, see Setting Local NDIS QoS Parameters.

- If the local DCBX Willing state is enabled, the local Willing bit must be set to one in the DCBX TLVs. In this case, the operational QoS parameters must be resolved from the remote QoS parameters. For more information on these parameters, see Receiving Remote NDIS QoS Parameters.

  Note If local DCBX Willing state is enabled, the miniport driver can also resolve its operational QoS parameters based on any proprietary QoS settings that are defined by the independent hardware vendor (IHV). The driver can only do this for QoS parameters that are not configured remotely by the peer or locally by the operating system.

The miniport driver manages the local DCBX Willing state in the following way:

- When the miniport driver is initialized through a call to its *MiniportInitializeEx* function, it should enable the local DCBX Willing state based on proprietary QoS settings that are defined by the IHV.
- The DCB component (Msdcb.sys) issues an object identifier (OID) method request of OID_QOS_PARAMETERS to configure the local QoS parameters on a network adapter. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for this OID request contains a pointer to an **NDIS_QOS_PARAMETERS** structure.

  If the **NDIS_QOS_PARAMETERS_WILLING** flag is set in the **Flags** member of this structure, the miniport driver enables the DCBX Willing state. If this bit is not set, the miniport driver disabled the DCBX Willing state.

For more information about LLDP, refer to the IEEE 802.1AB-2005 standard.

For more information about the local DCBX Willing bits and TLVs, refer to the IEEE 802.1Qaz draft standard.

Note Starting with Windows Server 2012, the DCB component can be configured through a PowerShell cmdlet to set or clear the **NDIS_QOS_PARAMETERS_WILLING** flag when it issues an OID_QOS_PARAMETERS request. This causes the miniport driver to respectively enable or disable the local DCBX Willing state.

Setting Local NDIS QoS Parameters

Article • 03/14/2023

Local NDIS Quality of Service (QoS) parameters specify the locally provisioned QoS settings for a miniport driver and its network adapter. Miniport drivers obtain the local NDIS QoS parameters in the following ways:

- Through an object identifier (OID) method request of OID_QOS_PARAMETERS that is issued by the Data Center Bridging (DCB) component (Msdcb.sys). This OID request contains an **NDIS_QOS_PARAMETERS** structure that specifies the local NDIS QoS parameters.

  For more information on the DCB component, see NDIS QoS Architecture for Data Center Bridging.

  Note Starting with Windows Server 2012, the DCB component is installed and enabled with the Microsoft Data Center Bridging (DCB) server feature. This feature is not installed by default.

- Through proprietary settings that are stored in the system registry and defined by the independent hardware vendor (IHV) for the network adapter. The miniport driver reads these settings when its *MiniportInitializeEx* function is called by NDIS.

- Through proprietary settings issued to the miniport driver through a management application developed by the IHV.

When the DCB component issues an OID method request of OID_QOS_PARAMETERS, the **NDIS_QOS_PARAMETERS_WILLING** flag of the **NDIS_QOS_PARAMETERS.Flags** member specifies how the miniport driver resolves its operational QoS parameters from the local NDIS QoS parameters. Based on this flag, the driver resolves the local QoS parameters in the following ways:

- If the **NDIS_QOS_PARAMETERS_WILLING** flag is set, the miniport driver must enable the local DCB Exchange (DCBX) Willing state. This allows the driver to be remotely configured with QoS parameters. In this case, the driver resolves its operational QoS parameters based on the remote QoS parameters.

  The miniport driver can also resolve its operational QoS parameters based on any proprietary QoS settings that are defined by the IHV. The driver can only do this for QoS parameters that are not configured remotely by the peer or locally by the operating system.

For more information about this procedure, see Receiving Remote NDIS QoS Parameters.

- If the **NDIS_QOS_PARAMETERS_WILLING** flag is not set, the miniport driver must disable the local DCBX Willing state. This allows the driver to resolve its operational QoS parameters from its local QoS parameters instead of remote QoS parameters.

  Note If the local DCBX Willing state is disabled, the miniport driver can still accept the remote QoS parameters but cannot use them to resolve its operational QoS parameters.

If the local DCBX Willing state is disabled, the miniport driver must follow these guidelines when it manages its local QoS parameters:

- The miniport driver must disable or override any local QoS parameter for which the related **NDIS_QOS_PARAMETERS_*Xxx*_CONFIGURED** flag is not set in the **NDIS_QOS_PARAMETERS.Flags** member.

  For example, the miniport driver can override an unconfigured local QoS parameter with its proprietary settings for the QoS parameter that are defined by the IHV. If there are no proprietary settings for local QoS parameters that are not specified with an **NDIS_QOS_PARAMETERS_*Xxx*_CONFIGURED** flag, the driver must disable the use of these QoS parameters on the network adapter.

  Note NDIS guarantees that both the **NDIS_QOS_PARAMETERS_ETS_CONFIGURED** and **NDIS_QOS_PARAMETERS_PFC_CONFIGURED** flags are set or cleared together.

- The miniport driver should *apply* the local QoS parameters that are contained in the **NDIS_QOS_PARAMETERS** structure when it resolves its operational NDIS QoS parameters. If the driver applies these local QoS parameters, it must not use any remote QoS parameters that it received from the remote peer.

  For more information on this procedure, see Resolving Operational NDIS QoS Parameters.

For more information about the local DCBX Willing state, see Managing the Local DCBX Willing State.

Receiving Remote NDIS QoS Parameters

Article • 03/14/2023

Remote NDIS Quality of Service (QoS) parameters are those that are received from a remote peer that the network adapter is connected to over the data link. The miniport driver discovers these parameters through the Data Center Bridging Exchange (DCBX) protocol that is specified by the IEEE 802.1Qaz draft standard.

The driver must follow these guidelines for managing remote QoS parameters:

- The miniport driver must issue an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication when its remote NDIS QoS parameters are either received from a peer for the first time or change later. For more information on this procedure, see Indicating Changes to the Remote NDIS QoS Parameters.

- The miniport driver can use the remote NDIS QoS parameters to resolve its operational NDIS QoS parameters only if the local Data Center Bridging Exchange (DCBX) Willing state is enabled on the network adapter. The miniport driver can also resolve its operational QoS parameters based on any proprietary QoS settings that are defined by the independent hardware vendor (IHV).

  For more information about this procedure, see Resolving Operational NDIS QoS Parameters.

  For more information about the local DCBX Willing state, see Managing the Local DCBX Willing State.

For more information about NDIS QoS parameters, see Overview of NDIS QoS Parameters.

Resolving Operational NDIS QoS Parameters

Article • 03/14/2023

Operational NDIS Quality of Service (QoS) parameters are those that the miniport driver uses for traffic prioritization over the data-link connection to a remote peer. The miniport driver derives, or *resolves*, its operational NDIS QoS parameters from one of the following:

- The local NDIS QoS parameters that are administered locally on the miniport driver. For more information about how the miniport driver obtains its local QoS parameters, see Setting Local NDIS QoS Parameters.

- The remote NDIS QoS parameters that the miniport driver receives from a remote peer on the data link. For more information about how the miniport driver obtains its remote QoS parameters, see Receiving Remote NDIS QoS Parameters.

Local, remote, and operational NDIS QoS parameters consist of the following types of data:

- Priority level and flow control settings. These settings define the IEEE 802.1p priority level and optional flow control algorithms for the transmit, or *egress*, traffic.

  For more information, see Priority Levels and Flow Control.

- Traffic selection algorithm (TSA) settings. These settings define how the network adapter selects egress traffic from its transmit queues.

  For example, the adapter could use the strict priority TSA and select egress packets based only on IEEE 802.1p priority. The adapter could also use the Enhanced Transmission Selection (ETS) TSA that moderates egress traffic among traffic classes based on their bandwidth allocation.

  For more information, see Transmission Selection Algorithms (TSAs).

- Traffic classifications that specify the assignment of IEEE 802.1p priority levels to packets that contain data which matches a classification condition, such as an EtherType or destination TCP port. For more information, see NDIS QoS Traffic Classifications.

  Note Traffic classifications are also known as "application priorities" in the IEEE 802.1 specifications.

The miniport driver resolves its operational settings from its local or remote NDIS QoS parameters by following these guidelines:

- If the local Data Center Bridging Exchange (DCBX) Willing state is enabled, the miniport driver must resolve its operational QoS parameters from its remote QoS parameters.

  For more information about how the miniport driver handles remote NDIS QoS parameters, see Receiving Remote NDIS QoS Parameters.

- If the local DCBX Willing state is disabled, the miniport driver must resolve its operational QoS parameters from its local QoS parameters.

  For more information about how the miniport driver handles local NDIS QoS parameters, see Setting Local NDIS QoS Parameters.

- The miniport driver can also resolve its operational QoS parameters based on any proprietary QoS settings that are defined by the independent hardware vendor (IHV). The driver can only do this for QoS parameters that are not configured remotely by the peer or locally by the operating system.

- The miniport driver must issue an NDIS status indication when its operational QoS parameters are either resolved for the first time or change later. For example, the driver may change its operational NDIS QoS parameters because it received a different set of QoS parameters from its remote peer. For more information on how to generate this status indication, see Indicating Changes to the Operational NDIS QoS Parameters.

For more information about the local DCBX Willing state, see Managing the Local DCBX Willing State.

For more information on the methods used to resolve QoS operational parameters, refer to the IEEE 802.1Qaz draft standard.

Querying NDIS QoS Parameters

Article • 03/14/2023

Overlying protocol and filter drivers can query the NDIS Quality of Service (QoS) parameters of a network adapter in the following ways:

- The overlying driver can query the operational NDIS QoS parameters through an object identifier (OID) query request of OID_QOS_OPERATIONAL_PARAMETERS.
- The overlying driver can query the remote NDIS QoS parameters through an OID query request of OID_QOS_REMOTE_PARAMETERS.

Note Overlying drivers cannot query the local NDIS QoS parameters.

For more information about local, remote, and operational NDIS QoS parameters, see Overview of NDIS QoS Parameters.

NDIS handles these OID requests for the miniport driver and returns the requested QoS parameters within an **NDIS_QOS_PARAMETERS** structure. NDIS handles these OID requests in the following ways:

- When NDIS handles the OID query request of OID_QOS_OPERATIONAL_PARAMETERS, it returns the operational NDIS QoS parameters that it had cached from the previous **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication that was issued by the miniport driver. The driver issues this status indication when its operational QoS parameters are either resolved for the first time or change later.

  If the overlying driver issues the OID query request before the miniport driver issues the **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication, NDIS returns an **NDIS_QOS_PARAMETERS** structure with all the members (with the exception of the **Header** member) set to zero.

  Note NDIS also returns this structure if the miniport driver issues an **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication with an **NDIS_QOS_PARAMETERS** structure whose members (with the exception of the **Header** member) are set to zero.

- When NDIS handles the OID query request of OID_QOS_REMOTE_PARAMETERS, it returns the remote NDIS QoS parameters that it had cached from the previous **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication that was issued by the miniport driver. The driver issues this status indication when its remote QoS parameters are either resolved for the first time or change later.

If the overlying driver issues the OID query request before the miniport driver issues the NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication, NDIS returns an NDIS_QOS_PARAMETERS structure with all the members (with the exception of the **Header** member) set to zero.

Note NDIS also returns this structure if the miniport driver issues an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication with an NDIS_QOS_PARAMETERS structure whose members (with the exception of the **Header** member) are set to zero.

Indicating NDIS QoS Parameter Status

Article • 03/14/2023

A miniport driver that supports NDIS Quality of Service (QoS) for the IEEE 802.1 Data Center Bridging (DCB) interface must issue NDIS status indications whenever one of the following events occurs:

- The driver's operational NDIS QoS parameters are either resolved for the first time or change later.

  For more information about how to issue this type of NDIS status indication, see Indicating Changes to the Operational NDIS QoS Parameters.

- The driver's remote NDIS QoS parameters are either received from a data-link peer for the first time or change later.

  For more information about how to issue this type of NDIS status indication, see Indicating Changes to the Remote NDIS QoS Parameters.

For more information about operational and remote NDIS QoS parameters, see Overview of NDIS QoS Parameters.

Indicating Changes to the Operational NDIS QoS Parameters

Article • 03/14/2023

The miniport driver that supports NDIS Quality of Service (QoS) issues an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication when the driver's operational NDIS QoS parameters are resolved for the first time or when they change later. The miniport driver configures the network adapter with these operational parameters to perform QoS packet transmission.

The miniport driver must follow these guidelines for issuing an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication:

- The miniport driver must issue an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication after it has resolved its operational NDIS QoS parameters and configured the network adapter with them.

  Note If the miniport driver is provisioned with proprietary local NDIS QoS parameters in the registry, the driver must issue an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication during or immediately after the call to *MiniportInitializeEx*. In this case, the driver initializes an NDIS_QOS_PARAMETERS structure with its proprietary local NDIS QoS parameter settings.

  For more information about how the driver resolves its operational NDIS QoS parameter settings, see Resolving Operational NDIS QoS Parameters.

- After this initial status indication, the miniport driver should issue an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication when its operational NDIS QoS parameters are changed. For example, the operational NDIS QoS parameters could change under the following conditions:

  - The operational NDIS QoS parameters change because of changes to the local NDIS QoS parameters. These parameters could change through an OID method request of OID_QOS_PARAMETERS or through a management application developed by the independent hardware vendor (IHV).

  - The operational NDIS QoS parameters change because of conflicts with the QoS settings from the remote peer.

    The miniport driver uses the IEEE 802.1Qaz Data Center Bridging Exchange (DCBX) protocol to discover the QoS parameters for a remote peer. If the DCBX Willing state is enabled, the driver must resolve the differences between its QoS parameters and the remote peer's QoS parameters by following the procedures that are defined for the DCBX state engine. For more information about this state engine, refer to the IEEE 802.1Qaz draft standard.

    For more information about the local DCBX Willing state, see Managing the Local DCBX Willing State.

Note When the miniport driver receives local or remote NDIS QoS parameters, it should not issue an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication if there have been no changes to the operational NDIS QoS parameters. If the driver makes this unnecessary status indication, NDIS may not pass the indication to overlying drivers.

- The miniport driver should issue an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication when it needs to override the local NDIS QoS parameters that were used to resolve the operational NDIS QoS parameters.

  The miniport driver notifies NDIS and the overlying driver that it has overridden the local NDIS QoS parameters by issuing an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication. For this type of indication, the driver must set the appropriate **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags in the **Flags** member of the NDIS_QOS_PARAMETERS structure to specify the reason for overriding the local NDIS QoS parameters.

  For more information on how the miniport driver manages the local QoS parameters, see Setting Local NDIS QoS Parameters.

  For more information on how the miniport driver resolves its operational QoS parameters, see Resolving Operational NDIS QoS Parameters.

Note The miniport driver must issue NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indications if its NDIS QoS capabilities are currently enabled through the ***QOS** keyword standardized INF keyword. For more information, see Standardized INF Keywords for NDIS QoS.

Guidelines for Issuing the NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE Status Indication

The miniport driver follows these steps when it issues the NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication:

1. The miniport driver allocates a buffer that is large enough to contain the following:

   - An NDIS_QOS_PARAMETERS structure that contains the NDIS QoS configuration settings as well as global operational parameters for the NDIS QoS traffic classes.

   - An array of NDIS_QOS_CLASSIFICATION_ELEMENT structures. Each of these structures specifies a traffic classification as defined by a packet data pattern (*condition*) and associated IEEE 802.1p priority level (*action*). If the network adapter finds a pattern in the transmit, or *egress*, packet that matches a condition, it assigns the associated priority level to the packet. The adapter also applies the other NDIS QoS policies to the packet based on the priority level.

2. The miniport initializes the NDIS_QOS_PARAMETERS structure with the operational NDIS QoS parameters. The driver must provide the complete set of operational parameters, including those parameters that may not be configured on the network adapter.

When the miniport driver initializes the **Header** member, it sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_QOS_PARAMETERS. The miniport driver sets the **Revision** member of **Header** to NDIS_QOS_PARAMETERS_REVISION_1 and the **Size** member to NDIS_SIZEOF_QOS_PARAMETERS_REVISION_1.

The miniport driver sets the appropriate **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags in the **Flags** member if the corresponding members contain data that has changed since the miniport driverissued an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication.

Note Setting the **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags is optional. NDIS always assumes that the members of the NDIS_QOS_PARAMETERS are current even if they have not changed from the previous NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication.

For more information on how to set the **Flags** member, see Guidelines for Setting the **Flags** Member.

3. The miniport driver initializes an NDIS_QOS_CLASSIFICATION_ELEMENT structure for each traffic classification from the operational NDIS QoS parameters. The driver adds these elements at the end of the NDIS_QOS_PARAMETERS structure in the buffer.

   Note The miniport driver must not set the NDIS_QOS_CLASSIFICATION_ENFORCED_BY_MINIPORT flag in the **Flags** member of any NDIS_QOS_CLASSIFICATION_ELEMENT structures.

   The driver sets the **NumClassificationElements** member of the NDIS_QOS_PARAMETERS structure to the number of classification elements in the array. The driver sets the **FirstClassificationElementOffset** member to the byte offset of the first element from the start of the buffer. The driver also sets the **ClassificationElementSize** member to the length, in bytes, of each element in the array.

   Note Starting with NDIS 6.30, the miniport driver must set the **ClassificationElementSize** member to `sizeof(NDIS_QOS_CLASSIFICATION_ELEMENT)`.

4. The miniport driver initializes an NDIS_STATUS_INDICATION structure for the status indication in the following way:

   - The **StatusCode** member must be set to NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE.

   - The **StatusBuffer** member must be set to the pointer to the buffer that contains the operational NDIS QoS parameters.

   - The **StatusBufferSize** member must be set to the length, in bytes, of the buffer.

5. The miniport driver issues the status indication by calling NdisMIndicateStatusEx. The driver must pass a pointer to the NDIS_STATUS_INDICATION structure to the *StatusIndication* parameter.

Guidelines for Setting the Flags Member

The miniport driver sets the following flags in the **Flags** member of the NDIS_QOS_PARAMETERS structure to specify which operational NDIS QoS parameters have been configured or changed on the network adapter:

NDIS_QOS_PARAMETERS_ETS_CONFIGURED
If this flag is set, the miniport driver has configured the network adapter with the ETS parameters contained in the following members:

- **NumTrafficClasses**
- **PriorityAssignmentTable**
- **TcBandwidthAssignmentTable**
- **TsaAssignmentTable**

Note The miniport driver must support ETS in order to support NDIS QoS for DCB. However, the setting of this flag does not specify whether the network adapter supports ETS. Instead, the setting of this flag specifies only whether ETS parameters are configured on the network adapter.

NDIS_QOS_PARAMETERS_ETS_CHANGED
If this flag is set, one or more ETS parameters have changed in the following members:

- **NumTrafficClasses**
- **PriorityAssignmentTable**
- **TcBandwidthAssignmentTable**
- **TsaAssignmentTable**

NDIS_QOS_PARAMETERS_PFC_CONFIGURED
If this flag is set, the miniport driver has configured the network adapter with the PFC settings contained in the **PfcEnable** member.

Note The miniport driver must support PFC in order to support NDIS QoS for DCB. The setting of this flag does not specify whether the network adapter supports PFC. Instead, the setting of this flag specifies only whether PFC parameters are enabled on the network adapter.

NDIS_QOS_PARAMETERS_PFC_CHANGED
If this flag is set, one or more PFC settings have changed in the **PfcEnable** member.

NDIS_QOS_PARAMETERS_CLASSIFICATION_CONFIGURED
If this flag is set, the miniport driver has configured the network adapter with the QoS traffic classifications parameters specified in the following members:

- **NumClassificationElements**
- **ClassificationElementSize**
- **FirstClassificationElementOffset**

NDIS_QOS_PARAMETERS_CLASSIFICATION_CHANGED
If this flag is set, one or more QoS traffic classification parameters have changed in the following members:

- **NumClassificationElements**
- **ClassificationElementSize**
- **FirstClassificationElementOffset**

Note The **NDIS_QOS_PARAMETERS_*Xxx*_CONFIGURED** flags must be set if the NDIS_QOS_PARAMETERS structure contains NDIS QoS parameter settings. The miniport driver must set these flags regardless of whether the settings have changed. However, the driver must set the **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags only for those settings that have changed.

Indicating Changes to the Remote NDIS QoS Parameters

Article • 03/14/2023

The miniport driver that supports NDIS Quality of Service (QoS) issues an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication when its remote NDIS QoS parameters are either received from a peer for the first time or change later. The miniport driver receives these QoS parameters from a remote peer through the IEEE 802.1Qaz Data Center Bridging Exchange (DCBX) protocol.

The miniport driver must follow these guidelines for issuing an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication:

- If the miniport driver has not received a DCBX frame from a remote peer, it must not issue an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication.

- The miniport driver must issue an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication after it has first received the QoS settings from a remote peer.

  Note The miniport driver must issue this status indication if the network adapter receives remote QoS parameter settings from a peer before the driver's local QoS parameters are set. For more information, see Setting Local NDIS QoS Parameters.

- After this initial status indication, the miniport driver should only issue an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication when it determines a change in the QoS settings on the remote peer.

  Note Miniport drivers should not issue an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication if there have been no changes to the remote NDIS QoS parameters. If the driver does make this type of status indication, NDIS may not pass the indication to overlying drivers.

Note The miniport driver must issue NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indications if its NDIS QoS capabilities are currently enabled. Starting with Windows Server 2012, these indications allow system administrators to view NDIS QoS and Data Center Bridging (DCB) settings regardless of whether the Microsoft DCB server feature is installed.

Guidelines for Issuing the NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE

Status Indication

The miniport driver follows these steps when it issues the NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication:

1. The miniport driver allocates a buffer that is large enough to contain the following:

    - An NDIS_QOS_PARAMETERS structure that contains the NDIS QoS configuration settings as well as global operational parameters for the NDIS QoS traffic classes.

    - An array of NDIS_QOS_CLASSIFICATION_ELEMENT structures. Each of these structures specifies a traffic classification as defined by a packet data pattern (*condition*) and associated IEEE 802.1p priority level (*action*). If the network adapter finds a pattern in the transmit, or *egress*, packet that matches a condition, it assigns the associated priority level to the packet. The adapter also applies the other NDIS QoS policies to the packet based on the priority level.

2. The miniport initializes the NDIS_QOS_PARAMETERS structure with the remote NDIS QoS parameters. The driver must provide the complete set of remote parameters that were received from the DCBX frame sent by the remote peer.

    When the miniport driver initializes the **Header** member, it sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_QOS_PARAMETERS. The miniport driver sets the **Revision** member of **Header** to NDIS_QOS_PARAMETERS_REVISION_1 and the **Size** member to NDIS_SIZEOF_QOS_PARAMETERS_REVISION_1.

    The miniport driver sets the appropriate **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags if the corresponding members contain data that has changed since the miniport driver previously issued an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication.

    Note Setting these **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags is optional. NDIS always assumes that the members of the NDIS_QOS_PARAMETERS are specified even if they have not changed from the previous NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication.

    The miniport driver sets the **Flags** member to specify status information for the data that is contained in the NDIS_QOS_PARAMETERS structure members.

    For example, the miniport driver sets the appropriate **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags in the **Flags** member for those members which contain data that has changed since the miniport driver previously issued an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication.

For more information on how to set the **Flags** member, see Guidelines for Setting the **Flags** Member.

3. The miniport driver initializes an NDIS_QOS_CLASSIFICATION_ELEMENT structure for each traffic classification from the remote NDIS QoS parameters. The driver adds these elements past the end of the NDIS_QOS_PARAMETERS structure in the buffer.

   Note The miniport driver must not set the NDIS_QOS_CLASSIFICATION_ENFORCED_BY_MINIPORT flag in the **Flags** member of any NDIS_QOS_CLASSIFICATION_ELEMENT structures.

   The driver sets the **NumClassificationElements** member of the NDIS_QOS_PARAMETERS structure to the number of classification elements in the array. The driver sets the **FirstClassificationElementOffset** member to the byte offset of the first element from the start of the buffer. The driver also sets the **ClassificationElementSize** member to the length, in bytes, of each element in the array.

   Note Starting with NDIS 6.30, the miniport driver must set the **ClassificationElementSize** member to `sizeof(NDIS_QOS_CLASSIFICATION_ELEMENT)`.

4. The miniport driver initializes an NDIS_STATUS_INDICATION structure for the status indication in the following way:

   - The **StatusCode** member must be set to NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE.

   - The **StatusBuffer** member must be set to the pointer to the buffer that contains the remote NDIS QoS parameters.

   - The **StatusBufferSize** member must be set to the length, in bytes, of the buffer.

5. The miniport driver issues the status indication by calling **NdisMIndicateStatusEx**. The driver must pass a pointer to the NDIS_STATUS_INDICATION structure to the *StatusIndication* parameter.

Guidelines for Setting the Flags Member

The miniport driver sets the following flags in the **Flags** member of the NDIS_QOS_PARAMETERS structure to specify which operational NDIS QoS parameters have been configured or changed on the network adapter:

NDIS_QOS_PARAMETERS_ETS_CONFIGURED
If this flag is set, the miniport driver has configured the network adapter with the ETS parameters contained in the following members:

- **NumTrafficClasses**
- **PriorityAssignmentTable**
- **TcBandwidthAssignmentTable**
- **TsaAssignmentTable**

Note The miniport driver must support ETS in order to support NDIS QoS for DCB. However, the setting of this flag does not specify whether the network adapter supports ETS. Instead, the setting of this flag specifies only whether ETS parameters are configured on the network adapter.

NDIS_QOS_PARAMETERS_ETS_CHANGED
If this flag is set, one or more ETS parameters have changed in the following members:

- **NumTrafficClasses**
- **PriorityAssignmentTable**
- **TcBandwidthAssignmentTable**
- **TsaAssignmentTable**

NDIS_QOS_PARAMETERS_PFC_CONFIGURED
If this flag is set, the miniport driver has configured the network adapter with the PFC settings contained in the **PfcEnable** member.

Note The miniport driver must support PFC in order to support NDIS QoS for DCB. The setting of this flag does not specify whether the network adapter supports PFC. Instead, the setting of this flag specifies only whether PFC parameters are enabled on the network adapter.

NDIS_QOS_PARAMETERS_PFC_CHANGED
If this flag is set, one or more PFC settings have changed in the **PfcEnable** member.

NDIS_QOS_PARAMETERS_CLASSIFICATION_CONFIGURED
If this flag is set, the miniport driver has configured the network adapter with the QoS traffic classifications parameters specified in the following members:

- **NumClassificationElements**
- **ClassificationElementSize**
- **FirstClassificationElementOffset**

NDIS_QOS_PARAMETERS_CLASSIFICATION_CHANGED
If this flag is set, one or more QoS traffic classification parameters have changed in the

following members:

- **NumClassificationElements**
- **ClassificationElementSize**
- **FirstClassificationElementOffset**

Note The **NDIS_QOS_PARAMETERS_*Xxx*_CONFIGURED** flags must be set if the NDIS_QOS_PARAMETERS structure contains NDIS QoS parameter settings. The miniport driver must set these flags regardless of whether the settings have changed. However, the driver must only set the **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags for those settings that have changed.

Guidelines for Indicating Invalid Remote NDIS QoS Parameters

The miniport driver must invalidate its remote QoS parameters if the following conditions are true:

- The time-to-live (TTL) value expires for the remote QoS parameters.

  Note DCBX is carried over the Link Layer Discovery Protocol (LLDP) protocol as specified in the IEEE 802.1AB-2005 standard. LLDP frames always contain a TTL field.

- Another data-link peer sends a DCBX frame before the TTL value expires. This scenario is known as a *multi-peer* condition. DCBX requires that the miniport driver maintain only one set of remote QoS parameters that were received from a single data-link peer.

  When a multi-peer condition occurs, the miniport driver must invalidate all of the remote QoS parameters until the TTL value expires for all of the received DCBX frames.

When the miniport driver invalidates its remote NDIS QoS parameters, it must follow these steps when it issues the NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication:

1. Because the miniport driver is not reporting any valid remote NDIS QoS parameters, it must first fill an NDIS_QOS_PARAMETERS structure with zeros.

   When the miniport driver initializes the **Header** member of this structure, it sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_QOS_PARAMETERS. The miniport driver sets the **Revision** member of **Header** to NDIS_QOS_PARAMETERS_REVISION_1 and the **Size** member to NDIS_SIZEOF_QOS_PARAMETERS_REVISION_1.

The miniport driver sets the appropriate **NDIS_QOS_PARAMETERS_*Xxx*_CHANGED** flags in the **Flags** member.

2. The miniport driver initializes an NDIS_STATUS_INDICATION structure for the status indication in the following way:

   - The **StatusCode** member must be set to NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE.

   - The **StatusBuffer** member must be set to the address of the NDIS_QOS_PARAMETERS structure.

   - The **StatusBufferSize** member must be set to `sizeof(NDIS_QOS_PARAMETERS)`.

3. The miniport driver issues the status indication by calling NdisMIndicateStatusEx. The driver must pass a pointer to the NDIS_STATUS_INDICATION structure to the *StatusIndication* parameter.

Standardized INF Keywords for NDIS QoS

Article • 03/14/2023

A standardized INF keyword is defined to enable or disable support for NDIS Quality of Service (QoS) on a miniport driver.

The INF file for the miniport driver of an adapter that supports NDIS QoS must specify the ***QOS** standardized INF keyword. After the driver is installed, administrators can update the ***QOS** keyword value in the **Advanced** property page for the adapter. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

Note The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter.

The ***QOS** INF keyword is an enumeration keyword. The following table describes the possible INF entries for the ***QOS** INF keyword. The columns in this table describe the following attributes for an enumeration keyword:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params\** key for the network adapter.

ParamDesc
The display text that is associated with SubkeyName.

Note The independent hardware vendor (IHV) can define any descriptive text for SubkeyName.

Value
The enumeration integer value that is associated with each SubkeyName in the list.

EnumDesc
The display text that is associated with each value that appears in the menu.

SubkeyName	ParamDesc	Value	EnumDesc
***QOS**	NDIS QoS	0	QoS Disabled
		1 (Default)	QoS Enabled

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver must do the following:

- The miniport driver must register the NDIS QoS hardware capabilities that the network adapter supports.
- The miniport driver must also read the ***QOS** keyword value in the registry to register the current status of the adapter's NDIS QoS hardware capabilities.

The miniport driver must follow these guidelines when it registers the current status of the NDIS QoS hardware capabilities:

- If the ***QOS** keyword has a value of one, the miniport driver must register all NDIS QoS hardware capabilities as currently enabled. The driver must enable its NDIS QoS hardware capabilities regardless of the following:

    - Whether the Microsoft Data Center Bridging (DCB) server feature is installed or enabled on Windows Server 2012 and later versions of Windows Server. For more information about this server feature and related components, see NDIS QoS Architecture for Data Center Bridging.
    - Whether the local Data Center Bridging Exchange (DCBX) Willing state is enabled on the network adapter. When this state is enabled, the network adapter and miniport driver can resolve its operational NDIS QoS parameters from remote NDIS QoS parameters that were received from the remote peer. For more information, see Managing the Local DCBX Willing State.

    For more information on how to register QoS hardware and current capabilities, see Registering NDIS QoS Capabilities.

    Note The miniport driver must always issue **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** and **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indications if its NDIS QoS hardware capabilities are currently enabled. Starting with Windows Server 2012, these status indications report on the current operational and remote QoS parameter settings, respectively. These indications allow system administrators to view NDIS QoS and DCB settings regardless of whether the Microsoft DCB server feature is installed. For more information, see Indicating NDIS QoS Parameter Status.

- If the ***QOS** keyword has a value of zero, the miniport driver must report all NDIS QoS hardware capabilities as currently disabled. In this case, the operating system will not configure the driver with NDIS QoS capabilities.

Note The driver must disable DCB and DCBX on the network adapter if the ***QOS** keyword has a value of zero.

- If the ***QOS** keyword is not present in the registry, the miniport driver must not report any NDIS QoS hardware capabilities. In this case, the operating system will not configure the driver with NDIS QoS capabilities.

  Note The driver must disable DCB and DCBX on the network adapter if the ***QOS** keyword is not present in the registry.

In addition to the ***QOS** keyword, the miniport driver must read the ***PriorityVLANTag** keyword. This keyword specifies whether the network adapter is enabled to insert the 802.1Q tags for packet priority and virtual LANs (VLANs).

The following table summarizes the relationship between the ***QOS** and ***PriorityVLANTag** keyword values.

QOS keyword setting	**PriorityVLANTag keyword setting**	***PriorityVLANTag setting description**
0 or not present	0	Packet priority & VLAN disabled
0 or not present	1	Packet priority enabled
0 or not present	2	VLAN Enabled
0 or not present	3 (Default)	Packet priority and VLAN enabled
1	0	Packet priority enabled
1	1	Packet priority enabled
1	2	Packet priority and VLAN enabled
1	3 (Default)	Packet priority and VLAN enabled

For more information about the ***PriorityVLANTag** keyword, see Enumeration Keywords.

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

For more information on how to register NDIS QoS capabilities, see Registering NDIS QoS Capabilities.

Initializing NDIS Timers

Article • 03/14/2023

The NDIS_TIMER_CHARACTERISTICS structure defines characteristics of a one-shot or periodic timer. Any NDIS driver can have more than one timer. Each timer object is associated with a different NetTimerCallback function that is specified in the **TimerFunction** member. NDIS calls the associated *NetTimerCallback* function when the timer expires.

To allocate and initialize a timer, your driver should call the NdisAllocateTimerObject function and provide a driver-allocated NDIS_TIMER_CHARACTERISTICS structure. The timer does not start until the driver calls the NdisSetTimerObject function.

To free a timer object, your driver should call the NdisFreeTimerObject function.

Setting and Clearing Timers

Article • 12/15/2021

After allocating and initializing a timer with the NdisAllocateTimerObject function, an NDIS 6.0 driver calls the NdisSetTimerObject function to set a timer object to fire after a specified interval or periodically.

The *DueTime* parameter of **NdisSetTimerObject** specifies the interval to elapse before a timer fires and NDIS calls the associated NetTimerCallback function. The expiration time is expressed in system time units (100-nanosecond intervals).

If the *MillisecondsPeriod* parameter of **NdisSetTimerObject** is not zero, the timer fires periodically and *MillisecondsPeriod* specifies the periodic time interval, in milliseconds, that elapses between each time a periodic timer fires and the next call to the *NetTimerCallback* function.

Your driver can call the NdisCancelTimerObject function to cancel a timer that is associated with a previous call to the **NdisSetTimerObject** function. NDIS might still call *NetTimerCallback* if the timeout has already expired before the call to **NdisCancelTimerObject**.

Servicing Timers

Article • 12/15/2021

NDIS calls the NetTimerCallback function when an NDIS 6.0 timer fires. The *FunctionContext* parameter of this function contains a pointer to a driver-supplied context area. The default value for *FunctionContext* is specified in an NDIS_TIMER_CHARACTERISTICS structure. The driver passed the structure to the NdisAllocateTimerObject function to allocate and initialize the associated timer object.

If the driver specified a non-NULL value in the *FunctionContext* parameter that is passed to the NdisSetTimerObject function, NDIS passes that value to the *FunctionContext* parameter of the *NetTimerCallback* function. Otherwise, NDIS passes the default value that is specified in the NDIS_TIMER_CHARACTERISTICS structure.

Any NDIS driver can have more than one *NetTimerCallback* function. Each such *NetTimerCallback* function must be associated with a different driver-allocated and initialized timer object.

A call to the NdisSetTimerObject function causes the *NetTimerCallback* function that is associated with the timer object to be run after a specified interval or periodically.

To stop calls to a *NetTimerCallback* function, call the NdisCancelTimerObject function. NDIS might still call *NetTimerCallback* if the timeout has already expired before the call to **NdisCancelTimerObject**.

If a *NetTimerCallback* function shares resources with other driver functions, the driver should synchronize access to those resources with a spin lock.

NDIS Configuration Functions

Article • 03/14/2023

NDIS includes the following functions to simplify driver configuration:

NdisOpenConfigurationEx

NdisMGetBusData

NdisMSetBusData

To obtain configuration information for an adapter, an NDIS miniport driver calls **NdisOpenConfigurationEx** and NdisReadConfiguration. The driver can call **NdisMGetBusData** to obtain bus-specific information. The driver can call **NdisMSetBusData** to set bus-specific information.

A protocol driver uses a registry path to an adapter name to read configuration parameters that are specific to the binding between the driver and the underlying adapter. NDIS provides the registry path in the call to the *ProtocolBindAdapterEx* function. The driver can pass this registry path to the NdisOpenProtocolConfiguration function or to direct registry calls. As an alternative, the driver can pass a *BindParameters* structure to the **NdisOpenConfigurationEx** function to read the same parameters.

NDIS Objects

Article • 03/14/2023

Components that do not have an NDIS handle use the **NdisAllocateGenericObject** function to allocate a generic NDIS object. A component must call the **NdisFreeGenericObject** function to free a generic object that was created with **NdisAllocateGenericObject**.

For information about using generic objects, see Obtaining Pool Handles.

NDIS I/O Work Items

Article • 03/14/2023

Drivers can queue I/O work item callback functions for later execution. NDIS calls the driver-specified callback function at IRQL = PASSIVE_LEVEL. This improves system performance by allowing the current function to return immediately and the driver to do work later at a lower IRQL.

NDIS 6.0 and later provide wrapper functions for the kernel I/O work item routines **IoAllocateWorkItem**, **IoFreeWorkItem**, and **IoQueueWorkItem**. Instead of the private **IO_WORKITEM** structure, NDIS uses the private **NDIS_IO_WORKITEM** structure.

NDIS 6.0 drivers call the **NdisAllocateIoWorkItem** function to allocate a work item. NDIS miniport drivers pass **NdisAllocateIoWorkItem** the adapter handle that NDIS passed to the *MiniportInitializeEx* function. **NdisAllocateIoWorkItem** gets the device object associated with the handle and passes the device object to the **IoAllocateWorkItem** routine. Filter drivers pass in a filter handle.

Note Protocol drivers cannot use **NdisAllocateIoWorkItem** because NDIS does not associate protocol drivers with device objects.

NDIS drivers call the **NdisQueueIoWorkItem** function to queue work items. NDIS work items use the **CriticalWorkQueue** queue type.

NDIS drivers must call the **NdisFreeIoWorkItem** function to free the resources associated with a work item that **NdisAllocateIoWorkItem** allocated.

Related topics

System Worker Threads

Components and Files Used for Network Component Installation

Article • 05/30/2023

The following components and files are used to install network drivers:

- One or more information (INF) files
- A required class installer and optional co-installer for miniport drivers
- INetCfg for protocol and filter drivers
- An optional notify object

In addition to one or more of the above components, a vendor also optionally supplies the following files:

- One or more device driver image (.sys) files and driver library (.dll) files
- A driver catalog file
- A text-mode setup information file (txtsetup.oem)

INF files

Each network component must have an information (INF) file that the network class installer uses to install the component. Network INF files are based on the common INF file format. For more information about the INF file format, see INF File Sections and Directives.

For detailed information about creating INF files for network components, see Creating Network INF Files.

Starting with Windows OS build version 25319, you can create a network driver package that can be executed from the Driver Store. An INF that is using 'run from Driver Store' means that the INF uses **DIRID 13** to specify the location for driver package files on install.

You can't install a driver package through the network configuration interfaces and use the driver store feature on older Windows versions. To successfully install the driver package in this scenario, you need to have a minimum OS build number of 25319. For more information, see Manufacturer Section in a Network INF File.

INetCfg

Currently, NDIS protocol and filter drivers are installed by calling into the `INetCfg` family of Network Configuration Interfaces. For example, to install or remove network components, a driver writer calls into the INetCfgClassSetup interface.

Driver writers can either call into this interface programmatically or they can use netcfg.exe, which calls `INetCfg` on their behalf.

For more information about protocol driver installation, see NDIS protocol driver installation.

For more information about filter driver installation, see NDIS Filter Driver Installation.

Notify object

A software component, such as a network protocol, client, or service, can have a *notify object*. A notify object can display a user interface, notify the component of binding events so that the component can exercise some control over the binding process, and conditionally install or remove software components. On older versions of Windows you can't create a driver package with a notify object that is executed from the Driver Store. To successfully install a driver package in this scenario, you need to have a minimum OS build number of 25341. For more information about notify objects, see Notify Objects for Network Components.

A network adapter can't have a notify object. It can have co-installers. For more information about co-installers, see Writing a Co-installer.

Vendor-supplied files

A vendor supplies one or more drivers for the device, which typically consists of a driver image (.sys) file and a driver library (.dll) file. A vendor may also supply an optional driver *catalog file*. A vendor gets a digital signature by submitting its driver package to the Windows Hardware Quality Lab (WHQL) for testing and signing. WHQL returns the package with a catalog (.cat) file. The vendor must list the catalog file in the INF file for the device.

An optional text-mode setup information file (txtsetup.oem) may also be supplied by the vendor. If a network device is required to boot the machine, the driver or drivers for the device must be included in the operating system kit or the vendor of such a device must provide a txtsetup.oem file. The txtsetup.oem file contains information that is used by the system setup components to install the device during text-mode setup.

Creating Network INF Files

Article • 12/15/2021

A *network INF file* is based on the standard INF file format but also includes network-specific items, such as network-specific sections, directives, section entries, and values. The following description of network INF files assumes an understanding of base INF files. Read the description of base INF files before attempting to create a network INF file. For more information about base INF files, see INF File Sections and Directives.

A single INF file can be used to install a network component on various Windows platforms. For more information, see INF File Sections and Directives.

The INF file requirements vary by type of network component. For more information about the INF file requirements for each type of network component, see Summary of Installation Requirements for Network Components.

This section includes:

In this section

Version Section in a Network INF File

Article • 12/15/2021

The **Version** section in a network INF file is based on the generic **INF Version section**.

The **Version** section in a network INF file has the following network-specific entries:

- Class
- ClassGuid
- Signature and Operating System Entries
- PnpLockDown
- CatalogFile
- Version Section Example

Class

The **Version** section should contain a **Class** entry which identifies the class of network component that is installed by the file.

There are four network classes:

Net
Specifies a physical or virtual network adapter. NDIS intermediate drivers, which export virtual network adapters, are included in the Net class.

NetTrans
Specifies a network protocol, such as TCP/IP, IPX, a connection-oriented client, or a connection-oriented call manager.

NetClient
Specifies a network client, such as the Microsoft Client for Networks or the NetWare Client. A NetClient component is considered to be a network provider and, if it provides print services over the network, it is also considered to be a print provider.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

NetService
Specifies a network service, such as a file service or a print service.

Note Infrared Data Association (IrDA) compliant devices are not categorized as any of the previous four network classes, even though they are installed by the network class installer. An INF file that is used to install an IrDA device should have a **Class** value of **Infrared**. This class includes both Serial-IR and Fast-IR devices.

Note Support for IrDA miniport drivers has been removed from NDIS 6.30 (Windows 8) and later.

ClassGuid

The **Version** section must contain a **ClassGuid** entry. The network class installer uses the **ClassGuid** entry to determine the class of network component being installed.

There are four network **ClassGuid** values, each of which corresponds to a network class:

Network class	ClassGuid
Net	{4D36E972-E325-11CE-BFC1-08002BE10318}
NetTrans	{4D36E975-E325-11CE-BFC1-08002BE10318}
NetClient	{4D36E973-E325-11CE-BFC1-08002BE10318}
NetService	{4D36E974-E325-11CE-BFC1-08002BE10318}

An INF file for an IrDA device should have a **ClassGuid** value of

{6bdd1fc5-81d0-bec7-08002be2092f}.

Signature and Operating System Entries

The **Signature** entry must be **$Windows NT$**.

PnpLockDown

The **PnpLockDown** entry should be set to 1 to prevent applications from directly modifying the files that your driver package's INF file specifies. For more information about this entry, see INF Version Section.

CatalogFile

The **CatalogFile** entry is used to declare an optional driver-supplied .cat file. For more information, see the Vendor-supplied files section of Components and Files Used for Network Component Installation.

Version Section Example

The following is an example of a **Version** section for an INF file that installs a network adapter:

INF

```
[Version]
Signature = $Windows NT$
Class=Net
ClassGuid = {4D36E972-E325-11CE-BFC1-08002BE10318}
Provider = %Msft%
DriverVer=06/22/2010,6.1.7065.0
PnpLockDown = 1
CatalogFile = netvmini630.cat
```

Note The **Provider** entry indicates the developer of the INF file, not the developer of the component that is installed by the INF file.

Manufacturer Section in a Network INF File

Article • 05/30/2023

The Manufacturer section in a network INF file is based on the generic **INF Manufacturer section**.

Starting with Windows OS build version 25319, you can create a network driver package that can be executed from the Driver Store. An INF that is using 'run from Driver Store' means that the INF uses **DIRID 13** to specify the location for driver package files on install.

You can't install a driver package through the network configuration interfaces and use the driver store feature on older Windows versions. To successfully install the driver package in this scenario, you need to have a minimum OS build number of 25319.

To use DIRID 13 for installation in newer builds, it's useful to create an INF Manufacturer section that includes multiple *models-section-name* entries that specify target operating system versions. Different **INF *Models* sections** can be specified for different versions of the operating system. The *models-section-name* entries indicate operating system versions with which the INF ***Models*** sections are used.

The following example shows how to create an OS-specific INF Manufacturer section using two *models-section-name* entries. OS builds 25319 and later will use `MyMfg.NT$ARCH$.10.0...25319`. All other builds will use `MyMfg.NT$ARCH$`. This example uses build 25319 because it's the first build that allows for installation using DIRID 13.

INF

```
[Manufacturer]
%ManufacturerName%=Standard,NT$ARCH$,NT$ARCH$.10.0...25319

[Standard.NT$ARCH$.10.0...25319]
%NDISPROT_Desc%=InstallA, MS_NDISPROT

[Standard.NT$ARCH$]
%NDISPROT_Desc%=InstallB, MS_NDISPROT

[InstallA]    ; OS build numbers 25319 and higher
AddReg=Inst_Ndi
Characteristics=0x0 ;
CopyFiles=CpyFiles_Sys_A

[InstallB]    ; OS build numbers lower than 25319
AddReg=Inst_Ndi
```

```
Characteristics=0x0 ;
CopyFiles=CpyFiles_Sys_B
```

For an example of how an OS-specific Manufacturer section can allow for installation using DIRID 13 for new builds and DIRID 12 for older builds, see the Sample NDIS Protocol Driver.

Models Section in a Network INF File

Article • 12/15/2021

The **Models** section in a network INF file is based on the generic **INF Models section**.

The **Models** section in an INF file contains an entry of the following format for each type of component installed by the INF file:

[*device-description*= *install-section.name*, *hw-id*[, *compatible-id*...]

For a detailed description of this entry, see Creating an INF File.

The *hw-id* (also known as the device, hardware, or component ID) for a network adapter must match the hardware ID supplied by the adapter to the PnP manager.

The *hw-id* for a network software component should consist of a provider name, followed by an underscore, and a manufacturer name or the product name, for example:

- MS_DLC
- MS_IBMDLC

A *provider name* identifies the provider of the INF file. A *manufacturer name* identifies the manufacturer of the software component. The *product name* identifies the software component.

DDInstall Section in a Network INF File

Article • 12/15/2021

A *DDInstall* section in a network INF file is based on the generic **INF DDInstall section**.

A *DDInstall* section in a network INF file has the following network-specific entries:

- Characteristics
- BusType
- Port1DeviceNumber and Port1FunctionNumber

Characteristics

Each *DDInstall* section in a network INF file must have a **Characteristics** entry. The **Characteristics** entry specifies certain characteristics of the network component being installed and may limit the user's actions regarding that component. For example, the **Characteristics** entry can specify whether the component supports a user interface, whether it can be removed, or whether it is hidden from the user.

The **Characteristics** entry can have one or more of the following values (multiple values are summed together):

Hex value	Name	Description
0x1	NCF_VIRTUAL	Component is a virtual adapter. The device is not on a physical bus, such as the PCI bus or USB, but is on the root bus. This flag is only applicable to drivers which use the Net device setup class.
0x2	NCF_SOFTWARE_ENUMERATED	Component is a software-enumerated adapter. This flag is only applicable to drivers which use the Net device setup class.

Hex value	Name	Description
0x4	NCF_PHYSICAL	Component is a physical adapter that the driver communicates with directly (for example, through the PCI bus) or indirectly (for example, through USB). Select this option if the driver supports a physical network interface.[1] This flag is only applicable to drivers which use the Net device setup class.
0x8	NCF_HIDDEN	Component should not be shown in any user interface.
0x10	NCF_NO_SERVICE	Component does not have an associated service (device driver).
0x20	NCF_NOT_USER_ REMOVABLE	Component cannot be removed by the user (for example, through Control Panel or Device Manager).
0x80	NCF_HAS_UI	Component supports a user interface (for example, the Advanced Page or a custom properties sheet).
0x400	NCF_FILTER	Component is a Filter Intermediate driver. Filter Intermediate drivers are not supported in Windows 10 or later.
0x4000	NCF_NDIS_PROTOCOL	Component requires the unload event that is provided by the binding engine to the **NetTrans** device setup class (typically used by filter Intermediate drivers which use the **NetService** device setup class).

Hex value	Name	Description
0x40000	NCF_LW_FILTER	Component is a lightweight filter driver. This flag is only applicable to drivers which use the NetService device setup class.

[1] When using Windows Server 2012 R2, at least one network interface on the system must be marked with NCF_PHYSICAL in order to be eligible for DHCPv6 client.

The following combinations of **Characteristics** values are not allowed:

- NCF_VIRTUAL, NCF_SOFTWARE_ENUMERATED, and NCF_PHYSICAL are mutually exclusive.

- NCF_NO_SERVICE cannot be used with NCF_VIRTUAL, NCF_SOFTWARE_ENUMERATED, or NCF_PHYSICAL. A virtual, software-enumerated, or physical adapter must always have an associated service (device driver).

The following is an example of a **Characteristics** entry for a physical adapter that supports a user interface:

INF

```
Characteristics = 0x84; NCF_PHYSICAL, NCF_HAS_UI
```

BusType

A *DDInstall* section for a physical network adapter must contain a **BusType** entry that specifies the type of bus (such as PCI or ISA) on which the adapter can function. The possible values for the **BusType** entry are specified by the INTERFACE_TYPE enumeration in the NDIS header file (ndis.h) as follows:

BusType Entry	Value
ISA	1
EISA	2
MicroChannel	3
TurboChannel	4
PCIBus	5

BusType Entry	Value
VMEbus	6
NuBus	7
PCMCIABus	8
Cbus	9
MPIBus	10
MPSABus	11
PNPISABus	14
PNPBus	15

Note If an adapter can function on more than one type of bus, the INF file that installs that adapter should contain a *DDInstall* section for each bus type.

For example, if an adapter can function on both the ISA bus and the PnPISA bus, the INF file for that adapter should contain a *DDInstall* section for ISA and a *DDInstall* section for PnPISA. The **BusType** entry in each such *DDInstall* section should specify the appropriate bus type for that section as follows:

INF

```
[a1.isa]
BusType=1

[a1.pnpisa]
BusType=14
```

Port1DeviceNumber and Port1FunctionNumber

The *DDInstall* section of an INF file that installs a multiport network adapter must include either a **Port1DeviceNumber** entry or a **Port1FunctionNumber** entry. Specifying such an entry causes the adapter's port information to be displayed in the **Connection Properties** dialog box (which is accessed through the **Network** and **Dial-Up Connections** folder) when you select the adapter name or icon.

- If an adapter's port numbers map sequentially to PCI device numbers, use the **Port1DeviceNumber** entry. Set **Port1DeviceNumber** to the first PCI device number in the sequence. For example, if PCI device number 4 maps to port 1, PCI device number 5 maps to port 2, PCI device number 6 maps to port 3, and so forth, use the following entry:

```INF
Port1DeviceNumber = 4
```

- If an adapter's port numbers map sequentially to PCI function numbers, use the **Port1FunctionNumber** entry. Set **Port1FunctionNumber** to the first PCI function number in the sequence. For example, if PCI function number 2 maps to port 1, PCI function number 3 maps to port 2, PCI function number 4 maps to port 3, and so forth, use the following entry:

```INF
Port1FunctionNumber = 2
```

Note It is assumed that the mapping of PCI device numbers or PCI functions to port numbers is static. It is also assumed that the adapter's ports are numbered sequentially.

The **Port1DeviceNumber** and **Port1FunctionNumber** entries are mutually exclusive. If both entries are present in a given *DDInstall* Section, only the **Port1DeviceNumber** entry is used.

Remove Section in a Network INF File

Article • 12/15/2021

Remove sections are supported for **NetClient**, **NetTrans**, and **NetService** components but not for **Net** components (adapters).

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

The network class installer does not keep track of adapter instances. A **Remove** section that removes files that are shared by other adapters or by multiple instances of an adapter could render those adapters or adapter instances inoperative. If it is necessary to remove a driver file that is used by a **Net** component, use a co-installer that keeps track of all drivers that are using the file. Such a co-installer should also track multiple instances of the same device, as well as drivers for multiple devices. For more information about co-installers, see Creating an INF File.

ControlFlags Section in a Network INF File

Article • 12/15/2021

A **ControlFlags** section in a network INF file is based on the generic **INF ControlFlags section**.

The **ControlFlags** section in a network INF file typically has one or more **ExcludeFromSelect** entries. Each **ExcludeFromSelect** entry specifies a network component that will not be displayed to the end user as an option during a manual installation.

The **ControlFlags** section in a network INF file must contain an **ExcludeFromSelect** entry for each Plug and Play adapter that it installs, and for any software component that should be added programmatically rather than manually by the user.

Adapters that are not compatible with Plug and Play must be added manually by the user and therefore should not be listed in the *ControlFlags* section. For example, non-PnP ISA adapters and EISA adapters must be manually added by the user. Note that Windows XP and later operating systems do not support non-PnP ISA adapters and EISA adapters.

Note An **ExcludeFromSelect** entry performs a different function than does the NCF_HIDDEN value of the **Characteristics** entry in the *DDInstall* section. For more information, see DDInstall Section.

An **ExcludeFromSelect** entry prevents an adapter or software component from being listed in the **Select Component for Installation** dialog box. The adapter or component, however, can still be listed in the **Connections** dialog box. The NCF_HIDDEN value prevents the adapter or component from being displayed in any part of the user interface, including the **Connections** dialog box.

Add-registry-sections in a Network INF File

Article • 12/15/2021

An INF file contains one or more *add-registry-sections* for each component that it installs. An *add-registry-section* adds keys and values to the registry. The **DDInstall** section of an INF file contains an **AddReg** directive that references one or more *add-registry-sections*. For more information about the *add-registry-section* and the **AddReg** directive, see **INF AddReg Directive**.

Adding Keys and Values to Instance Keys

One or more *add-registry-sections* can add keys and values to the instance key for a component to accomplish any of the following:

- Set static parameters for a component -- that is, configuration parameters that cannot be modified through a user interface. For more information, see Setting Static Parameters.

- Specify the number of endpoints (also known as channels, circuits or bearer channels) for a WAN adapter. For more information, see Specifying WAN Endpoints for a WAN Adapter.

- Specify keys and values for an ISDN adapter. For more information, see Specifying ISDN Keys and Values for an ISDN Adapter.

- Require the installation of another network component. For more information, see Requiring the Installation of Another Network Component.

- Specify values that support a custom properties sheet for a network adapter. For more information, see Specifying Custom Property Pages for Network Adapters.

Adding Keys and Values to a NetClient Component

An *add-registry-section* in an INF file for a **NetClient** component must add a **NetworkProvider** key to the *service* key for that component. The **NetworkProvider** key has two values: a **Name** that specifies the name of the network provider, and a **ProviderPath** that specifies the full path to the network provider DLL. For more information, see Specifying the Name and Provider Path for a NetClient Component.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

Creating the Ndi Key

Each network INF file must contain at least one *add-registry-section* that adds an **Ndi** key for the component installed by the file. The **Ndi** key is a network-specific key that is added to the instance key for the component. The keys and values that are added to the **Ndi** key vary according to the type of network component being installed and its capabilities. The **Ndi** key specifies the following:

- **HelpText** value for a **NetTrans**, **NetClient**, or **NetService** component. For more information, see Adding a HelpText Value.

- Values for a notify object. For more information, see Adding Registry Values for a Notify Object.

- Service-related values. For more information, see Adding Service-Related Values to the Ndi Key.

- Binding interfaces. For more information, see Specifying Binding Interfaces.

- Adapter configuration parameters for the **Advanced** page. For more information, see Specifying Configuration Parameters for the Advanced Properties Page.

- Bundle membership. For more information, see Specifying Bundle Membership.

For a list of **Ndi** registry keys and values that are available in Windows 95/98/Me but not used in Windows 2000 and later versions, see Ndi Values and Keys Not Used in Windows 2000 and Later Versions.

Setting Static Parameters

Article • 12/15/2021

An INF file sets a static parameter only once. This parameter cannot be reconfigured through a properties page in the user interface.

An *add-registry-section* adds a static parameter as a REG_SZ value to a component's instance key as shown in the following example:

INF

```
[a1.staticparams.reg]
HKR,, MediaType, 0, "1"
HKR,, InternalId, 0, "232"
```

An *add-registry-section* can add any vendor-defined static parameter to a component's instance key.

Specifying WAN Endpoints for a WAN Adapter

Article • 12/15/2021

A INF file for a WAN adapter must add a **WanEndpoints** value to the instance key for the adapter. **WanEndpoints** is a REG_DWORD value that specifies the number of endpoints (also known as channels, circuits or bearer channels) that are supported by the WAN adapter. For example, the **WanEndpoints** value for a *basic rate interface* (BRI) ISDN adapter is 2, whereas the **WanEndpoints** value for a *primary rate interface* (PRI) ISDN adapter is typically 23.

Note ISDN drivers are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

The following is an example of an *add-registry-section* that adds a **WanEndpoints** value of 2 for a BRI ISDN adapter:

INF

```
[a1.reg]
HKR,, WanEndpoints, 0x00010001, 2
```

Specifying ISDN Keys and Values for an ISDN Adapter

Article • 12/15/2021

In addition to a **WanEndpoints** value, an INF file for an ISDN adapter must add (through an *add-registry-section*) the following keys and values to the instance key for the adapter. For more information, see Specifying WAN Endpoints for a WAN Adapter.

Note ISDN drivers are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

- **IsdnNumDChannels**

  Specifies the number of D-channels that are supported by the ISDN adapter.

- **IsdnAutoSwitchDetect** (Optional)

  Specifies whether the ISDN adapter supports automatic switch detection. A value of 1 indicates that the adapter supports automatic switch detection. A value of zero indicates that the adapter does not support automatic switch detection.

- **IsdnSwitchTypes**

  Specifies the switch types that are supported by the ISDN adapter.

Switch	Description
ISDN_SWITCH_AUTO	Auto Detect (North America only)
ISDN_SWITCH_ATT	ESS5 (AT&T, North America)
ISDN_SWITCH_NI1	National ISDN 1 (NI-1)
ISDN_SWITCH_NI2	National ISDN 2 (NI-2)
ISDN_SWITCH_NT1	Northern Telecom DMS 100 (NT-1)
ISDN_SWITCH_INS64	NTT INS64 (Japan)
ISDN_SWITCH_1TR6	German National (1TR6). This switch type is rarely used.
ISDN_SWITCH_VN3	French National (VN3). This switch type is rarely used.
ISDN_SWITCH_NET3	European ISDN (DSS1)
ISDN_SWITCH_DSS1	European ISDN (DSS1)

Switch	Description
ISDN_SWITCH_AUS	Australian National. This switch type is rarely used.
ISDN_SWITCH_BEL	Belgium National. This switch type is rarely used.
ISDN_SWITCH_VN4	French National (VN4)
ISDN_SWITCH_SWE	Swedish National
ISDN_SWITCH_ITA	Italian National
ISDN_SWITCH_TWN	Taiwanese

To specify multiple switch types, simply add the individual switch type values together.

The ISDN Wizard, which runs automatically during the installation of an ISDN component, allows the user to select one of the switch types specified by **IsdnSwitchTypes**. The selected switch type determines which other ISDN parameters the ISDN Wizard subsequently displays for configuration. These ISDN parameters include the phone number, the SPID (service profile identifier), the subaddress, and the multisubscriber number.

- An **IsdnNumBChannels** value and a *D-channel* key for each D-channel

  The *D-channel* key is an zero-based index from 0 through 9 that identifies the D-channel. **IsdnNumBChannels** is a REG_DWORD value added to the *D-channel* key. **IsdnNumBChannels** specifies the number of B-channels supported by the D-channel.

The following is an example of an *add-registry-section* that adds ISDN keys and values to the instance key of an ISDN adapter. Two D-channels are specified for the adapter, and two B-channels are specified for each D-channel.

INF

```
[ISDNadapter.reg]
HKR,,  WanEndPoints,          0x00010001, 4
HKR,,  IsdnNumDChannels,      0x00010001, 2
HKR,,  IsdnAutoSwitchDetect, 0x00010001, 1
HKR,,  IsdnSwitchTypes,       0x00010001, 0x00000004  ;NI1

HKR, 0, IsdnNumBChannels,     0x00010001, 2

HKR, 1, IsdnNumBChannels,     0x00010001, 2
```

The ISDN Wizard itself also adds ISDN keys and values to the instance key for an ISDN adapter, based on the parameter values specified by the user. The ISDN Wizard adds the following keys and values:

- **IsdnSwitchType**

  A REG_DWORD that indicates the switch type that was selected by the user for the ISDN adapter.

- **IsdnMultiSubscriberNumbers** value for each D-channel

  A REG_MULTI_SZ value that indicates the multisubscriber numbers that were specified by the user for the D-channel.

- A *B-channel* key and an **IsdnSpid**, **IsdnPhoneNumber**, and/or an **IsdnSubaddress** value for each B-channel:

Key or Value	Description
B-channel key	A zero-based index that identifies the B-channel. The maximum value for a *B-channel* key is one less than the **IsdnNumBchannels** value assigned to the D-channel to which the B-channel belongs.
IsdnSpid	A REG_SZ value that indicates the SPID, if any, specified by the user for the B-channel.
IsdnPhoneNumber	The phone number, if any, specified by the user for the B-channel.
IsdnSubaddress	The subaddress, if any, specified by the user for the B-channel.

The following example is an ISDN adapter's registry section layout . Each registry key is enclosed in square brackets, for example: [*KeyName*]. The ISDN keys and values that were added by the INF file for the ISDN adapter are highlighted in boldface text; the ISDN keys and values that were added by the ISDN Wizard appear in normal (nonboldface) text.

INF

```
[...Enum\emumeratorID\device-instance-id]  ;ISDN adapter instance key
WanEndpoints=4
IsdnNumDChannels=2
IsdnAutoSwitchDetect=1
IsdnSwitchType=0x4  ;National ISDN 1
```

```
[...Enum\emumeratorID\device-instance-id\0]  ;D-channel 0
IsdnNumBChannels=2
IsdnMultiSubscriberNumbers=1234567 2345678 3456789

[...Enum\emumeratorID\device-instance-id\0\0]  ;B-channel 0 for D-channel 0
IsdnSpid=00555121200
IsdnPhoneNumber=5551212
IsdnSubaddress=

[...Enum\emumeratorID\device-instance-id\0\1]  ;B-channel 1 key for D-
channel 0
IsdnSpid=00555121300
IsdnPhoneNumber=5551213
IsdnSubaddress=

[...Enum\emumeratorID\device-instance-id\1]  ;D-channel 1 key
IsdnNumBChannels=2
IsdnMultiSubscriberNumbers=8675309 2390125 7658156

[...Enum\emumeratorID\device-instance-id\1\0]  ;B-channel 0 for D-channel 1
IsdnSpid=00555987600
IsdnPhoneNumber=5559876
IsdnSubaddress=

[...Enum\emumeratorID\device-instance-id\1\0]  ;B-channel 1 for D-channel 1
IsdnSpid=00555876500
IsdnPhoneNumber=5558765
IsdnSubaddress=
```

Installing a Multiprotocol WAN NIC

Article • 12/15/2021

A multiprotocol WAN NIC provides more than one WAN protocol. For example, such a NIC might allow the user to select ISDN, Frame Relay, or channelized T1. The user selects the WAN protocol during installation of the NIC or when configuring the NIC.

> ⓘ Note
>
> ISDN capabilities have been deprecated in Windows 10 and later.

A vendor of a multiprotocol WAN NIC must provide a co-installer that installs a wizard page. (For more information about co-installers, see Writing a Co-installer). The wizard page prompts the user to select a WAN protocol:

- If the user selects ISDN, the ISDN Wizard is displayed. The ISDN Wizard prompts the user for the ISDN switch type and, depending on the selected switch type, other ISDN parameter values. For more information, see Specifying ISDN Keys and Values for an ISDN Adapter.

- If the user selects a WAN protocol other than ISDN, the Wizard adds the **ShowIsdnPages** registry value to the WAN NIC's instance key. The Wizard, in this case, sets **ShowIsdnPages** to zero to suppress the display of the ISDN Wizard. As long as **ShowIsdnPages** is zero, the ISDN Wizard is suppressed.

After the WAN NIC has been installed, the user can reconfigure the NIC, using the NIC's property page:

- If the user changes the protocol from ISDN to another WAN protocol, the property page adds the **ShowIsdnPages** registry value to the WAN NIC's instance key, if necessary. The property page sets **ShowIsdnPages** to zero to suppress the display of the ISDN Wizard.

- If the user changes the protocol to ISDN, the property page for the WAN NIC displays a dialog box that prompts the user to apply the change. When the user applies the change, the property page sets **ShowIsdnPages** to 1. When the user again opens the NIC's property page, the ISDN Wizard is displayed.

Note that the **LowerRange** binding interface for a multiprotocol WAN NIC that supports ISDN must be set to **isdn**. For more information, see Specifying Binding Interfaces. If the **ShowIsdnPages** registry value is not present and if the NIC's **LowerRange** is set to **isdn**, the ISDN Wizard is displayed during installation and configuration of the NIC. If

ShowIsdnPages is set to zero, the ISDN Wizard is not displayed. If **ShowIsdnPages** is set to 1, the ISDN Wizard is displayed during configuration of the NIC.

Requiring the Installation of Another Network Component

Article • 12/15/2021

A network component may require the installation of one or more other network components in order to function properly. A network INF file specifies each such dependency with a **RequiredAll** value. The **RequiredAll** value is added (through an *add-registry-section*) to the **Ndi** key of the network component that requires the installation of another network component.

The following example shows a **RequiredAll** entry in an *add-registry-section*:

INF

```
[ndi.reg]
HKR, Ndi, RequiredAll, 0, "component id"
```

The *component ID* is the *hw-id* of the required network component. For more information, see INF Models Section. If a network component requires the installation of more than one other network component, use one **RequiredAll** entry for each network component that must be installed, as shown in the following example:

INF

```
HKR, Ndi, RequiredAll, 0, "component1 id, component2 id"
```

Note The **RequiredAll** value should only be used to install hidden network components that cannot be installed by the user. Such components should not support a user interface. Any network components specified by **RequiredAll** cannot be removed until the network component that required their installation through **RequiredAll** is itself removed.

For example, if the INF file for component A specifies, through **RequiredAll**, a dependency on component B, component B cannot be removed until component A is removed. **RequiredAll** should therefore install only network components that are absolutely required for the operation of another network component. For example, if an INF file for a Net component (an adapter) uses **RequiredAll** to specify that TCP/IP must be installed, the user will not be able to remove TCP/IP until that adapter is removed. Since the adapter does not require TCP/IP to operate, the INF for the adapter should not use **RequiredAll** to specify a dependency on TCP/IP.

The INF file that specifies a **RequiredAll** dependency must ensure that the INF file for the required network component is present in the inf directory. Typically, this is accomplished with a **CopyINF** directive. For more information about the **CopyINF** directive, see **INF CopyINF Directive**. For more information about copying INF files, see Copying INFs.

If the installation of a network component specified by a **RequiredAll** entry fails, the installation of the network component that requires the specified component fails as well.

Specifying the Name and Provider Path for a NetClient Component

Article • 12/15/2021

An INF file that installs a NetClient component must add a **NetworkProvider** key to the *service* key for the component. The INF file adds the **NetworkProvider** key through an *add-registry-section* that is referenced by an **AddReg** directive in the *service-install* section for the component.

The **NetworkProvider** key has two values: a **Name** that specifies the name of the network provider, and a **ProviderPath** that specifies the full path to the network provider DLL.

The following is an example of an *add-registry-section* that adds the **NetworkProvider** key to the instance key for a component:

```INF
[NWCWorkstation.AddReg]
HKR, NetworkProvider, Name, 0, "NetWare or Compatible Network"
HKR, NetworkProvider, ProviderPath, 0x20000, "%11%\nwprovau.dll"
```

Note An INF file that installs a **NetClient** component does not have to update the **ProviderOrder** value under the component's ... **Control\Network\Provider\Order** key. This is done automatically by the network class installer.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

Adding a HelpText Value

Article • 12/15/2021

The INF file for a **NetTrans**, **NetClient**, or **NetService** network component that is visible in a user interface should add a **HelpText** value (REG_SZ) to the component's **Ndi** key.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

The **HelpText** value is a localizable string that explains how the component benefits the user. For example, the **HelpText** value for a **NetClient** component should not simply identify the client but indicate what the client allows the user to connect to. The **HelpText** value is displayed at the bottom of the **General** page of the **Connection Properties** dialog box when a component on the page is selected.

Note Net components (adapters) and IrDA components do not support a **HelpText** value.

The following is an example of an *add-registry-section* that adds a **HelpText** value to the **Ndi** key:

INF

```
[MS_Protocol.ndi_reg]
HKR, Ndi, HelpText, 0, %MyTransport_Help%
```

The **HelpText** value is a % *strkey*% token that is defined in the **Strings** section of the INF file. For more information about the **Strings** section, see the INF Strings Section.

Note For Multilingual User Interface (MUI) support, the **HelpText** value can be an indirect string in the form `@filename,resource`. For example: "@%SystemRoot%\System32\drivers\mydriver.sys,-1000". The target string is located in the specified file. The resource value identifies the specific string within the file. If the resource value is zero or greater, the number is used as an index of the string in a binary file. If the resource value is negative, it is used as a resource identifier in a resource file.

Adding Registry Values for a Notify Object

Article • 12/15/2021

A **NetTrans**, **NetClient**, or **NetService** component can have a notify object that performs one or more of the following actions:

- Displays a user interface for the component
- Notifies the component of binding events so that the component can exercise some control over the binding process
- Conditionally installs or removes software components

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

For more information about notify objects, see Notify Objects for Network Components.

Note **Net** components (adapters) do not support notify objects; therefore, these components should use a co-installer.

For more information about co-installers, see Writing a Co-installer.

If a component has a notify object, the INF file for that component must add (through an *add-registry-section*) the following values to the component's **Ndi** key:

ClsID
A REG_SZ value that specifies the GUID (globally unique identifier) for the notify object. Obtain this GUID by running the uuidgen.exe utility. For more information about this utility, see the Microsoft Windows SDK.

ComponentDll
A REG_SZ value that specifies the path to the notify object DLL. The **ComponentDll** must specify the complete path to the DLL if the DLL is not in the Windows\System32 directory.

The following is an example of an *add-registry-section* that adds **ClsID** and **ComponentDll** values to the **Ndi** key:

INF

```
[MS_Protocol.ndi.reg]
HKR, Ndi, ClsID, 0, "GUID"
```

```
HKR, Ndi, ComponentDll, 0, "notifyobject.dll"
```

The *DDInstall* section for a component that has a notify object must also contain a **CopyFiles** directive which references a *file-list-section* that copies the notify object DLL to the destination directory specified by the **DestinationDirs** section. For more information about the **CopyFiles** directive and **DestinationDirs** sections, see INF File Sections and Directives.

Adding Service-Related Values to the Ndi Key

Article • 12/15/2021

If a component has an associated service (device driver), the *add-registry-section* referenced by the *DDInstall* section for that component must add a **Service** value to the Ndi key. The **Service** value is a REG_SZ value that specifies the primary service associated with the component. The **Service** value must match the *ServiceName* parameter of the **AddService** directive that references the *service-install-section* for the primary service. For more information, see INF DDInstall.Services Section.

If a component has one or more associated services, the *add-registry-section* referenced by the *DDInstall* section for that component must add a **CoServices** value to the **Ndi** key. The **CoServices** value is a MULTI_SZ value that specifies all the services that the component installs, including the primary service specified by the **Service** value. The **CoServices** value is required for all **NetTrans**, **NetClient**, and **NetService** components.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

Note **Net** components (adapters) should not have a **CoServices** value, because only one service can be associated with an adapter.

Except for shutting down services, all service-related actions are performed on the **CoServices** in the order that they are listed. For example, services are started in the order that they are listed. Services are stopped, however, in reverse order. Service-related actions for a component are performed on a service only if that service is listed in **CoServices**.

Specifying Binding Interfaces

Article • 05/16/2022

For each network component that it installs, a network INF file must specify the upper and lower binding interfaces for the component by adding the **Interfaces** key to the **Ndi** key.

The **Interfaces** key has at least two values:

UpperRange
A REG_SZ value that defines the interfaces to which the component can bind at its top edge.

LowerRange
A REG_SZ value that defines the interfaces to which the component can bind at its lower edge. For physical adapters, this interface should always be the network media, such as Ethernet, to which the adapter connects.

> ⓘ Note
>
> The **DefUpper** and **DefLower** values in Windows 95/98/Me network INF files, however, are not supported for INF files that will be used on Windows 2000 and later versions of the operating system.

The following table lists the Microsoft-supplied **UpperRange** values:

Value	Description
netbios	NetBIOS
ipx	IPX
tdi	TDI interface to TCP/IP
ndis5	NDIS 5.x (ndis2, ndis3, and ndis4 should no longer be used). This value should be specified for any non-ATM network component, such as a non-ATM adapter, that interfaces with NDIS at its upper edge.
Ndisatm	NDIS 5.x with ATM support. Specified value for any ATM network component, such as an ATM adapter, whose upper edge interfaces with NDIS

Value	Description
ndiswan	Upper edge for a WAN adapter. When specified, this value causes the operating system to automatically enable the WAN adapter for use with RAS
Ndiscowan	Upper edge for a WAN adapter over which connection-oriented NDIS runs
noupper	Upper edge for any component that does not expose an upper edge for binding; such a component typically has a private interface at its upper edge
winsock	The Windows socket interface
ndis5_atalk	Upper edge for an NDIS 5.x Net component (adapter) that binds only to an AppleTalk interface at its upper edge
ndis5_dlc	Upper edge for an NDIS 5.x Net component (adapter) that binds only to a DLC interface at its upper edge
ndis5_ip	Upper edge for an NDIS 5.x Net component (adapter) that binds only to a TCP/IP interface at its upper edge
ndis5_ipx	Upper edge for an NDIS 5.x Net component (adapter) that binds only to an IPX interface at its upper edge
ndis5_nbf	Upper edge for an NDIS 5.x Net component (adapter) that binds only to a NetBEUI interface at its upper edge
ndis5_streams	Upper edge for an NDIS 5.x Net component (adapter) that binds only to a streams interface at its upper edge. This value is obsolete for Windows XP and later operating systems.
flpp4	A mobile broadband (MB) device that supports IPv4.
flpp6	A mobile broadband (MB) device that supports IPv6.

The following table lists the Microsoft-supplied **LowerRange** values:

Value	Description
ethernet	Lower edge for an Ethernet adapter
atm	Lower edge for an ATM adapter
tokenring	Lower edge for a token ring adapter
serial	Lower edge for a serial adapter
fddi	Lower edge for an FDDI adapter
baseband	Lower edge for a baseband adapter
broadband	Lower edge for a broadband adapter
bluetooth	Lower edge for a Bluetooth adapter
arcnet	Lower edge for an Arcnet adapter
isdn	Lower edge for an ISDN adapter
localtalk	Lower edge for a LocalTalk adapter
wan	Lower edge for a WAN adapter
nolower	Lower edge for any component that does not expose a lower edge for binding
ndis5	NDIS 5.x. (ndis2, ndis3, and ndis4 should no longer be used.) For any network component whose lower edge interfaces through NDIS with non-ATM components
Ndisatm	Ndis 5.x with ATM support. For any network component whose lower edge interfaces through NDIS with ATM components
Wlan	Lower edge for a native 802.11 wireless LAN adapter.
ppip	Lower edge for a mobile broadband (MB) adapter
vwifi	Lower edge for a virtual wifi interface

The **UpperRange** and **LowerRange** values specify the types of interfaces -- not the actual components -- to which a component can bind. The binding engine binds a network component to all components that provide the specified interface at the appropriate (upper or lower) edge. For example, a protocol with a **LowerRange** of ndis5 binds to all components that have an **UpperRange** of ndis5, such as physical or virtual adapters.

> **Note**
>
> NDIS LWF drivers can't attach to adapters that have nolower in their **LowerRange** of their INF file. NDIS LWF drivers aren't allowed to have nolower in their **FilterMediaTypes**.

If an NDIS 5.x Net component (adapter) works only with one or more specific protocols, then its **UpperRange** should be assigned one or more protocol-specific values, such as ndis5_atalk, ndis5_dlc, ndis5_ip, ndis5_ipx, ndis5_nbf, or ndis5_streams. Such a net class component should not be assigned an **UpperRange** value of ndis5, because this would cause that component to bind to all protocols that provide an ndis5 lower edge.

An INF-file-writer can define and use vendor-specific **UpperRange** and **LowerRange** values for private binding interfaces. For example, if a vendor wants to bind its adapter only to its own proprietary protocol driver, the INF-file-writer could specify **XXX** for the **UpperRange** of the adapter and **XXX** for the **LowerRange** of the proprietary protocol. The Windows 2000 binding engine will bind all components that have an **UpperRange** of **XXX** (in this case, the adapter) with all components that have a **LowerRange** of **XXX** (in this case, the proprietary protocol).

The following is an example of an *add-registry-section* that adds **UpperRange** and **LowerRange** values for an ATM adapter:

INF

```
[addreg-section]
HKR, Ndi\Interfaces, UpperRange, 0, "ndisATM"
HKR, Ndi\Interfaces, LowerRange, 0, "atm"
```

Specifying Configuration Parameters for the Advanced Properties Page

Article • 12/15/2021

> ⓘ Note
>
> Prior to Windows 10, version 1703, driver upgrades and Windows updates could result in changes to INF values that the driver had previously defined in the **Advanced** properties page. Starting in Windows 10, version 1703, advanced properties that a driver specifies in its INF file persist through these updates.

An INF file that installs a Net component (adapter) can specify adapter configuration parameters for display in the **Advanced** properties page for the component. Configuration values specified by the user in the **Advanced** properties page are written to the root instance key for the component.

Note that if an adapter supports an **Advanced** properties page, the **Characteristics** entry in the *DDInstall* section for the adapter must include the NCF_HAS_UI value.

A network INF file specifies configuration parameters for display in the Advanced page through an *add-registry-section* that is referenced by the *DDInstall* section for the component. Such an *add-registry-section* adds one or more configuration subkeys to the **Ndi\params** key. The format for a configuration parameter subkey is **Ndi\params\***SubKeyName*, where *SubKeyName* is a REG_SZ value that specifies a vendor-specific parameter name. For example, the key for a parameter that specifies a transceiver type could be named **Ndi\params\TransceiverType**.

The following keywords are reserved and cannot be used as an **Ndi\params\***SubKeyName*: **BundleId**, **BusType**, **Characteristics**, **ComponentId**, **Description**, **DeviceInstanceId**, **DriverDate**, **DriverDesc**, **DriverVersion**, **InfPath**, **InfSection**, **InfSectionExt**,** *IfType** **InstallTimeStamp**, **Manufacturer**,** *MediaType, **NetCfgInstanceId, NetLuidIndex, PhysicalMediaType, **Provider, and **ProviderName*.

For each parameter subkey that is added to **Ndi\params**, the *add-registry-section* must add **ParamDesc**(parameter description) and **Type** values. The *add-registry-section* can also add **Default** and **Optional** values for the parameter and, if the parameter is numeric, **Min**, **Max**, and **Step** values. The following table describes the values that can be added to each **Ndi\params** key.

Name	Value	Description

Name	Value	Description
ParamDesc	*String*	Name displayed for the parameter on the **Advanced** page
Type	**int**, **long**, **Word**, **dword**, **edit**, or **enum**	Type of parameter: **int**, **long**, **Word**, and **dword** specify a numeric parameter; **edit** and **enum** specify a text parameter.
Default	*default value*	Default value for the parameter: for a numeric parameter, must be a numeric value (**int**, **long**, **Word**, or **dword**) that matches the specified parameter type; for a text parameter, must be a string. Default values must be specified for required parameters. Default values can also be specified for optional parameters. When a user selects the option to enter a value for an optional parameter, the default value, if specified, appears in the edit box for that parameter.
Optional	**0** or **1**	**0** required. Specify a value for the parameter or use the default value. **1** optional. Can be marked **Not Present** on the **Advanced** page.
Min	*numeric value*	Minimum value for a numeric parameter.
Max	*numeric value*	Maximum value for a numeric parameter.
Step	*numeric value*	Step (interval) between valid values for a numeric parameter. The minimum value is the starting point.

The range of values for an **enum** parameter are specified with a subkey that has the following format:

Ndi\params\*SubKeyName***\enum**

Each enumerated value must have a subkey. Each **enum** subkey specifies a numeric value (starting with zero for the first enumerated value) and a description for that value.

The following is an example of an *add-registry-section* that adds a configuration parameter named **TransType**.

INF

```
[a1.params.reg]
HKR, Ndi\params\TransType,      ParamDesc, 0, "Transceiver Type"
HKR, Ndi\params\TransType,      Type,      0, "enum"
HKR, Ndi\params\TransType,      Default,   0, "0"
HKR, Ndi\params\TransType,      Optional,  0, "0"
HKR, Ndi\params\TransType\enum, "0",       0, "Auto-Connector"
HKR, Ndi\params\TransType\enum, "1",       0, "Thick Net(AUI/DIX)"
HKR, Ndi\params\TransType\enum, "2",       0, "Thin Net (BNC/COAX)"
HKR, Ndi\params\TransType\enum, "3",       0, "Twisted-Pair (TPE)"
```

Specifying Custom Property Pages for Network Adapters

Article • 12/15/2021

If the **Advanced** property page is not suitable for displaying the configuration choices for a Net component (adapter), you can create one or more custom property pages.

To create a custom property page

1. Create a Microsoft Win32 property page. Then create a property sheet extension DLL that provides *AddPropSheetPageProc* and *ExtensionPropSheetPageProc* callback functions. For more information, see the Windows 2000 Platform SDK.

2. Use the *add-registry-section* that is referenced by the **DDInstall** section for the adapter to add the **EnumPropPages32** key to the instance key for the adapter. The **EnumPropPages32** key has two REG_SZ values: the name of the DLL that exports the *ExtensionPropSheetPageProc* function and the name of the *ExtensionPropSheetPageProc* function. The following is an example of an *add-registry-section* that adds the **EnumPropPages32** key:

   INF

   ```
   HKR, EnumPropPages32, 0, "DLL name, ExtensionPropSheetPageProc function
   name"
   ```

3. In the INF file for the adapter, include a **CopyFiles** section that copies the property sheet extension DLL to the Windows\System32 directory. For more information about the **CopyFiles** section, see INF File Sections and Directives.

4. In the **DDInstall** section for the adapter, specify NCF_HAS_UI as one of the **Characteristics** values to indicate that the adapter supports a user interface. For more information, see DDInstall Section.

5. After the user applies changes to a property page, the property sheet extension DLL must:

   - Call **SetupDiGetDeviceInstallParams**

   - Set the DI_FLAGSEX_PROPCHANGE_PENDING flag in the SP_DEVINSTALL_PARAMS structure supplied by **SetupDiGetDeviceInstallParams**

- Pass the updated SP_DEVINSTALL_PARAMS structure to **SetupDiSetDeviceInstallParams**.

  This reloads the driver so that it can read the changed parameter values.

Specifying Bundle Membership

Article • 12/15/2021

ⓘ **Note**

Bundle membership has been deprecated in Windows Vista and later.

An INF file that installs **Net** components (physical or virtual network adapters) can specify that those network adapters belong to the same bundle of adapters. Note that NDIS intermediate drivers and filter drivers, which export virtual network adapters, are included in the Net class. An NDIS driver can use adapters that it installed to balance its workload by distributing the workload over the bundle of adapters. For more information about load balancing, see Load Balancing and Failover.

To specify that an adapter belongs to a particular bundle of adapters, the INF file for the driver that installs the adapter must contain the **BundleId** keyword and a case-insensitive string value (REG_SZ). This string value identifies the driver's bundle of adapters. The registry is configured using the bundle-identifier information.

The following is an example of an *add-registry-section* in a driver's INF file that adds the **BundleId** subkey to the **Ndi\params** key and gives the **ParamDesc** (parameter description) for **BundleId** a string value of "Bundle1".

INF

```
[a1.params.reg]
HKR, Ndi\params\BundleId, ParamDesc, 0, "Bundle1"
```

Deprecated Ndi Values and Keys

Article • 12/15/2021

Important The following **Ndi** registry keys and values are no longer used in the Windows operating system. If you are migrating network drivers from Windows 95/98/Me to later versions of the operating system, do not use these values.

DeviceVxD

DevLoader

DriverDesc

InfFile

InfSelection

Ndi\CardType

Ndi\Compability

Ndi\DeviceID

Ndi\*filename*\...

Ndi\Install

Ndi\InstallInf

Ndi\Interfaces\DefLower

Ndi\Interfaces\DefUpper

Ndi\Interfaces\ExcludeAny

Ndi\Interfaces\RequireAny

Ndi\NdiInstaller

Ndi\*param-key-name***\resc**

Ndi\Params\*param-key-name***\flag**

Ndi\Params\*param-key-name***\location**

Ndi\Remove

NDIS\...

StaticVxD

Because Windows does not support **Ndi\***param-key-name***\resc** and **Ndi\Params\param-key-name\flag** values, a user cannot specify adapter resources through the **Advanced** properties page.

DDInstall.Services Section in a Network INF File

Article • 12/15/2021

A *DDInstall*.**Services** section in a network INF file is based on the generic INF DDInstall.Services section.

A *DDInstall*.**Services** section contains one or more **AddService** directives, each of which references an INF-writer-defined *service-install- section* that specifies how and when the services of particular component drivers are loaded.

A *DDInstall*.**Services** section is required in an INF file that installs a Net component (adapter); it is optional in an INF file that installs a **NetTrans**, **NetClient**, or **NetService** component.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

An **AddService** directive in a *DDInstall*.**Services** section can also reference an *error-log-install-section* that installs an error log for a component. An error log is optional for all network components.

For more information, see INF AddService Directive.

The following is an example of a *DDInstall*.**Services** section, a *service-install-section*, an *error-log-install-section*, and an *add-registry-section* that is referenced by an **AddReg** directive in the *error-log-install-section*:

```C++
[a1.ndi.NT.Services]
AddService = a1, 2, a1.AddService, a1.AddEventLog

[a1.AddService]
DisplayName = %Adapter1.DispName%
ServiceType = 1 ;SERVICE_KERNEL_DRIVER
StartType = 2 ;SERVICE_AUTO_START
ErrorControl = 1 ;SERVICE_ERROR_NORMAL
ServiceBinary = %13%\a1.sys
LoadOrderGroup = NDIS

[a1.AddEventLog]
AddReg = a1.AddEventLog.reg

[a1.AddEventLog.reg]
```

```
HKR,,EventMessageFile,0x00020000,"%%SystemRoot%%\System32\netevent.dll"
HKR,,TypesSupported,0x00010001,7
```

The *ServiceName* parameter of the **AddService** directive, which in the above example is **a1**(the first **AddService** parameter), must match the component's **Ndi\Service** value. For more information, see Adding Service-Related Values to the Ndi Key.

NetworkProvider and PrintProvider Sections in a Network INF File

Article • 12/15/2021

NetClient components are considered to be network providers because they provide network services to user applications. The Microsoft Client for Networks and the NetWare Client are examples of **NetClient** components.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

In addition to being a network provider, a **NetClient** component can also be a print provider. A print provider provides print services to user applications over a network.

A **NetClient** component is always installed as a network provider. An INF file that installs a **NetClient** component does not require a NetworkProvider section for that component unless at least one of the following is true:

- An alternative device name is specified for the component.

- A short name for the component is specified for use with the **net view** command. For more information, see Including a NetworkProvider Section.

An INF that installs a **NetClient** component that is a print provider must contain a **PrintProvider** section for that component. For more information, see Including a PrintProvider Section.

An INF file that installs a **NetClient** component must also contain an *add-registry-section* (referenced by a **AddReg** directive in the *service-install-section* for a component) that adds a **NetworkProvider** key to the component's **service** key. For more information, see Specifying the Name and Provider Path for a NetClient Component.

Including a NetworkProvider Section

Article • 12/15/2021

A **NetworkProvider** section specifies either a substitute device name for a **NetClient** component or a short name for use with the NetWare **net view** command, or both.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

To create a **NetworkProvider** section, add the **NetworkProvider** extension to the *DDInstall* section for the component, as shown in the following example:

INF

```
[DDInstall] ; Install section
[DDInstall.NetworkProvider] ; NetworkProvider section
```

Specifying a Device Name

The network class installer usually creates the device name for a network provider by copying the **Ndi\Service** value for the component to the NetworkProvider key under the component's **Service** key. For more information, see Adding Service-Related Values to the Ndi Key. To specify a different device name for the component, include a **DeviceName** entry in the **NetworkProvider** section for the component, as shown in the following example:

INF

```
[DDInstall-section.NetworkProvider]
DeviceName = "nwrdr"
```

The **DeviceName** is optional and should be specified only if the **Ndi\Service** value for the component is inadequate as a device name for the network provider.

Specifying a Short Name

To specify a short name for a network provider for use with the NetWare **net view** command, include a **ShortName** entry in the **NetworkProvider** section for the component, as shown in the following example:

INF

```
[DDInstall-section.NetworkProvider]
ShortName = "nw"
```

The following is an example of a short name used with the **net view** command:

INF

```
net view /n:nw
```

The **ShortName** is easier to remember and type than the entire name of the network provider.

The **ShortName** is optional and should only be specified if needed.

Including a PrintProvider Section

Article • 12/15/2021

An INF file that installs a **NetClient** component that is a print provider must contain a **PrintProvider** section for that component.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

To create a **PrintProvider** section, add the **PrintProvider** extension to the *DDInstall* section for the component, as shown in the following example:

C++

```
[DDInstall-section] ; Install section
[DDInstall-section.PrintProvider] ; PrintProvider section
```

The **PrintProvider** section must include the following entries:

PrintProviderName
A nonlocalized string that specifies the name of the print provider.

PrintProviderDll
The file name of the print provider DLL.

DisplayName
A localizable string that specifies the name of the print provider. The **DisplayName** can differ from the **PrintProviderName**.

The **PrintProviderName** and **PrintProviderDll** entries supply information that is used as input (in a PROVIDOR_INFO_1 structure) to the **AddPrintProvidor** function. The **AddPrintProvidor** function adds the print provider component as a print provider. For more information about the **AddPrintProvidor** function, see the Microsoft Windows SDK.

The following is an example of a **PrintProvider** section:

C++

```
[DDnstall-section.PrintProvider]
PrintProviderName = "NetWare or Compatible Network"
PrintProviderDll  = "nwprovau.dll"
DisplayName       = "%NWC_Network_Display_Name%"
```

Winsock Sections in a Network INF File

Article • 12/06/2022

An INF file for a **NetTrans** component that provides a Winsock interface must specify this Winsock dependency. Such an INF file must contain a *Winsock-install* section. To create a Winsockinstall section, add the .Winsock extension to the *DDInstall* section name for the protocol. For example, if the *DDInstall* section for a protocol is named **Ipx**, the *Winsock-install* section for that protocol must be named Ipx.Winsock.

> **Note**
>
> Winsock dependency has been deprecated in Windows 8 and later.

A *Winsock-install* section must contain an **AddSock** directive. The **AddSock** directive specifies a vendor-named section that contains values to be added to the component's **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\*TransportDriverName*\Params\Winsock** key.

The vendor-named section referenced by the **AddSock** directive must contain the following required values:

Value Name	Description
TransportService	A REG_SZ value that specifies the service name of the protocol. This must be the same as the **Ndi\Service** value for the protocol. For more information, see Adding Service-Related Values to the Ndi Key.
HelperDllName	A REG_EXPAND_SZ value that specifies the path to the Windows Sockets helper (WSH) DLL for the protocol. For more information, see WSH DLL Function Summary.
MaxSockAddrLength	A REG_DWORD value that specifies the largest valid SOCKADDR size, in bytes, for the WSH DLL
MinSockAddrLength	A REG_DWORD value that specifies the smallest valid SOCKADDR size, in bytes, for the WSH DLL

If an optional **ProviderId** for a namespace provider is specified, the following values must also be specified:

Value Name	Description
ProviderId	A REG_SZ value that specifies the Globally Unique Identifier (GUID) that identifies the namespace provider. The GUID is used as a key to all subsequent references to the namespace provider. Obtain the GUID by running the uuidgen.exe utility. For more information about this utility, see the Microsoft Windows SDK.
LibraryPath	A REG_EXPAND_SZ value that specifies the complete path to the namespace provider DLL.
DisplayString	A localizable string that specifies the name displayed for the namespace provider in the user interface.
SupportedNameSpace	A REG_DWORD value which specifies the namespace that is supported by the namespace provider.
Version	An optional REG_DWORD value that specifies the version number of the namespace provider. If this value is not specified, the default value (1) is used for the version number.

The following namespace values can be assigned to SupportedNameSpace and are defined in Winsock2.h:

Namespace	Value
NS_ALL	0
NS_SAP	1
NS_NDS	2
NS_PEER_BROWSE	3
NS_TCPIP_LOCAL	10
NS_TCPIP_HOSTS	11
NS_DNS	12
NS_NETBT	13
NS_WINS	14
NS_NBP	20

Namespace	Value
NS_MS	30
NS_STDA	31
NS_CAIRO	32
NS_X500	40
NS_NIS	41
NS_WRQ	50

For more information about namespace providers, see the Windows SDK documentation.

The following example shows Winsock sections for an IPX protocol:

INF

```
[Ipx.Winsock]
AddSock = Install.IpxWinsock

[Install.IpxWinsock]
TransportService = nwlinkipx
HelperDllName = "%%SystemRoot%%\System32\wshisn.dll"
MaxSockAddrLength = 0x10
MinSockAddrLength = 0xe
ProviderId = "GUID"
LibraryPath = "%SystemRoot%\\System32\\nwprovau.dll"
DisplayString = %NwlnkIpx_Desc%
SupportedNameSpace = 1
Version = 2
```

An INF file can remove a Winsock dependency for a protocol by including a *Winsock-remove* section. To create a *Winsock-remove* section, add the .Winsock extension to the *Remove* section name for the protocol. For example, if the *Remove* section for a protocol is named Ipx.Remove, the *Winsock-remove* section for the protocol must be named Ipx.Remove.Winsock.

The *Winsock-remove* section contains a **DelSock** directive that specifies an INF-writer-named section. The INF-writer-named section must specify the transport service to remove. If a **ProviderId** was previously registered for the protocol, the vendor-named section must also specify the **ProviderId** to remove.

The following example shows two sections that remove the Winsock dependency for an IPX protocol:

INF

```
[Ipx.Remove.Winsock]
DelSock = Remove.IpxWinsock

[Remove.IpxWinsock]
TransportService = nwlinkipx
ProviderId = "GUID"
```

Installation Requirements for Network Adapters

Article • 12/15/2021

This topic summarizes the installation requirements for network adapters.

Note NDIS 6.0 and later drivers support a set of standardized INF keywords for network devices.

General Requirements

INF File Section	Status	Comments
Version Section	Required	**Class**= Net **ClassGuid**= {4D36E972-E325-11CE-BFC1-08002BE10318}
INF SourceDisksNames Section and **INF SourceDisksFiles Section**	Required if ...	Required if the INF file is not distributed with Windows 2000. If the INF file is distributed with Windows 2000, a **LayoutFile** entry must be specified in the **Version** section, and the **SourceDisksNames** and **SourceDisksFiles** sections are not used. No network-specific requirements.
INF DestinationDirs Section	Required	No network-specific requirements.
ControlFlags Section	Required	Must contain an **ExcludeFromSelect** entry for each Plug and Play (PnP) adapter installed by the INF file. Non-PnP adapters, such as non-PnP ISA and EISA adapters, should not be listed. Note that Windows XP and later operating systems do not support non-PnP ISA adapters and EISA adapters.
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* must match the hardware ID supplied by the adapter to the PnP manager.

INF File Section	Status	Comments
DDInstall Section	Required	**Characteristics** entry Allowable values: NCF_VIRTUAL, NCF_SOFTWARE_ENUMERATED, NCF_PHYSICAL, NCF_MULTIPORT_INSTANCED_ADAPTER, NCF_HAS_UI, NCF_HIDDEN, NCF_NOT_USER_REMOVABLE NCF_VIRTUAL, NCF_SOFTWARE_ENUMERATED, and NCF_PHYSICAL are mutually exclusive. The **BusType** entry is required for a physical adapter. The **EisaCompressedId** entry is required for an EISA adapter. This entry specifies both an EISA Compressed ID and an adapter mask for the adapter. Windows XP and later operating systems do not support EISA adapters. A **Port1DeviceNumber** or **Port1FunctionNumber** entry is required for a multiport network adapter.
DDInstall.Services Section	Required	No network-specific requirements.

INF File Section	Status	Comments
Add-registry-sections	Required	Creating the Ndi Key Specifying service-related values Specifying Bundle Membership(only for LBFO miniport drivers) Specifying Binding Interfaces Allowable binding interfaces: **UpperRange**: ndis5, ndisatm, ndiswan, ndiscowan, noupper, ndis5_atalk, ndis5_dlc, ndis5_ip, ndis5_ipx, ndis5_nbf, ndis5_streams **LowerRange**: ethernet, atm, tokenring, serial, fddi, baseband, broadband, arcnet, isdn, localtalk, wan
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component Specifying Configuration Parameters for the Advanced Properties Page Specifying Custom Property Pages for Network Adapters
INF Strings Section	Required	No network-specific requirements.

Additional Requirements for WAN Adapters

WAN adapters have additional installation requirements that are described in the following topics:

Specifying WAN Endpoints for a WAN Adapter

Specifying ISDN Keys and Values for an ISDN Adapter

Installing a Multiprotocol WAN NIC

Note The Remove Section and Notify Objects for Network Components are not supported.

Installation Requirements for Network Protocols

Article • 12/15/2021

This topic summarizes the installation requirements for network protocols.

INF File Section	Status	Comments
Version Section	Required	**Class**= NetTrans **ClassGuid**= {4D36E975-E325-11CE-BFC1-08002BE10318}
INF SourceDisksNames Section and **INF SourceDisksFiles Section**	Required if ...	Required if the INF file is not distributed with Windows 2000. If the INF file is distributed with Windows 2000, a **LayoutFile** entry must be specified in the **Version** section, and the **SourceDisksNames** and **SourceDisksFiles** sections are not used. No network-specific requirements.
INF DestinationDirs Section	Required	No network-specific requirements.
ControlFlags Section	Optional	No network-specific requirements.
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name, for example: MS_DLC.

INF File Section	Status	Comments
DDInstall Section	Required	**Characteristics** entry Allowable values: NCF_HIDDEN NCF_NO_SERVICE NCF_NOT_USER_REMOVABLE NCF_HAS_UI
DDInstall.Services Section	Optional	No network-specific requirements.
Add-registry-sections	Required	Required: Creating the Ndi Key Specifying Binding Interfaces Allowable binding interfaces: **UpperRange**: netbios, ipx, tdi, winsock, noupper **LowerRange**: ndis5, ndisatm, nolower
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component Specifying service-related values Adding a HelpText Value Adding Registry Values for a Notify Object
Remove Section	Optional	

INF File Section	Status	Comments
Winsock Sections	Optional	For a protocol that provides a Winsock interface, a **Winsock-install** section is required and a **Winsock-remove** section is optional.
INF Strings Section	Required	No network-specific requirements.

Installation Requirements for Network Filter Drivers

Article • 12/15/2021

This topic summarizes the INF file requirements for network filter drivers. Filter drivers are supported in NDIS 6.0 and later versions. For more information about how to install filter drivers, see NDIS Filter Driver Installation.

INF File Section	Status	Comments
Version Section	Required	**Class**= NetService **ClassGuid**= {4D36E974-E325-11CE-BFC1-08002BE10318}
INF SourceDisksNames Section and **INF SourceDisksFiles Section**	Optional	No network-specific requirements.
INF DestinationDirs Section	Required	No network-specific requirements.
ControlFlags Section	Optional	No network-specific requirements.
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name (for example, MS_DLC).
DDInstall Section	Required	**Characteristics** entry: NCF_LW_FILTER (0x40000) is set. Filter drivers must not set the NCF_FILTER (0x400) flag. The values of the NCF_*Xxx* flags are defined in Netcfgx.h. For more information about NCF_*Xxx* flags, see DDInstall Section in a Network INF File. Set the **NetCfgInstanceId** entry.

INF File Section	Status	Comments
DDInstall.Services Section	Optional	No network-specific requirements.
Add-registry-sections	Required	Creating the Ndi Key The **ServiceBinary** entry in the **service-install** section of the INF file specifies the path to the binary for the filter driver. Set the **FilterType** and **FilterRunType** . See Types of Filter Drivers. Set the **UpperRange**, **LowerRange**, and **FilterMediaTypes** entries. See Specifying Filter Driver Binding Relationships. Specify the primary service name of the filter for the **CoServices** attribute. Specify the **FilterClass** to determine the order in a stack of modifying filters. See Configuring an INF File for a Modifying Filter Driver. See also Configuring an INF File for a Monitoring Filter Driver, Adding Service-Related Values to the Ndi Key, and DDInstall.Services Section in a Network INF File.
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component Adding a HelpText Value Adding Registry Values for a Notify Object
Remove Section	Optional	

INF File Section	Status	Comments
INF Strings Section	Required	No network-specific requirements.

Installation Requirements for Network MUX Intermediate Drivers

Article • 12/15/2021

This topic summarizes the installation requirements for network MUX intermediate drivers. For more information about installation requirements for MUX intermediate drivers, see Installing an Intermediate Driver.

Two INF files are required to install a network MUX intermediate driver :

- Driver protocol (**Class**= NetTrans)
- Driver device (**Class**= Net)

Protocol INF File for a Network MUX Intermediate Driver

INF File Section	Status	Comments
Version Section	Required	**Class**= NetTrans **ClassGuid**= {4D36E975-E325-11CE-BFC1-08002BE10318}
INF SourceDisksNames Section and **INF SourceDisksFiles Section**	Required if ...	Required if the INF file is not distributed with Windows 2000. If the INF file is distributed with Windows 2000, a **LayoutFile** entry must be specified in the **Version** section, and the **SourceDisksNames** and **SourceDisksFiles** sections are not used. No network-specific requirements.
INF DestinationDirs Section	Required	No network-specific requirements.
ControlFlags Section	Optional	No network-specific requirements.
INF Manufacturer Section	Required	No network-specific requirements.

INF File Section	Status	Comments
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name--for example: MS_DLC.
DDInstall Section	Required	**Characteristics** entry: NCF_HAS_UI is required. The device INF must be copied to the system INF directory, see Copying INFs.
DDInstall.Services Section	Optional	No network-specific requirements.
Add-registry-sections	Required	Creating the Ndi Key Specifying Binding Interfaces UpperRange: noupper **LowerRange**: ethernet, atm, tokenring, serial, fddi, baseband, broadband, arcnet, isdn, localtalk, wan
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component Specifying service-related values Adding a HelpText Value Adding Registry Values for a Notify Object
Remove Section	Optional	
INF Strings Section	Required	No network-specific requirements.

Device INF File for a Network MUX Intermediate Driver

INF File Section	Status	Comments
Version Section	Required	**Class**= Net **ClassGuid**= {4D36E972-E325-11CE-BFC1-08002BE10318}
ControlFlags Section	Required	This section must contain an **ExcludeFromSelect** entry for the device.
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name--for example: MS_DLC.
DDInstall Section	Required	**Characteristics** entry: NCF_VIRTUAL is required. NCF_HIDDEN and NCF_NOT_USER_REMOVABLE are optional.
DDInstall.Services Section	Required	No network-specific requirements.

INF File Section	Status	Comments
Add-registry-sections	Required	Creating the Ndi Key Specifying service-related values Specifying Binding Interfaces Allowable binding interfaces: **UpperRange**: ndis5, ndisatm, ndiswan, ndiscowan, noupper, ndis5_atalk, ndis5_dlc, ndis5_ip, ndis5_ipx, ndis5_nbf, ndis5_streams LowerRange: nolower
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component
INF Strings Section	Required	No network-specific requirements.

Installation Requirements for Network Filter Intermediate Drivers

Article • 12/15/2021

Note Filter intermediate drivers are not supported in NDIS 6.0 and later. You should use the NDIS filter driver interface instead. For more information about NDIS filter drivers, see NDIS Filter Drivers.

This topic summarizes the INF file requirements for NDIS 5.*x* network filter intermediate drivers.

Two INF files are required to install a network filter intermediate driver:

- Driver service (**Class**= NetService)
- Driver device (**Class**= Net)

Service INF File for a Network Filter Intermediate Driver

INF File Section	Status	Comments
Version Section	Required	**Class**= NetService **ClassGuid**= {4D36E974-E325-11CE-BFC1-08002BE10318}
INF SourceDisksNames Section and **INF SourceDisksFiles Section**	Required if ...	Required if the INF file is not distributed with Windows 2000. If the INF file is distributed with Windows 2000, a **LayoutFile** entry must be specified in the **Version** section, and the **SourceDisksNames** and **SourceDisksFiles** sections are not used. No network-specific requirements.
INF DestinationDirs Section	Required	No network-specific requirements.
ControlFlags Section	Optional	No network-specific requirements.

INF File Section	Status	Comments
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name--for example: MS_DLC.
DDInstall Section	Required	**Characteristics** entry: NCF_FILTER is required. NCF_HAS_UI and NCF_NO_SERVICE are optional. The device INF must be copied to the system INF directory, see Copying INFs.
DDInstall.Services Section	Optional	No network-specific requirements.
Add-registry-sections	Required	Creating the Ndi Key FilterClass, FilterDeviceInfId, FilterMediaTypes Specifying Binding Interfaces Allowable binding interfaces: UpperRange: noupper LowerRange: nolower
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component Adding a HelpText Value Adding Registry Values for a Notify Object
Remove Section	Optional	
INF Strings Section	Required	No network-specific requirements.

Device INF File for a Network Filter Intermediate Driver

INF File Section	Status	Comments
Version Section	Required	**Class**= Net **ClassGuid**= {4D36E972-E325-11CE-BFC1-08002BE10318}
ControlFlags Section	Required	This section must contain an **ExcludeFromSelect** entry for the device.
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name--for example: MS_DLC.
DDInstall Section	Required	**Characteristics** entry: NCF_VIRTUAL is required. NCF_HIDDEN and NCF_NOT_USER_REMOVABLE are optional.
DDInstall.Services Section	Required	The *ServiceName* value of the **AddService** directive must match the filter component's Service value under the **Ndi** key.
Add-registry-sections	Required	Creating the Ndi Key Specifying service-related values
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component
INF Strings Section	Required	No network-specific requirements.

Installation Requirements for Network Clients

Article • 12/15/2021

This topic summarizes the installation requirements for network clients.

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

INF File Section	Status	Comments
Version Section	Required	**Class**= NetClient **ClassGuid**= {4D36E973-E325-11CE-BFC1-08002BE10318}
INF SourceDisksNames Section and **INF SourceDisksFiles Section**	Required if ...	Required if the INF file is not distributed with Windows 2000. If the INF file is distributed with Windows 2000, a **LayoutFile** entry must be specified in the **Version** section, and the **SourceDisksNames** and **SourceDisksFiles** sections are not used. No network-specific requirements.
INF DestinationDirs Section	Required	No network-specific requirements.
ControlFlags Section	Optional	No network-specific requirements.
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name--for example: MS_DLC.

INF File Section	Status	Comments
DDInstall Section	Required	**Characteristics** entry Allowable values: NCF_HIDDEN NCF_NO_SERVICE NCF_NOT_USER_REMOVABLE NCF_HAS_UI
DDInstall.Services Section	Optional	No network-specific requirements.
Add-registry-sections	Required	Creating the Ndi Key Specifying Binding Interfaces Allowable binding interfaces: UpperRange: noupper **LowerRange**: ipx, tdi, winsock, netbios, nolower
Add-registry-sections	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component Specifying service-related values Adding a HelpText Value Adding Registry Values for a Notify Object
Remove Section	Optional	
NetworkProvider and PrintProvider Sections	Required if ...	Required if an alternative device name is specified for the network client or a short name for the component is specified for use with the **net view** command.
NetworkProvider and PrintProvider Sections	Required if ...	Required if the network client is a print provider.

INF File Section	Status	Comments
INF Strings Section	Required	No network-specific requirements.

Installation Requirements for Network Services

Article • 12/15/2021

This topic summarizes the installation requirements for network services.

INF File Section	Status	Comments

Version Section	Required	Class = NetService
		ClassGuid = {4D36E974-E324-11CE-BFC1-08002BE10318}

INF SourceDisksNames Section and **INF SourceDisksFiles Section**	Required if ...	Required if the INF file is not distributed with Windows 2000. If the INF file is distributed with Windows 2000, a LayoutFile entry must be specified in the Version section, and the SourceDisksNames and SourceDisksFiles sections are not used. No network-specific requirements.
INF DestinationDirs Section	Required	No network-specific requirements.
ControlFlags Section	Optional	No network-specific requirements.
INF Manufacturer Section	Required	No network-specific requirements.
Models Section	Required	The *hw-id* should consist of a provider name followed by an underscore and a manufacturer name or the product name--for example: MS_DLC.

INF File Section	Status	Comments
DDInstall Section	Required	Characteristics entry--allowable values: NCF_HIDDEN NCF_NO_SERVICE NCF_NOT_USER_REMOVABLE NCF_HAS_UI
DDInstall.Services Section	Optional	No network-specific requirements.
Add-registry-sections	Required	Creating the Ndi Key Specifying Binding Interfaces Allowable binding interfaces: UpperRange: noupper LowerRange: ipx, tdi, winsock, netbios, nolower
	Optional	Setting static parameters for the component Requiring the Installation of Another Network Component Specifying service-related values Adding a HelpText Value Adding Registry Values for a Notify Object
Remove Section	Optional	
INF Strings Section	Required	No network-specific requirements.

Notify Objects for Network Components

Article • 12/15/2021

A *notify object* processes notifications that are sent by the network configuration subsystem to the object on behalf of a specific network component. A notify object is served up by a dynamic-link library (DLL). A notify object is used to display Property pages for a network component and to give that component programmatic control over the network configuration.

Note A network component does not generally require a notify object if both of the following conditions are true:

- A network component can be installed and removed through its information (INF) file
- Reacting to changes in network configuration is not a requirement

The following sections describe notify objects and explain how to develop them:

About Notify Objects

Creating a Notify Object

For reference information for the interface methods that support notify objects, see Notify Objects.

About Notify Objects

Article • 12/15/2021

A notify object processes notifications that the network configuration subsystem sends to the object on behalf of a specific network component. This network component owns the notify object. Network components that can own a notify object are:

- Transports such as a protocol driver
- Services such as an intermediate driver
- Clients such as a Client for Microsoft Networks

Note Network cards do not support and cannot own notify objects. Physical or virtual network cards that participate in either configuring the network or installing and uninstalling must use INF files or the device co-installer mechanism. For more information, see Writing a Co-installer.

A notify object performs the following actions:

- Exposes interface methods to the network configuration subsystem so that the network configuration subsystem can inform the notify object about the occurrence of events on which the notify object requested notification.
- Calls methods of the network configuration subsystem's public interfaces to perform actions that include but are not limited to installing and removing network devices. For more information, see Network Configuration Interfaces.

To request and receive notifications and to communicate with each other, the notify object and the network configuration subsystem implement Component Object Model (COM) interfaces.

Notify objects are COM objects that reside within dynamic-link libraries (DLLs). These DLLs are COM *component servers*. Each type of network component is associated with a *class installer* which installs specific types of network components and registers COM *class objects* that are owned by these network components. After the main install phase for network components is complete, the objects are registered. To register a COM class object, the class installer calls the object's DLL entry-point function.

Whenever the operating system installs, upgrades, or removes networking functionality, or whenever applications configure the network, the operating system or those applications must start the network configuration subsystem. After the network configuration subsystem starts, it creates an instance of a notify object, and the notify object performs particular operations.

The following topics describe the types of notifications that notify objects receive and the operations that notify objects perform:

Notify Object Diagram

Processing Notifications

Installing Network Components

Removing Network Components

Upgrading Network Components

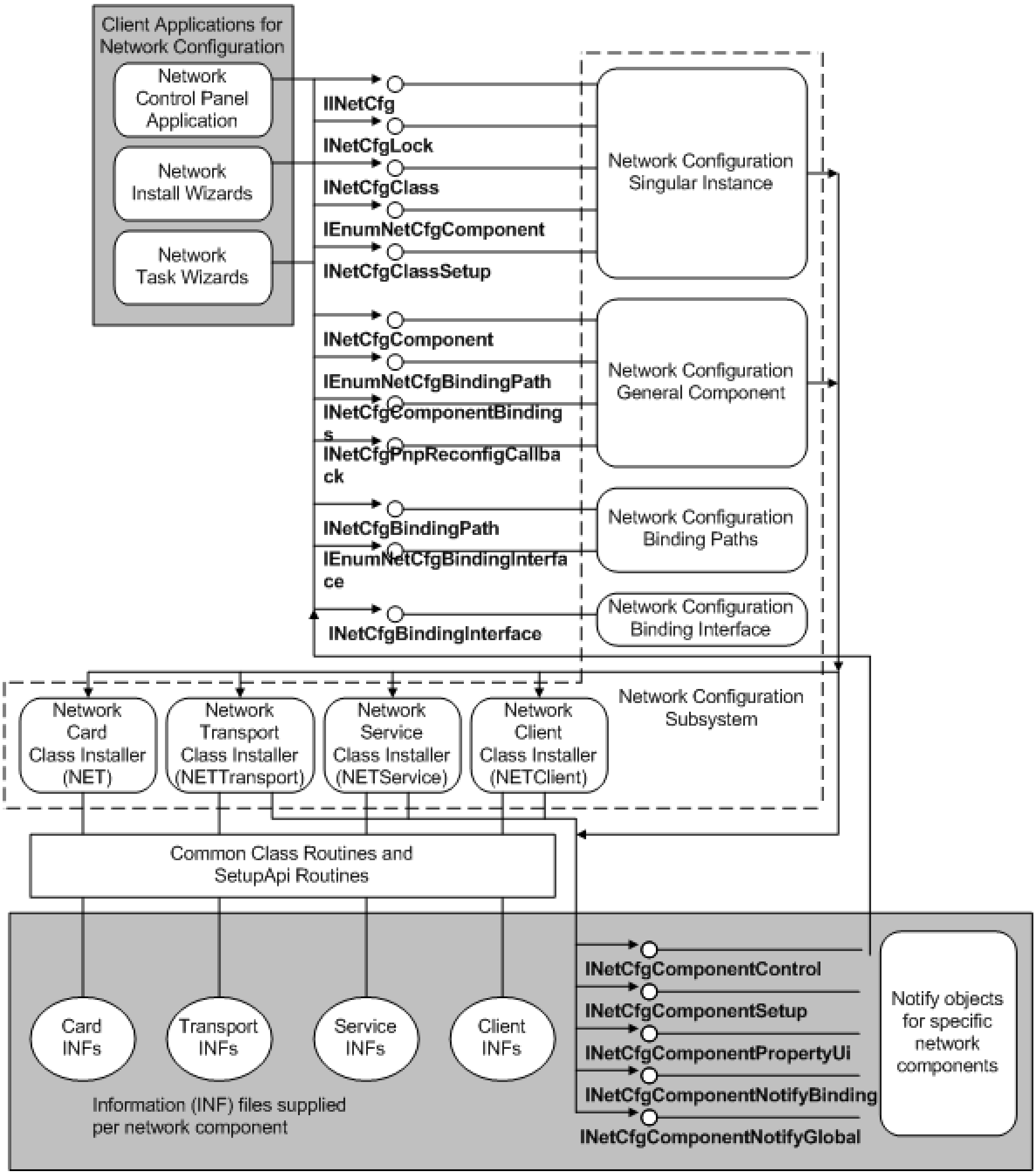
Displaying and Changing Properties

Configuring the Network

Notify Object Diagram

Article • 12/15/2021

The following diagram shows how client applications that install or control networking call the *network configuration subsystem*. This subsystem calls network class installers to install network components and to register notify objects for those components. Notify objects call back to the subsystem to configure the network on behalf of those components that own the objects.


Client Applications for Network Configuration
Network Control Panel Application
Network Install Wizards
Network Task Wizards
IINetCfg
INetCfgLock
INetCfgClass
IEnumNetCfgComponent
INetCfgClassSetup
INetCfgComponent
IEnumNetCfgBindingPath
INetCfgComponentBindings
INetCfgPnpReconfigCallback
INetCfgBindingPath
IEnumNetCfgBindingInterface
INetCfgBindingInterface
Network Configuration Singular Instance
Network Configuration General Component
Network Configuration Binding Paths
Network Configuration Binding Interface
Network Configuration Subsystem
Network Card Class Installer (NET)
Network Transport Class Installer (NETTransport)
Network Service Class Installer (NETService)
Network Client Class Installer (NETClient)
Common Class Routines and SetupApi Routines
Card INFs
Transport INFs
Service INFs
Client INFs
Information (INF) files supplied per network component
INetCfgComponentControl
INetCfgComponentSetup
INetCfgComponentPropertyUi
INetCfgComponentNotifyBinding
INetCfgComponentNotifyGlobal
Notify objects for specific network components

Processing Notifications

Article • 12/15/2021

The network configuration subsystem sends notifications to notify objects at the following intervals:

- During networking setup--including operating system installation, installing networking capability on an operating system that did not previously support networking, upgrading the operating system, or uninstalling networking features
- During network configuration--including adding, removing, enabling, and disabling network components, changing network components, and changing how the network configuration subsystem binds network components together
- After an application directs the subsystem to display the properties of network components that own notify objects

To process notifications, a notify object performs the following general sequence of operations:

1. When the notify object is loaded, it reads the system registry to form a model of the current network configuration in its internal data structures.
2. After the network configuration subsystem sends notifications to the notify object about networking changes that the notify object previously requested, the notify object modifies its internal data structures to keep track of those changes.
3. When the network configuration subsystem is done sending notifications to the notify object, the subsystem calls the notify object's **INetCfgComponentControl::ApplyRegistryChanges** method to commit the changes to the system registry.

Note The notifications mentioned in the preceding sequence can also include a call to the notify object's **INetCfgComponentControl::CancelChanges** method in which case the notify object should revert back to the original network configuration. Before modifying the original network configuration, the notify object should make two copies of the configuration. The notify object can modify one copy to include changes and leave the other copy in the original condition. The notify object can use the unmodified copy when reverting back to the original network configuration.

Installing Network Components

Article • 05/30/2023

Network components are installed by the network configuration subsystem.

To create a driver package with a notify object that is executed from the Driver Store, you must have a minimum OS build number of 25341. You can't successfully install a driver package in this scenario on older versions of Windows.

To install a network component

1. The network configuration subsystem calls the class installer for the particular component type. The class installer then calls the Setup API to retrieve information from the component's INF file and to install the component.

   If the component owns a notify object, the class installer retrieves the name of the DLL that houses the notify object. This DLL appears in the component's INF file as follows:

   INF

   ```
   HKR, Ndi, ComponentDll,     0,     "notifyobject.dll"
   ```

   The class installer calls the DLL's entry-point function to register the notify object. The network configuration subsystem creates an instance of the notify object and calls the object's **INetCfgComponentControl::Initialize** method. This method initializes the object and provides access to the component and all aspects of network configuration.

2. To perform operations required to install the component, the network configuration subsystem calls the notify object's **INetCfgComponentSetup::Install** method.

   If installation of the component is unattended, the network configuration subsystem calls the notify object's **INetCfgComponentSetup::ReadAnswerFile** method. This method opens and retrieves the component's parameters from a file for unattended setup that is known as an *answer file*.

3. After the network configuration subsystem creates an instance of and initializes the notify object, the subsystem calls the notify object's **INetCfgComponentNotifyGlobal::GetSupportedNotifications** method to retrieve the types of notifications required by the object. The subsystem uses this information to send required notifications to the object. The object can use these

notifications to control aspects of networking setup and configuration that might affect the component that owns the object. For example, if the subsystem calls the **INetCfgComponentNotifyGlobal::SysNotifyComponent** method to notify the object that the subsystem installed or removed another network component, the object has the opportunity to perform operations related to the change.

After the network configuration subsystem creates an instance of and initializes the notify object, the subsystem also calls any of the methods of the notify object's INetCfgComponentNotifyBinding interface to notify the object about changes to the way the subsystem binds other network components to the component that owns the notify object.

4. When the network configuration subsystem is ready to apply the component's properties to the operating system, it calls the notify object's **INetCfgComponentControl::ApplyRegistryChanges** method to assign the component's parameters under the component's registry key. The notify object calls its component's **INetCfgComponent::OpenParamKey** method to open and retrieve the component's registry key.

5. To configure the component's driver, the network configuration subsystem calls the notify object's **INetCfgComponentControl::ApplyPnpChanges** method and passes the **INetCfgPnpReconfigCallback** interface. The notify object calls the **INetCfgPnpReconfigCallback::SendPnpReconfig** method to send configuration information to its component's driver.

For more information about the Setup API and on files for unattended setup, see the Microsoft Windows SDK.

Removing Network Components

Article • 12/15/2021

Network components are removed by the network configuration subsystem.

To remove a network component

1. The network configuration subsystem creates an instance of the notify object and calls the object's INetCfgComponentControl::Initialize method. This method initializes the object and provides access to the component and all aspects of network configuration.

2. The subsystem calls the notify object's INetCfgComponentSetup::Removing method to perform operations required to remove the component. The **Removing** method performs cleanup operations to prepare for the component's removal.

3. The subsystem calls the notify object's INetCfgComponentControl::ApplyRegistryChanges method to remove information about the network component from the registry.

Upgrading Network Components

Article • 12/15/2021

Network components are upgraded by the network configuration subsystem.

To upgrade a network component

1. The network configuration subsystem creates an instance of the notify object and calls the object's INetCfgComponentControl::Initialize method. This method initializes the object and provides access to the component and all aspects of network configuration.

2. When the operating system is installed or upgraded to a different version, the network configuration subsystem calls the notify object's INetCfgComponentSetup::Upgrade method.

3. The subsystem calls the notify object's INetCfgComponentControl::ApplyRegistryChanges method to modify information about the network component in the registry and then calls the notify object's INetCfgComponentControl::ApplyPnpChanges method and passes the INetCfgPnpReconfigCallback interface to configure the component's driver with the upgraded information.

Displaying and Changing Properties

Article • 12/15/2021

The network configuration subsystem displays Property pages for a network component and changes the component's parameters.

A component's properties can be displayed and modified from Control Panel. When you click the **Network** icon, you start the network configuration subsystem, which creates an instance of the notify object and calls the object's INetCfgComponentControl::Initialize method. This method initializes the object and provides access to the component and all aspects of network configuration.

The application calls the component's INetCfgComponent::RaisePropertyUi method to display the component's properties. The **RaisePropertyUi** method then calls the following notify object methods:

- INetCfgComponentPropertyUi::QueryPropertyUi method to determine if a specific context is appropriate to display properties for the component.

- INetCfgComponentPropertyUi::SetContext method to direct the component's notify object to display the component's properties in the specified context.

- INetCfgComponentPropertyUi::MergePropPages method to create and merge custom pages for the component's property sheet into the default set.

If the user changes one of the component's parameters on one of the custom pages, **RaisePropertyUi** calls the notify object's INetCfgComponentPropertyUi::ApplyProperties method to store the change in memory.

To apply the change, the network configuration subsystem calls the notify object's INetCfgComponentControl::ApplyRegistryChanges method to modify information about the network component in the registry. To configure the component's driver with the modified information, the network configuration subsystem calls the notify object's INetCfgComponentControl::ApplyPnpChanges method and passes the INetCfgPnpReconfigCallback interface.

Configuring the Network

Article • 12/15/2021

A notify object can provide the network component that owns it with programmatic control over network configuration.

A network component's properties can be configured from the Network Control Panel application. When you click the **Network** icon, you start the network configuration subsystem, which creates an instance of the notify object and calls the object's **INetCfgComponentControl::Initialize** method. This method initializes the object and provides access to the component and all aspects of network configuration.

After the network configuration subsystem creates an instance of and initializes the notify object, the subsystem calls the notify object's **INetCfgComponentNotifyGlobal::GetSupportedNotifications** method to retrieve the types of notifications required by the object. Using this information, the subsystem can send required notifications to the object. The object can use these notifications to control aspects of networking setup and configuration that might affect the component that owns the object. For example, if the subsystem calls the notify object's **INetCfgComponentNotifyGlobal::SysQueryBindingPath** method to notify the object that the subsystem is about to add a binding path to which other network components belong, the object has the opportunity to request that the subsystem disable that binding path. In addition, the subsystem calls any of the methods of the notify object's INetCfgComponentNotifyBinding interface. These methods notify the object about changes to the way the subsystem binds other network components to the component that owns the notify object.

Creating a Notify Object

Article • 12/15/2021

A notify object should be created for a network component if the network component requires some control over networking setup and configuration and the ability to display custom property pages that users can use to modify the component's properties.

The following topics describe how to create a notify object:

Loading the Notify Object DLL and Class Object

Defining a Notify Class

Creating and Initializing an Instance of a Notify Object

Installing, Upgrading, and Removing the Component

Creating Property Pages for the Component

Setting Context to Display Properties

Evaluating Changes to Network Configuration

Applying Component Changes to the Registry

Configuring the Component's Driver

Retrieving Network Configuration Interface Pointers

Loading the Notify Object DLL and Class Object

Article • 12/15/2021

Notify objects for network components should be implemented as Component Object Model (COM) objects. These COM objects reside in DLLs that are COM component servers. For more information about developing DLL COM servers, see the Microsoft Windows SDK.

The DLL for a particular notify object should be implemented to export a set of entry-point functions:

- A **DllMain** function to let the network configuration subsystem load the DLL into the virtual address space for the subsystem.

- **DllRegisterServer** and **DllUnregisterServer** functions to put information into the operating system registry for the DLL's class objects. The network configuration subsystem uses this registry information to locate and load a network component's notify object.

- A **DllCanUnloadNow** function to let the network configuration subsystem determine whether the DLL is in use. If the DLL is not in use, the subsystem can safely unload the DLL from memory.

In order for a notify object DLL to be a COM server, it must expose a class factory for the notify object the server supports. This class factory lets the network configuration subsystem create an instance of the notify object. The class factory should inherit from the **IClassFactory** interface. For more information about implementing classes that inherit from **IClassFactory**, see the Windows SDK.

Defining a Notify Class

Article • 12/15/2021

Notify classes must be implemented so that they inherit from the **INetCfgComponentControl** interface. However, if notify objects perform certain operations their notify classes must also be implemented to inherit from the following interfaces:

- If a notify object performs operations related to installing, upgrading, and removing the component that owns the object, the associated notify class must inherit from the **INetCfgComponentSetup** interface.

- If a notify object displays custom property pages for the component that owns the object, the associated notify class must inherit from the **INetCfgComponentPropertyUi** interface.

- If a notify object evaluates changes to the way the network configuration subsystem binds the component that owns the object to other network components, the associated notify class must inherit from the INetCfgComponentNotifyBinding interface.

- If a notify object evaluates changes to network configuration that might affect the component that owns the object, the associated notify class must inherit from the **INetCfgComponentNotifyGlobal** interface.

Certain data members within notify classes should be defined as common to all notify objects. Certain data members should be defined as specific to their component. Data members that all notify objects should define include:

- A pointer to an instance of the network component that owns the object of type **INetCfgComponent** interface. An instance of a notify object uses this pointer to access and control the component that owns the object.

- A pointer to an instance of the network configuration object of type **INetCfg** interface. An instance of a notify object uses this pointer to access all aspects of network configuration.

- Variables to store parameter information for the component that owns the notify object

- A variable that specifies the action that a notify object previously performed. Define constants to indicate the different actions that notify objects might perform. When the network configuration subsystem calls the notify object's

INetCfgComponentControl::ApplyRegistryChanges method to apply configuration changes to the registry, **ApplyRegistryChanges** uses this variable to determine how to make registry changes. For example, if a notify object previously performed operations relating to installing the component that owns the object in its INetCfgComponentSetup::Install method, **Install** should set this variable to indicate the action as install.

- A registry key of type **HKEY**. A notify object calls the INetCfgComponent::OpenParamKey method of the component that owns the object to open and retrieve the registry key that contains parameters for the component. The notify object then sets the **HKEY** member to that key.

Define a constructor and a destructor for your notify class. Also consider defining private methods that only the notify class can use.

All the **IUnknown** interface methods should be implemented for a notify class. If a notify class inherits from any of the optional interfaces noted in the preceding list, all the methods of those interfaces must be implemented. Note that E_NOTIMPL is not a valid return type for any of the methods of the notify object interfaces. If a notify object does not require an implementation for a particular method, simply implement the method to return S_OK.

Creating and Initializing an Instance of a Notify Object

Article • 12/15/2021

The network configuration subsystem must create an instance of the notify object and initialize the object before the subsystem can inform a notify object about changes to network configuration and display custom property pages for the component that owns the object.

The subsystem creates an instance of the notify object from the DLL's class factory. The class factory then calls the constructor for the notify class.

The class constructor should first assign initial values to class data members. Values that the constructor should initially assign include the following:

- The constructor should set the interface pointer to an instance of a network component, **INetCfgComponent**, to a **NULL** value.

- The constructor should set the interface pointer to an instance of the network configuration object, **INetCfg**, to a **NULL** value.

- The constructor should set the variable that specifies the action that the notify object previously performed to a constant that identifies an unknown action. For more information about this variable, see Defining a Notify Class.

After the network configuration subsystem creates an instance of the notify object, the subsystem calls the object's **INetCfgComponentControl::Initialize** method to initialize the object instance. In this call, the subsystem passes an **INetCfgComponent** interface pointer. This **INetCfgComponent** provides the notify object with an instance of the object's component that the object can use to access and control the component. In this call, the subsystem also passes an **INetCfg** interface pointer to provide the notify object with an instance of the network configuration object that the notify object uses to access all aspects of network configuration.

The **Initialize** method should assign the **INetCfgComponent** and **INetCfg** interface pointers provided by the network configuration subsystem to data members of the notify class. **Initialize** should then call:

- the **INetCfg::AddRef** method to increment the reference count of the network configuration object

- the **INetCfgComponent::AddRef** method to increment the reference count of the component that owns the notify object

No other notify object interface methods are called until **Initialize** returns.

Installing, Upgrading, and Removing the Component

Article • 12/15/2021

When the network configuration subsystem installs, upgrades, or removes a network component, the subsystem also calls the component's notify object to complete the installation, upgrade, and removal. The component's notify object can be implemented to perform operations that the component might require. For example:

- A notify object for a multiplexer for a virtual LAN can be implemented so that when the subsystem installs the multiplexer, the notify object will install virtual adapters that the multiplexer protocol binds to.

  To install a virtual adapter, the notify object calls the network configuration's INetCfgClassSetup::Install method. In this call, the notify object passes the identifier of the virtual adapter to install. The notify object can call **INetCfgClassSetup::Install**, for example, from its INetCfgComponentNotifyBinding::NotifyBindingPath or INetCfgComponentPropertyUi::ApplyProperties method.

  To complete the installation of the virtual adapter, the operating system requires the INF file for the virtual adapter. To ensure that this INF file can be located, it must be copied to the operating system when the multipexer is installed. For more information, see Copying INFs. This topic indicates that the **CopyINF** directive or a call to the **SetupCopyOEMInf** function by a co-installer or setup application can be used to copy INF files to the target system's INF directory. However, if the INF file for the multiplexer (original INF) is copied using **SetupCopyOEMInf**, then the INF file for the virtual adapter must also be copied using **SetupCopyOEMInf** because the operating system only handles a **CopyINF** directive if the original INF is not yet in the INF directory.

- The multiplexer's notify object can be implemented so that when the subsystem removes the multiplexer, the notify object will remove all virtual adapters. To remove a virtual adapter, the notify object calls the network configuration's INetCfgClassSetup::DeInstall method. In this call, the notify object passes the pointer to the **INetCfgComponent** interface of the virtual adapter. The notify object can call **INetCfgClassSetup::DeInstall**, for example, from its **INetCfgComponentNotifyBinding::NotifyBindingPath** or **INetCfgComponentPropertyUi::ApplyProperties** method.

- The component's notify object can be implemented so that when the subsystem upgrades the component, the notify object will change the order of the component's binding path. To change this order, a notify object's INetCfgComponentSetup::Upgrade method calls either the INetCfgComponentBindings::MoveBefore or the INetCfgComponentBindings::MoveAfter methods.

Creating Property Pages for the Component

Article • 12/06/2022

A notify object creates custom property pages after the network configuration subsystem calls the notify object's INetCfgComponentPropertyUi::MergePropPages method. Custom property pages can be merged into the default set of pages on the component's property sheet using the **MergePropPages** method. **MergePropPages** will return the appropriate number of default pages into which the custom pages can be merged.

To create custom property pages, **MergePropPages** calls the COM **CoTaskMemAlloc** function to allocate memory for handles to PROPSHEETPAGE structures. Each of these handles represents a property page to create. If **CoTaskMemAlloc** successfully allocates the memory for the handles, **MergePropPages** will declare and fill **PROPSHEETPAGE** structures for each property page. After **MergePropPages** fills these structures, it calls the Win32 **CreatePropertySheetPage** function for each property page. In this call, **MergePropPages** passes the address of the PROPSHEETPAGE structure to create.

A dialog-box callback function should also be implemented for each property page that **MergePropPages** creates. A dialog-box callback function processes messages that the operating system sends to the property page that is associated with that dialog-box function. To associate a property page with a dialog-box function, **MergePropPages** should point the **pfnDlgProc** member of each PROPSHEETPAGE structure for each page to the dialog-box function for the page.

A dialog-box function processes the following messages:

- The WM_INITDIALOG message, which is sent to the dialog-box function immediately before the operating system displays its associated property page. Dialog-box functions typically use this message to initialize the property page and to perform tasks that affect the appearance of the property page.

- The WM_NOTIFY message, which is sent to the dialog-box function after an event occurs in the property page. Other information sent with this message identifies what event has occurred. This event information is contained in a pointer to a NMHDR structure. Information that NMHDR can contain for a property sheet includes, for example:

  - The PSN_APPLY event, which indicates that a user clicks OK, Close, or Apply on the property page. If the user clicks OK, Close, or Apply, the dialog-box function can call the **PropSheet_Changed** macro to inform the property sheet that information in the page has changed. In this call, the dialog-box function passes handles to the property sheet and the page. The dialog-box function can call the Win32 **GetParent** function and pass the handle to the page to retrieve the handle to the property sheet.

    After the dialog-box function notifies the property sheet about the change, the network configuration subsystem calls the INetCfgComponentPropertyUi::ValidateProperties method to check the validity of all changes. If all changes are valid, the subsystem calls the notify object's INetCfgComponentPropertyUi::ApplyProperties method to cause all changes to take effect. The network configuration subsystem calls **ApplyProperties** before the operating system closes the dialog box.

    The **ApplyProperties** method can be implemented to retrieve information that the user enters and to set the information to the notify object's data members.

  - The PSN_RESET event, which indicates that the operating system is about to destroy a property page. A user might click Cancel on the property page to initiate this action. If the user clicks Cancel, the network configuration subsystem calls the INetCfgComponentPropertyUi::CancelProperties method to cause all changes to be disregarded. The network configuration subsystem calls **CancelProperties** before the dialog box is closed.

  - The PSN_KILLACTIVE event, which indicates that a property page is about to become inactive. This event occurs when a user activates another page or clicks OK.

Property-page callback functions can also be implemented for each property page that **MergePropPages** creates. A property-page callback function performs initialization and cleanup operations for the page. To associate a property page with a property-page callback function, **MergePropPages** should point the **pfnCallback** member of each PROPSHEETPAGE structure for each page to the property-page callback function for that page.

See the Microsoft Windows SDK documentation for more information about:

- creating property pages and structures, functions, and notifications for property pages

- dialog-box callback procedures, messages, and structures

Setting Context to Display Properties

Article • 12/15/2021

A notify object can set the context in which to display properties for the network component that owns the object. The notify object sets the display context after the network configuration subsystem calls the object's INetCfgComponentPropertyUi::SetContext method but before the subsystem calls the object's INetCfgComponentPropertyUi::MergePropPages method.

When the network configuration subsystem calls **SetContext**, it passes an **IUnknown** interface. **SetContext** calls the **QueryInterface** method on this **IUnknown** interface to determine the interface of the specific object that the subsystem supplied.

For example, the network configuration subsystem can supply the INetLanConnectionUiInfo interface when it calls **SetContext**. **SetContext** can use the GetDeviceGuid method of **INetLanConnectionUiInfo** to retrieve the GUID of a LAN device. The notify object can subsequently display properties for its network component in the context of this LAN device. For example, the notify object for the TCP/IP protocol can display an IP address that is associated with a particular LAN adapter in the context of that adapter. Doing so enables users to specify an IP address for that adapter.

Evaluating Changes to Network Configuration

Article • 12/15/2021

After the network configuration subsystem calls the methods of a notify object's **INetCfgComponentNotifyGlobal** and INetCfgComponentNotifyBinding interfaces, the notify object should evaluate the proposed change in network configuration that the subsystem sends and should perform operations related to the change. The methods of a notify object's **INetCfgComponentNotifyGlobal** and **INetCfgComponentNotifyBinding** interfaces should be implemented to process only the changes that affect the component that owns the object.

The following topics describe examples of how a notify object processes changes to network configuration:

Adding a Component

Changing Bindings for a Component

Adding a Component

Article • 12/15/2021

The network configuration subsystem can inform a notify object when the subsystem adds network components. After initializing a notify object, the subsystem calls the notify object's INetCfgComponentNotifyGlobal::GetSupportedNotifications method to retrieve the types of notifications required by the object. If the notify object specified that it required notification when network components are added, the subsystem calls the notify object's INetCfgComponentNotifyGlobal::SysNotifyComponent method and passes NCN_ADD to inform the notify object that the subsystem installed a network component. If the component that owns the notify object should bind to the specified component, the notify object should perform operations to facilitate the binding. For example, the following code shows how the notify object can bind its component to the specified component if the specified component is a required physical network card.

```C++
HRESULT CSample::SysNotifyComponent(DWORD dwChangeFlag,
        INetCfgComponent* pnccItem)
{
    HRESULT hr = S_OK;
    INetCfgComponentBindings *pncfgcompbind;
    // Retrieve bindings for the notify object's component (m_pncc)
    hr = m_pncc->QueryInterface(IID_INetCfgComponentBindings,
                                (LPVOID*)&pncfgcompbind);
    // Determine if notification is about adding a component
    if (SUCCEEDED(hr) && (NCN_ADD & dwChangeFlag)) {
        // Retrieve the characteristics of the added component
        DWORD dwcc;
        hr = pnccItem->GetCharacteristics(&dwcc);
        // Determine if the added component is a physical adapter
        if (SUCCEEDED(hr) && (dwcc & NCF_PHYSICAL)) {
            // Determine the component's ID
            LPWSTR pszwInfId;
            hr = pnccItem->GetId(&pszwInfId);
            if (SUCCEEDED(hr)) {
                // Compare the component's ID to the required ID
                // and if they are the same perform the binding.
                static const TCHAR c_szCompId[] = TEXT("BINDTO_NIC");
                if (!_tcsicmp(pszwInfId, c_szCompId)) {
                    hr = pncfgcompbind->BindTo(pnccItem);
                }
            }
        }
    }
    return hr;
}
```

Changing Bindings for a Component

Article • 12/15/2021

The network configuration subsystem always informs a notify object about changes in binding that affect the notify object's network component. The subsystem calls the notify object's INetCfgComponentNotifyBinding::NotifyBindingPath method and passes a value that specifies the change along with a pointer to the **INetCfgBindingPath** interface of the binding path involved in the change. If the subsystem passes NCN_DISABLE to disable the binding path that the notify object's network component shares with a specific network card, the notify object can activate the binding with another network card as shown in the following code.

C++

```
HRESULT CSample::NotifyBindingPath(DWORD dwChangeFlag,
        INetCfgBindingPath* pncbp1)
{
    INetCfgComponent *pnccLow;
    INetCfgComponentBindings *pncbind;
    IEnumNetCfgBindingPath *penumncbp;
    INetCfgBindingPath *pncbp2;
    IEnumNetCfgBindingInterface *penumncbi;
    INetCfgBindingInterface *pncbi;
    DWORD dwFlags = EBP_BELOW;
    ULONG celt = 1; // Request one enumeration element.
    HRESULT hr = S_OK;
    // Retrieve bindings for the notify object's component (m_pncc)
    hr = m_pncc->QueryInterface(IID_INetCfgComponentBindings,
                                (LPVOID*)&pncbind);
    // Determine if notification is about disabling a binding path.
    if (SUCCEEDED(hr) && (NCN_DISABLE & dwChangeFlag)) {
        // Retrieve enumerator for binding paths for the component.
        hr = pncbind->EnumBindingPaths(dwFlags, &penumncbp);
        // Reset the sequence and retrieve a binding path.
        hr = penumncbp->Reset();
        hr = penumncbp->Next(celt, &pncbp2, NULL);
        // Ensure the binding path is different.
        do {
            if (pncbp1 != pncbp2) break;
   hr = penumncbp->Skip(celt); // skip one element
            hr = penumncbp->Next(celt, &pncbp2, NULL);
        } while (SUCCEEDED(hr));
        if (SUCCEEDED(hr)) {
            // Retrieve enumerator for interfaces of the binding path.
            hr = pncbp2->EnumBindingInterfaces(&penumncbi);
            // Retrieve a binding interface for the binding path.
            hr = penumncbi->Next(celt, &pncbi, NULL);
            // Retrieve the lower network component.
            hr = pncbi->GetLowerComponent(&pnccLow);
            // If the component is a physical network card and binding
            // is currently disabled, enable binding.
            DWORD dwcc;
            hr = pnccLow->GetCharacteristics(&dwcc);
            if (SUCCEEDED(hr) && (dwcc & NCF_PHYSICAL)) {
                hr = pncbp2->IsEnabled(); // S_FALSE for disabled
                if (hr == S_FALSE)  hr = pncbp2->Enable(TRUE);
            }
        }
        else return hr;
    }
    return hr;
}
```

Applying Component Changes to the Registry

Article • 12/15/2021

After the network configuration subsystem calls a notify object's INetCfgComponentControl::ApplyRegistryChanges method, the notify object should set, modify, or delete information from the registry depending on the action previously performed by the notify object. After the notify object performs specific actions related to installing, removing, or modifying parameters of the component that owns the object, the notify object should set a data member that indicates the action performed. After the subsystem calls **ApplyRegistryChanges** to apply configuration changes to the registry, **ApplyRegistryChanges** should use this data member to determine how to make registry changes. For example:

- If a notify object previously performed operations related to installing the component that owns the object, the notify object should have set the data member that indicates the action as "install". After the subsystem calls **ApplyRegistryChanges** to apply configuration changes to the registry, **ApplyRegistryChanges** should set information about the component in the registry.

- If a notify object previously performed operations related to removing the component that owns the object, the notify object should have set the data member that indicates the action as "remove". After the subsystem calls **ApplyRegistryChanges** to apply configuration changes to the registry, **ApplyRegistryChanges** should remove information about the component from the registry.

- If a user displays one of a component's custom property pages and modifies one of the component's parameters, the component's notify object should have set the data member that indicates the action as "modify parameter". After the subsystem calls **ApplyRegistryChanges** to apply configuration changes to the registry, **ApplyRegistryChanges** should change information about the component's parameter in the registry.

To open and retrieve a component's registry key to modify information about the component, the **ApplyRegistryChanges** method should be implemented to call the component's INetCfgComponent::OpenParamKey method. To set values in the registry under the component's registry key, implement **ApplyRegistryChanges** to write registry data using Win32 functions. For example, **ApplyRegistryChanges** can call the

RegCreateKeyEx function to create a subkey to hold values, and the **RegSetValueEx** function to create and set those values.

For more information about the registry, writing data to it, and retrieving data from it, see the Microsoft Windows SDK.

Configuring the Component's Driver

Article • 12/15/2021

After the network configuration subsystem calls a notify object's INetCfgComponentControl::ApplyPnpChanges method, the notify object should send configuration information to the driver of the network component that owns the notify object. The network configuration subsystem calls **ApplyPnpChanges** after it calls the INetCfgComponentControl::ApplyRegistryChanges method and after drivers and services for the particular network component have started. In the **ApplyPnpChanges** call, the network configuration subsystem passes the INetCfgPnpReconfigCallback interface. The component's notify object can use the **INetCfgPnpReconfigCallback** interface to send configuration information to the component's driver. This driver must be either a TDI provider or an NDIS miniport driver.

The notify object can call INetCfgPnpReconfigCallback::SendPnpReconfig within its **ApplyPnpChanges** implementation to send configuration information to its component's driver. **SendPnpReconfig** passes configuration information to the driver.

Alternatively, the notify object can call the Win32 CreateFile function to open a connection to its component's driver. The notify object can call the Win32 DeviceIoControl function to send a control code along with input data directly to its component's driver.

The notify object is not required to use **INetCfgPnpReconfigCallback**. But, if the notify object uses **INetCfgPnpReconfigCallback**, a user will not be required to reboot the operating system to cause configuration changes to take effect in the driver.

Retrieving Network Configuration Interface Pointers

Article • 12/15/2021

When the network configuration subsystem initializes an instance of the notify object as described in Creating and Initializing an Instance of a Notify Object, the object receives INetCfgComponent and INetCfg interface pointers. **INetCfgComponent** points to the notify object's component interface that the object can use to access and control the component. **INetCfg** points to the root network configuration interface that the notify object can use to access all aspects of network configuration. The following code uses these **INetCfgComponent** and **INetCfg** interface pointers to retrieve other network configuration interfaces that the notify object might require.

C++

```cpp
// Using the notify object's component interface that the notify
// object received:
INetCfgComponent *pncfgcompThis, *pncfgcompUp, *pncfgcompLow;
INetCfgComponentBindings *pncfgcompbind;
IEnumNetCfgBindingPath *penumncfgbindpath;
INetCfgBindingPath *pncfgbindpath;
IEnumNetCfgBindingInterface *penumncfgbindintrfc;
INetCfgBindingInterface *pncfgbindintrfc;
HRESULT hr;
DWORD dwFlags;  // EBP_ABOVE or EBP_BELOW
ULONG celt, celtFetched; // Number of requested and returned elements

// Retrieve a pointer to INetCfgComponentBindings to control and
// retrieve information about bindings for the component.
hr = pncfgcompThis->QueryInterface(IID_INetCfgComponentBindings,
                                   (LPVOID*)&pncfgcompbind);
// Retrieve a pointer to IEnumNetCfgBindingPath to enumerate binding
// paths for the component.
hr = pncfgcompbind->EnumBindingPaths(dwFlags, &penumncfgbindpath);
// Retrieve a pointer to INetCfgBindingPath that points to one or more
// binding paths for the component.
hr = penumncfgbindpath->Next(celt, &pncfgbindpath, &celtFetched);
// Retrieve a pointer to IEnumNetCfgBindingInterface to enumerate
// the collection of binding interfaces for the binding path.
hr = pncfgbindpath->EnumBindingInterfaces(&penumncfgbindintrfc);
// Retrieve a pointer to INetCfgBindingInterface that points to one or
// more binding interfaces for the binding path.
hr = penumncfgbindintrfc->Next(celt, &pncfgbindintrfc, &celtFetched);
// Retrieve pointers to INetCfgComponent for network components
// above and below the binding interface.
hr = pcfgbindintrfc->GetUpperComponent(&pncfgcompUp);
hr = pcfgbindintrfc->GetLowerComponent(&pncfgcompLow);
```

```
// Using the root network configuration interface that the notify
// object received:
INetCfg *pnetcfg;
INetCfgLock *pncfglock;
INetCfgClass *pncfgclass;
INetCfgComponent *pncfgcompOther, *pncfgcompInstall;
INetCfgClassSetup *pncfgclsSetup
GUID *pguidClass; // For example, set to GUID_DEVCLASS_NETTRANS
IEnumNetCfgComponent *penumncfgcomp;
HWND hwndParent; // Handle to Window for selecting.
OBO_TOKEN *pOboToken; // Another component or the user installs.
DWORD dwSetupFlags, dwUpgradeFromBuildNo;

// Retrieve a pointer to INetCfgLock to obtain a lock on network
// configuration.
hr = pnetcfg->QueryInterface(IID_INetCfgLock, (LPVOID*)&pncfglock);
// Retrieve a pointer to INetCfgComponent for a specific component.
hr = pnetcfg->FindComponent(TEXT("MS_TCPIP"), &pncfgcompOther);
// Retrieve a pointer to IEnumNetCfgComponent to enumerate
// the collection of a particular type of component.
hr = pnetcfg->EnumComponents(pguidClass, &penumncfgcomp);
// Retrieve a pointer to INetCfgClass for a specific class of
// component.
hr = pnetcfg->QueryNetCfgClass(pguidClass, IID_INetCfgClass,
                              (LPVOID*)&pncfgclass);
// Retrieve a pointer to INetCfgComponent for a specific component.
hr = pncfgclass->FindComponent(TEXT("MS_TCPIP"), &pncfgcompOther);
// Retrieve a pointer to IEnumNetCfgComponent to enumerate
// the collection of a particular type of component.
hr = pncfgclass->EnumComponents(&penumncfgcomp);
// Retrieve a pointer to INetCfgComponent that points to one or
// more components for the particular type of component.
hr = penumncfgcomp->Next(celt, &pncfgcompOther, &celtFetched);
// Retrieve a pointer to INetCfgClassSetup that enables installation
// or removal of a particular type of component.
hr = pncfgclass->QueryInterface(IID_INetCfgClassSetup,
                               (LPVOID*)&pncfgclsSetup);
// Retrieve a pointer to INetCfgComponent for an installed component.
hr = pncfgclsSetup->SelectAndInstall(hwndParent, pOboToken,
                                     &pncfgcompInstall);
// Retrieve a pointer to INetCfgComponent for an installed component.
hr = pncfgclsSetup->Install(TEXT("MS_TCPIP"), pOboToken, dwSetupFlags,
                           dwUpgradeFromBuildNo, TEXT("AnswerFile"),
                    TEXT("AnswerFileSections"), &pncfgcompInstall);
```

Process for upgrading network components

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (Service Pack 1 [SP1] and later), Microsoft Windows Server 2003, and later operating systems.

The network upgrade process migrates parameter values for network components during an operating system upgrade. The network upgrade process thus eliminates the need to reconfigure upgraded network components after the new operating system is installed.

The network upgrade process upgrades network components from Microsoft Windows NT 3.51 or Windows NT 4.0 to Microsoft Windows 2000 or later versions of the operating system. The network upgrade process does not upgrade network components from Windows 2000 to later versions of the operating system.

Vendors whose network components are not released as part of Windows 2000 or later should provide upgrade support for these components by supplying the following:

- A network migration DLL that migrates the preupgrade parameter values for one or more network components.
- A netmap.inf file that maps the preupgrade device, hardware, or compatible ID of one or more network components, to the corresponding ID in the new operating system.
- Optional custom Help message files that provide information about upgrading network components.

The network upgrade process is described in the following topics:

Customizing the Network Upgrade Process

The Network Upgrade Process

Writing a Network Migration DLL

Creating a Netmap.inf File

There are two major steps involved in testing the upgrade of network components. These are described in the following topics:

Setting Up the Test System

Running the Upgrade Test and Examining the Results

Network components whose drivers are released as part of Windows 2000 or later operating systems are automatically upgraded when the operating system is installed. No additional upgrade support is required for such components.

Customizing the Network Upgrade Process

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

System administrators can customize the network upgrade process.

To customize the network upgrade process

1. Create a directory on the system for each network component to be upgraded.
2. Copy the vendor-supplied upgrade files for each network component to the appropriate directory that you created in Step 1. These files must include a netmap.inf file. NetSetup uses the netmap.inf file to identify which network components to upgrade.
3. Create a netupg.inf file that contains an **OemNetUpgradeDirs** section and place it a directory of your choice. Each line in the **OemNetUpgradeDirs** section of the netupg.inf file specifies a path to a directory created in Step 1. Each directory specified in the netupg.inf file must contain the vendor-supplied upgrade files for the network component, including a netmap.inf file.
4. Set the NET_UPGRD_INIT_FILE_DIR environment variable to the directory that contains the netupg.inf file.

During the Winnt32 phase of the network upgrade, NetSetup locates the netupg.inf file in the directory specified by the NETUPGRD_INIT_FILE_DIR environment variable. In each directory specified in the netupg.inf file, NetSetup then locates the netmap.inf file and other vendor files for the network component to be upgraded. NetSetup processes these files to upgrade the component. For more information, see The Network Upgrade Process.

Creating a Netupg.inf File

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

The netupg.inf file contains a single section called **OemNetUpgradeDirs**. Each entry in this section specifies the complete path to a directory that contains the vendor-supplied upgrade files for a non-Microsoft-supported network component. Every network component being upgraded must have a corresponding entry in the **OemNetUpgradeDirs** section.

The following is an example of a netupg.inf file:

INF

```
[OemNetUpgradeDirs]
c:\temp\adapter1
c:\temp\adapter2
c:\temp\protocol1
c:\temp\netclient1
c:\temp\netservice1
```

Each directory specified in the **OemNetUpgradeDirs** section must contain a netmap.inf file. This file, which is provided by the vendor of the network component, maps the preupgrade device, hardware or compatible ID of a network component to the corresponding ID in the upgraded operating system.

The Network Upgrade Process

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

The network upgrade process is divided into three distinct phases, which can be briefly summarized as follows:

- **Winnt32 phase**

  Winnt32.exe calls NetSetup. NetSetup writes network component-specific information to the AnswerFile and calls any vendor-supplied network migration DLLs. The DLLs write component-specific information to the AnswerFile. Winnt32.exe copies the Microsoft Windows 2000 or later files to the system being upgraded and prepares the boot sector on the system. The system then boots into text mode.

- **Text mode phase**

  Installation messages are displayed on a blue, text-based screen. Setup performs the basic Windows 2000 or later installation. The system then boots into GUI mode.

- **GUI mode phase**

  NetSetup processes the winnt.sif file, which is also known as the AnswerFile, and installs the network components. Network migration DLLs can display a user interface in which a user or system administrator can specify parameter values for network components. Either NetSetup or a network migration DLL writes a network component's preupgrade parameter values to the Windows 2000 or later registry.

The phases of the network upgrade process are described in more detail in the following topics:

Winnt32 Phase of the Network Upgrade Process

Text Mode Phase of the Network Upgrade Process

GUI Mode Phase of the Network Upgrade Process

Winnt32 Phase of the Network Upgrade Process

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

The user or system administrator starts the upgrade process by taking either of the following actions:

- Selecting the component upgrade in the user interface that is displayed after the Windows 2000 or later CD-ROM spins up
- Selecting and running \i386\winnt32.exe on the CD-ROM

If the user or system administrator has set the NETUPGRD_INIT_FILE_DIR environment variable on the system being upgraded, NetSetup searches for a netupg.inf file in the directory specified by that variable. The netupg.inf file contains only one section: **OemNetUpgradeDirs**. Each entry in this section specifies the complete path to a directory that contains the vendor-supplied upgrade files, including a netmap.inf file, for a network component. If the NETUPGRD_INIT_FILE_DIR environment variable is not set, NetSetup (netupgrd.dll) looks for netmap.inf files in its own directory.

NetSetup reads the netmap.inf files to identify the network components that do not have built-in upgrade support. If NetSetup is running in unattended mode, it displays a wizard; however, the user or system administrator cannot use the wizard. If NetSetup is not running in unattended mode, the wizard displays a list of the network components that do not have built-in upgrade support.

Using the wizard, a user or system administrator can:

- Click **Cancel** to abort the installation of the operating system.
- Click **Next** to install the operating system without upgrading the listed network components.
- Specify the drive and directory location of vendor-supplied upgrade files for listed network components.

  NetSetup reads the netmap.inf file at the specified location and copies the vendor-supplied upgrade files at that location to a temporary directory on the system's hard disk. This temporary directory becomes the working directory for the vendor-supplied network migration DLL. NetSetup also removes any component that has a netmap.inf file from the component list in the wizard.

NetSetup generates the winnt.sif file (also known as the AnswerFile) in the Win_nt.~bt directory, which is usually located on the C: drive.

NetSetup generates the AnswerFile as follows:

1. NetSetup reads the registry of the preupgraded system to enumerate each network component. For each network component that has built-in upgrade support, NetSetup writes the information that is read from the registry to the AnswerFile.

2. For each network component that does not have built-in upgrade support, NetSetup reads the component's netmap.inf file. The netmap.inf file maps the preupgrade device, hardware, or compatible ID of a network component to the corresponding ID in the upgraded operating system. If NetSetup matches the preupgrade ID of the network component that it read from the registry with a preupgrade ID in the **OemNetAdapters**, **OemNetProtocols**, **OemNetServices**, or **OemAsyncAdapters** section of the netmap.inf file, NetSetup writes vendor-provided information for the component to the AnswerFile.

3. Using the component's operating system device, hardware, or compatible ID, NetSetup reads the **OemUpgradeSupport** section of the netmap.inf file to determine which network migration DLL to load. NetSetup then loads the network migration DLL, and calls the DLL's PreUpgradeInitialize function. The **PreUpgradeInitialize** function supplies information that the DLL uses to initialize itself.

4. NetSetup calls the DLL's DoPreUpgradeProcessing function once for each network component supported by the network migration DLL. **DoPreUpgradeProcessing** reads a network component's preupgrade parameter values from the registry and calls the NetUpgradeAddSection and NetUpgradeAddLineToSection functions to write these parameters, along with other component-specific information, to the AnswerFile. **DoPreUpgradeProcessing** can also migrate binary data associated with the preupgraded component by making appropriate entries in the AnswerFile.

5. After the AnswerFile is completely generated, NetSetup copies the vendor-supplied upgrade files to the appropriate directories and then boots into the text mode phase of the upgrade process.

Text Mode Phase of the Network Upgrade Process

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

Setup strips all comments from the AnswerFile and writes the AnswerFile to the System32 directory under the name $Winnt$.inf. Then the system boots into GUI mode setup. No network-specific processing occurs during the text mode phase.

GUI Mode Phase of the Network Upgrade Process

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

Before the Windows 2000 or later operating system is installed on the system, NetSetup reads the network-specific information that was written to the AnswerFile during the Winnt32 phase.

If a network migration DLL wrote the **InfToRunBeforeInstall** key to a component's *OEM section* in the AnswerFile, NetSetup finds the INF file and section specified by the key and processes the INF directives in this section. This section usually contains the **AddReg**, **DelReg**, **AddService**, or **DelService** directives.

After the Windows 2000 or later operating system is installed, NetSetup installs each network component detected in the system, using the default parameter values specified for the component in the component's Windows 2000 or later INF file. NetSetup then installs network components listed in the AnswerFile.

If a network component's *OEM section* in the AnswerFile contains an OemDllToLoad key, NetSetup loads the network migration DLL if the DLL is not already loaded and then calls the DLL's **PostUpgradeInitialize** function. The **PostUpgradeInitialize** function supplies the DLL with information that the DLL uses to initialize itself. NetSetup then calls the DLL's **DoPostUpgradeProcessing** function once for each network component to be upgraded by the DLL. **DoPostUpgradeProcessing** can display a user interface that allows a user to specify parameter values for a component. **DoPostUpgradeProcessing** writes any user-specified parameter values to the registry.

If the miniport driver for a network adapter required the adapter's instance ID before the upgrade, it will probably require the adapter's instance ID after the upgrade. A network migration DLL can call **HrGetInstanceGuidOfPreNT5NetCardInstance** from its **DoPostUpgradeProcessing** function to obtain the Windows 2000 or later instance GUID for a network adapter.

NetSetup starts the installed network protocols, clients, and services.

NetSetup processes the entries in the **Identification** section of the AnswerFile and tries to connect the system to the workgroup or domain specified in that section.

If the system being upgraded contains any Async adapters, Setup calls the Async class installer, which upgrades each Async adapter as follows:

- The Async class installer locates the OEM section for the Async adapter in the AnswerFile.
- From the Async adapter's OEM section, the Async class installer reads the preupgrade parameter values for the adapter. These parameter values were written by the network migration DLL for the adapter during the Winnt32 phase of the upgrade.
- The Async class installer writes the adapter's preupgrade parameter values to the Windows 2000 or later registry.

Writing a Network Migration DLL

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

A network migration DLL migrates the parameter values for one or more network components from Microsoft Windows NT 3.51 or Windows NT 4.0 to Windows 2000 or later.

A network migration DLL must:

- **Load under the preupgrade operating system (Windows NT 3.51 or Windows 4.0)**

  The DLL cannot call any functions specific to Windows 2000 or later or use any features specific to Windows 2000 or later. If the DLL runs in the postupgrade (GUI mode) phase, it must also load under Windows 2000 and later operating systems.

- **Export the PreUpgradeInitializeandDoPreUpgradeProcessingfunctions**

  If the DLL runs in the GUI mode phase, it must export the PostUpgradeInitialize and DoPostUpgradeProcessing functions, as well.

- **Make no irreversible changes during the Winnt32 phase**

  The DLL must not make any irreversible changes, such as deleting files or modifying registry keys, during this phase because a user can cancel the upgrade of a network component or the operating system. The DLL can, however, modify files in its temporary working directory, which is specified by NetSetup in the call to **PreUpgradeInitialize**.

Creating a Netmap.inf File

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

The netmap.inf file is a vendor-supplied file that resides either in a directory specified by an entry in the **OemNetUpgradeDirs** section of a netupg.inf file or in the directory that contains netupgrd.dll. The netmap.inf file:

- Maps a network component's preupgrade device ID to the component's Microsoft Windows 2000 or later device ID

- Specifies the network migration DLL that NetSetup loads

- Optionally specifies an alternative Help message file

A network component that has built-in upgrade support in Windows 2000 or later operating systems does not require a vendor-supplied netmap.inf file because these components are automatically upgraded during the installation of Windows 2000 and later operating systems.

This section includes the following topics:

Mapping IDs in a Netmap.inf File

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

A netmap.inf file contains one or more of the following top-level sections. Each section contains ID mappings for the components listed in the **Map** column.

Section	Map
OemNetAdapters	Network adapters, excluding Async adapters
OemAsyncAdapters	Async network adapters
OemNetProtocols	Network protocol drivers
OemNetServices	Network services
OemNetClients	Network clients

Each entry in a section maps a network component's preupgrade device, hardware, or compatible ID to its corresponding Windows 2000 or later ID. Each entry specifies either *one-to-one* ID mapping or *one-to-many* ID mapping. These mapping strategies are described following.

Network clients are not defined as such in Windows NT 3.51 and Windows NT 4.0; therefore, if an earlier network service becomes a network client under Windows 2000 or later, its device ID mapping must be listed in the **OemNetClients** section, not in the **OemNetServices** section.

One-to-One ID Mapping

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

An entry in an **Oem*Xxx*** section of a netmap.inf file that specifies one-to-one ID mapping has the following format:

preupgrade-ID = postupgrade-ID

For example:

C++

```
netservice=netservice_2000
```

A one-to-one ID mapping must be used for network protocols, services, and clients. Either one-to-one ID mapping or one-to-many ID mapping can be used for network adapters.

One-to-Many ID Mapping

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

A one-to-many ID mapping maps a single preupgrade ID that represents more than one network adapter. The only way to differentiate the adapters associated with a single preupgrade ID is to inspect the values under the registry key that contains the parameter values for the network adapter instance.

An entry in an **OemAdapters** or **OemAsyncAdapters** section that specifies a one-to-many ID mapping has the following format:

preupgrade-ID = *mapping-method-number*, *section-name*

where:

mapping-method-number must be 0

section-name specifies a section in the netmap.inf file that contains the mapping information

The netmap.inf file section specified by *section-name* contains the following entries:

ValueName = "*Name*"

Specifies the value that NetSetup reads under the registry key that contains the parameter values for the network adapter instance. *Name* identifies a particular network adapter.

ValueType = *Type*

Specifies the registry value type for *ValueName*. *Type* is an integer that corresponds to a specific registry type.

ValueName= *postupgrade-ID*

ValueName is the value that NetSetup reads under the registry key that contains the parameter values for the network adapter instance. *postupgrade-ID* is the Windows 2000 or later device ID for the adapter. One *ValueName* entry should be provided for each adapter type that will be upgraded. If *ValueName* is set to the keyword **ValueNotPresent** and if NetSetup finds no parameters values for the adapter instance, NetSetup uses the *postuprgrade-ID* associated with **ValueNotPresent** for the adapter.

The following example shows a one-to-many device ID mapping:

C++

```
[OemAdapters]
DATAFIREU=0, DATAFIREU

[DATAFIREU]
ValueName  = "BoardType"
ValueType  = 1
DataFireIsaU = "DATAFIRE - ISA1U"
DataFireIsa1ST= "DATAFIRE - ISA1ST"
DataFireIsa4ST= "DATAFIRE - ISA4ST"
DataFireIsaGeneric = "ValueNotPresent"
```

The **OemAdapters** section in the above example contains a single entry that identifies the preupgrade device ID of the network adapter as DATAFIREU and specifies that the **DATAFIREU** section of the netmap.inf file contains the mapping information for this adapter.

The DATAFIREU section contains the following information:

- The **ValueName** entry directs NetSetup to look for the **BoardType** value under the **Parameters** key of the network adapter instance.

- The **ValueType** entry, which is set to 1, specifies that the **BoardType** value is a DWORD.

- Each remaining value specifies a preupgrade device ID and a corresponding Windows 2000 or later ID. For example, the ID for the DataFireIsaU board type is DATAFIRE - ISA1U. The **ValueNotPresent** keyword can be specified instead of a preupgrade ID.

NetSetup performs a one-to-many ID mapping as follows:

1. NetSetup reads the specified ValueName under the registry key that contains the parameter values for the network adapter instance.

2. NetSetup attempts to match the ValueName with one of the ValueNames listed in the specified section of the netmap.inf file. If no ValueName is listed under the registry key, NetSetup attempts to find the ValueNotPresent keyword in the specified section of the netmap.inf file.

3. If NetSetup finds a match, it installs the network adapter, using the INF file that has the same name as the mapped Windows 2000 or later ID.

If the registry keys or values for an adapter instance are identical for different adapter types, it is not possible to map a single preupgrade device ID to more than one Windows 2000 or later device ID without first modifying these registry keys or values.

The most effective way of handling this situation is as follows:

1. The network migration DLL's **PreUpgradeInitialize** function modifies the registry so that the registry contains unique values for each instance of the network adapter. These unique values should indicate the adapter type.

2. The **PreUpgradeInitialize** function sets the NUA_REQUEST_ABORT_UPGRADE flag, which causes NetSetup to display a message that prompts the user to restart winnt32.exe and abort the upgrade.

3. The user aborts the upgrade and then restarts winnt32.exe. The network migration DLL can now use the unique values to map the single preupgrade device ID to more than one Windows 2000 or later device ID.

Specifying the Upgrade DLL in a Netmap.inf File

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

A netmap.inf file must have an **OemUpgradeSupport** section. For each network component to be upgraded, the **OemUpgradeSupport** section must contain an entry that has the following format:

postupgrade-ID = *network-migration-DLL*[, *Inf-file-name*]

where:

postupgrade-ID is the network component's Windows 2000 or later device ID, which was obtained by NetSetup, as described in Winnt32 Phase of the Network Upgrade Process.

network-migration-DLL is the name of the network migration DLL that NetSetup must load to upgrade the network component. Only one migration DLL can be specified in a netmap.inf file. If the netmap.inf file contains device ID mappings for more than one component, then all such components must be upgraded by the same migration DLL.

Inf-file-name is the name of the INF file that installs the network component.

Specifying Alternative Help Message Files in a Netmap.inf File

Article • 05/05/2023

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

If NetSetup fails to find the device ID mapping for a network component in any of the netmap.inf files, it lists this component on the Compatibility Report page in the wizard. Associated with each such component is a Help message file.

By default, NetSetup displays a Help message contained in the \winntupg\unsupmsg.txt file or, if an HTML browser is installed on the system, in the \winntupg\unsupmsg.htm file. You can optionally supply a custom message file that overrides the unsupmsg.txt and unsupmsg.txt message files. For example, if a vendor provides upgrade support for only some of its network components, the vendor could supply a custom Help message file that indicates that upgrade support is not provided for certain components.

An optional **OemUpgradeHelpFiles** section in a netmap.inf file specifies one or more custom Help message files. Each entry in this section has the following format:

postupgrade-ID = *text-name*, *htm-file*

where:

postupgrade-ID is the Windows 2000 or later device ID of network component

text-name is the path and name of the text version of the custom Help message file

htm-file is the path and name of the HTML version of the custom Help message file.

If a full path name is not specified in *text-name* or *htm-file*, the specified path is appended to the i386 directory--for example: \i386\mydirectory\myfile.txt.

The following example of a netmap.inf file contains an **OemUpgradeHelpFiles** section.

INF

```
[Version]
signature="$Windows NT$

[OemNetProtocols]
Protocol1=Protoco1_2000
Protocol2=Protocol2_2000
```

```
[OemUpgradeSupport]
Protocol1=NotSupported
Protocol2=abc_upgrade.dll, abc.inf

[OemUpgradeHelpFiles]
Protoco11=helpmsg.txt, helpmsg.htm
```

Even though this sample netmap.inf file does not provide upgrade support for Protocol1, it provides a device ID mapping for Protocol1 in the **OemNetProtocols** section. This mapping specifies a Windows 2000 device ID for Protocol1. The Windows 2000 device ID is required to associate custom Help message files with a network component.

Notice that the keyword **NotSupported** is assigned to Protocol1 in the **OemUpgradeHelpFiles** section. This keyword indicates that there is no need to load a migration DLL to upgrade Protocol1.

In the **OemUpgradeHelpFiles** section of the previous example, the Protoco11=helpmsg.txt, helpmsg.htm entry specifies two custom Help message files for Protocol1. The custom Help message contained in these files could indicate, for example, that the vendor does not support the upgrade of Protocol1 and that the user must separately upgrade Protocol1 to Protocol2 before attempting to upgrade the system to Windows 2000 or later.

Setting Up the Test System

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

Before you upgrade network components, make sure that the network components to be upgraded are correctly installed and configured.

To set up the test system

1. Create one partition for the preupgrade operating system and another partition for the Microsoft Windows 2000 or later operating system. **Note** Do not install the preupgrade operating system and the upgrade operating system in the same partition. If the preupgrade operating system and Windows 2000 or later are installed in the same partition, they will share the same Program Files directory.

2. On the test system, boot an operating system build other than the one to be upgraded. Then copy the entire partition to be upgraded, except for the pagefile.sys file, into a back-up directory. There is no need to copy the pagefile.sys file, since it is created on the start-up of Windows 2000 or later.

   This method of creating a back-up installation is preferable to creating a disk-image program, because it allows you to use **xcopy**, which takes less time to copy files than a disk-image program. You can repeat an upgrade test by simply copying the contents of the back-up partition into a new partition to be upgraded; it is not necessary to reinstall the preupgrade operating system.

3. Create a test directory for storing the network migration DLL and the netmap.inf file, and then copy these files to the test directory.

4. Create another directory for storing the Windows 2000 or later files required for the Winnt32 upgrade phase.

5. Insert the Windows 2000 or later Driver Development Kit (DDK) CD-ROM that contains the checked build of Windows 2000 or later. From the \i386 directory on the CD-ROM, copy the following files to the back-up directory (Step 2):

   - winnt32.exe
   - winnt32u.dll
   - pidgen.dll
   - wetuplog.*

6. Create an upgrade directory named winntupg. Copy the files in the \i386\winntupg directory on the CD-ROM to the winntupg directory on the test system.

7. Enable the debugger on the text system or start debugmon.exe, which is included with the Resource Kit for Windows 2000 or later operating systems. Then copy a netcfg.ini file to %windir%. The netcfg.ini file enables debug tracing.

   The following is a sample netcfg.ini file:

   INI

   ```ini
   [DebugFlags]
   BreakOnAddLegacy=0
   BreakOnAlloc=0
   BreakonDoUnattend=0
   BreakonDwrefProblem=0
   BreakOnError=0
   BreakOnHr=0
   BreakOnHrInteraction=0
   BreakOnIteration=0
   BreakOnNetInstall=0
   BreakOnWizard=0
   DisableTray=0
   DumpLeaks=0
   DumpNetCfgTree=0
   NetShellBreakOnInit=0
   ShowIngnoreErrors=0
   ShowProcessAndThreadIds=0
   SkipLanEnum=0
   TracingTimeStamps=0

   [Default]
   OutputToDebug=1

   [EsLocks]
   OutputToDebug=0

   [ShellViewMsgs]
   OutputToDebug=0

   [OptErrors]
   OutputToDebug=0
   ```

Running the Upgrade Test and Examining the Results

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

Before you upgrade the system to Windows 2000 or later, note the parameter values in the registry for each network component to be upgraded.

To run the upgrade test

1. Make sure that the CD-ROM that contains the checked build of Windows 2000 or later is in the CD-ROM drive.

2. Run winnt32.exe on the test system. For example, use the following command to run winnt32.exe on an Intel-based system with the CD-ROM in drive O:

   CMD

   ```
   winnt32.exe /s:o\i386
   ```

3. After Windows 2000 or later is installed, verify that the upgraded network component's parameters have been correctly migrated to the new operating system.

Examining the AnswerFile

Article • 12/15/2021

Note Vendor-supplied network upgrades are not supported in Microsoft Windows XP (SP1 and later), Microsoft Windows Server 2003, and later operating systems.

Immediately before the "Setup is Copying Files" progress bar is displayed on a system being upgraded, the AnswerFile is created. NetSetup and vendor-supplied network migration DLLs create sections in the AnswerFile and then write entries to these sections during the Winnt32 upgrade phase.

You can examine the AnswerFile by copying c:\win_nt.~bt\winnt.sif to %TEMP%. After the AnswerFile has been copied, you can click **Cancel** to cancel file copying. You do not have to wait until file copying is finished.

The following table lists the top-level sections in the AnswerFile and the corresponding entries that each section contains that pertain to network components:

Section	Entries Contained
NetAdapters	Network adapters, including ISDN adapters
AsyncAdapters	Async adapters
NetProtocols	Network protocols
NetServices	Network services
NetClients	Network clients

Note **NetClient** components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

For each network component that it finds during the Winnt32 phase, NetSetup writes an entry to the appropriate top-level section of the AnswerFile. Each entry has the following format:

params.*postupgrade-ID*

The *postupgrade-ID* entry is the Windows 2000 or later device ID that NetSetup obtained from the netmap.inf file for the component.

Each entry specifies the name of the parameters section for that component in the AnswerFile. For example, if a component's Windows 2000 or later device ID is netadapter2, its entry in the **NetAdapters** section is **params.netadapter2**. The top-level sections and the parameter sections in an AnswerFile are not visible to a network migration DLL.

To the parameters section name for a component, NetSetup adds the extension **OemSection** to create the *OEM-section* name for the component. For example, if the parameters section for a component is params.netadapter2, the *OEM-section* name for the component is params.netadapter2.OemSection. NetSetup passes the *OEM-section* name as the *szSectionName* parameter to the DoPreUpgradeProcessing function supplied by the network migration DLL for the component. The **DoPreUpgradeProcessing** function calls the NetUpgradeAddSection function to create the *OEM-section* for a component in the AnswerFile. The **DoPreUpgradeProcessing** function then calls the NetUpgradeAddLineToSection to add component-specific information to the *OEM-section*.

The following portion of an AnswerFile shows the sections and entries for a network adapter whose Windows 2000 or later device ID is **adapter2**:

```
INF
```

```
[NetAdapter]              ;top-level adapters section
adapter2=params.adapter2      ;entry for adapter2
[params.adapter2]          ;parameters section for adapter2
InfID=adapter2            ;Windows 2000 or later device ID
OemSection=params.adapter2.OemSection  ;Identifies the OemSection

[params.adapter2.OemSection]  ;OemSection created by migration DLL
InfToRunAfterInstall="", adapter2.SectionToRun ;Written by DLL

[adapter2.SectionToRun]      ;Section created by migration DLL
AddReg=adapter2.SectionToRun.AddReg ;AddReg directive

[adapter2.SectionToRun.AddReg] ;AddReg section created by DLL
HKR,0\0,IsdnPhoneNumber,0,"111-1111" ;AddReg entries written by DLL
HKR,0\1,IsdnPhoneNumber,0,"222-2222"
HKR,0\0,IsdnSpid,0,"111"
HKR,0\1,IsdnSpid,0,"222"
HKR,0,IsdnSwitchType,0x00010001,1
```

During the GUI mode phase, NetSetup detects the InfToRunAfterInstall key written by the migration DLL to the **params.adapter2.OemSection** of the example AnswerFile. As directed by this key, NetSetup processes the **adapter2.SectionToRun.AddReg** section. The **adapter2.SectionToRun.AddReg** section directs NetSetup to add parameter values to adapater2's instance key in the Windows 2000 or later registry. These parameter values should match the preupgrade parameter values that the migration DLL read from adapter2's the registry during the Winnt32 phase of the upgrade.

If a network migration DLL is to be loaded during the GUI mode phase, its DoPreUpgradeProcessing function sets the NUA_LOAD_POST_UPGRADE flag. This flag causes NetSetup to write the **OemDllToLoad** entry to the component's parameters section in the AnswerFile. The **OemDllToLoad** entry causes NetSetup to load the migration DLL for the component during the GUI mode phase.

The following example shows the AnswerFile sections and entries for a component whose network migration DLL is loaded during the GUI mode phase:

INF

```
[NetAdapter]              ;top-level adapters section
adapter2=params.adapter2      ;entry for adapter2
[params.adapter2]          ;parameters section for adapter2
InfID=adapter2            ;postupgrade device ID
OemSection=params.adapter2.OemSection;Identifies the OemSection
OemDllToLoad=c:\temp\oem0001\migration.dll
```

Note the **OemDllToLoad** entry in the **params.adapter2** section. Also note that the migration DLL did not create a **params.adapter2.OemSection**. When the migration DLL is to be loaded during the GUI mode phase, it typically does not write an **InfToRunAfterInstall** key to the AnswerFile. The DLL performs the postinstallation upgrade; therefore, it does not need to create an *Oem-Section* name that contains directives for NetSetup to perform during the GUI mode phase.

NDIS general statistics OIDs

Article • 02/06/2024

A driver should respond to a query of a statistics OID with complete information so that the driver can supply the operating system and applications with information that they need to monitor network status, respond to security issues, and diagnose problems. If statistics counters are in hardware, the driver should read the appropriate statistics value from hardware each time that a statistics OID is queried.

Note: General statistics OIDs count all traffic through the network adapter including Network Direct Kernel (NDK) traffic. NDK statistics may be counted separately with OID_NDK_STATISTICS.

Miniport driver support for 64-bit counters

All one-Gbps and faster miniport drivers must support 64-bit counters for the following statistics OIDs. In addition, Microsoft recommends that all 100Mbps and faster miniport drivers support 64-bit counters for the following statistics OIDs:

- OID_GEN_STATISTICS
- OID_GEN_BYTES_RCV
- OID_GEN_BYTES_XMIT
- OID_GEN_RCV_DISCARDS
- OID_GEN_XMIT_DISCARDS
- OID_GEN_XMIT_OK
- OID_GEN_RCV_OK
- OID_GEN_XMIT_ERROR
- OID_GEN_RCV_ERROR
- OID_GEN_RCV_NO_BUFFER
- OID_GEN_DIRECTED_BYTES_XMIT
- OID_GEN_DIRECTED_FRAMES_XMIT
- OID_GEN_MULTICAST_BYTES_XMIT
- OID_GEN_MULTICAST_FRAMES_XMIT
- OID_GEN_BROADCAST_BYTES_XMIT
- OID_GEN_BROADCAST_FRAMES_XMIT
- OID_GEN_DIRECTED_BYTES_RCV
- OID_GEN_DIRECTED_FRAMES_RCV
- OID_GEN_MULTICAST_BYTES_RCV
- OID_GEN_MULTICAST_FRAMES_RCV
- OID_GEN_BROADCAST_BYTES_RCV

- OID_GEN_BROADCAST_FRAMES_RCV
- OID_GEN_RCV_CRC_ERROR
- OID_GEN_TRANSMIT_QUEUE_LENGTH
- OID_GEN_INIT_TIME_MS
- OID_GEN_RESET_COUNTS
- OID_GEN_MEDIA_SENSE_COUNTS

Miniport drivers can also support 64-bit counters for other statistics OIDs, such as OIDs that indicate transmit or receive errors.

System support for 64-bit counters is available in Windows XP and later operating systems.

Note: If an NDIS MUX driver exposes multiple miniport instances, querying the following general statistics OIDs should return data specific to that miniport instance. For example, if a MUX driver implements virtual local area network (VLAN) filtering and exposes one miniport per VLAN, the statistics values returned from the following OIDs are expected to be per VLAN.

- OID_GEN_STATISTICS
- OID_GEN_RCV_OK
- OID_GEN_XMIT_OK

NDIS network interface OIDs

Article • 03/14/2023

NDIS network interface object identifiers (OIDs) provide information about network interfaces to support the MIB (RFC 2863).

NDIS interface providers must support these OIDs. Drivers that are not registered interface providers should not support the OIDs in this section.

NDIS calls the ProviderQueryObject function to make a query request for information from the interface provider. The *ObjectId* parameter of this function contains the object identifier. The interface provider registered *ProviderQueryObject* when it called the NdisIfRegisterProvider function to register as an interface provider.

The handle at the *ProviderIfContext* parameter of the *ProviderQueryObject* function identifies the network interface. This handle was provided to NDIS when the interface provider called the NdisIfRegisterInterface function to register the interface. The *pOutputBuffer* parameter of the *ProviderQueryObject* function contains the result of the OID request.

For more information about NDIS network interface OIDs, see NDIS 6.0 Network Interfaces.

This section describes the following NDIS network interface OIDs:

- OID_GEN_ALIAS
- OID_GEN_ADMIN_STATUS
- OID_GEN_OPERATIONAL_STATUS
- OID_GEN_PROMISCUOUS_MODE
- OID_GEN_XMIT_LINK_SPEED
- OID_GEN_RCV_LINK_SPEED
- OID_GEN_UNKNOWN_PROTOS
- OID_GEN_DISCONTINUITY_TIME
- OID_GEN_LAST_CHANGE
- OID_GEN_INTERFACE_INFO
- OID_GEN_MEDIA_CONNECT_STATUS_EX
- OID_GEN_LINK_SPEED_EX
- OID_GEN_MEDIA_DUPLEX_STATE
- OID_TUNNEL_INTERFACE_RELEASE_OID
- OID_TUNNEL_INTERFACE_SET_OID

Mandatory OIDs for miniport drivers

Article • 12/15/2021

The following table lists the OIDs that are mandatory for all miniport drivers. Your miniport driver will be required to support additional OIDs, depending on its NDIS version and the services that it supports, such as:

- Connection-Oriented Objects
- CoNDIS
- Ethernet statistics OIDs
- Header-Data Split OIDs
- Hyper-V Extensible Switch OIDs
- IPsec Offload Version 2 OIDs
- MB OIDs
- Native 802.11 Wireless LAN OIDs
- NDIS TCP/IP Offload OIDs
- NDKPI OIDs
- Operational Power Management OIDs
- Overview of Receive Filter OIDs
- Receive Filter OIDs
- Receive Side Scaling OIDs
- Remote NDIS OIDs
- Required and Optional OIDs for Power Management
- SR-IOV OIDs
- Statistical Power Management OIDs
- Task Offload Objects
- VMQ OIDs

In the "O/M" columns in the table:

- "M" means "mandatory" and "O" means "optional."
- "N/A" in the "O/M for Query" column means that NDIS handles the OID query request and does not send it to the miniport driver, so the miniport driver only needs to support the OID set request.
- If there is no entry in the "O/M for Query" column, this OID is a set-only OID.
- If there is no entry in the "O/M for Set" column, this OID is a query-only OID.

OID	O/M for Query	O/M for Set	Comments

OID	O/M for Query	O/M for Set	Comments
OID_GEN_CURRENT_LOOKAHEAD	N/A	M	NDIS handles query and unsuccessful Set requests for the miniport driver. NDIS sends valid Set requests to the miniport driver. You can obtain the same information with OID_GEN_RECEIVE_BLOCK_SIZE.
OID_GEN_CURRENT_PACKET_FILTER	N/A	M	Query is not mandatory. Set is mandatory.
OID_GEN_INTERRUPT_MODERATION	M	M	
OID_GEN_LINK_PARAMETERS		M	
OID_GEN_MAXIMUM_TOTAL_SIZE	M		There is no other way to get this information.
OID_GEN_RCV_OK	M		NDIS does not handle this OID for miniport drivers and OID_GEN_STATISTICS does not include this information. **Note**: Statistics OIDs are mandatory unless NDIS handles them.
OID_GEN_RECEIVE_BLOCK_SIZE	M		NDIS does not handle this OID for miniport drivers.
OID_GEN_RECEIVE_BUFFER_SPACE	M		There is no other way to get this information.
OID_GEN_STATISTICS	M		
OID_GEN_TRANSMIT_BLOCK_SIZE	M		There is no other way to get this information.
OID_GEN_TRANSMIT_BUFFER_SPACE	M		There is no other way to get this information.
OID_GEN_VENDOR_DESCRIPTION	M		There is no other way to get this information.
OID_GEN_VENDOR_DRIVER_VERSION	M		There is no other way to get this information.
OID_GEN_VENDOR_ID	M		There is no other way to get this information. Independent hardware vendor's filter drivers or intermediate drivers might query this OID.

OID	O/M for Query	O/M for Set	Comments
OID_GEN_XMIT_OK	M		NDIS does not handle this OID and OID_GEN_STATISTICS does not include this information. **Note**: Statistics OIDs are mandatory unless NDIS handles them.

Ethernet statistics OIDs

Article • 12/15/2021

The following table summarizes the OIDs used to get Ethernet statistics for Network Interface Controllers (NICs).

Length	Query	Set	Name
4	O		OID_802_3_RCV_OVERRUN
4	O		OID_802_3_XMIT_DEFERRED
4	O		OID_802_3_XMIT_HEARTBEAT_FAILURE
4	O		OID_802_3_XMIT_LATE_COLLISIONS
4	O		OID_802_3_XMIT_MAX_COLLISIONS
4	O		OID_802_3_XMIT_TIMES_CRS_LOST
4	O		OID_802_3_XMIT_UNDERRUN

ⓘ Note

The following OIDs are obsolete in NDIS 6.0 and later:

- OID_802_3_RCV_ERROR_ALIGNMENT
- OID_802_3_XMIT_MORE_COLLISIONS
- OID_802_3_XMIT_ONE_COLLISION

General operational OIDs for connection-oriented miniport drivers

Article • 12/15/2021

The following table summarizes the OIDs used to get or set the general operational characteristics of connection-oriented miniport drivers and/or their NICs.

Tip

A connection-oriented miniport driver handles such requests in its **MiniportCoOidRequest** callback function.

In this table, M indicates an OID is mandatory, while O indicates it is optional.

Length	Query	Set	Name
2	M		OID_GEN_CO_DRIVER_VERSION
8	O		OID_GEN_CO_GET_NETCARD_TIME
8	O		OID_GEN_CO_GET_TIME_CAPS
4	M		OID_GEN_CO_HARDWARE_STATUS
4	M		OID_GEN_CO_LINK_SPEED
4	M		OID_GEN_CO_MAC_OPTIONS
4	M		OID_GEN_CO_MEDIA_CONNECT_STATUS
Arr(4)	M		OID_GEN_CO_MEDIA_IN_USE
Arr(4)	M		OID_GEN_CO_MEDIA_SUPPORTED
4	M		OID_GEN_CO_MINIMUM_LINK_SPEED
4		M	OID_GEN_CO_PROTOCOL_OPTIONS
Arr	O		OID_GEN_CO_SUPPORTED_GUIDS
Arr(4)	M		OID_GEN_CO_SUPPORTED_LIST
Var.	M		OID_GEN_CO_VENDOR_DESCRIPTION
4	M		OID_GEN_CO_VENDOR_DRIVER_VERSION
4	M		OID_GEN_CO_VENDOR_ID

General statistics OIDs for connection-oriented miniport drivers

Article • 05/20/2024

The following table summarizes the OIDs used to get or set the general statistics characteristics of connection-oriented miniport drivers and/or their NICs.

💡 Tip

A connection-oriented miniport driver handles such requests in its MiniportCoOidRequest callback function.

In this table, M indicates an OID is mandatory, while O indicates it is optional.

Expand table

Length	Query	Set	Name
4 or 8	O		OID_GEN_CO_BYTES_RCV
4 or 8	O		OID_GEN_CO_BYTES_XMIT
4 or 8	O		OID_GEN_CO_BYTES_XMIT_OUTSTANDING
4 or 8	O		OID_GEN_CO_NETCARD_LOAD
4 or 8	O		OID_GEN_CO_RCV_CRC_ERROR
4 or 8	M		OID_GEN_CO_RCV_PDUS_ERROR
4 or 8	M		OID_GEN_CO_RCV_PDUS_NO_BUFFER
4 or 8	M		OID_GEN_CO_RCV_PDUS_OK
4 or 8	O		OID_GEN_CO_TRANSMIT_QUEUE_LENGTH
4 or 8	M		OID_GEN_CO_XMIT_PDUS_ERROR
4 or 8	M		OID_GEN_CO_XMIT_PDUS_OK

Miniport driver support for 64-bit counters

All one-Gbps and faster connection-oriented miniport drivers must support 64-bit counters for the following statistics OIDs. In addition, Microsoft recommends that all

100Mbps and faster connection-oriented miniport drivers support 64-bit counters for the following statistics OIDs:

OID_GEN_CO_XMIT_PDUS_OK

OID_GEN_CO_RCV_PDUS_OK

OID_GEN_CO_BYTES_XMIT

OID_GEN_CO_BYTES_RCV

Such miniport drivers can also support 64-bit counters for other statistics OIDs, such as OIDs that indicate transmit or receive errors.

System support for 64-bit counters is available in Windows XP and later versions.

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

OIDs for connection-oriented call managers and clients

Article • 12/15/2021

The following table summarizes the OIDs that connection-oriented clients can send to call managers or MCM drivers and that call managers or MCM drivers can send to connection-oriented clients.

In this table, M indicates an OID is mandatory, while O indicates it is optional.

Length	Query	Set	Name
Varies		O	OID_CO_ADD_ADDRESS
Varies		O	OID_CO_ADD_PVC
0		O	OID_CO_ADDRESS_CHANGE
0		M	OID_CO_AF_CLOSE
Varies		O	OID_CO_DELETE_ADDRESS
Varies		O	OID_CO_DELETE_PVC
Varies	O		OID_CO_GET_ADDRESSES
			OID_CO_GET_CALL_INFORMATION
0		O	OID_CO_SIGNALING_DISABLED
0		O	OID_CO_SIGNALING_ENABLED

TAPI extension OIDs for connection-oriented NDIS

Article • 03/14/2023

The following table summarizes OIDs that allow TAPI calls to be made over connection-oriented media. Connection-oriented clients send these OIDs to call managers or integrated miniport call manager (MCM) drivers.

In this table, M indicates an OID is mandatory, while O indicates it is optional.

Length	Query	Set	Name
Varies	O		OID_CO_TAPI_ADDRESS_CAPS
Sizeof(CO_TAPI_CM_CAPS)	O		OID_CO_TAPI_CM_CAPS
Varies	O		OID_CO_TAPI_GET_CALL_DIAGNOSTICS
Varies	O		OID_CO_TAPI_LINE_CAPS
Varies	O		OID_CO_TAPI_TRANSLATE_NDIS_CALLPARAMS
Varies	O		OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS
Varies	O		OID_CO_TAPI_TRANSLATE_TAPI_SAP

In its call to NdisCoRequest, the client that queries any of the TAPI extension OIDs must specify an *NdisAfHandle* that identifies the address family to which the request applies. The client can specify an *NdisVcHandle* that identifies the virtual connection (VC) to which the request applies. From this VC handle, the call manager or MCM driver may be able to derive the particular line and perhaps the address to which the request applies.

NDIS_STATUS_MEDIA_CONNECT

Article • 03/14/2023

The NDIS_STATUS_MEDIA_CONNECT status indicates that the status of a device's network connection has changed from disconnected to connected. For example, a device connects when it comes within range of an access point (for a wireless device) or when the user connects the device's network cable.

Remarks

NDIS translates NDIS_STATUS_MEDIA_CONNECT status indications to **NDIS_STATUS_LINK_STATE** status indications for overlying NDIS 6.0 drivers.

NDIS 5.*x* and earlier miniport drivers indicate an **NDIS_STATUS_MEDIA_DISCONNECT** status when a miniport driver determines that the network connection has been lost. When the connection is restored, the driver indicates an NDIS_STATUS_MEDIA_CONNECT status.

For more information about NDIS_STATUS_MEDIA_CONNECT, see Indicating Connection Status (NDIS 5.1) and Media Status Indications for 802.11 Networks.

Requirements

Version	Not supported in NDIS 6.0 and later (use **NDIS_STATUS_LINK_STATE** instead). Supported only for NDIS 5.1 drivers in Windows Vista and Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_LINK_STATE

NDIS_STATUS_MEDIA_DISCONNECT

NDIS_STATUS_MEDIA_DISCONNECT

Article • 03/14/2023

The NDIS_STATUS_MEDIA_DISCONNECT status indicates that the status of a network connection has changed from connected to disconnected. For example, the network device loses the connection because it is out of range (for a wireless device), or the user unplugs the device's network cable.

Remarks

NDIS translates NDIS_STATUS_MEDIA_DISCONNECT status indications to **NDIS_STATUS_LINK_STATE** status indications for overlying NDIS 6.0 drivers.

NDIS 5.*x* and earlier miniport drivers indicate an **NDIS_STATUS_MEDIA_CONNECT** status when the connection is restored.

For more information about NDIS_STATUS_MEDIA_DISCONNECT, see Indicating Connection Status (NDIS 5.1) and Media Status Indications for 802.11 Networks.

Requirements

Version	Not supported in NDIS 6.0 and later (use **NDIS_STATUS_LINK_STATE** instead). Supported only for NDIS 5.1 drivers in Windows Vista and Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_LINK_STATE

NDIS_STATUS_MEDIA_CONNECT

NDIS_STATUS_RESET_START

Article • 03/14/2023

The NDIS_STATUS_RESET_START status indicates that a miniport adapter is being reset.

Remarks

Miniport drivers should not call the **NdisMIndicateStatusEx** function to signal the start and finish of each reset operation because NDIS notifies overlying drivers when a reset operation begins and ends.

A miniport driver resets a miniport adapter when NDIS calls the miniport driver's *MiniportResetEx* function. NDIS calls the *ProtocolStatusEx* function of each bound protocol and intermediate driver and the *FilterStatus* function of the overlying filter modules with a status of NDIS_STATUS_RESET_START. When the miniport driver completes the reset, NDIS notifies the overlying drivers with a status of **NDIS_STATUS_RESET_END**.

When a protocol driver receives an NDIS_STATUS_RESET_START status indication, it should:

- Hold any data that is ready to transmit until its *ProtocolStatusEx* function receives an NDIS_STATUS_RESET_END status indication.

- Not make any NDIS calls that are directed to the underlying miniport driver, except calls to return resources such as received data buffers with the **NdisReturnNetBufferLists** function.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ndis.h (include Ndis.h)

See also

FilterStatus

MiniportResetEx

NDIS_STATUS_RESET_END

NdisMIndicateStatusEx

NdisReturnNetBufferLists

ProtocolStatusEx

NDIS_STATUS_RESET_END

Article • 03/14/2023

The NDIS_STATUS_RESET_END status indicates that a miniport adapter reset operation is complete.

Remarks

Miniport drivers should not call the NdisMIndicateStatusEx function to signal the start and finish of each reset operation because NDIS notifies overlying drivers when a reset operation begins and ends.

When a miniport driver starts a reset operation, NDIS notifies the overlying drivers with an NDIS_STATUS_RESET_START status indication.

After a bound protocol driver receives an NDIS_STATUS_RESET_END status indication, the protocol driver can resume sending data and making OID requests.

After an overlying filter or intermediate driver receives an NDIS_STATUS_RESET_END status indication, the driver can resume sending data and making OID requests to overlying drivers.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_RESET_START

NdisMIndicateStatusEx

NDIS_STATUS_MEDIA_BUSY

Article • 03/14/2023

The NDIS_STATUS_MEDIA_BUSY status indicates that the IRDA media is busy.

Remarks

For more information about NDIS_STATUS_MEDIA_BUSY, see OID_IRDA_MEDIA_BUSY.

Requirements

Version	Not supported in NDIS 6.0 and later. Supported only for NDIS 5.1 drivers in Windows Vista and Windows XP.
Header	Ndis.h (include Ndis.h)

See also

OID_IRDA_MEDIA_BUSY

NDIS_STATUS_MEDIA_SPECIFIC_INDICATION

Article • 03/14/2023

The NDIS_STATUS_MEDIA_SPECIFIC_INDICATION status indicates a media-specific status.

Remarks

Miniport drivers make media-specific status indications by calling the NdisMIndicateStatusEx function with the **StatusCode** member of the NDIS_STATUS_INDICATION structure set to NDIS_STATUS_MEDIA_SPECIFIC_INDICATION. The **StatusBuffer** member of this structure points to a driver-allocated buffer. The buffer contains data in a format that is specific to the status indication that is identified in the **StatusCode** member.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NdisMIndicateStatusEx

NDIS_STATUS_LINK_SPEED_CHANGE

Article • 03/14/2023

The NDIS_STATUS_LINK_SPEED_CHANGE status indicates a link speed change.

Remarks

NDIS translates NDIS_STATUS_LINK_SPEED_CHANGE status indications to **NDIS_STATUS_LINK_STATE** status indications for overlying NDIS 6.0 drivers. When NDIS receives an NDIS_STATUS_LINK_SPEED_CHANGE status, NDIS issues an OID query request of OID_GEN_LINK_SPEED. NDIS uses the results of the OID_GEN_LINK_SPEED query to issue an NDIS_STATUS_LINK_STATE status to overlying NDIS 6.0 drivers.

The NDIS 5.*x* or earlier miniport driver supplies a DWORD-type value at the *StatusBuffer* parameter of the **NdisMIndicateStatus** function. For more information about NDIS_STATUS_LINK_SPEED_CHANGE, see OID_IRDA_RATE_SNIFF.

Requirements

Version	Not supported in NDIS 6.0 and later (use **NDIS_STATUS_LINK_STATE** instead). Supported only for NDIS 5.1 drivers in Windows Vista and Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_LINK_STATE

NdisMIndicateStatus

OID_GEN_LINK_SPEED

OID_IRDA_RATE_SNIFF

NDIS_STATUS_LINK_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_LINK_STATE status indication to notify NDIS and overlying drivers that there has been a change in the physical characteristics of a medium.

Remarks

Overlying drivers should not use the OID_GEN_LINK_STATE OID to determine the link state. Instead, use the NDIS_STATUS_LINK_STATE status indication for link state updates.

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains the **NDIS_LINK_STATE** structure. This structure specifies the physical state of the medium.

Miniport drivers should avoid sending the NDIS_STATUS_LINK_STATE status indication if there have been no changes in the physical state of the medium. However, avoiding this status indication is not a requirement.

If a miniport adapter transitions to a low power state, NDIS 6.0 and later miniport drivers should indicate a connection status of **MediaConnectStateUnknown**. When the miniport adapter transitions back to the working power state, the miniport driver should indicate a status of **MediaConnectStateConnected** after the link has been re-established. NDIS 6.30 miniport drivers should indicate **MediaConnectStateUnknown** during a low power transition only when a wake on link change and selective suspend are disabled. In other words, a miniport driver must indicate a connection state of **MediaConnectStateUnknown** during a low power transition, if it is impossible to detect and wake on a connection state change from a low power state.

NDIS might not pass a status indication to overlying drivers if there are no changes in the link state as specified in the previously indicated link state. However, this behavior is not guaranteed. Overlying drivers that receive this status indication must determine which characteristics of the medium, if any, have changed.

If an overlying driver is an NDIS 5.*x* or earlier protocol driver, NDIS translates the NDIS_STATUS_LINK_STATE status indication to appropriate NDIS 5.1 status indications. NDIS indicates link speed changes with the **NDIS_STATUS_LINK_SPEED_CHANGE** status indication. NDIS indicates changes in the connection state with **NDIS_STATUS_MEDIA_CONNECT** and **NDIS_STATUS_MEDIA_DISCONNECT** status indications.

NDIS also translates the NDIS 5.*x* miniport driver status for overlying NDIS 6.0 and later drivers. NDIS uses status indications or media state changes that NDIS identified in an NDIS 5.*x* OID query to create NDIS_STATUS_LINK_STATE status indications. NDIS performs the following translations:

- The **NDIS_STATUS_MEDIA_CONNECT** status indication is translated to **MediaConnectStateConnected** in the **NDIS_LINK_STATE** structure.

- The **NDIS_STATUS_MEDIA_DISCONNECT** status indication is translated to **MediaConnectStateDisconnected** in the **NDIS_LINK_STATE** structure.

- The **NDIS_STATUS_LINK_SPEED_CHANGE** status indication and the OID_GEN_LINK_SPEED OID are used to generate the link speed status.

For more information about link status, see OID_GEN_LINK_STATE.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_LINK_STATE

NDIS_STATUS_INDICATION

NDIS_STATUS_LINK_SPEED_CHANGE

NDIS_STATUS_MEDIA_CONNECT

NDIS_STATUS_MEDIA_DISCONNECT

OID_GEN_LINK_SPEED

OID_GEN_LINK_STATE

NDIS_STATUS_PORT_STATE

Article • 03/14/2023

Miniport drivers that support NDIS ports use the NDIS_STATUS_PORT_STATE status indication to indicate changes in the state of an NDIS port.

Remarks

Miniport drivers must set the port number in the **PortNumber** member of the NDIS_STATUS_INDICATION structure. The **StatusBuffer** member of this structure contains a pointer to an NDIS_PORT_STATE structure.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_PORT_STATE

NDIS_STATUS_INDICATION

NDIS_STATUS_OPER_STATUS

Article • 03/14/2023

The NDIS_STATUS_OPER_STATUS status indicates the current operational state of an NDIS network interface to overlying drivers.

Remarks

NDIS generates this status indication; NDIS miniport drivers should not generate this status indication.

NDIS supplies an NDIS_OPER_STATE structure in the **StatusBuffer** member of the NDIS_STATUS_INDICATION structure.

The **StatusBufferSize** member of the NDIS_STATUS_INDICATION structure is set to sizeof(NDIS_OPER_STATE).

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_OPER_STATE

NDIS_STATUS_INDICATION

NDIS_STATUS_NETWORK_CHANGE

Article • 03/14/2023

The NDIS_STATUS_NETWORK_CHANGE status indicates a network change to allow overlying drivers to initiate renegotiation of network addresses.

Remarks

NDIS miniport drivers can generate this status indication to request the overlying protocol drivers to renegotiate layer three addresses.

NDIS generates NDIS_STATUS_NETWORK_CHANGE status indications for the older 802.1X wireless miniport drivers that emulate 802.3. These miniport drivers report a media type of **NdisMedium802_3** and a physical media type of **NdisPhysicalMediumWirelessLan**. When such a miniport driver generates an NDIS_STATUS_MEDIA_CONNECT status indication and the associated miniport adapter is in a connected state, NDIS generates an NDIS_STATUS_NETWORK_CHANGE status indication for the miniport adapter.

NDIS 6.0 and later miniport drivers should generate the NDIS_STATUS_NETWORK_CHANGE status indication only after they are ready to handle network data. For example, in native 802.11, this status indication is generated after authentication is completed successfully and full layer two connectivity is achieved.

Note Although the media-connected state is not precisely defined, this state can be loosely defined as - the state in which the miniport adapter is able to transmit and receive network data. Media-connected is not directly related to link authentication status. The native WiFi 802.3 interface is unable to send or receive packets until after the link is authenticated. In this case, the media-connected state is coincident with the link-authenticated state in native 802.11.

NDIS supplies one of the following NDIS_NETWORK_CHANGE_TYPE values in the **StatusBuffer** member of the NDIS_STATUS_INDICATION structure:

NdisPossibleNetworkChange
The miniport driver detected that there might be a network change. In this case, the overlying protocols must detect the network change, if any, and renegotiate the addresses if necessary.

NDIS also uses this value when it generates NDIS_STATUS_NETWORK_CHANGE status indications for older 802.1X wireless miniport drivers that emulate 802.3. However, NDIS uses **NdisNetworkChangeFromMediaConnect** instead of **NdisPossibleNetworkChange** when it translates the same event for Windows Management Instrumentation (WMI).

NdisDefinitelyNetworkChange
The miniport driver detected that there is a network change, so the overlying protocols must renegotiate the addresses.

NdisNetworkChangeFromMediaConnect
An older 802.1X wireless miniport driver that emulates 802.3 generated an NDIS_STATUS_MEDIA_CONNECT status indication when it was in a connected state. This value is used in the WMI event notification for GUID_NDIS_STATUS_NETWORK_CHANGE. **NdisNetworkChangeFromMediaConnect** is not used in the NDIS_STATUS_NETWORK_CHANGE status indication.

The **StatusBufferSize** member of the NDIS_STATUS_INDICATION structure is set to sizeof(NDIS_NETWORK_CHANGE_TYPE).

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NDIS_STATUS_MEDIA_CONNECT

NDIS_STATUS_PACKET_FILTER

Article • 03/14/2023

The NDIS_STATUS_PACKET_FILTER status indicates a packet filter change to overlying drivers. NDIS generates this status indications for a miniport adapter to notify overlying drivers that there might be a change in the miniport adapter's packet filter setting.

Remarks

NDIS does not guarantee that the packet filter has changed when NDIS generates the NDIS_STATUS_PACKET_FILTER status indication.

NDIS filter drivers can also generate the NDIS_STATUS_PACKET_FILTER status indication.

NDIS supplies a bitwise OR of the filter type flags in the **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure. For a list of the filter type flags, see the OID_GEN_CURRENT_PACKET_FILTER OID. For additional information about packet filters, see OID_GEN_SUPPORTED_PACKET_FILTERS.

The **StatusBufferSize** member of the **NDIS_STATUS_INDICATION** structure is set to sizeof(ULONG).

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

OID_GEN_CURRENT_PACKET_FILTER

OID_GEN_SUPPORTED_PACKET_FILTERS

NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES

Article • 03/14/2023

The miniport driver issues an **NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES** status indication when its currently enabled receive filtering capabilities change.

Note This status indication should only be made by miniport drivers that support NDIS receive filters.

When the miniport driver makes this status indication, it sets the **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure to a pointer to an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure. The driver initializes this structure with its currently enabled receive filter capabilities.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.

- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.

- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

The miniport driver issues the **NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES** status indication when one of the following conditions is true:

- The currently enabled receive filter capabilities change on a single network adapter. For example, receive filters can be enabled or disabled through a management application developed by the independent hardware vendor (IHV).

- The currently enabled receive filter capabilities change for one or more network adapters that belong to a load balancing failover (LBFO) team managed by a MUX intermediate driver. For more information, see NDIS MUX Intermediate Drivers.

The miniport driver follows these steps when it issues the **NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES** status indication:

1. The miniport initializes the NDIS_RECEIVE_FILTER_CAPABILITIES structure with the receive filter capabilities that are currently enabled on the network adapter.

   When the miniport driver initializes the **Header** member, it sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT. The miniport driver sets the **Revision** member of **Header** to NDIS_RECEIVE_FILTER_CAPABILITIES_REVISION_2 and the **Size** member to NDIS_SIZEOF_RECEIVE_FILTER_CAPABILITIES_REVISION_2.

2. The miniport driver initializes an NDIS_STATUS_INDICATION structure for the status indication in the following way:

   - The **StatusCode** member must be set to **NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES**.

   - The **StatusBuffer** member must be set to the address of the NDIS_RECEIVE_FILTER_CAPABILITIES structure.

   - The **StatusBufferSize** member must be set to `sizeof(NDIS_RECEIVE_FILTER_CAPABILITIES)`.

3. The miniport driver issues the status indication by calling NdisMIndicateStatusEx. The driver must pass a pointer to the NDIS_STATUS_INDICATION structure to the *StatusIndication* parameter.

Note Overlying drivers can use the **NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES** status indication to determine the currently enabled receive filter capabilities of the network adapter. Alternatively, these drivers can also issue OID query requests of OID_RECEIVE_FILTER_CURRENT_CAPABILITIES to obtain the currently enabled receive filter capabilities at any time.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NdisMIndicateStatusEx

NDIS_STATUS_INDICATION

NDIS_RECEIVE_FILTER_CAPABILITIES

OID_RECEIVE_FILTER_CURRENT_CAPABILITIES

NDIS_STATUS_RECEIVE_FILTER_HARDWARE_CAPABILITIES

Article • 03/14/2023

The miniport driver issues an **NDIS_STATUS_RECEIVE_FILTER_HARDWARE_CAPABILITIES** status indication when its hardware receive filtering capabilities change. These capabilities include the hardware capabilities that are currently disabled by INF file settings or through the **Advanced** properties page.

Note This status indication should only be made by miniport drivers that support NDIS receive filters.

When the miniport driver makes this status indication, it sets the **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure to a pointer to an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure. The driver initializes this structure with its currently enabled receive filter capabilities.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.
- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.
- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

The miniport driver issues the **NDIS_STATUS_RECEIVE_FILTER_HARDWARE_CAPABILITIES** status indication when one of the following conditions is true:

- The hardware receive filter capabilities change on a single network adapter. For example, receive filters can be enabled or disabled through a management application developed by the independent hardware vendor (IHV).
- The hardware receive filter capabilities change for the load balancing failover (LBFO) team of network adapters that are managed by a MUX intermediate driver.

For example, the hardware receive filter capabilities could change when an adapter is added to or removed from the team.

For more information, see NDIS MUX Intermediate Drivers.

The miniport driver follows these steps when it issues the **NDIS_STATUS_RECEIVE_FILTER_HARDWARE_CAPABILITIES** status indication:

1. The miniport initializes the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure with the receive filter capabilities that are currently enabled on the network adapter.

   When the miniport driver initializes the **Header** member, it sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT. The miniport driver sets the **Revision** member of **Header** to NDIS_RECEIVE_FILTER_CAPABILITIES_REVISION_2 and the **Size** member to NDIS_SIZEOF_RECEIVE_FILTER_CAPABILITIES_REVISION_2.

2. The miniport driver initializes an **NDIS_STATUS_INDICATION** structure for the status indication in the following way:

   - The **StatusCode** member must be set to **NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES**.

   - The **StatusBuffer** member must be set to the address of the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure.

   - The **StatusBufferSize** member must be set to `sizeof(NDIS_RECEIVE_FILTER_CAPABILITIES)`.

3. The miniport driver issues the status indication by calling **NdisMIndicateStatusEx**. The driver must pass a pointer to the **NDIS_STATUS_INDICATION** structure to the *StatusIndication* parameter.

Note Overlying drivers can use the **NDIS_STATUS_RECEIVE_FILTER_HARDWARE_CAPABILITIES** status indication to determine the currently enabled receive filter capabilities of the network adapter. Alternatively, these drivers can also issue OID query requests of OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES to obtain the hardware receive filter capabilities at any time.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NdisMIndicateStatusEx

NDIS_STATUS_INDICATION

NDIS_RECEIVE_FILTER_CAPABILITIES

OID_RECEIVE_FILTER_CURRENT_CAPABILITIES

NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE

Article • 03/14/2023

The miniport driver that supports NDIS Quality of Service (QoS) issues an **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication when its operational NDIS QoS parameters are either resolved for the first time or changed later. The miniport driver configures the network adapter with these operational parameters to perform QoS packet transmission.

When the miniport driver makes this status indication, it sets the **StatusBuffer** member of the NDIS_STATUS_INDICATION structure to a pointer to an NDIS_QOS_PARAMETERS structure. The driver initializes this structure with its operational NDIS QoS parameters.

Note This NDIS status indication is valid only for miniport drivers that support the IEEE 802.1 Data Center Bridging (DCB) interface.

Remarks

The miniport driver issues an **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication under the following conditions:

- The miniport driver must issue an **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication after it has initially resolved its operational NDIS QoS parameters and configured the network adapter with them.

- After this initial status indication, the miniport driver must issue an **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication when its operational NDIS QoS parameters are changed. This can happen when either the local or remote NDIS QoS parameters are changed.

- Miniport drivers obtain the local NDIS QoS parameters from the Windows operating system when the Data Center Bridging (DCB) component (Msdcb.sys) issues an object identifier (OID) method request of OID_QOS_PARAMETERS. This OID request contains an **NDIS_QOS_PARAMETERS** structure that specifies the local NDIS QoS parameters.

  There may be situations when the miniport driver has to override the local NDIS QoS parameters when it resolves its operational NDIS QoS parameters. This is especially true if the local QoS parameters compromise the operational QoS parameters that are being used by any underlying protocols or technologies that are currently enabled on the network adapter. For example, the driver can override the local QoS parameters if the network adapter is enabled for remote boot through the Fibre Channel over Ethernet (FCoE) protocol.

The miniport driver notifies NDIS and overlying drivers of its intention to override the local NDIS QoS parameters by issuing an **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication.

For more information, see Managing NDIS QoS Parameters.

Note Overlying drivers can use the **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication to determine the operational NDIS QoS parameters. Alternatively, these drivers can also issue OID query requests of OID_QOS_OPERATIONAL_PARAMETERS to obtain the operational NDIS QoS parameters at any time.

For information on how the miniport driver issues an **NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE** status indication, see Indicating Changes to the Operational NDIS QoS Parameters.

For more information about the various types of NDIS QoS parameters, see Overview of NDIS QoS Parameters.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NDIS_QOS_PARAMETERS

OID_QOS_OPERATIONAL_PARAMETERS

NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE

Article • 03/14/2023

The miniport driver that supports NDIS Quality of Service (QoS) issues an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication when its remote NDIS QoS parameters are either received from a peer for the first time or change later. The miniport driver receives these QoS parameters from a remote peer through the IEEE 802.1Qaz Data Center Bridging Exchange (DCBX) protocol.

When the miniport driver makes this status indication, it sets the **StatusBuffer** member of the NDIS_STATUS_INDICATION structure to a pointer to an NDIS_QOS_PARAMETERS structure. The driver initializes this structure with its remote NDIS QoS parameters.

Note This NDIS status indication is valid only for miniport drivers that support the IEEE 802.1 Data Center Bridging (DCB) interface.

Remarks

The miniport driver uses the DCBX protocol to receive the QoS parameters for a remote peer. The miniport driver resolves its operational NDIS QoS parameters based on its local and remote QoS settings. Once the operational parameters are resolved, the miniport driver configures the network adapter with these parameters for QoS packet transmission.

For more information about how the driver resolves its operational NDIS QoS parameter settings, see Resolving Operational NDIS QoS Parameters.

The miniport driver must follow these guidelines for issuing an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication:

- If the miniport driver has not received a DCBX frame from a remote peer, it must not issue an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication.

- The miniport driver must issue an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication after it has first received the QoS settings from a remote peer.

  Note The miniport driver must issue this status indication if the network adapter receives remote QoS parameter settings from a peer before the driver's local QoS parameters are set. For more information, see Setting Local NDIS QoS Parameters.

- After this initial status indication, the miniport driver must only issue an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication when it determines a change in the QoS settings on the remote peer.

  Note Miniport drivers must not issue an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication if there have been no changes to the remote NDIS QoS parameters. If the driver does make this type of status indication, NDIS may not pass the indication to overlying drivers.

Note Overlying drivers can use the **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication to determine the remote NDIS QoS parameters. Alternatively, these drivers can also issue OID query requests of OID_QOS_REMOTE_PARAMETERS to obtain the remote NDIS QoS parameters at any time.

For more information on how the miniport driver issues an **NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE** status indication, see Indicating Changes to the Remote NDIS QoS Parameters.

For more information about the remote NDIS QoS parameters, see Overview of NDIS QoS Parameters.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NDIS_QOS_PARAMETERS

OID_QOS_REMOTE_PARAMETERS

NDIS_STATUS_PM_WAKE_REASON

Article • 03/14/2023

The **NDIS_STATUS_PM_WAKE_REASON** status indication provides information about the wake-up event that was generated by a network adapter.

Remarks

Starting with NDIS 6.30, the miniport driver issues an NDIS status indication of **NDIS_STATUS_PM_WAKE_REASON**. This status indication notifies NDIS and overlying drivers about the reason for a wake-up event generated by the network adapter.

If the miniport driver supports this type of status indication, the miniport driver must issue an **NDIS_STATUS_PM_WAKE_REASON** status indication if the network adapter generated a wake-up signal. The driver does this while it is handling the OID set request of OID_PNP_SET_POWER for the transition of the adapter to a full-power state.

When the miniport driver makes this status indication, it sets the **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure to a pointer to an **NDIS_PM_WAKE_REASON** structure.

For more information about how to issue an **NDIS_STATUS_PM_WAKE_REASON** indication, see Issuing NDIS Wake Reason Status Indications.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_PM_WAKE_REASON

NDIS_STATUS_INDICATION

NDIS_STATUS_WAN_LINE_UP

Article • 03/14/2023

The NDIS_STATUS_WAN_LINE_UP status indicates that a WAN-capable miniport driver has established a connection with a remote node.

Remarks

NDIS 4.*x* and earlier NDIS WAN miniport drivers use this status indication. NDIS 5.0 and later WAN miniport drivers must use the CoNDIS WAN interface. For more information about the CoNDIS WAN interface, see Implementing CoNDIS WAN Miniport Drivers (NDIS 5.1).

The *StatusBuffer* parameter of the **NdisMIndicateStatus** function contains a pointer to an **NDIS_MAC_LINE_UP** structure.

For more information about NDIS_STATUS_WAN_LINE_UP, see Line-Up Indication (NDIS 5.1) and Indicating NDIS WAN Miniport Driver Status (NDIS 5.1).

Requirements

Version	Not supported for NDIS 6.0 drivers or NDIS 5.1 drivers in Windows Vista or Windows XP. Supported for NDIS 4.x drivers.
Header	Ndis.h (include Ndis.h)

See also

NDIS_MAC_LINE_UP

NdisMIndicateStatus

NDIS_STATUS_WAN_LINE_DOWN

Article • 03/14/2023

The NDIS_STATUS_WAN_LINE_DOWN status indicates that a WAN-capable miniport driver has lost an established connection with a remote node.

Remarks

NDIS 4.*x* and earlier NDIS WAN miniport drivers use this status indication. NDIS 5.0 and later WAN miniport drivers must use the CoNDIS WAN interface. For more information about the CoNDIS WAN interface, see Implementing CoNDIS WAN Miniport Drivers (NDIS 5.1).

The *StatusBuffer* parameter of the **NdisMIndicateStatus** function contains a pointer to an **NDIS_MAC_LINE_DOWN** structure. The **NdisLinkContext** member of NDIS_MAC_LINE_DOWN identifies the link that is no longer valid.

For more information about NDIS_STATUS_WAN_LINE_DOWN, see Indicating NDIS WAN Miniport Driver Status (NDIS 5.1).

Requirements

Version	Not supported for NDIS 6.0 drivers or NDIS 5.1 drivers in Windows Vista or Windows XP. Supported for NDIS 4.x drivers.
Header	Ndis.h (include Ndis.h)

See also

NDIS_MAC_LINE_DOWN

NdisMIndicateStatus

NDIS_STATUS_WAN_FRAGMENT

Article • 03/14/2023

The NDIS_STATUS_WAN_FRAGMENT status indicates that a WAN-capable miniport driver has received a partial packet from a remote node.

Remarks

NDIS 4.*x* and earlier NDIS WAN miniport drivers use this status indication. NDIS 5.0 and later miniport drivers should use the CoNDIS WAN interface. For more information about NDIS_STATUS_WAN_FRAGMENT, see **NDIS_STATUS_WAN_CO_FRAGMENT**.

The *StatusBuffer* parameter of the **NdisMIndicateStatus** function contains a pointer to an **NDIS_MAC_FRAGMENT** structure. NDIS_MAC_FRAGMENT identifies a particular link and describes the reason that the partial packet was received.

For more information about NDIS_STATUS_WAN_FRAGMENT, see Indicating NDIS WAN Miniport Driver Status (NDIS 5.1).

Requirements

Version	Not supported for NDIS 6.0 drivers or NDIS 5.1 drivers in Windows Vista or Windows XP. Supported for NDIS 4.x drivers.
Header	Ndis.h (include Ndis.h)

See also

NDIS_MAC_FRAGMENT

NDIS_STATUS_WAN_CO_FRAGMENT

NdisMIndicateStatus

NDIS_STATUS_TAPI_INDICATION

Article • 03/14/2023

The NDIS_STATUS_TAPI_INDICATION status indicates that a TAPI event occurred. A WAN-capable miniport driver can indicate TAPI status.

Remarks

NDIS 4.*x* and earlier NDIS WAN miniport drivers use this status indication. NDIS 5.0 and later WAN miniport drivers must use the CoNDIS WAN interface. For more information about the CoNDIS WAN interface, see Implementing CoNDIS WAN Miniport Drivers (NDIS 5.1).

The *StatusBuffer* parameter of the **NdisMIndicateStatus** function contains a pointer to an **NDIS_TAPI_EVENT** structure.The NDIS_TAPI_EVENT structure describes the TAPI line or call event that occurs (for example, changes in line and call states, the arrival of an incoming call, and the closing by a remote node or by the miniport driver of an existing call or line).

For more information about NDIS_STATUS_TAPI_INDICATION, see Indicating NDIS WAN Miniport Driver Status (NDIS 5.1).

Requirements

Version	Not supported for NDIS 6.0 drivers or NDIS 5.1 drivers in Windows Vista or Windows XP. Supported for NDIS 4.x drivers.
Header	Ndis.h (include Ndis.h)

See also

NDIS_TAPI_EVENT

NdisMIndicateStatus

NDIS_STATUS_RING_STATUS

Article • 03/14/2023

The NDIS_STATUS_RING_STATUS status indicates the ring status of a line. A WAN-capable miniport driver can use this status to report a ring failure.

Remarks

NDIS 4.*x* and earlier NDIS WAN miniport drivers use this status indication. NDIS 5.0 and later WAN miniport drivers must use the CoNDIS WAN interface. For more information about the CoNDIS WAN interface, see Implementing CoNDIS WAN Miniport Drivers (NDIS 5.1).

The *StatusBuffer* parameter of the **NdisMIndicateStatus** function contains a ULONG value with one of the following status values:

NDIS_RING_LOBE_WIRE_FAULT

NDIS_RING_HARD_ERROR

NDIS_RING_SIGNAL_LOSS

These values specify ring conditions that are the reason for the status indication. For more information about NDIS_STATUS_RING_STATUS, see Reporting Hardware Status (NDIS 5.1).

Requirements

Version	Not supported for NDIS 6.0 drivers or NDIS 5.1 drivers in Windows Vista or Windows XP. Supported for NDIS 4.x drivers.
Header	Ndis.h (include Ndis.h)

See also

NdisMIndicateStatus

NDIS_STATUS_WW_INDICATION

Article • 03/14/2023

The NDIS_STATUS_WW_INDICATION status is the same as the NDIS_STATUS_MEDIA_SPECIFIC_INDICATION status.

Remarks

For more information about NDIS_STATUS_WW_INDICATION, see OID_WW_GEN_INDICATION_REQUEST.

Requirements

Version	Not supported in NDIS 6.0 and later. Supported only for NDIS 5.1 drivers in Windows Vista and Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_MEDIA_SPECIFIC_INDICATION

OID_WW_GEN_INDICATION_REQUEST

NDIS_STATUS_WAN_CO_FRAGMENT

Article • 03/14/2023

The NDIS_STATUS_WAN_CO_FRAGMENT status indicates that a CoNDIS WAN miniport driver has received a partial packet from the endpoint of a VC.

Remarks

The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure contains a pointer to an NDIS_WAN_CO_FRAGMENT structure. The NDIS_WAN_CO_FRAGMENT structure describes the reason that the partial packet was received.

For more information about NDIS_STATUS_WAN_CO_FRAGMENT, see Indicating CoNDIS WAN Miniport Driver Status. For more information about the CoNDIS WAN interface, see Implementing CoNDIS WAN Miniport Drivers.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NDIS_WAN_CO_FRAGMENT

NDIS_STATUS_WAN_CO_LINKPARAMS

Article • 03/14/2023

The NDIS_STATUS_WAN_CO_FRAGMENT status indicates that parameters for a particular VC that is active on a CoNDIS miniport adapter have changed.

Remarks

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains a pointer to a **WAN_CO_LINKPARAMS** structure. The WAN_CO_LINKPARAMS structure describes new parameters for the VC.

For more information about NDIS_STATUS_WAN_CO_LINKPARAMS, see Indicating CoNDIS WAN Miniport Driver Status. For more information about the CoNDIS WAN interface, see Implementing CoNDIS WAN Miniport Drivers.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

WAN_CO_LINKPARAMS

NDIS_STATUS_WAN_CO_MTULINKPARAMS

Article • 03/14/2023

The NDIS_STATUS_WAN_CO_MTULINKPARAMS status indicates that the link speed and send window parameters have changed for a particular VC that is active on a CoNDIS miniport adapter.

Remarks

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains a pointer to a **WAN_CO_MTULINKPARAMS** structure. The WAN_CO_MTULINKPARAMS structure describes new parameters for the VC.

For more information about NDIS_STATUS_WAN_CO_MTULINKPARAMS, see Indicating CoNDIS WAN Miniport Driver Status. For more information about the CoNDIS WAN interface, see Implementing CoNDIS WAN Miniport Drivers.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

WAN_CO_MTULINKPARAMS

Introduction to Header-Data Split

Article • 12/15/2021

This section describes header-data split services that are available in NDIS 6.1 and later versions. *Header-data split* improves network performance by splitting the header and data in received Ethernet frames into separate buffers. Separating the headers and the data enables the headers to be collected together into smaller regions of memory. As a result, more headers will fit into a single memory page and more headers will fit into the system caches, so the overhead for memory accesses in the driver stack is reduced.

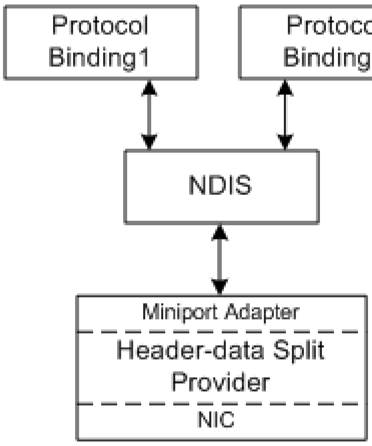
This section includes:

Header-Data Split Architecture

Article • 12/15/2021

A header-data split provider improves network performance by splitting the headers and data in received Ethernet frames into separate buffers. A header-data split provider includes a network interface card (NIC) and an NDIS 6.1 or later miniport driver that services the NIC.

The following figure shows the header-data split architecture.



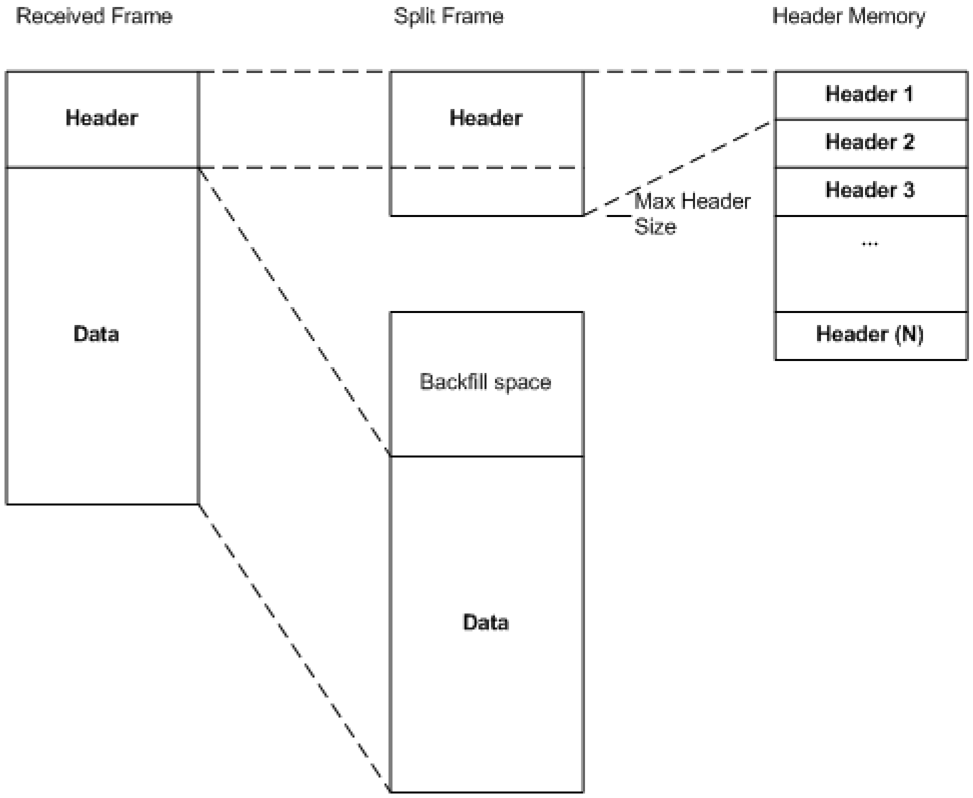
The miniport driver receives configuration information from NDIS to set up the NIC for header-data split receive operations. Also, the miniport driver exposes the NIC's services to NDIS for run-time operations such as send and receive operations.

A NIC that is capable of header-data split operations receives Ethernet frames and splits the headers and data into separate receive buffers.

The miniport driver uses the normal NDIS receive functions to indicate the received data to NDIS. Also, the driver must assign exactly one **NET_BUFFER** structure to a **NET_BUFFER_LIST** structure when indicating received data. For more information, see Indicating Received Ethernet Frames.

For header-data split, the **NET_BUFFER** structures in the receive indications split the received Ethernet frame by using separate memory descriptor lists (MDLs) for the header and the data. Also, the **NET_BUFFER_LIST** structure contains header-data split information in the NET_BUFFER_LIST information.

The following figure shows the received frame, the split buffers, and the memory layout of the header buffers.



The header buffers should all be in a contiguous block of storage.

An *upper-layer protocol* is an IP transport protocol such as TCP, UDP, or ICMP.

Note IPsec is not considered an upper-layer protocol for the purposes of defining header-data split requirements. For more information about splitting IPsec frames, see Splitting IPsec Frames.

Where to Split Header and Data

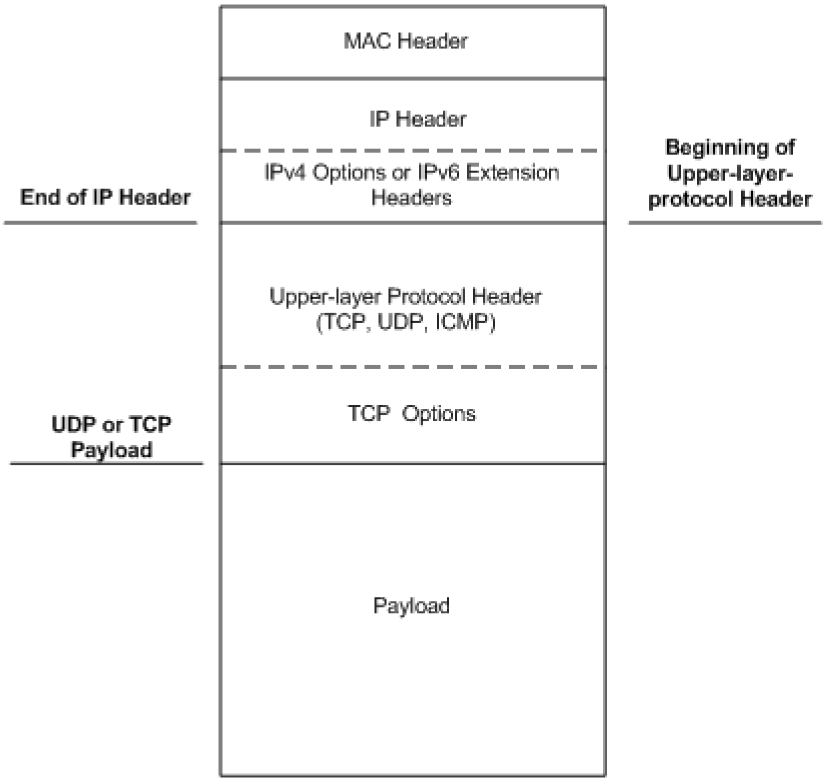
Article • 12/15/2021

The following are the only valid places where a header-data split provider can split an Ethernet frame:

- Beginning of upper-layer-protocol header.
- Beginning of UDP payload.
- Beginning of TCP payload.

Note These requirements apply only to header-data split providers. For more information about splitting frames in cases where header-data split is not used, see Cases Where Header-Data Split Is Not Used.

The following figure shows the major parts of the Ethernet frame and the valid split locations.


MAC Header
IP Header
IPv4 Options or IPv6 Extension Headers
End of IP Header
Beginning of Upper-layer-protocol Header
Upper-layer Protocol Header (TCP, UDP, ICMP)
TCP Options
UDP or TCP Payload
Payload

Cases Where Header-Data Split Is Not Used

Article • 12/15/2021

This topic provides an overview of the cases where a header-data split provider must not split Ethernet frames. For a listing of the minimum requirements that a provider must meet to support header-data split, see Minimum Requirements for Supporting Header-Data Split.

Note There are cases where a received frame can be split outside of the header-data split provider requirements. That is, the header-data split requirements only apply to header-data split providers. In these cases, never split Ethernet frames in the middle of the IP header, IPv4 options, IPsec headers, IPv6 extension headers, or upper-layer-protocol headers, unless the first MDL contains at least as many bytes as NDIS specified for lookahead size.

All Ethernet frames that are not split must follow the general NDIS rules and requirements. For example, the first MDL in the chain of MDLs in a received **NET_BUFFER** structure must contain either the lookahead part of the frame or the entire Ethernet frame (whichever is smaller) in a virtually contiguous buffer. NDIS sets the size of lookahead with the OID_GEN_CURRENT_LOOKAHEAD OID.

Header-data split providers:

- Do not split non-IP frames.

- Do not split frames if they cannot be split in one of these locations: at the beginning of the upper-layer-protocol header, the beginning of the TCP payload, or the beginning of the UDP payload.

- Do not split frames that would exceed the maximum configured header size unless the header can be split at the beginning of the upper-layer-protocol header. For more information about the maximum header size, see Allocating the Header Buffer.

- Do not split frames that contain IPv4 options that the NIC does not recognize.

- Do not split frames that contain IPv6 extension headers that the NIC does not recognize.

- Do not split frames that contain TCP options that the NIC does not recognize unless they can be split at the beginning of the TCP header.

Minimum Requirements for Supporting Header-Data Split

Article • 12/15/2021

This topic summarizes the minimum requirements that a provider must meet to support header-data split. For a complete listing of the rules that apply to splitting Ethernet frames, see Splitting Ethernet Frames.

The following list contains the minimum requirements for header-data split support:

- Providers must not split frames that the Cases Where Header-Data Split Is Not Used topic describes.

- Providers must move virtual LAN (VLAN) tags to the **NET_BUFFER_LIST** structure OOB data. For more information about VLAN requirements, see Receive Indications with Header-Data Split.

- Providers must support splitting IPv4 frames without options. For more information about splitting IPv4 frames, see Splitting IPv4 Frames.

- Providers must support splitting IPv6 frames without extension headers. For more information about splitting IPv6 frames, see Splitting IPv6 Frames.

- Providers must support splitting TCP frames at the TCP payload with no TCP options and with only the timestamp option. For more information about splitting TCP frames, see Splitting Frames at the TCP Payload.

- Providers must support splitting UDP frames at the UDP payload. For more information about splitting UDP frames, see Splitting Frames at the UDP Payload.

- Providers must support the header-data split initialization attributes. For more information about these attributes, see Initializing a Header-Data Split Provider.

- Providers must support the header-data split receive indication requirements, including setting the header-data split flags in the **NblFlags** member of the **NET_BUFFER_LIST** structures, header size requirements, and data backfill requirements. For more information about receive requirements, see Receive Indications with Header-Data Split.

- Providers must support the OID_GEN_HD_SPLIT_PARAMETERS OID, the OID_GEN_HD_SPLIT_CURRENT_CONFIG OID, the **NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG** status indication, and registry settings. For more information about header-data split parameters and settings, see Header-Data Split Administration and Configuration.

For more information about header-data split requirements for protocol drivers and filter drivers, see Supporting Header-Data Split in Protocol Driver and Filter Drivers.

NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG status indication to notify NDIS and overlying drivers that there has been a change in the header-data split configuration of a miniport adapter.

Remarks

When a miniport driver receives an OID_GEN_HD_SPLIT_PARAMETERS set request, the driver must use the contents of the **NDIS_HD_SPLIT_PARAMETERS** structure to update the current configuration of the miniport adapter. After the update, the miniport driver must report the changes with the NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG status indication. The status indication ensures that all of the overlying drivers are updated with the new information.

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains an **NDIS_HD_SPLIT_CURRENT_CONFIG** structure. This structure specifies the current header-data split configuration of a miniport adapter.

Requirements

Version	Supported in NDIS 6.1 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_HD_SPLIT_CURRENT_CONFIG

NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG

NDIS_STATUS_INDICATION

OID_GEN_HD_SPLIT_PARAMETERS

Initializing a Header-Data Split Provider

Article • 12/15/2021

To support header-data split, a miniport driver must register as an NDIS 6.1 or later driver. The sources file for the miniport driver must specify DNDIS61_MINIPORT=1 instead of DNDIS60_MINIPORT=1. The miniport driver must also specify NDIS 6.1 or a later version in the NDIS_MINIPORT_DRIVER_CHARACTERISTICS structure.

To register its header-data split attributes, an NDIS 6.1 miniport driver calls the NdisMSetMiniportAttributes function from its *MiniportInitializeEx* function and passes **NdisMSetMiniportAttributes** an initialized NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

The NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure contains the following information:

- The **HDSplitAttributes** member of NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES contains a pointer to an NDIS_HD_SPLIT_ATTRIBUTES structure that specifies the header-data split capabilities that a miniport adapter provides.

- The **HardwareCapabilities** member of NDIS_HD_SPLIT_ATTRIBUTES contains the header-data split capabilities that the miniport adapter supports. These capabilities can include capabilities that are currently disabled by INF file settings or through the **Advanced** properties page.

- The **CurrentCapabilities** member of NDIS_HD_SPLIT_ATTRIBUTES contains the current header-data split capabilities that the miniport adapter supports. If header-data split is enabled through the ***HeaderDataSplit** standardized INF keyword, the miniport driver uses the same flags as the **HardwareCapabilities** member to indicate the current header-data split configuration. For more information about ***HeaderDataSplit**, see Standardized INF Keywords for Header-Data Split.

- The **HDSplitFlags** member of NDIS_HD_SPLIT_ATTRIBUTES contains header-data split configuration flags. The miniport driver should set this member to zero before calling NdisMSetMiniportAttributes. NDIS sets this member with a bitwise OR of the configuration flags. After **NdisMSetMiniportAttributes** successfully returns, the miniport driver must check the flag settings in **HDSplitFlags** and configure the hardware accordingly.

NDIS uses the NDIS_HD_SPLIT_ENABLE_HEADER_DATA_SPLIT flag to enable header-data split for the miniport adapter. NDIS will not set

NDIS_HD_SPLIT_ENABLE_HEADER_DATA_SPLIT if the miniport driver did not set the NDIS_HD_SPLIT_CAPS_SUPPORTS_HEADER_DATA_SPLIT flag in the **CurrentCapabilities** member of the **NDIS_HD_SPLIT_ATTRIBUTES** structure. The miniport driver should enable header-data split in the NIC if NDIS sets the NDIS_HD_SPLIT_ENABLE_HEADER_DATA_SPLIT flag.

The miniport driver should set the **BackfillSize** member of the NDIS_HD_SPLIT_ATTRIBUTES structure to zero before calling **NdisMSetMiniportAttributes**. NDIS sets the **BackfillSize** member if the miniport driver must pre-allocate backfill storage in the data buffer of the split frames. After **NdisMSetMiniportAttributes** successfully returns, the miniport driver must use the **BackfillSize** value that NDIS specified and pre-allocate the data buffers. For more information about the data buffer backfill size, see Allocating Backfill for the Data Buffer.

The miniport driver should set the **MaxHeaderSize** member of the **NDIS_HD_SPLIT_ATTRIBUTES** structure to zero before calling **NdisMSetMiniportAttributes**. NDIS sets this member to the maximum size that is allowed for the header buffer of the split frames. After **NdisMSetMiniportAttributes** successfully returns, the miniport driver must use the **MaxHeaderSize** value that NDIS specified. For more information about the maximum header size, see Allocating the Header Buffer.

Splitting Ethernet Frames Overview

Article • 12/15/2021

This section describes the specific header-data split requirements that apply to header-data split providers, depending on the type of Ethernet frame that the provider is splitting.

Note After you read the general requirements in this topic, you can use the subsequent topics to understand the specific requirements for each type of Ethernet frame. The later topics build on the requirements in the earlier topics. For example, if a frame contains IPv4 and UDP information, you should read the Splitting IPv4 Frames and Splitting Frames at the UDP Payload topics.

If the header-data split provider splits a frame in compliance with the header-data split requirements, the indicated **NET_BUFFER_LIST** structures must have the NDIS_NBL_FLAGS_HD_SPLIT flag set in the **NblFlags** member. If the header-data split provider does not split a frame, the frame must be indicated with the following flags cleared in **NblFlags** :

- NDIS_NBL_FLAGS_HD_SPLIT
- NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_HEADER
- NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_PAYLOAD

For more information about setting header-data split NET_BUFFER_LIST flags and other receive indication requirements, see Receive Indications with Header-Data Split.

There are cases where a header-data split provider can split a received frame outside of the header-data split provider requirements. In these cases, the provider should never split Ethernet frames in the middle of the IP header, IPv4 options, IPsec headers, IPv6 extension headers, or upper-layer-protocol headers, unless the first MDL contains at least as many bytes as NDIS specified for the lookahead size. For more information about the lookahead size, see OID_GEN_CURRENT_LOOKAHEAD.

This section includes:

Splitting IPv4 Frames

Splitting IPv6 Frames

Splitting Fragmented IP Frames

Splitting Frames at the Beginning of the Upper-Layer-Protocol Headers

Splitting Frames at the TCP Payload

Splitting Frames at the UDP Payload

Splitting Frames Other Than TCP and UDP

Splitting IPv4 Frames

Article • 12/15/2021

To support header-data split, a NIC must support splitting IPv4 Ethernet frames that have no IPv4 options. The NIC must be able to split such frames at the beginning of upper-layer-protocol header.

Support for IPv4 Ethernet frames with IPv4 options is optional. The NIC can support some IPv4 options and not the others. The NIC must not split IPv4 frames that contain IPv4 options that it does not recognize. The header portion of a split frame must contain the entire IPv4 header and all of the IPv4 options that are present.

The NIC can also support header-data split for fragmented IPv4 frames. For more information about fragmented IPv4 frames, see Splitting Fragmented IP Frames.

Note Supporting an IPv4 option, an IPv6 extension header or a TCP option, for the purposes of header-data requirements, implies the ability of the NIC to recognize the element, determine its length, include it in the header MDL and locate its end and the beginning of the next element in the frame.

If the header-data split provider splits an IPv4 frame, the indicated **NET_BUFFER_LIST** structures must have the NDIS_NBL_FLAGS_IS_IPV4 flag set in the **NblFlags** member. For complete information about setting header-data split flags in the NET_BUFFER_LIST structure, see Setting NET_BUFFER_LIST Information.

Additional Ethernet frame characteristics determine how to split IPv4 frames. If the IP frame is fragmented, see Splitting Fragmented IP Frames. If the frame contains TCP information, see Splitting Frames at the TCP Payload. If the frame contains UDP information, see Splitting Frames at the UDP Payload. For all other cases, see Splitting Frames Other Than TCP and UDP.

Splitting IPv6 Frames

Article • 12/15/2021

To support header-data split, a NIC must support splitting IPv6 Ethernet frames without any IPv6 extension headers. The NIC must be able to split such frames at the beginning of upper-layer-protocol header.

Support for IPv6 Ethernet frames with IPv6 extension headers is optional. A NIC can support some IPv6 options and not support others. The NIC must not split IPv6 frames that contain IPv6 extension headers that is does not support. The header portion of a split frame must contain the entire IPv6 header and all of the IPv6 extension headers that are present.

The NIC can also support header-data split for fragmented IPv6 frames. For more information about fragmented IPv4 frames, see Splitting Fragmented IP Frames.

Note Supporting an IPv4 option, an IPv6 extension header or a TCP option, for the purposes of header-data requirements, implies the ability of the NIC to recognize the element, determine its length, include it in the header MDL and locate its end and the beginning of the next element in the frame.

If the header-data split provider splits an IPv6 frame, the indicated **NET_BUFFER_LIST** structures must have the NDIS_NBL_FLAGS_IS_IPV6 flag set in the **NblFlags** member. For complete information about setting header-data split flags in the NET_BUFFER_LIST structure, see Setting NET_BUFFER_LIST Information.

Additional Ethernet frame characteristics determine how to split IPv6 frames. If the frame is fragmented, see Splitting Fragmented IP Frames. If the frame contains TCP information, see Splitting Frames at the TCP Payload. If the frame contains UDP information, see Splitting Frames at the UDP Payload. For all other cases, see Splitting Frames Other Than TCP and UDP.

Splitting Fragmented IP Frames

Article • 12/15/2021

If a fragmented IP frame contains the upper-layer-protocol header, a NIC must split the frame at the beginning of upper-layer-protocol header or must not split the frame. That is, the NIC must not split fragmented IP frames at the beginning of the TCP or UDP payload.

If a fragmented IPv4 frame does not contain the upper-layer-protocol header, the NIC must split the frame at the beginning of the UDP or TCP payload or must not split the frame.

If a fragmented IPv6 frame does not contain the upper-layer-protocol header, the NIC must not split the frame.

For more information about splitting frames at the beginning of the upper-layer-protocol header, see Splitting Frames at the Beginning of the Upper-Layer-Protocol Headers.

Splitting Frames at the Beginning of the Upper Layer-Protocol Headers

Article • 12/15/2021

An *upper-layer protocol* is an IP transport protocol such as TCP, UDP, or ICMP.

Note IPsec is not considered an upper-layer-protocol in the header-data split requirements. For more information about splitting IPsec frames, see Splitting IPsec Frames.

If a NIC splits an Ethernet frame at the beginning of the upper-layer-protocol header, the indicated **NET_BUFFER** must contain exactly two MDLs. The buffer that the first MDL describes must begin with the first byte of the Ethernet frame (MAC header) and the buffer that the second MDL describes must start with the first byte of the upper-layer-protocol header.

Note The NIC can split TCP and UDP frames at the TCP or UDP payload. For more information, see Splitting Frames at the TCP Payload and Splitting Frames at the UDP Payload.

If the header-data split provider splits the frame at the beginning of the upper-layer-protocol header, the indicated **NET_BUFFER_LIST** structures must have the NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_HEADER flag set in the **NblFlags** member. For more information about setting header-data split NET_BUFFER_LIST flags, see Setting NET_BUFFER_LIST Information.

The NIC must not split a frame if the resulting header buffer has a greater length than the maximum header size. For more information about splitting frames when the maximum header size is exceeded, see Allocating the Header Buffer.

Splitting Frames at the TCP Payload

Article • 12/15/2021

NDIS miniport adapters that support header-data split must support splitting frames at the upper-layer-protocol header for TCP frames. However, if the TCP header does not contain any TCP options, the NIC should split the frame at the beginning of the TCP payload.

The NIC might not be able to split a TCP frame if the resulting header buffer has a greater length than the maximum header size. For more information about splitting frames when the maximum header size is exceeded, see Allocating the Header Buffer.

NICs must also support splitting TCP headers with only the timestamp option. That is, the timestamp option is the only TCP option that is mandatory. Otherwise, support for TCP headers with TCP options is optional. If the TCP header of a frame contains an unrecognized TCP option, the NIC must either split the frame at the beginning of TCP header (that is, at the upper-layer-protocol header) or not split the frame.

Note Supporting an IPv4 option, an IPv6 extension header or a TCP option, for the purposes of header-data requirements, implies the ability of the NIC to recognize the element, determine its length, include it in the header MDL and locate its end and the beginning of the next element in the frame.

For more information about splitting frames at the beginning of the upper-layer-protocol header, see Splitting Frames at the Beginning of the Upper-Layer-Protocol Headers.

If the header-data split provider splits the frame at the TCP payload, the indicated **NET_BUFFER_LIST** structures must have the NDIS_NBL_FLAGS_IS_TCP and NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_PAYLOAD flags set in the **NblFlags** member. For more information about setting header-data split NET_BUFFER_LIST flags, see Setting NET_BUFFER_LIST Information.

Splitting Frames at the UDP Payload

Article • 12/15/2021

NDIS miniport adapters that support header-data split must support splitting frames at the upper-layer-protocol header for UDP frames. However, the NIC must first try to split the frame at the beginning of UDP payload.

The NIC might not be able to split a UDP frame if the resulting header buffer has a greater length than the maximum header size. For more information about splitting frames when the maximum header size is exceeded, see Allocating the Header Buffer.

If the NIC cannot split the frame at the UDP payload, the NIC should split the frame at the beginning of the upper-layer-protocol header or should not split the frame. For more information about splitting frames at the beginning of the upper-layer-protocol header, see Splitting Frames at the Beginning of the Upper-Layer-Protocol Headers.

If the header-data split provider splits the frame at the UDP payload, the indicated **NET_BUFFER_LIST** structures must have the NDIS_NBL_FLAGS_IS_UDP and NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_PAYLOAD flags set in the **NblFlags** member. For more information about setting header-data split NET_BUFFER_LIST flags, see Setting NET_BUFFER_LIST Information.

Splitting ICMP Frames and Other Upper-Layer-Protocol Frames

Article • 12/15/2021

A NIC must split IP frames with upper-layer-protocols other than TCP or UDP (for example, ICMP frames) at the beginning of upper-layer-protocol header or must not split such frames.

For more information about splitting at the upper-layer-protocol header, see Splitting Frames at the Beginning of the Upper-Layer-Protocol Headers.

Splitting IPsec Frames

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

A NIC can split IPsec frames at the beginning of the upper-layer-protocol header, the beginning of the TCP payload, or the beginning of the UDP payload. The NIC should treat the IPsec information the same as an IPv4 option or IPv6 extension header.

The NIC might not be able to split the frame if the resulting header buffer has a greater length than the maximum header size. For more information about the maximum header size, see Allocating the Header Buffer.

Receive Indications with Header-Data Split

Article • 12/15/2021

A miniport driver that supports header-data split must indicate received data in the format that header-data split requires. For example, the header buffers should all be in a contiguous block of storage and the data buffers must include backfill space.

The header information in split frames must never include virtual LAN (VLAN) tags. Header-data split requires support for VLAN in hardware and requires removing VLAN tags from the incoming frames and placing them in the **Ieee8021QNetBufferListInfo** OOB information in the **NET_BUFFER_LIST** structure. The miniport driver must specify support for VLAN in the **MacOptions** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure and in response to the OID_GEN_MAC_OPTIONS OID query.

NDIS and overlying drivers or user-mode applications use the OID_GEN_HD_SPLIT_PARAMETERS OID to set the current header-data split settings of a miniport adapter. If the NDIS_HD_SPLIT_COMBINE_ALL_HEADERS flag in the **HDSplitCombineFlags** member of the **NDIS_HD_SPLIT_PARAMETERS** structure is set, the miniport adapter must combine all split frames. If header-data split is enabled in the hardware, the miniport driver must combine the header and data before indicating the frame to NDIS. For more information about OID_GEN_HD_SPLIT_PARAMETERS and other administrative and configuration issues, see Header-Data Split Administration and Configuration.

This section includes:

Allocating the Header Buffer

Allocating Backfill for the Data Buffer

Setting NET_BUFFER_LIST Information

Allocating the Header Buffer

Article • 12/15/2021

NDIS specifies the maximum header size that a miniport driver should allocate in the **MaxHeaderSize** member of the **NDIS_HD_SPLIT_ATTRIBUTES** structure. For more information about setting header-data split attributes, see Initializing a Header-Data Split Provider.

When a NIC splits the header and data in a received Ethernet frame, the size of the header portion of the indicated Ethernet frame must not exceed the **MaxHeaderSize** value.

If an IP header contains IPv4 options, IPsec headers, or IPv6 extension headers, and if the header exceeds the **MaxHeaderSize** value, the NIC must not split the frame.

If a header that includes the UDP header, TCP header, or TCP options exceeds the **MaxHeaderSize** value, the NIC must either split the frame at the beginning of the upper-layer-protocol header or must not split the frame.

Allocating Backfill for the Data Buffer

Article • 12/15/2021

NDIS specifies the amount of data backfill space that the miniport driver should allocate in the **BackfillSize** member of the **NDIS_HD_SPLIT_ATTRIBUTES** structure. For more information about setting header-data split attributes, see Initializing a Header-Data Split Provider.

When a NIC splits the header and data in a received Ethernet frame, the miniport driver must pre-allocate backfill storage of at least the number of bytes that **BackfillSize** specifies before the starting address of data portion of the frame. The backfill storage must not cross a page boundary.

The driver stack can use the pre-allocated backfill storage to copy the header portion of the frame and create a virtually contiguous frame for network drivers that cannot handle split Ethernet frames.

Setting NET_BUFFER_LIST Information

Article • 12/15/2021

A header-data split provider must set the header-data split flags in the **NblFlags** member of the NET_BUFFER_LIST structures for receive indications. For split frames, a NIC must also provide the physical address of the data portion of the received frame in the **DataPhysicalAddress** member of each NET_BUFFER structure.

Note A miniport driver can set the **DataPhysicalAddress** member of the NET_BUFFER structure, even if the NET_BUFFER is not associated with a split frame. In this case, **DataPhysicalAddress** contains the physical address of the header MDL.

The header-data split provider combines the flags in the **NblFlags** member with a bitwise OR operation.

The header-data split provider can set the following flags even if it does not split a frame:

NDIS_NBL_FLAGS_IS_IPV4
All of the frames in the NET_BUFFER_LIST are IPv4 frames. If this flag is set, the NDIS_NBL_FLAGS_IS_IPV6 flag must not be set.

NDIS_NBL_FLAGS_IS_IPV6
All of the frames in the NET_BUFFER_LIST are IPv6 frames. If this flag is set, the NDIS_NBL_FLAGS_IS_IPV4 flag must not be set.

NDIS_NBL_FLAGS_IS_TCP
All of the frames in the NET_BUFFER_LIST are TCP frames. If this flag is set, NDIS_NBL_FLAGS_IS_UDP must not be set. And either NDIS_NBL_FLAGS_IS_IPV4 or NDIS_NBL_FLAGS_IS_IPV6 must be set.

NDIS_NBL_FLAGS_IS_UDP
All of the frames in the NET_BUFFER_LIST are UDP frames. If this flag is set, NDIS_NBL_FLAGS_IS_TCP must not be set. And either NDIS_NBL_FLAGS_IS_IPV4 or NDIS_NBL_FLAGS_IS_IPV6 must be set.

Any NDIS driver can set the preceding flags for debugging, testing, or other purposes. If a driver sets these flags, the values must accurately describe the contents of the received frame. Setting these flags is recommended.

The header-data split provider can set the following header-data split flags:

NDIS_NBL_FLAGS_HD_SPLIT
The header and data are split in all of the Ethernet frames that are associated with the **NET_BUFFER_LIST** structure.

NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_HEADER
All of the frames in the NET_BUFFER_LIST structure are split at the beginning of the upper-layer-protocol header. If this flag is set, either NDIS_NBL_FLAGS_IS_IPV4 or NDIS_NBL_FLAGS_IS_IPV6 must be set. Also, either NDIS_NBL_FLAGS_IS_TCP or NDIS_NBL_FLAGS_IS_UDP can be set. And NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_PAYLOAD must not be set.

NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_PAYLOAD
All of the frames in a NET_BUFFER_LIST structure are split at the beginning of the TCP payload or beginning of the UDP payload. If this flag is set, either NDIS_NBL_FLAGS_IS_IPV4 or NDIS_NBL_FLAGS_IS_IPV6 must be set. Either NDIS_NBL_FLAGS_IS_TCP or NDIS_NBL_FLAGS_IS_UDP must be set. Also, NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_HEADER must not be set.

If the header-data split provider does not split a frame, the frame must be indicated with the following flags cleared in **NblFlags** :

- NDIS_NBL_FLAGS_HD_SPLIT
- NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_HEADER
- NDIS_NBL_FLAGS_SPLIT_AT_UPPER_LAYER_PROTOCOL_PAYLOAD

Setting the Current Header-Data Split Configuration

Article • 12/15/2021

NDIS and overlying drivers or user-mode applications use the OID_GEN_HD_SPLIT_PARAMETERS OID to set the current header-data split settings of a miniport adapter. NDIS 6.1 and later miniport drivers that provide header-data split services must support this OID. Otherwise, this OID is optional.

A system administrator can use the GUID that is associated with this OID through the WMI interface. For more information about header-data split WMI GUIDs, see WMI Support for Header-Data Split.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains an **NDIS_HD_SPLIT_PARAMETERS** structure.

If the NDIS_HD_SPLIT_COMBINE_ALL_HEADERS flag in the **HDSplitCombineFlags** member of NDIS_HD_SPLIT_PARAMETERS is set, the miniport adapter must combine all split frames. If header-data split is enabled in the hardware, the miniport driver must combine the header and data before the driver indicates the frame to NDIS.

For example, NDIS might use the OID_GEN_HD_SPLIT_PARAMETERS OID to set the NDIS_HD_SPLIT_COMBINE_ALL_HEADERS flag when an NDIS 5.*x* protocol driver binds to an NDIS 6.1 miniport adapter. NDIS processes this OID before it passes the OID to the miniport driver and updates the miniport adapter's ***HeaderDataSplit** standardized keyword, if required. If header-data split is disabled, NDIS does not send this OID to the miniport adapter.

Getting the Current Header-Data Split Configuration

Article • 12/15/2021

To get the current header-data split settings of a miniport adapter, overlying drivers or user-mode applications can query the OID_GEN_HD_SPLIT_CURRENT_CONFIG OID. However, overlying drivers should use the information that NDIS provides to them during initialization and with status indications.

A system administrator can use the GUID that is associated with the OID_GEN_HD_SPLIT_CURRENT_CONFIG OID through the WMI interface. For more information about header-data split WMI GUIDs, see WMI Support for Header-Data Split.

NDIS handles OID_GEN_HD_SPLIT_CURRENT_CONFIG on behalf of the miniport driver. NDIS maintains the current header-data split configuration information based on the miniport driver initialization attributes and the **NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG** status indication.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains an **NDIS_HD_SPLIT_CURRENT_CONFIG** structure. NDIS also provides the NDIS_HD_SPLIT_CURRENT_CONFIG structure to overlying drivers during initialization and with the status indication.

When a miniport driver receives an OID_GEN_HD_SPLIT_PARAMETERS set request, the driver must use the contents of the **NDIS_HD_SPLIT_PARAMETERS** structure to update the current configuration of the miniport adapter. After the update, the miniport driver must report the changes with the **NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG** status indication. The status indication ensures that all of the overlying drivers are updated with the new information.

When NDIS calls the *ProtocolBindAdapterEx* function of NDIS 6.1 or later protocol drivers, NDIS provides an **NDIS_BIND_PARAMETERS** structure with a pointer to an **NDIS_HD_SPLIT_CURRENT_CONFIG** structure.

When NDIS calls the *FilterAttach* function of NDIS 6.1 or later filter drivers, NDIS provides an **NDIS_FILTER_ATTACH_PARAMETERS** structure with a pointer to an NDIS_HD_SPLIT_CURRENT_CONFIG structure.

Standardized INF Keywords for Header-Data Split

Article • 12/15/2021

A *standardized keyword* is defined to enable or disable support for header-data split for a miniport adapters.

The following table describes the possible INF entries for the header-data split keyword.

SubkeyName	ParamDesc	Value	EnumDesc
***HeaderDataSplit**	Header Data Split	0 (Default)	Disabled
		1	Enabled

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

WMI Support for Header-Data Split

Article • 12/15/2021

NDIS 6.1 and later supports header-data split GUIDs for WMI applications to query and set the header-data split configuration and to provide notification of changes.

The following GUIDs support header-data split:

- GUID_NDIS_HD_SPLIT_CURRENT_CONFIG
- GUID_NDIS_HD_SPLIT_PARAMETERS
- GUID_NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG

Supporting Header-Data Split in Protocol Drivers and Filter Drivers

Article • 12/15/2021

NDIS 6.0 and later protocol drivers and filter drivers must support receive indications with the header and data in non-contiguous buffers.

You must not assume that there is only a single MDL in a NET_BUFFER structure. Protocol drivers and filter drivers are not required to do anything specific to support header-data split registration. But, the driver receive handling code must handle more than one MDL in the MDL chain and must use the following NDIS MDL macros to access the MDL chain:

- NET_BUFFER_FIRST_MDL
- NET_BUFFER_CURRENT_MDL
- NET_BUFFER_CURRENT_MDL_OFFSET

With split buffers, the data length that is associated with the NET_BUFFER structure (in the **DataLength** member of the NET_BUFFER_DATA structure) is split across multiple MDLs. For example, if a protocol driver tried to access the entire data buffer in the first MDL, the driver could access invalid data.

Note After the receive indication call returns to a miniport driver, the miniport driver can reclaim the header MDLs. The overlying drivers or their clients must not access the header MDLs after the receive indication call returns to the miniport driver. This restriction also applies even when the miniport driver is not indicating the received data with a status of NDIS_RECEIVE_FLAGS_RESOURCES.

Overview of Network Direct Kernel Provider Interface (NDKPI)

Article • 12/15/2021

The Network Direct Kernel Provider Interface (NDKPI) is an extension to NDIS that allows IHVs to provide kernel-mode Remote Direct Memory Access (RDMA) support in a network adapter (also called an RNIC). To expose the adapter's RDMA functionality, the IHV must implement the NDKPI interface as defined in the NDKPI Reference.

- NDKPI and RDMA
- The NDK provider
- The NDK consumer

NDKPI and RDMA

A NIC vendor implements RDMA as a combination of software, firmware, and hardware. The hardware and firmware portion is a network adapter that provides NDK/RDMA functionality. This type of adapter is also called an RDMA-enabled NIC (RNIC). The software portion is an NDK-capable miniport driver, which implements the NDKPI interface.

The Windows implementation of RDMA is called Network Direct (ND). The kernel portion is called Network Direct Kernel (NDK).

NDK providers must support Network Direct connectivity via both IPv4 and IPv6 addresses assigned to NDK-capable miniport adapters.

For more information about RDMA, see Background Reading on RDMA.

The NDK provider

An NDK provider is a miniport driver that implements the NDKPI interface.

The NDK provider is loaded and initialized by the PnP Manager. For more information, see Initializing an NDK-Capable Miniport Driver and Initializing an NDK Miniport Adapter.

Once the NDK provider is loaded and initialized, it is ready to handle requests from the NDK consumer. These requests arrive as calls to provider functions.

When handling requests from an NDK consumer, the provider can call the consumer's NDK callback functions. These are documented in NDKPI Consumer Callback Functions.

NDK providers must implement all elements of the NDKPI interface that are documented in the NDKPI Reference, except for the NDKPI Consumer Callback Functions.

The NDK consumer

NDK consumers are kernel-mode Windows components, such as SMB server and client.

Note This documentation does not discuss how to implement an NDK consumer. The NDKPI consumer device driver interface (DDI) is a proprietary Windows-internal interface.

An NDK consumer calls the provider's *NdkOpenAdapter* (*OPEN_NDK_ADAPTER_HANDLER*) callback function to create an adapter object and *NdkCloseAdapter* (*NDK_FN_CLOSE_OBJECT*) to close it. Once the provider has created the adapter object, the consumer calls other provider callback functions to create additional NDK objects.

NDK consumers implement the NDKPI Consumer Callback Functions, which are called by NDK providers.

NDKPI Terminology

Article • 12/15/2021

The NDKPI documentation uses the following terms to describe NDK providers and consumers.

provider function

An NDKPI function in the function dispatch table of an NDK object. Provider functions are implemented by NDK providers and called by NDK consumers. All provider functions have as the first parameter a pointer to the object on which they operate. This pointer is similar to the "this" pointer in C++. This pointer is always passed explicitly by the consumer to the provider function.

consumer callback

An NDKPI function implemented by NDK consumers and called by NDK providers. There are 2 types of consumer callbacks: completion callbacks and event callbacks.

completion callback

A consumer callback that is called by the NDK provider to signal the completion of an asynchronous provider function. In NDKPI 1.1 and 1.2, there are 3 completion callbacks:

- *NdkCloseCompletion* (*NDK_FN_CLOSE_COMPLETION*)
- *NdkCreateCompletion* (*NDK_FN_CREATE_COMPLETION*)
- *NdkRequestCompletion* (*NDK_FN_REQUEST_COMPLETION*)

event callback

A consumer callback that can be called by the NDK provider to indicate certain events on an NDK object asynchronously without being triggered by an asynchronous provider function. In NDKPI 1.1 and 1.2, there are 4 event callbacks:

- *NdkCqNotificationCallback* (*NDK_FN_CQ_NOTIFICATION_CALLBACK*)
- *NdkConnectEventCallback* (*NDK_FN_CONNECT_EVENT_CALLBACK*)
- *NdkDisconnectEventCallback* (*NDK_FN_DISCONNECT_EVENT_CALLBACK*)
- *NdkSrqNotificationCallback* (*NDK_FN_SRQ_NOTIFICATION_CALLBACK*)

parent object

An NDK object whose function dispatch table contains one or more *NdkCreate*Xxx** provider functions to create other objects. in NDKPI versions 1.1 and 1.2, there are 2 parent objects:

The NDK adapter object (**NDK_ADAPTER**) is the parent of:

- **NDK_CONNECTOR**
- **NDK_CQ**
- **NDK_LISTENER**
- **NDK_LOGICAL_ADDRESS_MAPPING**
- **NDK_PD**
- **NDK_SHARED_ENDPOINT**

The NDK protection domain (PD) object (**NDK_PD**) is the parent of:

- **NDK_MR**
- **NDK_MW**
- **NDK_QP**
- **NDK_SRQ**

child object

An NDK object which is created by calling one of the *NdkCreate*Xxx** provider functions in a parent object's dispatch table.

antecedent object

An NDK object that another object relies on in order to provide functionality. The antecedent object must be created before the successor object. Note that all parent objects are antecedent objects, but the reverse is not true.

successor object

An NDK object that relies on an antecedent object. The antecedent object must be created before the successor object. Note that all child objects are successor objects but the reverse is not true. Note that an antecedent/successor relationship may be required, optional, and/or deferred to a point after the successor creation in some cases.

The following antecedent/successor relationships are defined by NDKPI versions 1.1 and 1.2 (in addition to the parent/child relationships, which are antecedent/successor relationships by definition):

Antecedent	Successor
NDK_CQ	NDK_QP
NDK_MR	NDK_MW (See *NdkBind* (*NDK_FN_BIND*).)
NDK_SRQ	NDK_QP
NDK_QP	NDK_CONNECTOR (See *NdkConnect* (*NDK_FN_CONNECT*), *NdkAccept* (*NDK_FN_ACCEPT*), and *NdkConnectWithSharedEndpoint* (*NDK_FN_CONNECT_WITH_SHARED_ENDPOINT*).)
NDK_SHARED_ENDPOINT	NDK_CONNECTOR (See *NdkConnectWithSharedEndpoint* (*NDK_FN_CONNECT_WITH_SHARED_ENDPOINT*).)
NDK_LISTENER	NDK_CONNECTOR (See *NdkConnectEventCallback* (*NDK_FN_CONNECT_EVENT_CALLBACK*).)

endpoint

An implicit or explicit representation of a local address and NetworkDirect port number that identify the local point over which connections can be initiated or accepted, for example, 10.1.1.1:445:

- An NDK_LISTENER has an implicit endpoint (which the consumer specifies when calling *NdkListen* (*NDK_FN_LISTEN*)).

- An **NDK_CONNECTOR** that is connected by calling *NdkConnect* (*NDK_FN_CONNECT*) also has an implicit endpoint.
- An **NDK_SHARED_ENDPOINT** represents an explicit endpoint. The NDK consumer directly creates the endpoint and uses it explicitly to initiate one or more connections by calling *NdkConnectWithSharedEndpoint* (*NDK_FN_CONNECT_WITH_SHARED_ENDPOINT*).

Note An NDK endpoint is not the same as the NDSPI version 1 **INDEndpoint** interface or the NDSPI version 2 **INDQueuePair** interface.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

Background Reading on RDMA

Article • 12/15/2021

RDMA is a networking technology that provides high-throughput, low-latency communication that minimizes CPU usage. NDK currently supports the following RDMA technologies:

- Infiniband (IB)
- Internet Wide Area RDMA Protocol (iWARP)
- RDMA over Converged Ethernet (RoCE)

For more information about RDMA, Infiniband, iWARP, and RoCE, see the following resources:

- RFC 5040: A Remote Direct Memory Access Protocol Specification
- RFC 5041: Direct Data Placement over Reliable Transports
- RFC 5042: Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security
- RFC 5043: Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation
- RFC 5044: Marker PDU Aligned Framing for TCP Specification
- RDMA Consortium
- Internet Draft: RDMA Protocol Verbs Specification
- Infiniband Trade Association (for downloadable specifications for Infiniband and RoCE)

Related topics

Network Direct Kernel Provider Interface (NDKPI)

Initializing an NDK-Capable Miniport Driver

Article • 12/15/2021

A miniport driver that supports Network Direct kernel (NDK) is initialized in the same way as other miniport drivers. However, it must also register additional NDKPI entry points.

- DriverEntry function
- MiniportSetOptions function
- Related topics

DriverEntry function

Every miniport driver's *DriverEntry* function initializes an **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure and passes it to **NdisMRegisterMiniportDriver** as described in the following pages:

- Initializing a Miniport Driver
- **DriverEntry of NDIS Miniport Drivers function**

The NDK-capable miniport driver must do the following when initializing the **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure:

- In the **OidRequestHandler** member, the miniport driver must register a *MiniportOidRequest* function that supports:

    - All NDKPI OIDs.

    - Any OIDs that are mandatory for NDIS miniport drivers in general.

      Note For a list of these mandatory OIDs, see Mandatory OIDs for Miniport Drivers.

- In the **SetOptionsHandler** member, the miniport driver must register a *MiniportSetOptions* function as described in Configuring Optional Miniport Driver Services and the following MiniportSetOptions function section.

MiniportSetOptions function

NDIS calls the *MiniportSetOptions* function immediately after the miniport driver's **DriverEntry** function returns. The *MiniportSetOptions* function is called in the context of the miniport driver's call to **NdisMRegisterMiniportDriver**.

In its *MiniportSetOptions* function, the NDK-capable miniport driver registers its NDK capability and registers the following required NDKPI function entry points as described in Configuring Optional Miniport Driver Services:

- *OpenNDKAdapterHandler* (*OPEN_NDK_ADAPTER_HANDLER*)
- *CloseNDKAdapterHandler* (*CLOSE_NDK_ADAPTER_HANDLER*)

To register NDKPI entry points for these functions, the miniport driver's *MiniportSetOptions* function must do the following:

1. Initialize an **NDIS_NDK_PROVIDER_CHARACTERISTICS** structure.

   Note Pay particular attention to the **Header** member description. The miniport driver must set this member correctly to identify itself as an NDK-capable miniport driver.

2. Store the function entry points in the **OpenNDKAdapterHandler** and **CloseNDKAdapterHandler** members of the structure.

3. Call the **NdisSetOptionalHandlers** function, passing the structure in the *OptionalHandlers* parameter.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

Initializing an NDK Miniport Adapter

Article • 12/15/2021

A Network Direct kernel (NDK) miniport adapter is initialized in the same way as other miniport adapters: NDIS calls the miniport adapter's *MiniportInitializeEx* function as described in Initializing a Miniport Adapter. This topic describes the NDK-specific requirements for the miniport adapter's *MiniportInitializeEx* function.

In its *MiniportInitializeEx* function, the miniport driver must do the following:

1. Populate an **NDIS_MINIPORT_ADAPTER_NDK_ATTRIBUTES** structure for the adapter as follows:

    - The miniport driver sets the **Header** member as described in the member description to identify the adapter as an NDK-capable miniport adapter.

    - The miniport driver sets the **Enabled** member to **TRUE** if its NDK functionality is enabled, or **FALSE** otherwise.

        > ⓘ Note
        >
        > For more information about querying and setting the current state of the miniport driver's NDK functionality, see **Enabling and Disabling NDK Functionality**.

    - In the **NdkCapabilities** member, the miniport driver stores a pointer to an **NDIS_NDK_CAPABILITIES** structure that specifies the capabilities of the adapter.

2. Call **NdisMSetMiniportAttributes** to set these attributes for the adapter.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

Implementing NDKPI Functions

Article • 12/15/2021

An NDK-capable miniport driver must register entry points for all NDK_FN_*XXX* callback functions. All of the NDKPI provider callback functions are mandatory; none are optional.

To register support for these functions, the miniport driver stores their entry points in the structures listed in the "Object's Dispatch Table" column of the following table:

Object Type	Created By This Function	Object's Dispatch Table
NDK_ADAPTER	*OPEN_NDK_ADAPTER_HANDLER*	NDK_ADAPTER_DISPATCH
NDK_CONNECTOR	*NDK_FN_CREATE_CONNECTOR*	NDK_CONNECTOR_DISPATCH
NDK_CQ	*NDK_FN_CREATE_CQ*	NDK_CQ_DISPATCH
NDK_LISTENER	*NDK_FN_CREATE_LISTENER*	NDK_LISTENER_DISPATCH
NDK_MR	*NDK_FN_CREATE_MR*	NDK_MR_DISPATCH
NDK_MW	*NDK_FN_CREATE_MW*	NDK_MW_DISPATCH
NDK_PD	*NDK_FN_CREATE_PD*	NDK_PD_DISPATCH
NDK_QP	*NDK_FN_CREATE_QP* or *NDK_FN_CREATE_QP_WITH_SRQ*	NDK_QP_DISPATCH
NDK_SHARED_ENDPOINT	*NDK_FN_CREATE_SHARED_ENDPOINT*	NDK_SHARED_ENDPOINT_DISPATCH
NDK_SRQ	*NDK_FN_CREATE_SRQ* or *NDK_FN_CREATE_QP_WITH_SRQ*	NDK_SRQ_DISPATCH

Related topics

Network Direct Kernel Provider Interface (NDKPI)

INF Requirements for NDKPI

Article • 06/02/2023

The INF file for a miniport driver that supports Network Direct kernel (NDK) must meet the following requirements.

NDIS upper range value

The miniport driver's INF file must specify an NDIS upper range value of "ndis5" in order for Windows components to discover and use the NDK-capable miniport adapters that are serviced by the driver. This value is specified as follows:

INF

```
HKR, Ndi\Interfaces, UpperRange, 0, "ndis5"
```

*NetworkDirect INF keyword

The INF file must specify the ***NetworkDirect** keyword value as follows:

- Once the driver is installed, administrators can update the ***NetworkDirect** keyword value in the **Advanced** property page for the adapter.

Note: The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter.

INF

```
HKR, Ndi\Params\*NetworkDirect,        ParamDesc,  0, "NetworkDirect
Functionality"
HKR, Ndi\Params\*NetworkDirect,        Type,       0, "enum"
HKR, Ndi\Params\*NetworkDirect,        Default,    0, "1"
HKR, Ndi\Params\*NetworkDirect\enum,   "0",        0, "Disabled"
HKR, Ndi\Params\*NetworkDirect\enum,   "1",        0, "Enabled"
```

*NetworkDirectTechnology INF keyword

The INF file must specify the ***NetworkDirectTechnology** keyword value as follows:

- Once the driver is installed, administrators can update the ***NetworkDirectTechnology** keyword value in the **Advanced** property page for the adapter. The enumerations are mutually exclusive, meaning the selection of a NetworkDirectTechnology value excludes all others. This allows for the Platform to define strict device behavior.

- A device must express only the supported transports. The transport values are identifiers which map to WDK **NDK_RDMA_TECHNOLOGY**. A redefinition of the identifiers is prohibited.
- The behavior of devices with multiple concurrent transports is undefined. The device **must** specify a transport type.

Note: The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter.

INF

```
HKR, Ndi\Params\*NetworkDirectTechnology,        ParamDesc,  0,
"NetworkDirect Technology"
HKR, Ndi\Params\*NetworkDirectTechnology,        Default,    0,  "1"
HKR, Ndi\Params\*NetworkDirectTechnology,        Type,       0,  "enum"
HKR, Ndi\Params\*NetworkDirectTechnology\enum,   1,          0,  "iWARP"
HKR, Ndi\Params\*NetworkDirectTechnology\enum,   2,          0,
"InfiniBand"
HKR, Ndi\Params\*NetworkDirectTechnology\enum,   3,          0,  "RoCE"
HKR, Ndi\Params\*NetworkDirectTechnology\enum,   4,          0,  "RoCEv2"
HKR, Ndi\Params\*NetworkDirectTechnology,        Optional,   0,  "0"
```

*NetworkDirectRoCEFrameSize INF keyword

The INF file for a miniport driver that supports ***NetworkDirectRoCEFrameSize** must meet the following requirements:

- The ***NetworkDirectRoCEFrameSize** keyword specifies the administrator requested maximum transmission unit for NetworkDirect communications. Adapters supporting the ***NetworkDirect** keyword with **RoCE** or **RoCEv2** must additionally support this keyword.

- The acceptable registry values for ***NetworkDirectRoCEFrameSize** are 256, 512, 1024, 2048, and 4096. The value of 1024 is required.

- The adapter must use the largest supported size for ***NetworkDirectRoCEFrameSize** that doesn't exceed ***JumboPacket**.

- If the configured value of ***NetworkDirectRoCEFrameSize** differs from the operational (active) RoCE MTU, the driver must log an event in the system event log indicating operational (active) RoCE MTU.

Note: The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter unless the change can be made effective without the restart.

The following table describes the ***NetworkDirectRoCEFrameSize** keyword and values that can be edited. The min and max values define the required limits for supported values. An individual adapter can support a lower minimum value or higher maximum value but must support at least these values.

SubkeyName	ParamDesc	Type	Default value	Min	Max
***NetworkDirectRoCEFrameSize**	Network Direct Maximum Transmission Unit	enum	1024	256	4096

For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

For more information about using standardized INF keywords, see Standardized INF Keywords for Network Devices.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

Enabling and Disabling NDK Functionality

Article • 12/15/2021

To enable or disable NDK functionality, NDIS issues an OID_NDK_SET_STATE OID request. An NDK-capable miniport driver must register support for this OID in its *MiniportOidRequest* function.

Determining whether NDK functionality can be enabled

The ***NetworkDirect** keyword determines whether the miniport driver's NDK functionality can be enabled.

If this keyword value is set to 1 ("Enabled"), NDK functionality can be enabled.

If it is set to 0 ("Disabled"), NDK functionality cannot be enabled.

When the miniport driver is installed, its INF file sets this keyword value to 1 ("Enabled") by default. For more information, see INF Requirements for NDKPI.

After the miniport driver is installed, administrators can update the ***NetworkDirect** keyword value by setting a new value in the **Advanced** property page for the adapter. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

Note The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter.

When to enable or disable NDK functionality

This state change can be triggered by an OID_NDK_SET_STATE OID request, or by a success or failure in the adapter itself.

Enabling or disabling NDK functionality

To enable or disable its NDK functionality, the miniport driver must send a **NetEventNDKEnable** or **NetEventNDKDisable** Plug and Play (PnP) event to NDIS.

To send the PnP event, the miniport driver calls the **NdisMNetPnPEvent** function, setting the **NetPnPEvent** member of the **NET_PNP_EVENT_NOTIFICATION** structure that the *NetPnPEvent* parameter points to as follows:

- **NetEventNDKEnable** if NDK functionality is to be enabled.
- **NetEventNDKDisable** if NDK functionality is to be disabled.

The **NetEventNDKDisable** PnP event triggers NDIS and upper layer drivers to start closing their opened **NDK_ADAPTER** instances over the adapter where the NDK functionality is being disabled. The PnP event will remain pending until all of the opened **NDK_ADAPTER** instances over the adapter are closed.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

NDKPI Object Lifetime Requirements

Article • 10/05/2022

How NDK Objects Are Created, Used, and Closed

An NDK consumer initiates a create request for an NDK object by calling the NDK provider's create function for that object.

When the consumer calls a create function, it passes an *NdkCreateCompletion* (*NDK_FN_CREATE_COMPLETION*) as a parameter.

The consumer initiates various requests by calling provider functions in the object's dispatch table, passing an *NdkRequestCompletion* (*NDK_FN_REQUEST_COMPLETION*) completion callback as a parameter.

When an object is no longer needed, the consumer calls the provider's *NdkCloseObject* (*NDK_FN_CLOSE_OBJECT*) function to initiate a close request for the object, passing an *NdkCloseCompletion* (*NDK_FN_CLOSE_COMPLETION*) callback as a parameter.

The provider calls the consumer's callback function to complete the request asynchronously. This call indicates to the consumer that the provider has completed the operation (for example, closing the object) and is returning control to the consumer. If the provider completes the close request synchronously, either successfully or in error, it won’t call the consumer’s callback function.

The Rules for Completion Callbacks

When a provider has created an object at the request of a consumer, the provider calls the consumer's *NdkCreateCompletion* callback to indicate that the object is ready for use.

The consumer can call other provider functions for the same object without waiting for the first callback to return.

The consumer will not call the *NdkCloseObject* function for an object until all provider functions for that object have returned.

However, if the provider function initiates a completion request, the consumer is free to call *NdkCloseObject* from inside that completion callback, even if the provider function hasn't returned.

A provider function can initiate a completion request before returning from a callback by doing one of the following:

- Calling the completion callback directly
- Queuing the completion request to another thread

By initiating a completion request, the provider effectively returns control to the consumer. The provider must assume that the object can be closed at any time after the provider initiates the completion request.

Note To prevent deadlock after initiating a completion request, the provider must either:

- Not perform other operations on the object until the completion callback returns.
- Take the necessary measures to keep the object intact, if the provider absolutely must touch the object.

Example: Consumer-Provider Interaction

Consider the following scenario:

1. The consumer creates a connector (**NDK_CONNECTOR**) and then calls *NdkConnect* (*NDK_FN_CONNECT*).
2. The provider processes the connect request, hits a failure, and calls the consumer's completion callback in the context of the *NdkConnect* call (as opposed to returning inline failure due to an internal implementation choice).
3. The consumer calls *NdkCloseObject* in the context of this completion callback, even though the *NdkConnect* call has not yet returned to the consumer.

To avoid deadlock, the provider must not touch the connector object after step 2 (the point when it initiated the completion callback inside the *NdkConnect* call).

Closing Antecedent and Successor Objects

The provider must be prepared for the consumer to call the *NdkCloseObject* function to close an antecedent object before the consumer calls *NdkCloseObject* for successor objects. If the consumer does this, here's what the provider must do:

- The provider must not close the antecedent object until all the successor objects are closed, i.e., provider must return STATUS_PENDING from the close request and complete it (by calling the registered *NdkCloseCompletion* function for the close request) once all successor objects are closed.

- The consumer will not use the antecedent object after calling *NdkCloseObject* on it, so the provider does not have to add any handling for failing further provider functions on the antecedent object (but it may if it chooses to).
- The provider may treat the close request like a simple dereference which has no other side-effect until the last successor object is closed, unless otherwise required (see the NDK listener close case below which has a required side-effect).

The provider must not complete the close request on an antecedent object (including the NDK_ADAPTER close request) before any in-progress close completion callback on any successor object returns to the provider. This is to allow NDK consumers to unload safely.

An NDK consumer will not call *NdkCloseObject* for an NDK_ADAPTER object (which is a blocking call) from inside a consumer callback function.

Closing Adapter Objects

Consider the following scenario:

1. The consumer calls *NdkCloseObject* on a completion queue (CQ) object.
2. The provider returns STATUS_PENDING, and later calls the consumer's completion callback.
3. Inside this completion callback, the consumer signals an event that it's now OK to close the NDK_ADAPTER.
4. Another thread wakes up upon this signal, and closes the NDK_ADAPTER and proceeds to unload.
5. However, the thread in which the consumer's CQ close completion callback was called might still be inside the consumer's callback function (for example, the function epilog), so it's not safe for the consumer driver to unload.
6. Because the completion callback context is the only context the consumer can signal the event, the consumer driver can't solve the safe-unload issue itself.

There must be a point at which the consumer can be assured that all of its callbacks have returned control. In NDKPI, this point is when the close request on a NDK_ADAPTER returns control. Note that **NDK_ADAPTER** close request is a blocking call. When an **NDK_ADAPTER** close request returns, it's guaranteed that all callbacks on all objects that descend from that **NDK_ADAPTER** object have returned control to the provider.

Completing Close Requests

The provider must not complete a close request on an object until:

- All pending asynchronous requests on the object have been completed (in other words, their completion callbacks have returned to the provider).
- All of the consumer's event callbacks (for example, *NdkCqNotificationCallback* (*NDK_FN_CQ_NOTIFICATION_CALLBACK*) on a CQ, *NdkConnectEventCallback* (*NDK_FN_CONNECT_EVENT_CALLBACK*) on a Listener) have returned to the provider.

The provider must guarantee that no more callbacks will happen after the close completion callback is called or after the close request returns STATUS_SUCCESS. Note that a close request must also initiate any needed flushing or cancellation of pending asynchronous requests.

Note It logically follows from the above that an NDK consumer must not call *NdkCloseObject* for an **NDK_ADAPTER** object (which is a blocking call) from inside a consumer callback function.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

NDKPI Listeners, Connectors, and Endpoints

Article • 12/15/2021

An NDK consumer connects an NDK connector by calling the *NdkConnect* (*NDK_FN_CONNECT*) or *NdkConnectWithSharedEndpoint* (*NDK_FN_CONNECT_WITH_SHARED_ENDPOINT*) function.

Each connector that is in a connected state also has an underlying endpoint that represents the local end of the established NDK connection:

- A connector that is established by accepting an incoming connection over an NDK listener automatically inherits the listener's implicit endpoint as its local implicit endpoint.
- A connector that is connected via the *NdkConnect* function has its own dedicated implicit local endpoint.
- A connector that is connected via the *NdkConnectWithSharedEndpoint* function has an explicit local endpoint that can be shared with other connectors that are also connected via the *NdkConnectWithSharedEndpoint* function.

The NDK provider must keep some sort of reference count for each implicit or explicit endpoint, and release the endpoint (i.e., mark the address/port as available to be used again) when the reference count reaches zero:

Reference Counting for (Non-Shared) Endpoints

When the consumer calls the *NdkListen* (*NDK_FN_LISTEN*) function, the provider creates an implicit endpoint. For this implicit endpoint, the provider must maintain a reference count as follows:

- Add a reference for the listener itself to the endpoint's reference count.
- Add a reference for each connector that is accepted over that listener.
- Remove a reference when a connector that was previously accepted over the listener is closed.
- Remove a reference when the listener itself is closed. **Note** You can't close the listener until all the connectors are closed.
- Release the endpoint when its reference count returns to zero. (This is the case only when the listener and all the connectors accepted over the listener have been

closed.)

- Simply closing the listener does not release the endpoint as long as there are previously accepted connectors that are not yet closed. This means that new *NdkListen*, *NdkConnect*, and *NdkConnectWithSharedEndpoint* requests for the same local address and port will fail until all such connections are closed. Note that the close request on the listener will also remain pending until all such connections are closed (due to the antecedent/successor rules outlined in NDKPI Object Lifetime Requirements). The provider must reject further incoming connections on the listener as soon as a close request is issued (so that no new connections are accepted while the close request is pending).

Reference Counting for Connectors

When the consumer calls *NdkConnect*, the provider creates and implicit endpoint. For this implicit endpoint, the provider must:

- Add a reference by the connector. There is only one connector, hence only one reference.
- Remove the connector's reference to the endpoint when the connector is closed.
- Release the endpoint when that reference is gone.

Reference Counting for Shared Endpoints

When the consumer calls *NdkConnectWithSharedEndpoint*, the provider creates an explicit shared endpoint. For this explicit shared endpoint, the provider must:

- Add a reference for the shared endpoint itself to the shared endpoint's reference count.
- Add a reference for each connector that is connected over that shared endpoint.
- Remove a reference when a connector that was previously connected over the shared endpoint is closed.
- Release the endpoint the reference count returns to zero. (This is the case when the shared endpoint and all the connectors connected over the shared endpoint have been closed.)
- Simply closing the shared endpoint does not release the endpoint as long as there are previously connected connectors that are not yet closed. This means that new *NdkListen*, *NdkConnect*, and *NdkConnectWithSharedEndpoint* requests for the same local address and port will fail until all such connections are closed. Note that the close request on the shared endpoint will also remain pending until all such

connections are closed (due to the antecedent/successor rules outlined in NDKPI Object Lifetime Requirements).

Related topics

Network Direct Kernel Provider Interface (NDKPI)

NDKPI Completion Handling Requirements

Article • 12/15/2021

NDK consumers and NDK providers must follow these requirements for NDKPI completion handling.

The Rules for NdkGetCqResults, NdkGetCqResultsEx, and NdkArmCq Functions

The consumer will always serialize its calls to these provider functions on the same completion queue (CQ) object (**NDK_CQ**):

- *NdkGetCqResults* (*NDK_FN_GET_CQ_RESULTS*)
- *NdkGetCqResultsEx* (*NDK_FN_GET_CQ_RESULTS_EX*)
- *NdkArmCq* (*NDK_FN_ARM_CQ*)

This means not only that the consumer will never call the same provider function multiple times concurrently, but also that it will never call any combination of these functions concurrently on the same CQ from multiple threads.

An **NdkOperationTypeReceiveAndInvalidate** completion that occurs as a result of a remote *NdkSendAndInvalidate* (*NDK_FN_SEND_AND_INVALIDATE*) call must still be retrievable using *NdkGetCqResults* (not *NdkGetCqResultsEx*n). Doing so must still invalidate the specified token on the receiver, but will not notify the receiving consumer of this invalidation (the consumer must use *NdkGetCqResultsEx* to get this information). A later *NdkInvalidate* (*NDK_FN_INVALIDATE*) for the same token will fail, as usual.

The Rules for Notification Callbacks

The provider must call the *NdkCqNotificationCallback* (*NDK_FN_CQ_NOTIFICATION_CALLBACK*) callback only once, and only after the consumer has armed the *NdkCqNotificationCallback* callback by calling *NdkArmCq*. That is, the provider must clear the arm and call the *NdkCqNotificationCallback* callback when the conditions for calling the *NdkCqNotificationCallback* callback occur (in other words, when request completions are queued in the CQ).

If there are completions already present in the CQ when the consumer calls *NdkArmCq*, the provider will behave as follows:

- If at least one of the completions was newly placed into the CQ since the last *NdkCqNotificationCallback* callback was called, the provider must satisfy the arm request immediately (see below for serialization requirements).
- However, if all completions in the CQ were present also when the last *NdkCqNotificationCallback* callback was called (in other words, the consumer called *NdkArmCq* without removing all the completions and no new completions got placed into the CQ), then the provider may satisfy the arm request immediately.

When the provider needs to call the *NdkCqNotificationCallback* callback, if there's already a *NdkCqNotificationCallback* callback in progress, then the provider must defer the invocation of the *NdkCqNotificationCallback* callback until after the existing call to the *NdkCqNotificationCallback* callback returns control to the provider. In other words, the provider is responsible for serializing the *NdkCqNotificationCallback* callbacks.

The following table shows the resulting arm type if *NdkArmCq* is called a second time before a previous *NdkArmCq* request is satisfied:

	2nd arm ANY	**2nd arm ERRORS**	**2nd arm SOLICITED**
1st arm ANY	ANY	ANY	ANY
1st arm ERRORS	ANY	ERRORS	SOLICITED
1st arm SOLICITED	ANY	SOLICITED	SOLICITED

Related topics

Network Direct Kernel Provider Interface (NDKPI)

NDKPI Work Request Posting Requirements

Article • 12/15/2021

Work Request Posting Rules for the Consumer

The NDK consumer will post the following types of work requests on the initiator queue:

- *NdkBind* (*NDK_FN_BIND*)
- *NdkFastRegister* (*NDK_FN_FAST_REGISTER*)
- *NdkInvalidate* (*NDK_FN_INVALIDATE*)
- *NdkRead* (*NDK_FN_READ*)
- *NdkSend* (*NDK_FN_SEND*)
- *NdkSendAndInvalidate* (*NDK_FN_SEND_AND_INVALIDATE*)
- *NdkWrite* (*NDK_FN_WRITE*)

The consumer will post *NdkReceive* (*NDK_FN_RECEIVE*) requests on the receive queue.

The consumer will post all of these requests to the same individual queue on an **NDK_QP** or **NDK_SRQ** in a serialized fashion. In other words, the consumer will never have two concurrent calls to any work request functions on the same individual queue belonging to an **NDK_QP** or **NDK_SRQ**.

This means, for example, that concurrent *NdkReceive* calls won't be issued, concurrent *NdkSend* and *NdkWrite* calls won't be issued, but concurrent *NdkReceive* and *NdkWrite* calls may be issued on the same **NDK_QP**.

Work Request Posting Rules for the Provider

The provider should not have any redundant locks inside the above work request functions, because they are guaranteed to be serialized by the consumer.

The provider must be able to handle *NdkFlush* (*NDK_FN_FLUSH*) calls that may be called concurrently with a work request call on the same **NDK_QP**.

The provider must be able to handle an **NdkCloseConnector** call (on the successor **NDK_CONNECTOR** object for the **NDK_QP**) that may be called concurrently with a work request call on the same **NDK_QP**.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

NetworkDirect Disconnect Scheme

Article • 12/15/2021

The scheme described here applies to both NDSPI version 2 and NDKPI. The following terms are used:

- ND is used to refer to NDSPI or NDK.
- *NdDisconnect* is used to refer to the function call that an ND consumer makes in order to initiate a graceful disconnect. For NDSPI, this is **INDConnector::Disconnect**. For NDKPI, it is *NdkDisconnect* (*NDK_FN_DISCONNECT*).
- *NdDisconnectIndication* is used to refer to the indication delivered by an ND provider to an ND consumer when the ND provider receives a graceful disconnect from the peer or detects that the connection was aborted due to any reason (other than the local NDK consumer's own initiation such as issuing *NdDisconnect* or *NdCloseConnector*).

Below, A and B refer to the two sides of an ND connection. Consumer A refers to the ND consumer on side A, provider A refers to the ND provider on side A, and similarly Consumer B/Provider B refers to those same entities on side B. When consumer A calls *NdDisconnect*, provider A must send a graceful disconnect notification to side B and complete consumer A's *NdDisconnect* request only when both of the following conditions occur:

- Either:
    - A graceful disconnect notification is received from B (which leads to successful completion of consumer A's *NdDisconnect*), or
    - An error such as connection abortion or time-out occurred (which leads to consumer A's *NdDisconnect* to be completed with a failure).
- All DMA activity on the QP is finished (including DMA activity for work requests that were posted with silent-success flag).

When provider B receives a graceful disconnect notification from A or detects that the connection is aborted, provider B must deliver *NdDisconnectIndication* to consumer B if consumer B has not called *NdDisconnect* to provider B already. Since an incoming graceful disconnect notification or an abort event can race with the local consumer initiating *NdDisconnect*, local consumer must be prepared to handle an *NdDisconnectIndication* arriving after local consumer calls *NdDisconnect*. Note that an *NdDisconnectIndication* does not provide any guarantees in terms of work request completions.

A disconnected QP or connector cannot be reused by the consumer.

NetworkDirect does not have any notion of half-closed connections. Once *NdDisconnect* is completed (with success or failure), the connection is fully closed.

A consumer should typically call *NdDisconnect* only after it gets completions for all work requests it posted to the initiator queue. Otherwise, the *NdDisconnect* may not lead to a true graceful disconnect. Providers are not required to support graceful disconnect in the case where a consumer leaves such work requests outstanding.

Related topics

Network Direct Kernel Provider Interface (NDKPI)

NDKPI Deferred Processing Scheme

Article • 12/15/2021

There are many cases where an NDK consumer will post a chain of initiator requests to the queue pair (QP). For example, a consumer could post a number of fast register requests followed by a send request. The performance for such request patterns may be improved if the chain of requests is queued to the QP and then indicated to the hardware for processing as a batch, rather than indicating each request in the chain to the hardware, one by one.

The **NDK_OP_FLAG_DEFER** flag value can be used for this purpose with the following request types:

- *NdkBind* (*NDK_FN_BIND*)
- *NdkFastRegister* (*NDK_FN_FAST_REGISTER*)
- *NdkInvalidate* (*NDK_FN_INVALIDATE*)
- *NdkRead* (*NDK_FN_READ*)
- *NdkSend* (*NDK_FN_SEND*)
- *NdkSendAndInvalidate* (*NDK_FN_SEND_AND_INVALIDATE*)
- *NdkWrite* (*NDK_FN_WRITE*)

The presence of the flag is a hint to the NDK provider that it may defer indicating the request to hardware for processing, but the provider may process the new request at any time.

The presence of the **NDK_OP_FLAG_DEFER** flag on an initiator request does not change the NDK provider's existing responsibilities with respect to generating completions. A call to the initiator request that returns a failure status must not result in a completion being queued to the CQ for the failed request. Conversely, a call that returns a success status must eventually result in a completion being queued to the CQ as long as the consumer follows the additional requirements listed below.

In addition to all the existing NDK requirements, two additional requirements (one for the provider and one for the consumer) must be observed to prevent a situation in which requests are successfully posted to the QP with the **NDK_OP_FLAG_DEFER** flag, but are never indicated to the hardware for processing:

- When returning a failure status from a call to an initiator request, the provider must guarantee that all requests that were previously submitted with the **NDK_OP_FLAG_DEFER** flag are indicated to the hardware for processing.
- The consumer guarantees that, in the absence of an inline failure, all initiator request chains will be terminated by an initiator request that does not set the

NDK_OP_FLAG_DEFER flag.

For example, consider a case where a consumer has a chain of two fast register requests and a send that it needs to post to the QP:

1. The consumer posts the first fast register with the **NDK_OP_FLAG_DEFER** flag and *NdkFastRegister* returns STATUS_SUCCESS.
2. Again, the second fast register is posted with the **NDK_OP_FLAG_DEFER** flag set but now *NdkFastRegister* returns a failure status. In this case, the consumer will not post the send request.
3. When returning the inline failure for the second call to *NdkFastRegister*, the NDK provider makes sure that all previously unindicated requests (the first fast register in this case) are indicated to the hardware for processing.
4. Because the first call to *NdkFastRegister* succeeded, a completion must be generated to the CQ.
5. Because the second call to *NdkFastRegister* failed inline, a completion must not be generated to the CQ.

Related topics

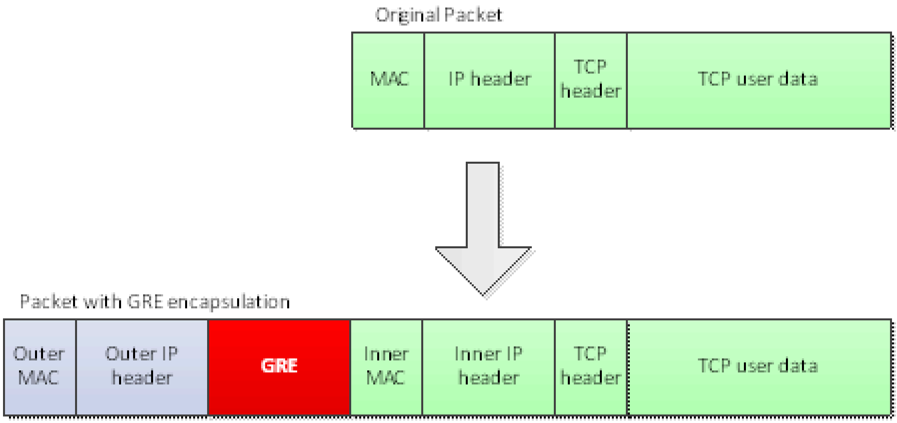
Network Direct Kernel Provider Interface (NDKPI)

About Network Virtualization using Generic Routing Encapsulation (NVGRE)

Article • 06/15/2023

Hyper-V Network Virtualization supports Network Virtualization using Generic Routing Encapsulation (NVGRE) as the mechanism to virtualize IP addresses. In NVGRE, the virtual machine's packet is encapsulated inside another packet. The header of this new, NVGRE-formatted packet has the appropriate source and destination provider area (PA) IP addresses. In addition, it has a 24-bit Virtual Subnet ID (VSID), which is stored in the GRE header of the new packet.

The following figure shows a GRE-encapsulated packet. On the wire, NVGRE-encapsulated packets look like IP-over-Ethernet packets, except that the payload of the outer IP header is a GRE-encapsulated IP packet (including the Ethernet header).



NDIS 6.30 (available in Windows Server 2012 and later) introduces NVGRE Task Offload, which makes it possible to use NVGRE-formatted packets with:

- Large Send Offload (LSO)
- Virtual Machine Queue (VMQ)
- Transmit (Tx) checksum offload (IPv4, TCP, UDP)
- Receive (Rx) checksum offload (IPv4, TCP, UDP)

NDIS 6.85 introduces support for NVGRE with UDP segmentation offload (USO).

Note: It is possible for a protocol driver to offload "mixed mode" packets, which means packets in which the inner and outer IP header versions are different. For example, a packet could have outer IP header as IPv6 and the inner IP header as IPv4.

Note: It is also possible for a protocol driver to offload an NVGRE-formatted packet that has no inner TCP or UDP header. For example, an IP packet could have an inner payload that is an Internet Control Message Protocol (ICMP) packet.

For more information about NVGRE, see the following Internet Draft:

- NVGRE: Network Virtualization using Generic Routing Encapsulation

NVGRE is based on Generic Routing Encapsulation (GRE). For more information about GRE, see the following resources:

- RFC 2784: Generic Routing Encapsulation (GRE)
- RFC 2890: Key and Sequence Number Extensions to GRE

This section includes:

- Overview of Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload
- Supporting NVGRE in Large Send Offload (LSO)
- Supporting NVGRE in UDP Segmentation Offload (USO)
- Supporting NVGRE in Checksum Offload
- Supporting NVGRE in RSS and VMQ Receive Task Offloads
- Locating the Transport Header for Encapsulated Packets in the Receive Path
- Determining the NVGRE Task Offload Capabilities of a Network Adapter
- Querying and Changing NVGRE Task Offload State
- Standardized INF Keywords for NVGRE Task Offload

Related topics

Offloading Checksum Tasks

Offloading the Segmentation of Large TCP Packets

TCP/IP Task Offload

Overview of Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload

Article • 06/15/2023

NVGRE Encapsulation Packet Format

In this case, a protocol or filter driver will generate the (non-LSO) packets, including the GRE encapsulation, and send the packets on the wire. On the receive side, these (non-RSS, VMQ) packets are passed to the protocol driver without any modifications. Note that the NVGRE Task Offload feature does not specify the offloading of the encapsulation and decapsulation operations.

Send and Receive Offloads

On the send path, the following task offloads need to account for encapsulation:

- Checksum computation of IPv4 and TCP or UDP payload
- Large Send Offload version 1 (LSO_v1) and Large Send Offload version 2 (LSO_v2)
- UDP Segmentation Offload (USO)

For send-side offloads, the miniport must perform corresponding operations on the tunnel (outer) IP header, the transport (inner) IP header, and the TCP header.

On the receive path, the following task offloads need to account for encapsulation:

- Checksum validation of IPv4 and TCP or UDP payload
- Receive side scaling (RSS)
- VMQ

For receive-side offloads, the NIC must parse the encapsulation protocol headers. For example, for GRE encapsulation, the NIC must parse the GRE header and perform task offloads on the transport (inner) and/or tunnel (outer) IP headers.

Supporting NVGRE in Large Send Offload (LSO)

Article • 12/15/2021

NDIS 6.30 (Windows Server 2012) introduces Network Virtualization using Generic Routing Encapsulation (NVGRE). NDIS miniport, protocol, and filter drivers and NICs that perform large send offload (LSO) version 2 (LSOV2) should do so in a way that supports NVGRE.

Note This page assumes that you are familiar with the information in Offloading the Segmentation of Large TCP Packets.

If NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**IsEncapsulatedPacket** is **TRUE** and the **TcpIpChecksumNetBufferListInfo** out-of-band (OOB) information is valid, this indicates that NVGRE support is required and the NIC must perform LSOV2 offload on the NVGRE-formatted packet, with the following conditions:

- Only the values in the NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO.**LsoV2Transmit** structure are valid. The NIC and miniport driver must not refer to the values in the **NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO.LsoV1Transmit** structure.
- The NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO.**LsoV2Transmit.TcpHeaderOffset** member does not have the correct offset value and must not be used by the NIC or miniport driver.

To support NVGRE in LSOV2, protocol and filter drivers must make the following changes:

- Reduce the **MSS** value in the NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO.**LsoV2Transmit** structure to account for the new GRE header.
- Send down a TCP payload length that may not be an exact multiple of the reduced **MSS** value.
- Adjust the **InnerFrameOffset**, **TransportIpHeaderRelativeOffset**, and **TcpHeaderRelativeOffset** values in the NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO structure to account for the GRE header.

NICs and miniport drivers may use the **InnerFrameOffset**, **TransportIpHeaderRelativeOffset**, and **TcpHeaderRelativeOffset** values provided in the NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO structure. The NIC or miniport driver may perform any needed header checks on the tunnel (outer) IP header or subsequent headers to validate these offsets.

Miniport drivers must handle the case where NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**InnerFrameOffset** may be in a different scatter-gather list than the beginning of the packet. The protocol driver will guarantee that all the prepended encapsulation headers (ETH, IP, GRE) will be physically contiguous and will be in the first MDL of the packet.

Protocol and filter drivers do not ensure that the total TCP payload length is an exact multiple of the reduced **MSS** value. For this reason, miniport drivers and NICs must update the tunnel (outer) IP header. NICs must generate as many full-sized segments as possible based on the reduced **MSS** value in the NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO.**LsoV2Transmit** OOB information. Only one sub-**MSS** segment may be generated per LSOv2 send.

Miniport drivers must do the following:

- Compute the checksum for the tunnel (outer) IP header.
- Increment the IP identification (IP ID) value of the tunnel (outer) IP header for every packet. The first packet must use the IP ID in the original tunnel (outer) IP header.
- Increment the IP ID of the transport (inner) IP header for every packet. The first packet must use the IP ID in the original transport (inner) IP header.
- Compute the checksum for the TCP header and the transport (inner) IP header.
- Ensure that the complete headers, including the encapsulation tunnel (outer) headers are added to every generated packet.

Related topics

Offloading the Segmentation of Large TCP Packets

Supporting NVGRE in UDP Segmentation Offload (USO)

Article • 06/15/2023

NDIS 6.85 introduces Network Virtualization using Generic Routing Encapsulation (NVGRE) with UDP segmentation offload (USO). NDIS miniport, protocol, and filter drivers, as well as NICs that perform USO, should support NVGRE and VXLAN encapsulations.

Note: This article presumes you're familiar with the information in UDP Segmentation Offload (USO).

If NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**IsEncapsulatedPacket** is **TRUE** and the **UdpSegmentationOffloadInfo** out-of-band (OOB) information is valid, NVGRE and VXLAN support is required. The NIC must perform USO offload on the NVGRE/VXLAN-encapsulated packet with the following condition:

- The NDIS_UDP_SEGMENTATION_OFFLOAD_NET_BUFFER_LIST_INFO.**Transmit.UdpHeaderOffset** member doesn't have the correct offset value and must not be used by the NIC or miniport driver.

To support NVGRE in USO, protocol and filter drivers must:

- Adjust the **InnerFrameOffset**, **TransportIpHeaderRelativeOffset**, and **TcpHeaderRelativeOffset** values in the NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO structure to account for the encapsulation header. The **TcpHeaderRelativeOffset** refers to the UDP header.

NICs and miniport drivers may use the **InnerFrameOffset**, **TransportIpHeaderRelativeOffset**, and **TcpHeaderRelativeOffset** values provided in the NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO structure. The NIC or miniport driver may perform any needed header checks on the tunnel (outer) IP header or subsequent headers to validate these offsets.

Miniport drivers must handle the case where NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**InnerFrameOffset** may be in a different scatter-gather list than the beginning of the packet. The protocol driver will guarantee that all the prepended encapsulation headers (ETH, IP, GRE/VXLAN) will be physically contiguous and will be in the first MDL of the packet.

Protocol and filter drivers don't ensure that the total UDP payload length is an exact multiple of the reduced **MSS** value when **UdpSegmentation.SubMssFinalSegmentSupported** is set in the NDIS_OFFLOAD capabilities. For this reason, miniport drivers and NICs with **SubMssFinalSegmentSupported** must update the tunnel (outer) IP header. NICs must generate as many full-sized segments as possible based on the reduced **MSS** value in the NDIS_UDP_SEGMENTATION_OFFLOAD_NET_BUFFER_LIST_INFO.**Transmit** OOB information. Only one sub-**MSS** segment may be generated per LSOv2 send.

Miniport drivers must:

- Compute the checksum for the tunnel (outer) IP header.
- Increment the IP identification (IP ID) value of the tunnel (outer) IP header for every packet. The first packet must use the IP ID in the original tunnel (outer) IP header.
- Increment the IP ID of the transport (inner) IP header for every packet. The first packet must use the IP ID in the original transport (inner) IP header.
- Compute the checksum for the UDP header and the transport (inner) IP header.
- Ensure that the complete headers, including the encapsulation tunnel (outer) headers are added to every generated packet.

Related articles

UDP Segmentation Offload (USO)

Supporting NVGRE in Checksum Offload

Article • 12/15/2021

NDIS 6.30 (Windows Server 2012) introduces Network Virtualization using Generic Routing Encapsulation (NVGRE). NDIS miniport, protocol, and filter drivers and NICs that offload checksum tasks should do so in a way that supports NVGRE.

Note This page assumes that you are familiar with the information in Offloading Checksum Tasks.

If NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**IsEncapsulatedPacket** is **TRUE** and the **TcpIpChecksumNetBufferListInfo** out-of-band (OOB) information is valid, this indicates that NVGRE support is required and the NIC must compute the checksum for the tunnel (outer) IP header, the transport (inner) IP header, and the TCP or UDP header.

The **IsIPv4** and **IsIPv6** flags in the NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO structure indicate the IP header version of the tunnel (outer) IP header. The NIC must parse the transport (inner) IP header to determine that header's IP version. Because mixed-mode packets are allowed (see NDIS_ENCAPSULATED_PACKET_TASK_OFFLOAD), the NIC must not assume that the inner and outer IP headers will have the same IP header version.

NICs and miniport drivers may use the **InnerFrameOffset**, **TransportIpHeaderRelativeOffset**, and **TcpHeaderRelativeOffset** values provided in the NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO structure. The NIC or miniport driver may perform any needed header checks on the tunnel (outer) IP header or subsequent headers to validate these offsets.

Note that when NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**IsEncapsulatedPacket** is TRUE, the existing header offset fields, NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO.**LsoV2Transmit.TcpHeaderOffset** and NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO.**Transmit.TcpHeaderOffset**, will not have correct values and must not be used by the NIC or driver.

Miniport drivers must handle the case where NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**InnerFrameOffset** may be in a different scatter-gather list than the beginning of the packet. The protocol driver will guarantee that all the prepended encapsulation headers (ETH, IP, GRE) will be physically contiguous and will be in the first MDL of the packet.

Checksum Validation

Checksum validation for NVGRE is largely the same as it would be otherwise.

If a miniport receives an OID_TCP_OFFLOAD_PARAMETERS OID request and succeeds it for **NDIS_ENCAPSULATION_TYPE_GRE_MAC** (see **NDIS_OFFLOAD_PARAMETERS**), the NIC must perform checksum validation on the tunnel (outer) IP header, transport (inner) IP header, and TCP or UDP header.

For encapsulated packets that have an IPv4 tunnel (outer) header and an IPv4 transport (inner) header, a miniport driver should set the **IpChecksumSucceeded** flag in the **NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO** structure only if both IP header checksum validations succeeded. For encapsulated packets that have both a tunnel (outer) IPv4 header and a transport (inner) IPv4 header, the miniport driver should set the **IpChecksumFailed** flag if either of the IP header checksum validations failed.

Supporting NVGRE in RSS and VMQ Receive Task Offloads

Article • 07/07/2022

NDIS 6.30 (Windows Server 2012) introduces Network Virtualization using Generic Routing Encapsulation (NVGRE). NDIS miniport drivers and NICs that perform Receive Side Scaling (RSS) and Virtual Machine Queue (VMQ) receive task offloads should do so in a way that supports NVGRE.

Note This page assumes that you are familiar with the information in Offloading the Segmentation of Large TCP Packets.

If the miniport driver supports RSS and VMQ for encapsulated packets, it must advertise those capabilities in the **RssSupported** and **VmqSupported** members of the **NDIS_ENCAPSULATED_PACKET_TASK_OFFLOAD** structure. If the miniport advertised these capabilities, received an OID_TCP_OFFLOAD_PARAMETERS OID request, and succeeded the OID, the NIC must perform RSS and VMQ on the advertised encapsulated packet types.

For supported encapsulated packets that it is able to parse, the NIC must perform RSS on the TCP or UDP header in the payload of the transport (inner) IP header and VMQ on the inner MAC header.

For performing RSS and VMQ, the NIC must get to the transport (inner) IP header of the encapsulated packet as described in Locating the Transport Header for Encapsulated Packets in the Receive Path and check the protocol number. If the NIC receives a packet that uses a protocol that the NIC can parse, the NIC should:

- Perform RSS by doing a 4-tuple hash on the transport (inner) IP header and the TCP or UDP header.
  - For encapsulated packets whose protocol the miniport cannot parse, the NIC should perform a 2-tuple hash on the source and destination address fields in the tunnel (outer) IP header.
  - For encapsulated packets that do not contain a TCP or UDP header immediately following the transport (inner) IP header, the NIC should perform a 2-tuple hash on the source and destination address fields in the tunnel (outer) IP header.
- Perform VMQ by using the Ethernet header in the encapsulated packet. For encapsulated packets that do not contain an Ethernet header (within the encapsulated packet), VMQ should be performed using the outermost Ethernet header.

Locating the Transport Header for Encapsulated Packets in the Receive Path

Article • 12/15/2021

On receiving a packet, a NIC that supports Network Virtualization using Generic Routing Encapsulation (NVGRE) must first determine whether the packet is encapsulated and, if so, the type of encapsulation.

Note In the send path, a packet is encapsulated if NDIS_TCP_SEND_OFFLOADS_SUPPLEMENTAL_NET_BUFFER_LIST_INFO.**IsEncapsulatedPacket** is **TRUE**.

In the receive path, the NIC must determine whether the packet is encapsulated by checking the protocol number in the **Protocol** field of the IPv4 tunnel (outer) header or the **NextHeader** field of the IPv6 tunnel (outer) header. The list of assigned protocol numbers can be found at https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml .

Once a packet is determined to be an encapsulated packet, the NIC must determine the offset to the transport (inner) IP header by parsing the encapsulated packet's protocol.

For NDIS 6.30 (Windows Server 2012) and later, only GRE IP encapsulation is supported. So the NIC should be able to parse the following, depending on the advertised capabilities:

- GRE (RFC 2784: Generic Routing Encapsulation (GRE)) headers
- RFC 2890: Key and Sequence Number Extensions to GRE
- IPv4 (RFC 791: Internet Protocol) headers
- IPv6 (RFC 2460: Internet Protocol, Version 6 (IPv6)) headers

If the NIC finds an unknown or unsupported encapsulation protocol, it must pass the packet unchanged to the host stack.

Thus, on the receive path, the miniport must parse the transport (inner) IP header to determine the IP version as well as to get to the TCP or UDP header. This is a new requirement for NDIS 6.30 (Windows Server 2012) and later.

Determining the NVGRE Task Offload Capabilities of a Network Adapter

Article • 12/15/2021

A miniport driver that supports Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload reports this capability by means of the **NDIS_OFFLOAD** structure that its *MiniportInitializeEx* function passes to **NdisMSetMiniportAttributes**.

Reporting NVGRE Task Offload Capability

In the **NDIS_OFFLOAD** structure, the **Header** member must be set as follows:

- The **Revision** member must be set to **NDIS_OFFLOAD_REVISION_3**.
- The **Size** member must be set to **NDIS_SIZEOF_NDIS_OFFLOAD_REVISION_3**.

To report its support for NVGRE task offload, a miniport driver sets the following members in the **NDIS_ENCAPSULATED_PACKET_TASK_OFFLOAD** structure, which is stored in the **EncapsulatedPacketTaskOffloadGre** member of the **NDIS_OFFLOAD** structure that the miniport driver's *MiniportInitializeEx* function passes to **NdisMSetMiniportAttributes**:

- Set the **MaxHeaderSizeSupported** member to the maximum header size from the beginning of the packet to the beginning of the inner TCP or UDP payload (the last byte of TCP or UDP inner header) that the NIC must support for all of these task offloads. The protocol driver is expected to not offload processing of a packet whose combined encapsulation headers exceed this size.

  Note 256 bytes is a good default value that should cover all possible cases.

- Set the other members to indicate which types of task offload the miniport driver supports for encapsulated packets. For a list of the flags that can be set for these members, see the Remarks section of **NDIS_ENCAPSULATED_PACKET_TASK_OFFLOAD**.

Querying NVGRE Task Offload Capability

To determine whether a miniport driver supports NVGRE task offload, protocol and filter drivers can issue the OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES OID request, which returns the **NDIS_OFFLOAD** structure.

Note To determine whether the miniport driver's NVGRE capability is currently enabled, use the OID_TCP_OFFLOAD_CURRENT_CONFIG OID request as described in Querying and Changing NVGRE Task Offload State.

Note To enable or disable the miniport driver's NVGRE capability, use the OID_TCP_OFFLOAD_PARAMETERS OID request as described in Querying and Changing NVGRE Task Offload State.

Querying and Changing NVGRE Task Offload State

Article • 12/15/2021

This section describes how to query or change the current Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload state of an NVGRE-capable miniport driver. NVGRE task offload can be enabled by default, but it must not be operationally active by default. A NIC should not begin performing task offloads on encapsulated packets until this feature is enabled explicitly by an NDIS protocol or filter driver.

Querying NVGRE Task Offload State

To query a miniport driver's current NVGRE task offload state, an NDIS protocol or filter driver uses the OID_TCP_OFFLOAD_CURRENT_CONFIG OID request. This will return an **NDIS_OFFLOAD** structure whose **EncapsulatedPacketTaskOffloadGre** member is an **NDIS_ENCAPSULATED_PACKET_TASK_OFFLOAD** structure that contains **NDIS_OFFLOAD_SUPPORTED** if those offloads are currently enabled for GRE-encapsulated packets and **NDIS_OFFLOAD_NOT_SUPPORTED** otherwise. NDIS handles this OID and does not pass it down to the miniport.

Note To determine whether a miniport driver supports NVGRE task offload, use the OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES OID request as described in Determining the NVGRE Task Offload Capabilities of a Network Adapter.

Changing NVGRE Task Offload State

An NDIS protocol or filter driver can enable or disable NVGRE task offload by issuing the OID_TCP_OFFLOAD_PARAMETERS OID request. This OID uses an **NDIS_OFFLOAD_PARAMETERS** structure. In this structure, the **EncapsulatedPacketTaskOffload** member can have the following values:

Term	Description
NDIS_OFFLOAD_SET_NO_CHANGE	The NVGRE task offload state is unchanged.
NDIS_OFFLOAD_SET_ON	Specify this flag to enable NVGRE task offload.
NDIS_OFFLOAD_SET_OFF	Specify this flag to disable NVGRE task offload.

After the miniport driver processes the OID_TCP_OFFLOAD_PARAMETERS OID request, it must issue an **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication with the updated offload state.

When a miniport driver receives a OID_TCP_OFFLOAD_PARAMETERS OID request in which the **NDIS_OFFLOAD_SET_OFF** flag is specified, the driver should indicate any existing encapsulated packets that are partially processed for task offloads up the stack before completing the OID request.

Base task offloads for normal packets are enabled by existing OIDs such as OID_OFFLOAD_ENCAPSULATION and OID_RECEIVE_FILTER_ALLOCATE_QUEUE. The **EncapsulatedPacketTaskOffload** member setting supplements these OIDs and instructs the NIC to also do these offloads for encapsulated packets.

Standardized INF Keywords for NVGRE Task Offload

Article • 12/15/2021

The ***EncapsulatedPacketTaskOffload** standardized enumeration keyword is defined to enable or disable support for Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload in miniport adapters.

The following table describes the possible INF entries for this keyword.

SubkeyName	ParamDesc	Value	EnumDesc
***EncapsulatedPacketTaskOffload**	Encapsulated Task Offload	0	Disabled
		1 (Default)	Enabled

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

Overview of Receive Segment Coalescing

Article • 01/12/2023

When receiving data, the miniport driver, NDIS, and TCP/IP must all look at each PDU's header information separately. When large amounts of data are being received, this creates a large amount of overhead. Receive segment coalescing (RSC) reduces this overhead by coalescing a sequence of received segments and passing them to the host TCP/IP stack in one operation, so that NDIS and TCP/IP need only look at one header for the entire sequence.

RSC is intended to support coalescing in a way that:

- Doesn't interfere with the normal operation of TCP's congestion and flow control mechanisms.
- Coalesces packets without discarding information that is used by the TCP stack.

RSC-capable miniport drivers for network cards must:

- Follow a standard set of rules when coalescing segments.
- Provide certain out-of-band information to the host TCP/IP stack.

The following sections provide an overview of RSC.

- Rules for Coalescing TCP/IP Segments
- Updating the IP Headers for Coalesced Segments
- Examples of Receive Segment Coalescing
- Indicating Coalesced Segments
- Exception Conditions that Terminate Coalescing

Rules for Coalescing TCP/IP Segments

Article • 12/15/2021

This section defines the rules that specify when a receive segment coalescing (RSC)-capable miniport driver must coalesce a segment for a given TCP connection. If any of the rules are violated, an exception is generated, and the miniport driver must abort the coalescing of the segment.

The miniport driver must update the IP and TCP headers for the single coalesced unit (SCU). The miniport driver must recompute the TCP and IPv4 checksums over the SCU and chain the TCP payload.

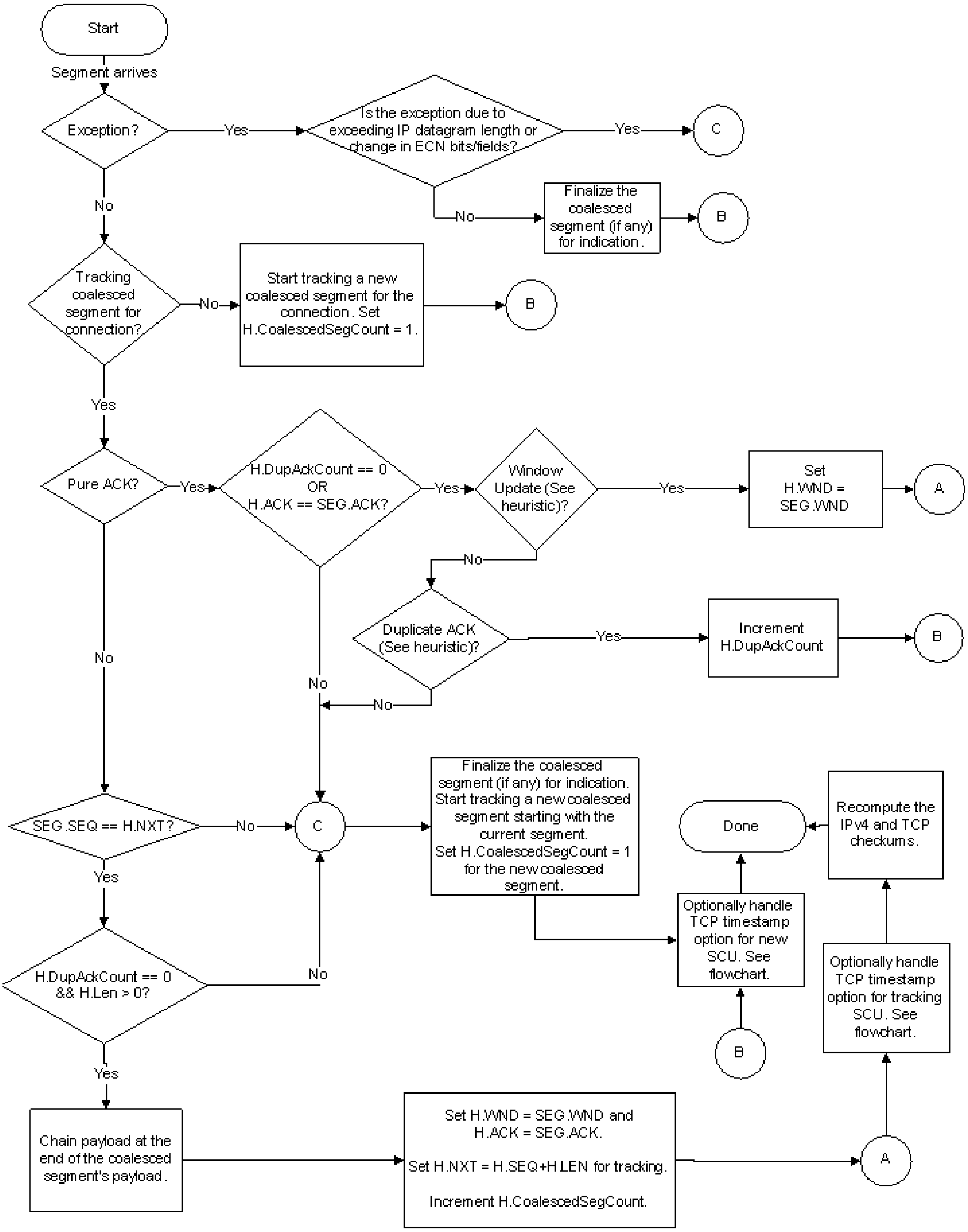
The first of the following two flowcharts describes the rules for coalescing segments and updating the TCP headers. This flowchart refers to mechanisms for distinguishing valid duplicate ACKs and window updates. The second flowchart describes these mechanisms.

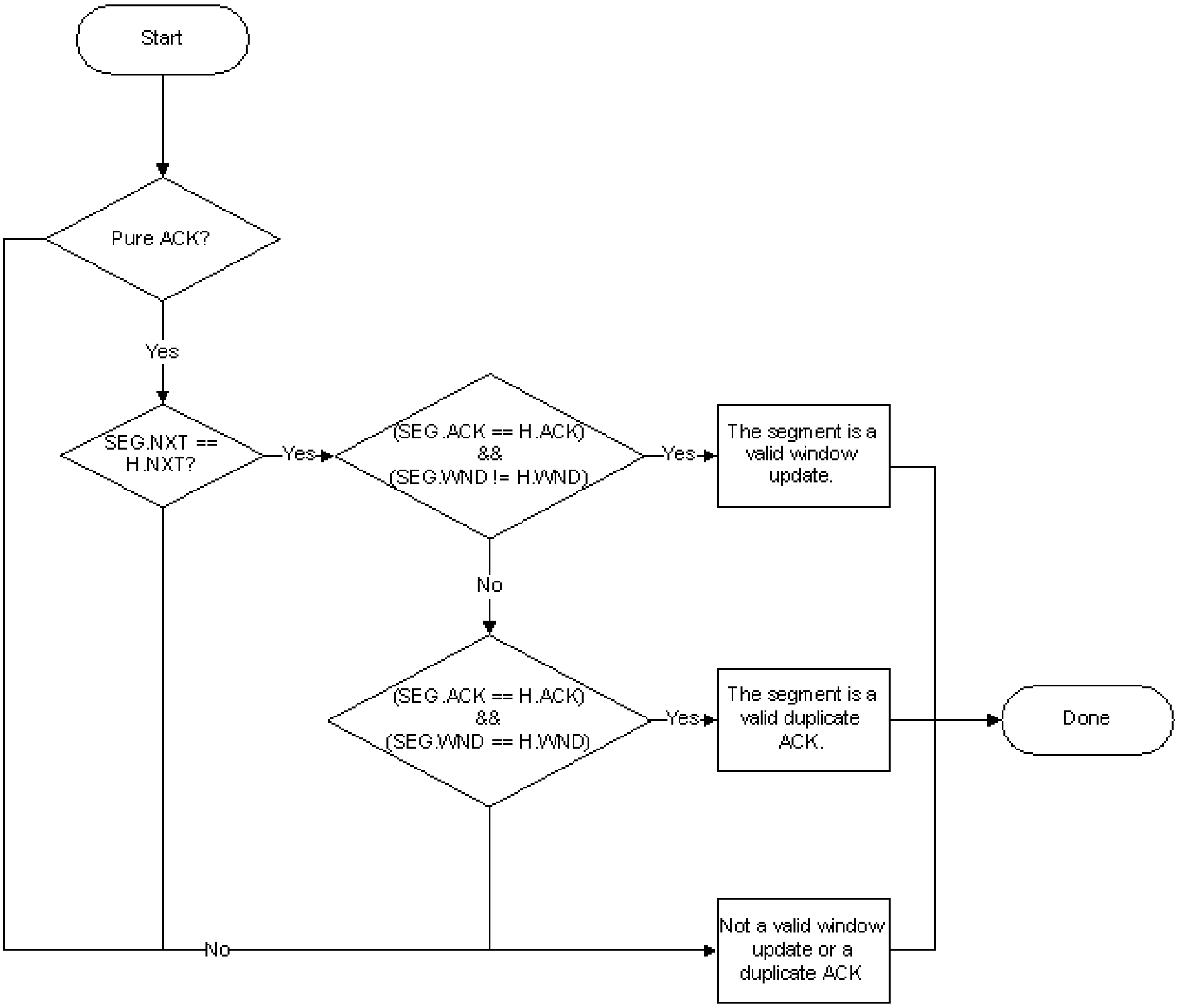
These flowcharts are provided as a reference for understanding the RSC rules. A hardware implementation can optimize the flowchart, as long as correctness is maintained.

The following terms are used in the flowcharts:

Term	Description
SEG.SEQ	Sequence number of the incoming segment.
H.SEQ	Sequence number of the currently tracked SCU.
SEG.ACK	Acknowledgment number of the incoming segment.
H.ACK	Acknowledgment number of the currently tracked SCU.
SEG.WND	The window that is advertised by the incoming segment.
H.WND	The window that is advertised by the currently tracked SCU.
SEG.LEN	TCP payload length of the incoming segment.
H.LEN	TCP payload length of the currently tracked SCU.
SEG.NXT	The sum of **SEG.SEQ** and **SEG.LEN**.
H.NXT	The sum of **H.SEQ** and **H.LEN**.
H.DupAckCount	The number of duplicate ACKs that have been coalesced into the SCU. This number should be zero.

Term	Description
SEG.Tsval	The **Timestamp** value in the currently received segment. The format for this value is defined in RFC 1323 .
H.Tsval	The **Timestamp** value in the currently tracked SCU.
SEG.TSecr	The **Timestamp Echo Reply** in the currently received segment.
H.TSecr	The **Timestamp Echo Reply** in the currently tracked SCU.


Start
Segment arrives
Exception?
Yes
Is the exception due to exceeding IP datagram length or change in ECN bits/fields?
Yes
C
No
Finalize the coalesced segment (if any) for indication.
B
No
Tracking coalesced segment for connection?
No
Start tracking a new coalesced segment for the connection. Set H.CoalescedSegCount = 1.
B
Yes
Pure ACK?
Yes
H.DupAckCount == 0 OR H.ACK == SEG.ACK?
Yes
Window Update (See heuristic)?
Yes
Set H.WND = SEG.WND
A
No
Duplicate ACK (See heuristic)?
Yes
Increment H.DupAckCount
B
No
No
No
SEG.SEQ == H.NXT?
No
C
Finalize the coalesced segment (if any) for indication. Start tracking a new coalesced segment starting with the current segment. Set H.CoalescedSegCount = 1 for the new coalesced segment.
Done
Recompute the IPv4 and TCP checksums.
Yes
Optionally handle TCP timestamp option for new SCU. See flowchart.
H.DupAckCount == 0 && H.Len > 0?
No
B
Optionally handle TCP timestamp option for tracking SCU. See flowchart.
Yes
Chain payload at the end of the coalesced segment's payload.
Set H.WND = SEG.WND and H.ACK = SEG.ACK.
Set H.NXT = H.SEQ+H.LEN for tracking.
Increment H.CoalescedSegCount.
A


Start
Pure ACK?
Yes
SEG.NXT == H.NXT?
Yes
(SEG.ACK == H.ACK) && (SEG.WND != H.WND)
Yes
The segment is a valid window update.
No
(SEG.ACK == H.ACK) && (SEG.WND == H.WND)
Yes
The segment is a valid duplicate ACK.
Done
No
Not a valid window update or a duplicate ACK

The flowcharts show that the miniport driver may coalesce segments with different ACK numbers. However, the miniport driver must obey the following rules regarding ACK numbers, as shown in the first flowchart above:

- After performing the sequence number check, an incoming pure ACK may be coalesced into the currently tracked SCU if it meets one or both of the following conditions:

    - **H.ACK** == **SEG.ACK**.

    - The duplicate-ACK count in the coalesced segment that is being tracked is zero. In other words, **H.DupAckCount** == 0.

      In other words, any pure ACK that is not a duplicate ACK or a window update triggers an exception and must not be coalesced. All such pure ACKs must be indicated as individual segments. This rule ensures that RSC does not affect the behavior or performance of the Windows TCP congestion control algorithms.

- An incoming data segment (**SEG.ACK** == **H.ACK**) or an incoming piggy-backed ACK (**SEG.ACK** > **H.ACK**) may be coalesced into the currently tracked SCU if both of the following conditions are met:

- The segment is contiguous to the SCU in the sequence space. In other words, **SEG.SEQ** == **H.NXT**.
- The duplicate-ACK count in the coalesced segment that is being tracked is zero. In other words, **H.DupAckCount** == 0.

Additional notes on Duplicate ACK coalescing

Duplicate ACK Behavior

The miniport driver should treat a duplicate ACK segment equivalent to a pure ACK and not coalesce it. In this case, it must finalize the current SCU (if any) for indication and indicate the duplicate ACK segment as an individual segment. Because Windows clients use selective acknowledgments (SACK) by default, a duplicate ACK segment will likely generate an exception. See Examples of Receive Segment Coalescing for an example. If a segment with **DupAckCount** > 0 is indicated, NDIS will disable RSC on the interface.

Handling Duplicate ACK when tracking a SCU consisting of data segments

When tracking an SCU with **H.LEN** > 0 (in other words, a coalesced segment that contains data), if a duplicate ACK arrives next, then the tracking SCU should be finalized as follows:

1. A new SCU should be tracked, starting with the duplicate ACK.

2. The **DupAckCount** for the new SCU should be set to zero.

3. The **DupAckCount** should be incremented if additional duplicate ACKs are received.

In this case, **DupAckCount** will be 1 less than the number of duplicate ACKs. The host stack will handle the counting correctly.

Handling Duplicate ACK when tracking a SCU consisting of a pure cumulative ACK

When tracking an SCU that consists of a single pure cumulative ACK (rules forbid coalescing multiple pure ACKs), if a duplicate ACK arrives next, then the **DupAckCount** for the tracking SCU should be incremented. It should also be incremented if additional duplicate ACKs are received. In this case, **DupAckCount** will be equal to the number of duplicate ACKs that are coalesced.

When the first segment that is received in a DPC is a duplicate ACK

In this case, the NIC cannot determine whether the received segment is a duplicate ACK, because it does not maintain any state. So the segment should be treated as a pure ACK instead as follows:

1. A new SCU should be tracked, starting with this segment.
2. The **DupAckCount** for the new SCU should be set to zero.
3. The **DupAckCount** should be incremented by 1 for each additional duplicate ACK that is received.

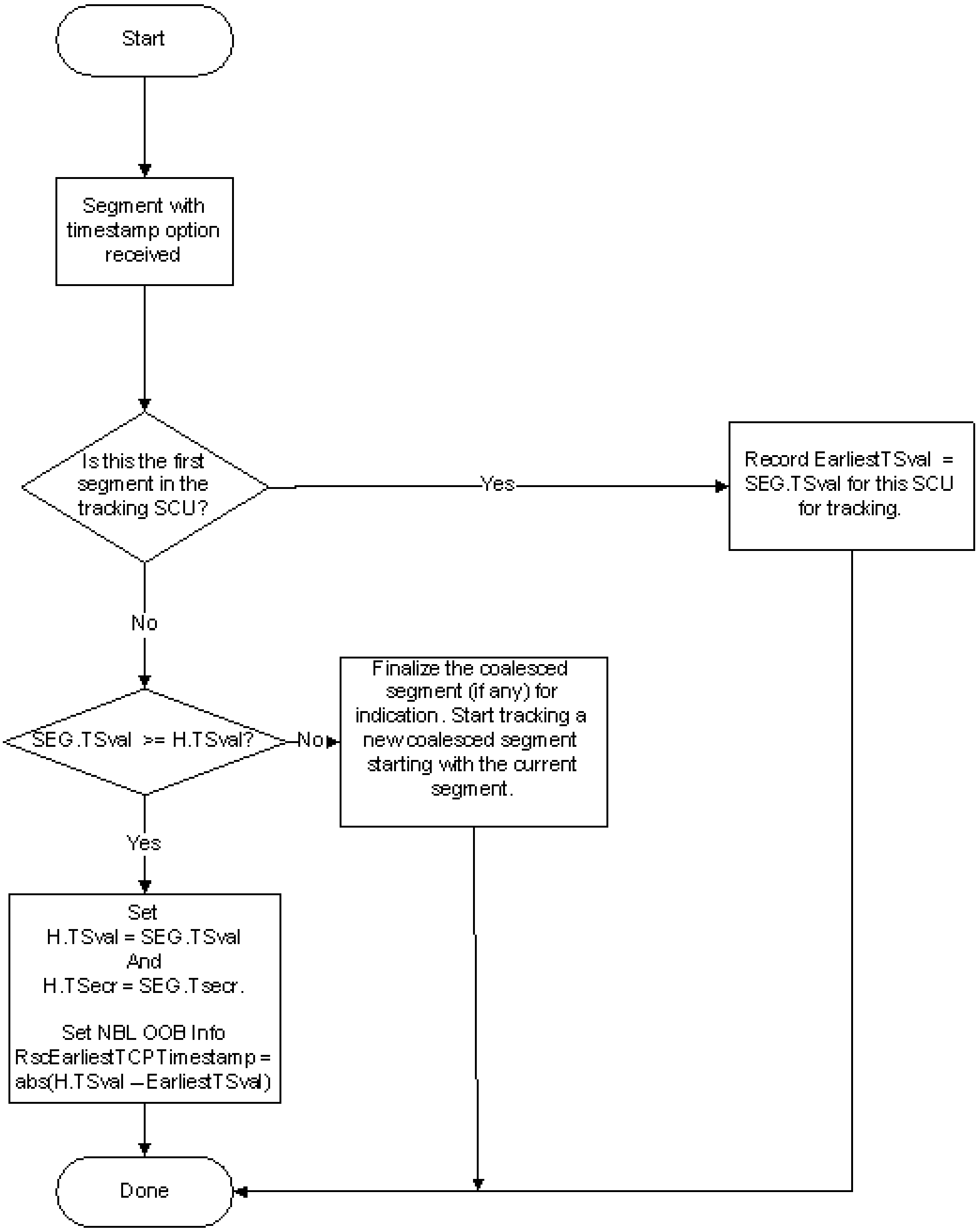
In this case, **DupAckCount** will be equal to 1 less than the actual number of duplicate ACKs. The host stack will handle the counting correctly.

Duplicate ACK Exemption

The miniport driver may treat a duplicate ACK segment equivalent to a pure ACK and not coalesce it. In this case, it must finalize the current SCU (if any) for indication and indicate the duplicate ACK segment as an individual segment. Because Windows clients use SACK by default, a duplicate ACK segment will likely generate an exception. For an example, see Examples of Receive Segment Coalescing. This exemption does not apply to window update segments.

Coalescing Segments with the Timestamp Option

The TCP timestamp option is the only option that may be legally coalesced. Coalescing segments with this option is left as an implementation-specific decision. If the miniport driver coalesces segments with the timestamp option, then it must follow the rules outlined in the following flowchart:


Start
Segment with timestamp option received
Is this the first segment in the tracking SCU?
Yes
Record EarliestTSval = SEG.TSval for this SCU for tracking.
No
SEG.TSval >= H.TSval?
No
Finalize the coalesced segment (if any) for indication. Start tracking a new coalesced segment starting with the current segment.
Yes
Set H.TSval = SEG.TSval And H.TSecr = SEG.Tsecr.
Set NBL OOB Info RscEarliestTCPTimestamp = abs(H.TSval – EarliestTSval)
Done

ⓘ **Note**

The check **SEG.TSval** >= **H.TSval** must be performed using modulo-232 arithmetic similar to that used for TCP sequence numbers. See RFC 793, section 3.3.

When indicating a coalesced segment, the following out-of-band information must be indicated as follows by setting the **NetBufferListInfo** member of the NET_BUFFER_LIST structure that describes the coalesced segment:

- The number of segments that were coalesced must be stored into the **NetBufferListInfo**[**TcpRecvSegCoalesceInfo**].**CoalescedSegCount** member. This number only represents data segments that were coalesced. Pure ACK coalescing is forbidden and window update segments must not be counted as part of this field.

- The duplicate ACK count must be stored into the **NetBufferListInfo**[**TcpRecvSegCoalesceInfo**].**DupAckCount** member. The first flowchart above explains how this value is calculated.

- When segments with the TCP timestamp option are coalesced, **NetBufferListInfo**[**RscTcpTimestampDelta**] must be filled with the absolute delta between the earliest and the latest TCP timestamp value seen in the sequence of coalesced segments comprising the SCU. The SCU itself should contain the latest TCP timestamp value seen in the sequence of coalesced segments.

The **DupAckCount** and **RscTcpTimestampDelta** members are interpreted if and only if the **CoalescedSegCount** member is greater than zero. If the **CoalescedSegCount** is zero, the segment is treated as a non-coalesced non-RSC segment.

For information about the contents of the **NetBufferListInfo** member, see **NDIS_NET_BUFFER_LIST_INFO** and **NDIS_RSC_NBL_INFO**.

The PSH bit should be ORed for all coalesced segments. In other words, if the PSH bit was set in any of the individual segments, the miniport driver should set the PSH bit in the SCU.

Finalizing an SCU involves:

- Recomputing the TCP and, if applicable, the IPv4 checksum.

- Updating the IP headers as described in Updating the IP Headers for Coalesced Segments.

- Setting the ECN bits and ECN fields in the TCP and IP headers to the same values that were set in the individual segments.

Handling TCP/IP IPsec segments

A network card may report both RSC and IPsec task offload capabilities. (See Determining the RSC Capabilities of a Network Adapter.) However, if it supports IPsec task offload, it must not attempt to coalesce segments that are protected by IPsec.

Updating the IP Headers for Coalesced Segments

Article • 12/15/2021

When finalizing a single coalescing unit (SCU), a receive segment coalescing (RSC)-capable miniport driver updates the fields in the IP headers as described in the following tables.

- Updating IPv4 header fields for coalesced segments
- Updating IPv6 header fields for coalesced segments

Updating IPv4 header fields for coalesced segments

Field	Description
Version	The value of this field must be the same for all coalesced segments.
Header Length	The length of a basic IPv4 header without any IP options.
Differentiated Services	The value of this field must be the same for all coalesced segments.
ECN bits	See Exception 8 in Exception Conditions that Terminate Coalescing. Datagrams should be coalesced if they all have the same values for the ECN bits.
Total Length	The value of this field must be recomputed every time a new segment with non-zero TCP payload length is coalesced into an existing SCU. See Exception Conditions that Terminate Coalescing for special cases that arise from the value in this field.
Identification	Must be set to the IP ID of the first coalesced segment.

Field	Description
Flags	- Datagrams may be coalesced as long as they have the same value for the DF (Don’t Fragment) bit: either all set or all clear. - Segments with the MF (More Fragments) bit set must not be coalesced.
Fragment Offset	Not applicable. Fragmented IP datagrams are not coalesced.
Time To Live	Must be set to the minimum time to live (TTL) value of the coalesced segments.
Protocol	Always set to 6, for TCP.
Header Checksum	The value of this field must be recomputed by the miniport driver.
Source Address	The value of this field must be the same for all coalesced segments.
Destination Address	The value of this field must be the same for all coalesced segments.

Updating IPv6 header fields for coalesced segments

Field	Description
Version	The value of this field must be the same for all coalesced segments.
Traffic Class	The value of this field must be the same for all coalesced segments.
Flow Label	The value of this field must be the same for all coalesced segments.
Payload Length	The value of this field must be recomputed whenever a new segment with nonzero TCP payload length is coalesced into an existing segment.
Next Header	Always set to 6, for TCP.

Field	Description
Hop Limit	Must be set to the minimum **Hop Limit** value of the coalesced segments.
Source Address	The value of this field must be the same for all coalesced segments.
Destination Address	The value of this field must be the same for all coalesced segments.

Examples of Receive Segment Coalescing

Article • 12/15/2021

This section illustrates the coalescing algorithm by using examples of segments that are received in order and processed in a single deferred procedure call (DPC).

This page uses X and X′ for labeling successive segments. All other segment and single coalesced unit (SCU) fields are as described in Rules for Coalescing TCP/IP Segments.

Example 1: Data segments

Segment Description

10 successive segments belonging to the same TCP connection are processed. All of the following conditions are true for each:

- X′.SEQ == X.NXT
- X′SEQ > X.SEQ
- X′.ACK == X.ACK

None of these segments generates an exception.

Result

A single SCU is formed out of the 10 segments. This is indicated as a single **NET_BUFFER** in a single **NET_BUFFER_LIST**.

Example 2: Data segments, followed by an exception, followed by data segments

Segment Description

5 successive segments belonging to the same TCP connection are processed. All of the following conditions are true for each:

- X′.SEQ == X.NXT

- X'SEQ > X.SEQ

- X'.ACK == X.ACK

None of these segments generates an exception. The 6th segment is a duplicate ACK segment with a TCP SACK option and generates an exception based on rule number 3 in Rules for Coalescing TCP/IP Segments.

Note In this case, the exception rule for handling a TCP option takes precedence and thus overrides the coalescing rule.

2 successive segments belonging to the same TCP connection are processed. All of the following conditions are true for each:

- X'.SEQ == X.NXT

- X'SEQ > X.SEQ

- X'.ACK == X.ACK

None of these segments generates an exception.

Result

A single SCU is formed out of the first 5 segments. The 6th segment does not form an SCU.

The 7th and 8th segments form an SCU together.

A **NET_BUFFER_LIST** chain is indicated with three **NET_BUFFER_LIST** structures each having a single **NET_BUFFER**. The ordering of received segments is maintained.

Example 3: Data segments, followed by multiple window updates

Segment Description

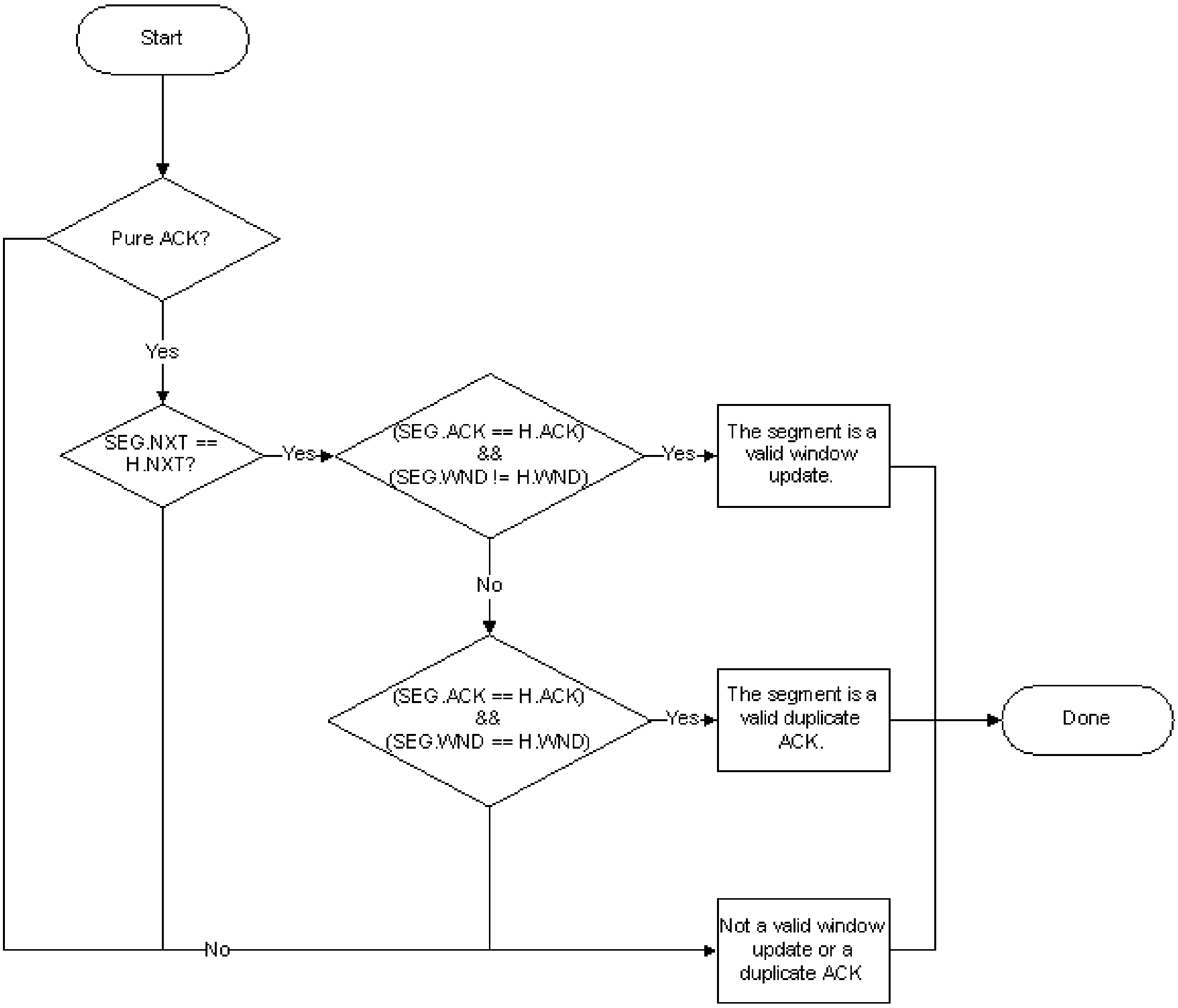
5 successive segments belonging to the same TCP connection are processed. All of the following conditions are true for each:

- X'.SEQ == X.NXT

- X'SEQ > X.SEQ

- X'.ACK == X.ACK

None of these segments generates an exception. The 6th segment is a pure ACK that is a window update with SEG.WND = 65535 as shown in the following flowchart.


Start
Pure ACK?
Yes
SEG.NXT == H.NXT?
Yes
(SEG.ACK == H.ACK) && (SEG.WND != H.WND)
Yes
The segment is a valid window update.
No
(SEG.ACK == H.ACK) && (SEG.WND == H.WND)
Yes
The segment is a valid duplicate ACK.
Done
No
Not a valid window update or a duplicate ACK

The 7th segment is a pure ACK that is a window update with SEG.WND = 131070 as shown in the same flowchart.

Result

A single SCU is formed out of the 7 segments. This is indicated as a single NET_BUFFER in a single NET_BUFFER_LIST.

The SCU.WND = 131070, and the checksum is updated based on this value.

Example 4: Piggybacked ACKs mixed with data segments

Segment Description

3 successive segments belonging to the same TCP connection are processed. All of the following conditions are true for each:

- X'.SEQ == X.NXT
- X'SEQ > X.SEQ
- X'.ACK == X.ACK

None of these segments generates an exception. 2 successive segments belonging to the same TCP connection are processed. All of the following conditions are true for each:

- X'.SEQ == X.NXT
- X'SEQ > X.SEQ
- X'.ACK == X.ACK

None of these segments generates an exception.

Result

A single SCU is formed out of the 5 segments. This is indicated as a single **NET_BUFFER** in a single **NET_BUFFER_LIST**. The SCU.ACK is set to the last SEG.ACK.

Indicating Coalesced Segments

Article • 12/15/2021

A single coalesced unit (SCU) is a sequence of TCP segments that are coalesced into a single TCP segment according to the rules defined in Rules for Coalescing TCP/IP Segments. This section describes how to indicate the resulting coalesced segments.

An SCU must:

- Be indicated by calling **NdisMIndicateReceiveNetBufferLists**.

- Look like a normal TCP segment that is received over the wire.

- Be no larger than the maximum legal IP datagram length, as defined in section 3.1 of RFC 791.

  Note Because segments with IPv6 extension headers cannot be coalesced (see Exception Conditions that Terminate Coalescing), the size of the SCU for IPv6 datagrams is also limited by the maximum legal datagram length.

The NIC or miniport driver should recompute the TCP and IPv4 checksums, if applicable, before indicating the coalesced segment. If the NIC or miniport driver validates the TCP and IPv4 checksums but does not recompute them for the coalesced segment, it must set the **TcpChecksumValueInvalid** and **IpChecksumValueInvalid** flags in the **NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO** structure. Additionally, in this case the NIC or miniport driver may optionally zero out the TCP and IPv4 header checksum values in the segment.

The NIC and miniport driver must always set the **IpChecksumSucceeded** and **TcpChecksumSucceeded** flags in the **NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO** structure before indicating the coalesced segment.

For more information about coalescing rules, see Rules for Coalescing TCP/IP Segments.

For more information about exceptions, see Exception Conditions that Terminate Coalescing.

Coalescing is expected to be performed on a best-effort basis. The hardware might not be able to coalesce in some cases, for example due to lack of resources. The requirements stated here are primarily to specify when not to coalesce and how to coalesce.

At a high level, the NIC and miniport driver must handle the receipt of a TCP segment over the wire as follows:

- Check the incoming segment for an exception as follows:

    1. If no exception was encountered, check whether the segment can be coalesced with the last segment that was received for the same TCP connection per the rules.

    2. If the segment triggered an exception, or if coalescing it with the previously received segment is not possible, then indicate the segment individually.

- The NIC and miniport driver must not indicate coalesced segments until the protocol driver enables RSC as described in Querying and Changing RSC State.

- For a given TCP connection, a data indication from the miniport adapter to the host TCP/IP stack may consist of one or more coalesced segments, separated by one or more individual segments that could not be coalesced.

- The NIC and miniport driver must not delay the indication of TCP segments, whether coalesced or not. Specifically, the NIC and miniport driver must not delay the indication of segments from one deferred procedure call (DPC) to the next in order to attempt to coalesce the segments.

- The NIC and miniport driver may use timers to determine the end of coalescing. However, the handling of latency sensitive workloads must be as effective as the DPC boundary requirement.

Exception Conditions that Terminate Coalescing

Article • 12/15/2021

This section defines the checks that a receive segment coalescing (RSC)-capable miniport driver must perform on a segment before it can be coalesced.

A segment must pass both of the following types of checks before it can be coalesced:

- Checks for presence of a certain condition in the segment. For example, the presence of a SYN flag in the TCP header would trigger an exception and the segment would not be coalesced. These types of checks are defined below.

- Checks that depend on inspecting and correlating information from previously coalesced segments and the currently examined segments. For example, checking if the received segment is a duplicate acknowledgment falls in this category of checks. These types of checks are defined in Rules for Coalescing TCP/IP Segments.

If a check fails, an exception is triggered, and the miniport driver must terminate coalescing for that TCP connection and treat segments as follows:

- TCP segments that were coalesced before the exception was detected should be indicated as a single unit.

- TCP segments that are coalesced after the exception is detected should be indicated as a separate unit.

Note For exceptions 7 and 8 below, the miniport driver should resume coalescing starting with the segment that triggered the exception.

Receiving a segment that meets any of the following criteria must trigger an exception:

1. The hardware resource constraints in the NIC prevent coalescing.

2. The segment has an invalid TCP or IP checksum.

3. The segment contains any of the SYN, URG, RST, FIN in its TCP header, as defined in section 3.1 of RFC 793 . More broadly, if the segment contains any flag other than PSH or ACK, it should trigger an exception. For ECN flags, see exception 8 below.

4. The segment contains one or more TCP options other than the TCP timestamp option. See RFC 1323 for a discussion of the TCP timestamp option.

5. The segment contains IPv4 options or IPv6 extension headers.

6. The segment is an IPv4 fragment.

7. Coalescing the currently received segment will cause the single coalesced unit to exceed the maximum legal IP Datagram length. This exception requires special handling. For more information, see:

   - The first flowchart in Rules for Coalescing TCP/IP Packets

   - "Responding to Queries for RSC Statistics" in Programming Considerations for RSC Drivers.

8. The segment contains ECN flags, as defined in RFC 3168, that meet one or both of the following criteria:

   a. The segment contains a different value for the ECN field (ECT, CE) in the IP header than the previous segment.

   b. The segment has a different value for the ECN flags (ECE and CWR) in the TCP header than the previous segment.

Determining the RSC Capabilities of a Network Adapter

Article • 12/15/2021

A receive segment coalescing (RSC)-capable miniport driver reports its RSC capability by means of the NDIS_OFFLOAD structure that it passes to NdisMSetMiniportAttributes.

Reporting RSC Capability

In the NDIS_OFFLOAD structure, the **Header** member must be set as follows:

- The **Revision** member must be set to **NDIS_OFFLOAD_REVISION_3**.
- The **Size** member must be set to **NDIS_SIZEOF_NDIS_OFFLOAD_REVISION_3**.

To report its support for RSC, a miniport driver can set the following members in the NDIS_TCP_RECV_SEG_COALESCE_OFFLOAD structure, which is stored in the **Rsc** member of the NDIS_OFFLOAD structure:

- Set the **IPv4.Enabled** member to **TRUE** to indicate support for RSC for IPv4.

- Set the **IPv6.Enabled** member to **TRUE** to indicate support for RSC for IPv6.

The miniport driver must support RSC for at least IEEE 802.3 encapsulation. In addition, it can support RSC for any other encapsulations. If it does not support RSC for some encapsulation, and it receives packets of that encapsulation, the driver must indicate the packets up the stack normally.

Querying RSC Capability

To determine whether a miniport driver supports RSC, protocol drivers and other drivers can issue the OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES OID request, which will return an NDIS_OFFLOAD structure.

Querying and Changing RSC State

Article • 12/15/2021

This section describes how to query or change the current receive segment coalescing (RSC) state of an RSC-capable miniport driver.

Querying RSC State

The current RSC state can be queried by issuing the OID_TCP_OFFLOAD_CURRENT_CONFIG OID request. NDIS handles this OID and does not pass it down to the miniport.

Changing RSC State

RSC can be enabled or disabled by issuing the OID_TCP_OFFLOAD_PARAMETERS OID request. This OID uses an **NDIS_OFFLOAD_PARAMETERS** structure. In this structure, the **RscIPv4** and **RscIPv6** members can have the following values:

Term	Description
NDIS_OFFLOAD_PARAMETERS_NO_CHANGE	The RSC state is unchanged.
NDIS_OFFLOAD_PARAMETERS_RSC_DISABLED	Specify this flag to disable RSC.
NDIS_OFFLOAD_PARAMETERS_RSC_ENABLED	Specify this flag to enable RSC.

After the miniport driver processes the OID_TCP_OFFLOAD_PARAMETERS OID request, it must give an **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication with the updated offload state.

When a miniport driver receives a OID_TCP_OFFLOAD_CURRENT_CONFIG OID request in which the **NDIS_OFFLOAD_PARAMETERS_RSC_DISABLED** flag is specified, the driver must indicate any existing coalesced segments up the stack before completing the OID request.

Standardized INF Keywords for RSC

Article • 12/15/2021

In Windows 8, Windows Server 2012, and later, the receive segment coalescing (RSC) interface supports standardized INF keywords that appear in the registry and are specified in INF files.

The following list shows the enumeration standardized INF keywords for RSC:

***RscIPv4**
Enable or disable support for RSC for the IPv4 datagram version.

***RscIPv6**
Enable or disable support for RSC for the IPv6 datagram version.

Enumeration standardized INF keywords have the following attributes:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with **SubkeyName**.

Value
The enumeration integer value that is associated with each option in the list. This value is stored in NDI\params\ *SubkeyName**Value*.

EnumDesc
The display text that is associated with each value that appears in the menu.

Default
The default value for the menu.

The following table describes the possible INF entries for the RSC enumeration keywords.

SubkeyName	ParamDesc	Value	EnumDesc
*RscIPv4	Recv Segment Coalescing (IPv4)	0	Disabled
		1 (Default)	Enabled
*RscIPv6	Recv Segment Coalescing (IPv6)	0	Disabled

SubkeyName	ParamDesc	Value	EnumDesc
		1 (Default)	Enabled

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

For more information about using enumeration keywords, see Enumeration Keywords.

Programming Considerations for RSC Drivers

Article • 12/15/2021

The following sections describe issues to consider when implementing a receive-segment coalescing (RSC)-capable miniport driver.

- Responding to Queries for RSC Statistics
- Forwarded TCP Packets
- RSC Support for Lightweight Filters and MUX Intermediate Drivers
- Windows Filtering Platform (WFP) Inspection and Callout Drivers

Responding to Queries for RSC Statistics

NDIS, overlying drivers, and user-mode applications use the OID_TCP_RSC_STATISTICS OID to get the RSC statistics of a miniport adapter. RSC-capable miniport drivers must support this OID.

Forwarded TCP Packets

The miniport driver shouldn't perform RSC on segments in TCP packets that aren't intended for the host but are being forwarded out on another interface.

The host TCP/IP stack will disable RSC on any interface that has forwarding enabled. Weak host forwarding does not affect RSC.

RSC Support for Lightweight Filters and MUX Intermediate Drivers

All NDIS 6.30 lightweight filter drivers must support receive packets that are larger than the link maximum transmission unit (MTU). For more information about segment size limits, see Indicating Coalesced Segments.

NDIS will disable RSC on an interface if any lightweight filter driver or MUX intermediate driver in the host stack is NDIS 6.20 or lower.

A MUX intermediate driver may disable RSC on an interface, even if the interface's NDIS version is 6.30 or higher.

Windows Filtering Platform (WFP) Inspection and Callout Drivers

WFP callout drivers provide additional filtering functionality by adding custom callout functions to the filter engine at one or more of the kernel-mode filtering layers. Callouts support deep inspection and packet as well as stream modification.

WFP callout drivers may support handling of support receive packets that are larger than the link MTU. (For more information about packet size limits, see Tracking and Indicating Coalesced Segments.) Such WFP callout drivers should do the following:

- Opt in during registration to handle large packets.
- Set the callout driver flag as specified in the reference page for the **FWPS_CALLOUT2** structure.

Whenever a callout driver that has not opted in to handle large packets is registered, WFP will notify TCP/IP in the context of the registration. As part of handling this notification, TCP/IP will disable RSC on the interface.

If there is active TCP traffic during callout registration, TCP/IP will notify WFP. WFP will delay calling the registered filters until RSC is disabled. This will protect callout drivers from large packets.

UDP Receive Segment Coalescing Offload (URO)

Article • 05/22/2024

Starting in Windows 11, version 24H2, UDP Receive Segment Coalescing Offload (URO) enables network interface cards (NICs) to coalesce UDP receive segments. NICs can combine UDP datagrams from the same flow that match a set of rules into a logically contiguous buffer. These combined datagrams are then indicated to the Windows networking stack as a single large packet.

Coalescing UDP datagrams reduces the CPU cost to process packets in high-bandwidth flows, resulting in higher throughput and fewer cycles per byte.

The following sections describe the rules for coalescing UDP packets and how to write a URO miniport driver.

- Rules for coalescing UDP packets
- Write a URO miniport driver
- Programming considerations for URO drivers

Rules for coalescing UDP packets

URO coalescing can only be attempted on packets that meet all the following criteria:

- **IpHeader.Version** is identical for all packets.
- **IpHeader.SourceAddress** and **IpHeader.DestinationAddress** are identical for all packets.
- **UdpHeader.SourcePort** and **UdpHeader.DestinationPort** are identical for all packets.
- **UdpHeader.Length** is identical for all packets, except the last packet, which may be less.
- **UdpHeader.Length** must be nonzero.
- **UdpHeader.Checksum**, if non-zero, must be correct on all packets. This means that receive checksum offload must validate the packet.
- **Layer 2 headers** must be identical for all packets.

If the packets are IPv4, they must also meet the following criteria:

- **IPv4Header.Protocol** == 17 (UDP) for all packets.
- **EthernetHeader.EtherType** == 0x0800 for all packets.

- The **IPv4Header.HeaderChecksum** on received packets must be correct. This means that receive checksum offload must validate the header.
- **IPv4Header.HeaderLength** == 5 (no IPv4 Option Headers) for all packets.
- **IPv4Header.ToS** is identical for all packets.
- **IPv4Header.ECN** is identical for all packets.
- **IPv4Header.DontFragment** is identical for all packets.
- **IPv4Header.TTL** is identical for all packets.
- **IPv4Header.TotalLength** == **UdpHeader.Length** + length(**IPv4Header**) for all packets.

If the packets are IPv6, they must also meet the following criteria:

- **IPv6Header.NextHeader** == 17 (UDP) for all packets (no extension headers).
- **EthernetHeader.EtherType** == 0x86dd (IPv6) for all packets.
- **IPv6Header.TrafficClass** and **IPv6Header.ECN** are identical for all packets.
- **IPv6Header.FlowLabel** is identical for all packets.
- **IPv6Header.HopLimit** is identical for all packets.
- **IPv6Header.PayloadLength** == **UdpHeader.Length** for all packets.

URO packet structure

The resulting Single Coalesced Unit (SCU) must have a single IP header and UDP header, followed by the UDP payload for all coalesced datagrams concatenated together.

URO indications must set the **IPv4Header.TotalLength** field to the total length of the SCU, or **IPv6Header.PayloadLength** field to the length of the UDP payload and **UdpHeader.Length** field to the length of coalesced payloads.

If Layer 2 (L2) headers are present in coalesced datagrams, the SCU must contain a valid L2 header. The L2 header in the SCU must resemble the L2 header of the coalesced datagrams.

Checksum validation and indication

URO indications must set the **IPv4Header.HeaderChecksum** and **UdpHeader.Checksum** fields to zero and fill out the checksum offload out-of-band information on the SCU indicating IPv4 and UDP checksum success.

A packet that matches all conditions for being coalesced but fails checksum validation must be indicated separately. Packets received after it must not be coalesced with packets received before it.

For example, suppose packets 1, 2, 3, 4, and 5 are received from the same flow, but packet 3 fails checksum validation. Packets 1 and 2 can be coalesced together, and packets 4 and 5 can be coalesced together, but packet 3 must not be coalesced with either SCU. Packets 1 and 2 must not be coalesced together with packets 4 and 5. Packet 2 is the last packet in an SCU and packet 4 starts a new SCU. Additionally, the SCU containing packets 1 and 2 must be indicated before packet 3 is indicated and packet 3 must be indicated before the SCU containing packets 4 and 5.

Packet coalescing and flow separation

Packets from multiple flows may be coalesced in parallel, as hardware and memory permit. Packets from different flows must not be coalesced together.

Packets from multiple receives interleaved may be separated and coalesced with their respective flows. For example, given flows A, B, and C, if packets arrive in the order A, A, B, C, B, A, the packets from the A flow may be coalesced into AAA, and the packets from the B flow coalesced into BB, while the packet from the C flow may be indicated normally or coalesced with a pending SCU from flow C.

The packets within a given flow must not be reordered with respect to each other. For example, the packets from the A flow must be coalesced in the order received, regardless of the packets from the B and C flows received in between.

INF keyword for controlling URO

The following keyword can be used to enable/disable URO with a registry key setting.

***UdpRsc**

Enumeration standardized INF keywords have the following attributes:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with SubkeyName.

Value
The enumeration integer value that is associated with each option in the list. This value is stored in NDI\params\ *SubkeyName**Value*.

EnumDesc
The display text that is associated with each value that appears in the menu.

Default
The default value for the menu.

Expand table

SubkeyName	ParamDesc	Value	EnumDesc
***UdpRsc**	URO	0	Disabled
		1 (Default)	Enabled

For more information about using enumeration keywords, see Enumeration Keywords.

Write a URO miniport driver

Starting in NDIS 6.89, the NDIS interface for URO facilitates communication between TCP/IP and the NDIS miniport driver.

Report URO capability

A miniport driver advertises support for URO in the **UdpRsc** member of the **NDIS_OFFLOAD** structure, which it passes to the **NdisMSetMiniportAttributes** function.

Query URO capability

To check if a miniport driver supports URO, NDIS drivers and other applications can query the OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES OID, which returns the **NDIS_OFFLOAD** structure.

Query URO state

To determine the current URO state, NDIS drivers and other applications can query the OID_TCP_OFFLOAD_CURRENT_CONFIG OID request. NDIS handles this OID and doesn't pass it down to the miniport.

Change URO state

URO can be enabled or disabled by issuing the **OID_TCP_OFFLOAD_PARAMETERS** OID request. This OID uses an **NDIS_OFFLOAD_PARAMETERS** structure. In this structure, the

UdpRsc.Enabled member can have the following values:

Expand table

Value	Meaning
NDIS_OFFLOAD_PARAMETERS_UDP_RSC_NO_CHANGE 0	The miniport driver shouldn't change the current setting.
NDIS_OFFLOAD_PARAMETERS_UDP_RSC_DISABLED 1	URO is disabled.
NDIS_OFFLOAD_PARAMETERS_UDP_RSC_ENABLED 2	URO is enabled.

When a driver processes a **OID_TCP_OFFLOAD_PARAMETERS** OID request with the `NDIS_OFFLOAD_PARAMETERS_UDP_RSC_DISABLED` flag set, the NIC must wait to complete the request until all existing coalesced segments and outstanding URO indications are indicated. This ensures synchronization of URO enable/disable events across NDIS components.

After the miniport driver processes the **OID_TCP_OFFLOAD_PARAMETERS** OID request, the miniport driver must issue an NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG status indication with the updated offload state.

The `NDIS_OFFLOAD_PARAMETERS_SKIP_REGISTRY_UPDATE` flag in NDIS_OFFLOAD_PARAMETERS allows for runtime-only disabling of URO. Changes made with this flag aren't saved to the registry.

Opt-out of URO in NDIS 6.89 and later

Drivers targeting NDIS 6.89 and later should understand URO packets and handle them gracefully. To opt-out of URO:

- Lightweight filter (LWF) drivers set the `NDIS_FILTER_DRIVER_UDP_RSC_NOT_SUPPORTED` flag in the NDIS_FILTER_DRIVER_CHARACTERISTICS structure.
- Protocol drivers set the `NDIS_PROTOCOL_DRIVER_UDP_RSC_NOT_SUPPORTED` flag in the NDIS_PROTOCOL_DRIVER_CHARACTERISTICS structure.

This approach ensures components that are unfamiliar with URO don't receive URO NBLs. NDIS disables URO on the miniport during binding if an LWF or protocol driver that doesn’t support URO is present.

Programming considerations for URO drivers

Consider the following issues when implementing a URO-capable miniport driver.

Winsock URO API

For information on the Winsock URO API, see IPPROTO_UDP socket options. See the information on **UDP_RECV_MAX_COALESCED_SIZE** and **UDP_COALESCED_INFO**.

Windows TCP/IP stack updates

The Microsoft TCP/IP transport enables URO at bind time with NDIS, unless configuration prevents it from doing so.

WFP callouts can use `FWP_CALLOUT_FLAG_ALLOW_URO` in **FWPS_CALLOUT2** to advertise their support for URO. If an incompatible WFP callout is registered at a URO-sensitive layer, then the OS will disable URO while the callout is registered.

If a socket opts-in to URO with a max coalesced size greater than or equal to the hardware offload size, then the stack will deliver the NBLs from hardware unmodified to the socket. If a socket opts-in to a smaller max coalesced size, the stack will break the coalesced receive into the smaller size for the socket.

If a socket doesn't opt-in to URO, then the stack will resegment receives for that socket. In the absence of hardware URO, the existing software URO feature will continue to be available.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Introduction to Receive Side Scaling

Article • 09/27/2024

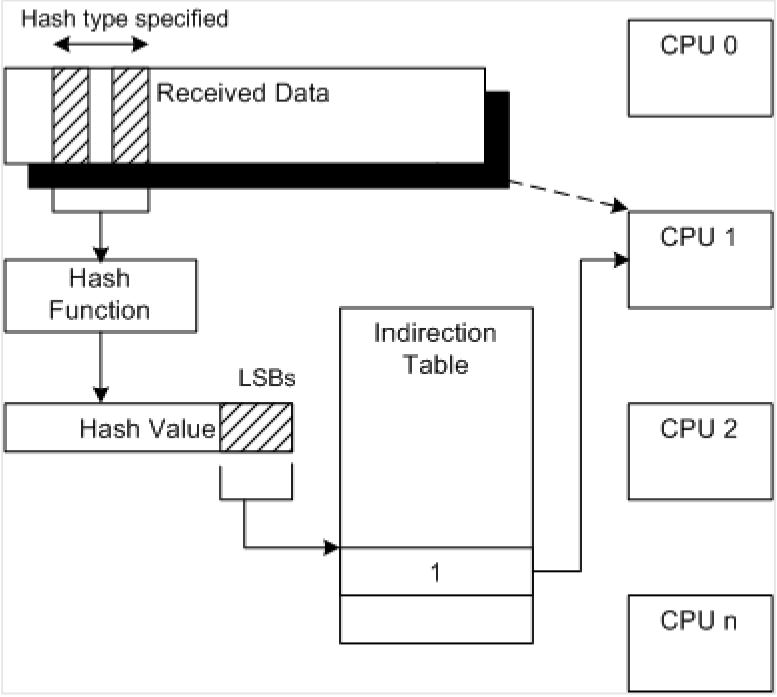
Receive side scaling (RSS) is a network driver technology that enables the efficient distribution of network receive processing across multiple CPUs in multiprocessor systems.

> ⓘ **Note**
>
> Because hyper-threaded CPUs on the same core processor share the same execution engine, the effect is not the same as having multiple core processors. For this reason, RSS does not use hyper-threaded processors.

To process received data efficiently, a miniport driver's receive interrupt service function schedules a deferred procedure call (DPC). Without RSS, a typical DPC indicates all received data within the DPC call. Therefore, all of the receive processing that is associated with the interrupt runs on the CPU where the receive interrupt occurs. For an overview of non-RSS receive processing, see Non-RSS Receive Processing.

RSS allows the NIC and miniport driver to schedule receive DPCs on other processors. The RSS design ensures that processing associated with a given connection stays on an assigned CPU. The NIC implements a hash function, and the resulting hash value helps select a CPU.

The following figure illustrates the RSS mechanism for determining a CPU.



A NIC uses a hashing function to compute a hash value over a defined area (hash type) within the received network data. The defined area can be noncontiguous.

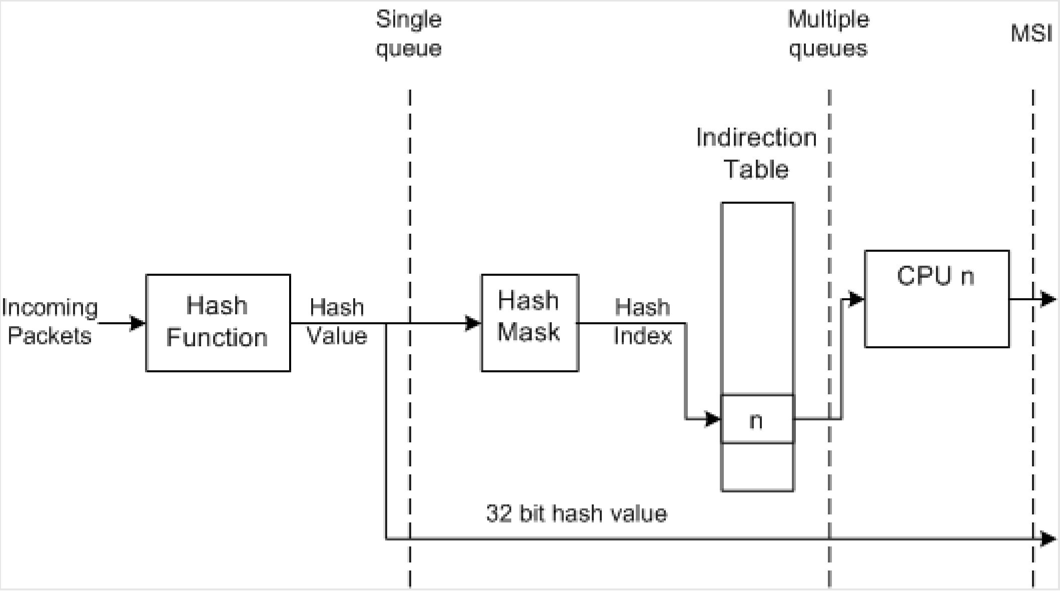
A number of least significant bits (LSBs) of the hash value are used to index an indirection table. The values in the indirection table are used to assign the received data to a CPU.

For more detailed information about specifying indirection tables, hash types, and hashing functions, see RSS Configuration.

With message signaled interrupt (MSI) support, a NIC can also interrupt the associated CPU. For more information about NDIS support for MSIs, see NDIS MSI-X.

Hardware support for RSS

The following figure illustrates the levels of hardware support for RSS.



There are three possible levels of hardware support for RSS:

- Hash calculation with a single queue: The NIC calculates the hash value, and the miniport driver assigns received packets to queues that are associated with CPUs.
- Hash calculation with multiple receive queues: The NIC assigns the received data buffers to queues associated with CPUs.
- Message signaled interrupts (MSIs): The NIC interrupts the CPU that should handle the received packets.

The NIC always passes on the 32-bit hash value.

How RSS improves system performance

RSS can improve network system performance by reducing:

- Processing delays by distributing receive processing from a NIC across multiple CPUs.

  Distributing receive processing helps to ensure that no CPU is heavily loaded while another CPU is idle.

- Spin lock overhead by increasing the probability that software algorithms that share data execute on the same CPU.

  Spin lock overhead occurs, for example, when a function executing on CPU0 possesses a spin lock on data that a function running on CPU1 must access. CPU1 spins (waits) until CPU0 releases the lock.

- Reloading of caches and other resources by increasing the probability that software algorithms that share data execute on the same CPU.

  Such reloading occurs, for example, when a function that is executing and accessing shared data on CPU0, executes on CPU1 in a subsequent interrupt.

To achieve these performance improvements in a secure environment, RSS provides the following mechanisms:

- Distributed processing

  RSS distributes the processing of receive indications from a given NIC in DPCs to multiple CPUs.

- In-order processing

  RSS preserves the order of delivery of received data packets. For each network connection, RSS processes receive indications on an associated CPU. For more information about RSS receive processing, see Indicating RSS Receive Data.

- Dynamic load balancing

  RSS provides a means to rebalance the network processing load between CPUs as host system load varies. To rebalance the load, overlying drivers can change the indirection table. For more information about specifying indirection tables, hash types, and hashing functions, see RSS Configuration.

- Send-side scaling

RSS enables driver stacks to process send and receive-side data for a given connection on the same CPU. Typically, an overlying driver (for example, TCP) sends part of a data block and waits for an acknowledgment before sending the balance of the data. The acknowledgment then triggers subsequent send requests. The RSS indirection table identifies a particular CPU for the receive data processing. By default, the send processing runs on the same CPU if it's triggered by the receive acknowledgment. A driver can also specify the CPU (for example, if a timer is used).

- Secure hash

  RSS includes a signature that provides added security. This signature protects the system from malicious remote hosts that might attempt to force the system into an unbalanced state.

- MSI-X support

  RSS, with support for MSI-X, runs the interrupt service routine (ISR) on the same CPU that later executes the DPC. This reduces spin lock overhead and reloading of caches.

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

Receive Side Scaling Version 2 (RSSv2)

Article • 12/15/2021

Receive Side Scaling improves the system performance related to handling of network data on multiprocessor systems. NDIS 6.80 and later support RSS Version 2 (RSSv2), which extends RSS by offering dynamic, per-VPort spreading of queues.

Overview

Compared to RSSv1, RSSv2 shortens the time between the measurement of CPU load and updating the indirection table. This avoids slowdown during high-traffic situations. To accomplish this, RSSv2 performs its actions at IRQL = DISPATCH_LEVEL, in the processor context of handling the request, and only operates on a subset of indirection table entries that point to the current processor. This means that RSSv2 can dynamically spread receive queues over multiple processors much more responsively than RSSv1.

Two OIDs, OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 and OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES, have been introduced in RSSv2 for miniport drivers to set proper RSS capabilities and control the indirection table respectively. OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 is a Regular OID, while OID_GEN_RSS_SET_INDIRECTION_ENTRIES is a Synchronous OID that cannot return NDIS_STATUS_PENDING. For more info about these OIDs, see their individual reference pages. For more info about Synchronous OIDs, see Synchronous OID request interface in NDIS 6.80.

RSSv2 terminology

This topic uses the following terms:

Term	Definition
RSSv1	The first generation receive side scaling mechanism. Uses OID_GEN_RECEIVE_SCALE_PARAMETERS.
RSSv2	The second generation receive side scaling mechanism supported in Windows 10, version 1803 and later, described in this topic.
Scaling entity	The miniport adapter itself in Native RSS mode, or a VPort in RSSv2 mode.

Term	Definition
ITE	An indirection table entry (ITE) of a given scaling entity. The total number of ITEs per VPort cannot exceed **NumberOfIndirectionTableEntriesPerNonDefaultPFVPort** or **NumberOfIndirectionTableEntriesForDefaultVPort** in VMQ mode or 128 in the Native RSS case. **NumberOfIndirectionTableEntriesPerNonDefaultPFVPort** and **NumberOfIndirectionTableEntriesForDefaultVPort** are members of the NDIS_NIC_SWITCH_CAPABILITIES structure.
Scaling mode	The per-VPort vmswitch policy that controls how its ITEs are handled at runtime. This can be static (no ITE moves due to load changes) or dyanmic (expansion and coalescing depending on current traffic load).
Queue	An underlying hardware object (queue) that backs an ITE. Depending on the hardware and indirection table, the configuration queue may back multiple ITEs. The total number of queues, including one that is used by the default queue, cannot exceed the preconfigured limit typically set by an administrator.
Default processor	A processor that receives packets for which the hash cannot be calculated. Each VPort has a default processor.
Primary processor	A processor specified as the **ProcessorAffinity** member of the NDIS_NIC_SWITCH_VPORT_PARAMETERS structure during VPort creation. This processor can be updated at runtime and specifies where VMQ traffic is directed.
Source CPU	The processor to which the ITE is currently mapped.
Target CPU	The processor to which the ITE is being re-mapped (using RSSv2).
Actor CPU	The processor on which RSSv2 requests are being made.

Advertising RSSv2 capability in a miniport driver

Miniport drivers advertise RSSv2 support by setting the **CapabilitiesFlags** member of the NDIS_RECEIVE_SCALE_CAPABILITIES structure with the *NDIS_RSS_CAPS_SUPPORTS_INDEPENDENT_ENTRY_MOVE* flag. This capability is required to enable RSSv2's CPU load balancing feature, along with the *NDIS_RECEIVE_FILTER_DYNAMIC_PROCESSOR_AFFINITY_CHANGE_SUPPORTED* flag that enables RSSv1 dynamic balancing for non-default VPorts (VMQs).

ⓘ Note

Upper layer protocols assume that the primary processor of the default VPort can be moved for RSSv2 miniport drivers.

If a miniport adapter does not advertise RSSv2 capability, all VMQ-enabled VPorts stay in static spreading mode even if these VPorts are requested to perform dynamic spreading. The RSSv1 OID for configuration of RSS parameters, OID_GEN_RECEIVE_SCALE_PARAMETERS, is used for these VPorts that are still in static spreading mode.

Miniport drivers only need to implement one RSS control mechanism - either RSSv1 or RSSv2. If the driver advertises RSSv2 support, NDIS will convert RSSv1 OIDs to RSSv2 OIDs if necessary to configure per-VPort spreading. The miniport driver must support the two new OIDs and modify the behavior of the RSSv1 OID_GEN_RECEIVE_SCALE_PARAMETERS OID as follows:

- OID_GEN_RECEIVE_SCALE_PARAMETERS is used only for Query requests in RSSv2 and not for setting RSS parameters.
- OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 is a Query and a Set OID used for configuring the scaling entity's parameters such as the number of queues, the number of ITEs, RSS enablement/disablement, and hash key updates.
- OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES is a Method OID used to perform modification of indirection table entries.

Handling RSSv2 OIDs

OID_GEN_RECEIVE_SCALE_PARAMETERS is only used to query the current RSS parameters of a given scaling entity. In RSSv1, this OID is used to set parameters. For RSSv2-capable miniport drivers, NDIS automatically performs this role conversion for the driver and issues the following two OIDs to set parameters instead.

OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 is a Regular OID and is handled the same as the OID_GEN_RECEIVE_SCALE_PARAMETERS OID was handled in RSSv1. This OID is not visible to NDIS light-weight filter drivers (LWFs) prior to NDIS 6.80.

OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES, however, is a Synchronous OID that cannot return NDIS_STATUS_PENDING. This OID must be executed and completed in the processor context which originated the OID. Like OID_GEN_RECEIVE_SCALE_PARAMETERS_V2, it is also not visible to NDIS LWFs prior to NDIS 6.80. LWFs in NDIS 6.80 and later are not permitted to delay this OID or move to another processor. Its payload contains an array of simple "move ITE" actions, each of which contains a command to move a single ITE for a scaling entity to a different target CPU. Elements of the array can reference different scaling entities (VPorts).

Each type of NDIS driver, miniport, filter, and protocol, have entry points to support the Synchronous OID request interface:

NDIS driver type	Synchronous OID handler(s)	Function to originate Synchronous OIDs
Miniport	*MiniportSynchronousOidRequest*	N/A
Filter	- *FilterSynchronousOidRequest* - *FilterSynchronousOidRequestComplete*	**NdisFSynchronousOidRequest**
Protocol	N/A	**NdisSynchronousOidRequest**

RSS state transitions, ITE updates, and primary/default processors

Steering parameters

In RSSv2, different parameters are used to steer traffic to the correct CPU depending on the RSS state (enabled or disabled). When RSS is disabled, only the primary processor is used for directing traffic. When RSS is enabled, both the default processor and all ITEs are used for directing traffic. These *steering parameters* are labele as "active" or "inactive", summarized in the following table:

Steering parameter	RSS disabled	RSS enabled
Primary processor	Active	Inactive
Default processor	Inactive	Active
ITE[0..N]	Inactive	Active

When a steering parameter is in the *active* state, it directs the traffic. From the moment of an RSS state transition that makes a parameter *inactive*, miniport drivers must track changes to the parameter until the reverse transition activates it again. This means that a miniport driver needs to track all updates to the default processor and indirection table entries while RSS is disabled for that scaling entity. When RSS is enabled, the current tracked state for the default processor and indirection table should take effect.

For example, consider the scenario when software vRSS is already enabled. In this case, the indirection table already exists in the upper layer protocol and is actively used by the upper layer's software spreading code. If, during hardware RSS enablement, all entries start pointing to the primary processor before the updates to *move* the indirection table entries are issued to and executed by the hardware, the primary processor might experience a short jam. If the miniport driver has tracked default processor and ITE information, it can direct traffic to where it is already expected by the upper layer.

Note that while miniport drivers must track all updates to inactive steering parameters, they should defer validation of those parameters until the RSS state change attempts to make these parameters *active*. For example, in the case of software spreading while hardware RSS is disabled, upper layer protocols can use any processor for spreading (including outside the adapter's RSS set). The upper layers ensure that, at the moment of RSS state transition, all *inactive* parameters are valid for the new RSS state. However, the miniport dirver should still validate the parameters and fail the RSS state transition if it discovers that any tracked *inactive* steering parameters are invalid.

Initial state and updates to steering parameters

The following table describes the initial state of the scaling entity after creation (for example, after VPort creation), as well as how the parameters can be updated:

Parameter	Description
Primary processor	• Initialized with the **Affinity** processor specified during VPort creation. • Can be updated using the OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OID with the **NDIS_RSS_SET_INDIRECTION_ENTRY_FLAG_PRIMARY_PROCESSOR** flag set. • Can be updated using the OID_NIC_SWITCH_VPORT_PARAMETERS OID with the **NDIS_NIC_SWITCH_VPORT_PARAMS_PROCESSOR_AFFINITY_CHANGED** flag set (this is the compatibility path for existing cmdlet's). • Can be read using the OID_NIC_SWITCH_VPORT_PARAMETERS OID with the **NDIS_NIC_SWITCH_VPORT_PARAMS_PROCESSOR_AFFINITY_CHANGED** flag (this is the compatibility path for existing cmdlet's). • Post-initialization moves of the primary processor do not affect the default processor or the contents of the indirection table.
Default processor	• Initialized with the **Affinity** processor specified during VPort creation. • Can be updated using the OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OID with the **NDIS_RSS_SET_INDIRECTION_ENTRY_FLAG_DEFAULT_PROCESSOR** flag set.

Parameter	Description
Indirection table	- **NumberOfIndirectionTableEntries** is set to **1**. - The only entry is initialized with the **Affinity** processor specified during VPort creation. - Can be updated using the OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OID.

Updates to ITEs and the primary/default processors (using OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES) is invoked from the processor to which the corresponding entry currently points. For a given VPort, the upper layer ensures that no OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OIDs to move ITEs or set the primary/default processors will be issued in these circumstances:

1. While OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 is in progress.
2. After the VPort deletion sequence is initiated. For example, the upper layer issues the set filter OID only after the last OID to move ITEs is completed.

RSS disablement

During RSS disablement, the upper layer protocol might choose to either point all the ITEs to the primary processor, then issue the OID to disable RSS, or it might choose to leave the indirection table as-is and disable RSS. In either case, receive traffic should target the primary processor.

RSSv2 maintains a requirement from RSSv1 that permits the upper layer protocol to delete a VPort without first disabling RSS. The upper layer can set the receive filter on the VPort to zero, thus ensuring that no receive traffic flows through the VPort, then proceed with VPort deletion without disabling RSS. The upper layer guarantees that no OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OIDs will be issued during or after VPort deletion.

During both RSS disablement and VPort deletion, the miniport driver should take care of any pending internal operations that might exist because of previous queue moves.

RSSv2 invariants

The upper layer protocol ensures that important invariants are not violated before performing management functions or ITE moves. For example:

1. Before reducing the number of queues, the upper layer ensures that the indirection table does not reference more processors than the new number of

queues for a VPort.

2. The upper layer should not request an indirection table update that violates the currently configured number of queues for a VPort. The miniport driver should enforce this and return a failure.
3. Before changing the number of indirection table entries for VMMQ-RESTRICTED adapters, the upper layer ensures that the contents of the indirection table are normalized to the power of 2.

Related links

OID_GEN_RECEIVE_SCALE_PARAMETERS_V2

OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES

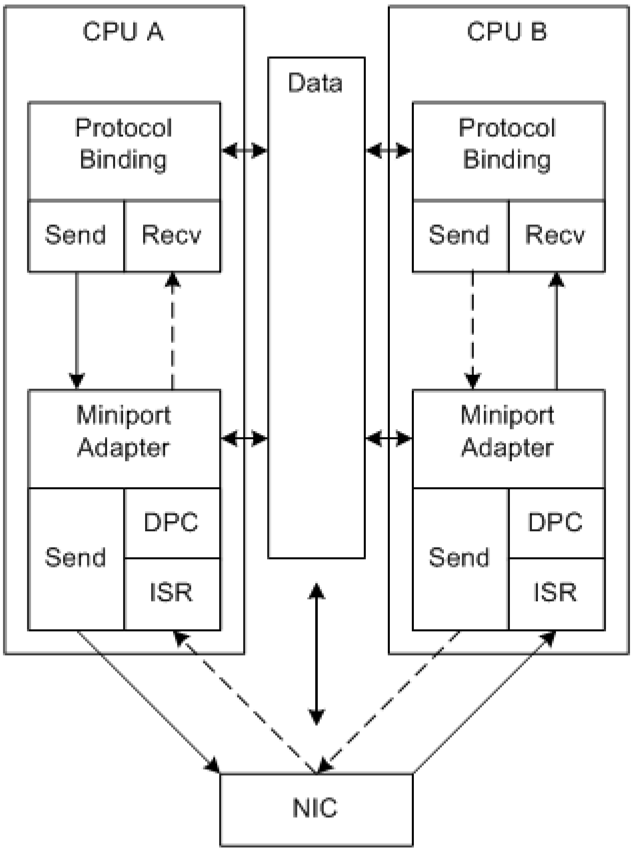
Synchronous OID request interface in NDIS 6.80

Non-RSS Receive Processing

Article • 12/15/2021

Miniport drivers that do not support RSS handle receive processing as described in this topic.

The following figure illustrates non-RSS receive processing.



In the figure, the dashed paths represent an alternate path for the send and receive processing. Because the system controls the scaling, the processing doesn't always occur on the CPU that provides the best performance. Connections are processed on the same CPU over successive interrupts only by chance.

The following process repeats for each non-RSS interrupt cycle:

1. The NIC uses DMA to fill a buffer with received data and interrupts the system.

   The miniport driver allocated the receive buffers in shared memory during initialization.

2. The NIC can continue to fill additional receive buffers at any time in this interrupt cycle. However, the NIC does not interrupt again until the miniport driver enables interrupts.

The received buffers that the system handles in one interrupt cycle can be associated with many different network connections.

3. NDIS calls the miniport driver's *MiniportInterrupt* function (ISR) on a system-determined CPU.

   Ideally, the ISR should go to the least busy CPU. However, in some systems, the system assigns the ISR to an available CPU or to a CPU that is associated with the NIC.

4. The ISR disables the interrupts and requests NDIS to queue a deferred procedure call (DPC) to process the received data.

5. NDIS calls the *MiniportInterruptDPC* function (the DPC) on the current CPU.

6. The DPC builds receive descriptors for all of the received buffers and indicates the data up the driver stack. For more information, see Receiving Network Data.

   There can be many buffers for many different connections and there is potentially a lot of processing to complete. The received data associated with subsequent interrupt cycles can be processed on other CPUs. The send processing for a given network connection can also run on a different CPU.

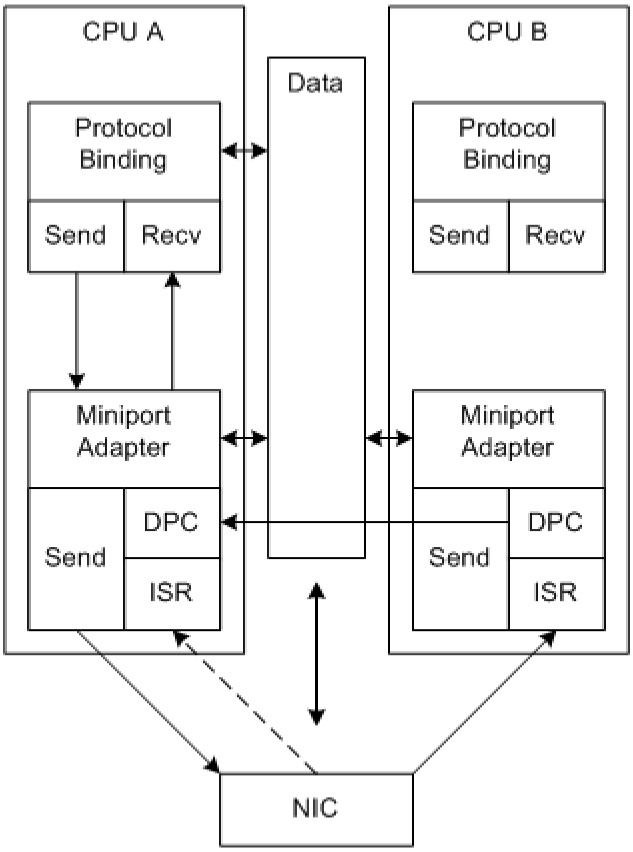
7. The DPC enables the interrupts. This interrupt cycle is complete and the process starts again.

RSS with a Single Hardware Receive Queue

Article • 12/15/2021

Miniport drivers can support RSS for NICs that support RSS hash calculation and a single receive descriptor queue.

The following figure illustrates RSS processing with a single receive descriptor queue.



In the figure, the dashed arrows represent an alternate path for the receive processing. RSS cannot control the CPU that receives the initial ISR call.

Unlike the non-RSS receive processing, RSS-based receive processing is distributed over multiple CPUs. Also, the processing for a given connection can be tied to a given CPU.

The following process repeats for each interrupt:

1. The NIC uses DMA to fill buffers with received data and interrupts the system.

   The miniport driver allocated the receive buffers in shared memory during initialization.

2. The NIC can fill additional receive buffers at any time but does not interrupt again until the miniport driver enables the interrupts.

The received buffers that the system handles in one interrupt can be associated with many different network connections.

3. NDIS calls the miniport driver's *MiniportInterrupt* function (ISR) on a system-determined CPU.

4. The ISR disables the interrupts and requests NDIS to queue a deferred procedure call (DPC) to process the received data.

5. NDIS calls the *MiniportInterruptDPC* function (DPC) on the current CPU. In the DPC:
   a. The miniport driver uses the hash values that the NIC calculated for each received buffer and reassigns each received buffer to a receive queue that is associated with a CPU.
   b. The current DPC requests NDIS to queue a DPC for each of the other CPUs that are associated with a non-empty receive queue.
   c. If the current DPC is running on a CPU that is associated with a non-empty queue, the current DPC processes the associated receive buffers and indicates the received data up the driver stack.

   Assigning queues, and queuing additional DPCs requires additional processing overhead. To achieve improved system performance, this overhead must be offset by better utilization of available CPUs.

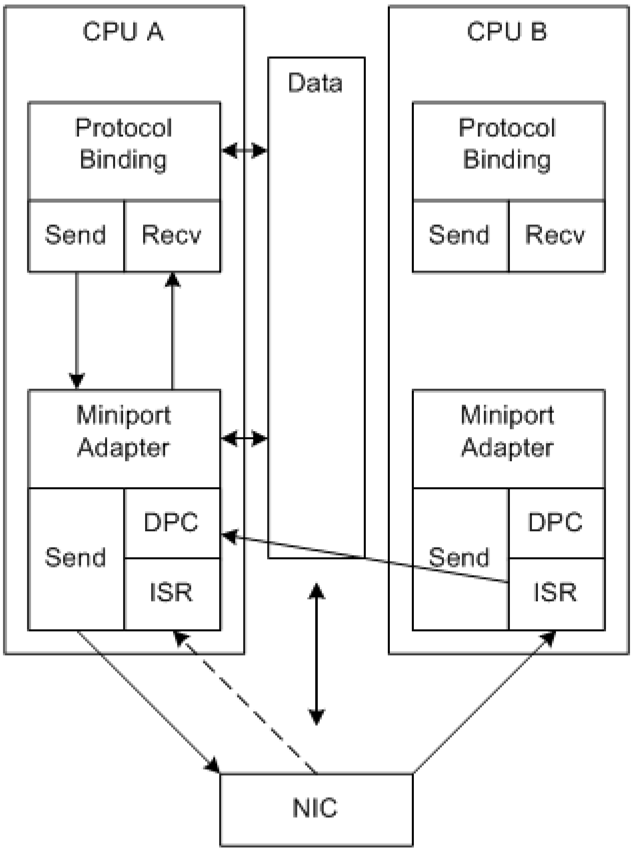
6. The DPC on a given CPU:
   a. Processes the receive buffers that are associated with its receive queue and indicates the data up the driver stack. For more information, see Indicating RSS Receive Data.
   b. Enables the interrupts, if it is the last DPC to complete. This interrupt is complete and the process starts again. The driver must use an atomic operation to identify the last DPC to complete. For example, the driver can use the **NdisInterlockedDecrement** function to implement an atomic counter.

RSS with Hardware Queuing

Article • 12/15/2021

RSS with hardware queuing improves system performance relative to RSS with a single hardware receive queue solution. NICs that support hardware queuing assign received data to multiple receive queues. The receive queues are associated with a CPU. The NIC assigns received data to CPUs based on hash values and an indirection table.

The following figure illustrates RSS with NIC receive queuing.



In the figure, the dashed arrows represent an alternate path for the receive processing. RSS cannot control the CPU that receives the initial ISR call. The driver does not have to queue the data so it can immediately schedule the initial DPCs on the correct CPUs.

The following process repeats for each interrupt:

1. The NIC:

   a. Uses DMA to fill buffers with received data.

      The miniport driver allocated the receive buffers in shared memory during initialization.

   b. Computes a hash value.

c. Queues the buffer for a CPU and provides the queue assignments to the miniport driver.

   For example, the NIC could loop steps 1-3 and DMA a list of CPU assignments after some number of packets are received. The specific mechanism is left to the NIC design.

d. Interrupts the system.

   The received buffers that the system handles in one interrupt are distributed between the CPUs.

2. NDIS calls the miniport driver's *MiniportInterrupt* function (ISR) on a system-determined CPU.

3. The miniport driver requests NDIS to queue deferred procedure calls (DPCs) for each of the CPUs that have a non-empty queue.

   Note that all the DPCs must complete before the driver enables interrupts. Also, note that the ISR might be running on a CPU that has no buffers to process.

4. NDIS calls the *MiniportInterruptDPC* function for each queued DPC. The DPC on a given CPU:

   a. Builds receive descriptors for all of the received buffers in its queue and indicates the data up the driver stack.

      For more information, see Indicating RSS Receive Data.

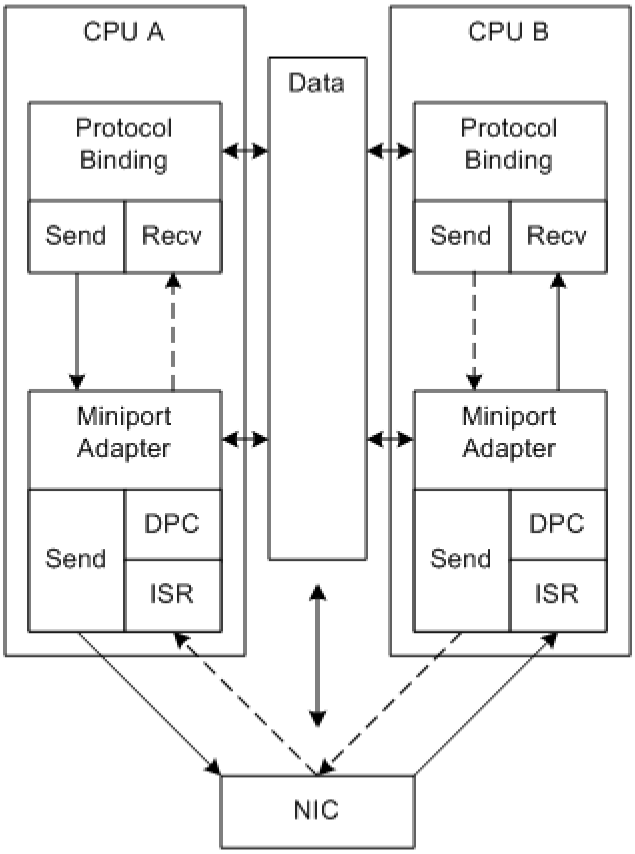
   b. Enables the interrupts, if it is the last DPC to complete. This interrupt is complete and the process starts again. The driver must use an atomic operation to identify the last DPC to complete. For example, the driver can use the **NdisInterlockedDecrement** function to implement an atomic counter.

RSS with Message Signaled Interrupts

Article • 12/15/2021

Miniport drivers can support message signaled interrupts (MSIs) to improve RSS performance. MSIs enable the NIC to request an interrupt on the CPU that will process the received data. For more information about NDIS support for MSI, see NDIS MSI-X.

The following figure illustrates RSS with MSI-X.



In the figure, the dashed arrows represent processing on a different connection. RSS with MSI-X allows the NIC to interrupt the correct CPU for a connection.

The following process repeats for each interrupt:

1. The NIC:

    a. Uses DMA to fill buffers with received data.

    The miniport driver allocated the receive buffers in shared memory during initialization.

    b. Computes a hash value.

    c. Queues the buffer to a CPU and provides the queue assignments to the miniport driver. For example, the NIC could loop steps 1-3 and DMA a list of

CPU assignments after some number of packets are received. The specific mechanism is left to the NIC design.

d. Using MSI-X, interrupts the CPU that is associated with a non-empty queue.

2. The NIC can fill additional receive buffers and add them to the queue at any time but does not interrupt that CPU again until the miniport driver enables the interrupts for that CPU.

3. NDIS calls the miniport driver's ISR (*MiniportInterrupt*) on the current CPU.

4. The ISR disables interrupts on the current CPU and queues a DPC on the current CPU.

   Interrupts can still occur on the other CPUs while the DPC is running on the current CPU.

5. NDIS calls the *MiniportInterruptDPC* function for each queued DPC. Each DPC:
   a. Builds receive descriptors for all of the received buffers in its queue and indicates the data up the driver stack. For more information, see Indicating RSS Receive Data.
   b. Enables interrupts for the current CPU. This interrupt is complete and the process starts again. Note that no atomic operation is required to track the progress of other DPCs.

RSS Hashing Types

Article • 12/15/2021

Overview

The RSS hashing type specifies the portion of received network data that a NIC must use to calculate an RSS hash value.

Overlying drivers set the hash type, function, and indirection table. The hash type that the overlying driver sets can be a subset of the type that the miniport driver can support. For more information, see RSS Configuration.

The hash type is an OR of valid combinations of the following flags:

- NDIS_HASH_IPV4
- NDIS_HASH_TCP_IPV4
- NDIS_HASH_UDP_IPV4
- NDIS_HASH_IPV6
- NDIS_HASH_TCP_IPV6
- NDIS_HASH_UDP_IPV6
- NDIS_HASH_IPV6_EX
- NDIS_HASH_TCP_IPV6_EX
- NDIS_HASH_UDP_IPV6_EX

These are the sets of valid flag combinations:

- IPv4 (combinations of NDIS_HASH_IPV4, NDIS_HASH_TCP_IPV4, and NDIS_HASH_UDP_IPV4)
- IPv6 (combinations of NDIS_HASH_IPV6, NDIS_HASH_TCP_IPV6, and NDIS_HASH_UDP_IPV6)
- IPv6 with extension headers (combinations of NDIS_HASH_IPV6_EX, NDIS_HASH_TCP_IPV6_EX, and NDIS_HASH_UDP_IPV6_EX)

A NIC must support one of the combinations from the IPv4 set. The other sets and combinations are optional. A NIC can support more than one set at a time. In this case, the type of data received determines which hash type the NIC uses.

In general, if the NIC cannot interpret the received data correctly, it must not compute the hash value. For example, if the NIC only supports IPv4 and it receives an IPv6 packet, which it cannot interpret correctly, it must not compute the hash value. If the NIC receives a packet for a transport type that it does not support, it must not compute the hash value. For example, if the NIC receives a UDP packet when it is supposed to be calculating hash values for TCP packets, it must not compute the hash value. In this case, the packet is processed as in the non-RSS case. For more information about the non-RSS receive processing, see Non-RSS Receive Processing.

IPv4 hash type combinations

The valid hash type combinations in the IPv4 set are:

- NDIS_HASH_IPV4
- NDIS_HASH_TCP_IPV4
- NDIS_HASH_UDP_IPV4
- NDIS_HASH_TCP_IPV4 | NDIS_HASH_IPV4
- NDIS_HASH_UDP_IPV4 | NDIS_HASH_IPV4
- NDIS_HASH_TCP_IPV4 | NDIS_HASH_UDP_IPV4 | NDIS_HASH_IPV4

NDIS_HASH_IPV4

If this flag alone is set, the NIC should compute the hash value over the following IPv4 header fields:

- Source-IPv4-Address
- Destination-IPv4-Address

> ⓘ **Note**
>
> If a NIC receives a packet that has both IP and TCP headers, NDIS_HASH_TCP_IPV4 should not always be used. In the case of a fragmented IP packet, NDIS_HASH_IPV4 must be used. This includes the first fragment which contains both IP and TCP headers.

NDIS_HASH_TCP_IPV4

If this flag alone is set, the NIC should parse the received data to identify an IPv4 packet that contains a TCP segment.

The NIC must identify and skip over any IP options that are present. If the NIC cannot skip over any IP options, it should not calculate a hash value.

The NIC should compute the hash value over the following fields:

- Source-IPv4-Address
- Destination-IPv4-Address
- Source TCP Port
- Destination TCP Port

NDIS_HASH_UDP_IPV4

If this flag alone is set, the NIC should parse the received data to identify an IPv4 packet that contains a UDP datagram.

The NIC must identify and skip over any IP options that are present. If the NIC cannot skip over any IP options, it should not calculate a hash value.

The NIC should compute the hash value over the following fields:

- Source-IPv4-Address
- Destination-IPv4-Address
- Source UDP Port
- Destination UDP Port

NDIS_HASH_TCP_IPV4 | NDIS_HASH_IPV4

If this flag combination is set, the NIC should perform the hash calculations as specified for the NDIS_HASH_TCP_IPV4 case. However, if the packet does not contain a TCP header, the NIC should compute the hash value as specified for the NDIS_HASH_IPV4 case.

NDIS_HASH_UDP_IPV4 | NDIS_HASH_IPV4

If this flag combination is set, the NIC should perform the hash calculations as specified for the NDIS_HASH_UDP_IPV4 case. However, if the packet does not contain a UDP header, the NIC should compute the hash value as specified for the NDIS_HASH_IPV4 case.

NDIS_HASH_TCP_IPV4 | NDIS_HASH_UDP_IPV4 | NDIS_HASH_IPV4

If this flag combination is set, the NIC should perform the hash calculation as specified by the transport in the packet. However, if the packet does not contain a TCP or UDP header, the NIC should compute the hash value as specified for the NDIS_HASH_IPV4 case.

IPv6 hash type combinations

The valid hash type combinations in the IPv6 set are:

- NDIS_HASH_IPV6
- NDIS_HASH_TCP_IPV6
- NDIS_HASH_UDP_IPV6
- NDIS_HASH_TCP_IPV6 | NDIS_HASH_IPV6
- NDIS_HASH_UDP_IPV6 | NDIS_HASH_IPV6
- NDIS_HASH_TCP_IPV6 | NDIS_HASH_UDP_IPV6 | NDIS_HASH_IPV6

NDIS_HASH_IPV6

If this flag alone is set, the NIC should compute the hash over the following fields:

- Source-IPv6-Address
- Destination-IPv6-Address

NDIS_HASH_TCP_IPV6

If this flag alone is set, the NIC should parse the received data to identify an IPv6 packet that contains a TCP segment. The NIC must identify and skip over any IPv6 extension headers that are present in the packet. If the NIC cannot skip over any IPv6 extension headers, it should not calculate a hash value.

The NIC should compute the hash value over the following fields:

- Source-IPv6 -Address
- Destination-IPv6 -Address
- Source TCP Port
- Destination TCP Port

NDIS_HASH_UDP_IPV6

If this flag alone is set, the NIC should parse the received data to identify an IPv6 packet that contains a UDP datagram. The NIC must identify and skip over any IPv6 extension headers that are present in the packet. If the NIC cannot skip over any IPv6 extension headers, it should not calculate a hash value.

The NIC should compute the hash value over the following fields:

- Source-IPv6-Address

- Destination-IPv6-Address
- Source UDP Port
- Destination UDP Port

NDIS_HASH_TCP_IPV6 | NDIS_HASH_IPV6

If this flag combination is set, the NIC should perform the hash calculations as specified for the NDIS_HASH_TCP_IPV6 case. However, if the packet does not contain a TCP header, the NIC should compute the hash as specified for the NDIS_HASH_IPV6 case.

For example, if the packet is fragmented, then it may not contain the TCP header. In that case, the NIC should compute the hash only over the IP header.

NDIS_HASH_UDP_IPV6 | NDIS_HASH_IPV6

If this flag combination is set, the NIC should perform the hash calculations as specified for the NDIS_HASH_UDP_IPV6 case. However, if the packet does not contain a UDP header, the NIC should compute the hash as specified for the NDIS_HASH_IPV6 case.

For example, if the packet is fragmented, then it may not contain the UDP header. In that case, the NIC should compute the hash only over the IP header.

NDIS_HASH_TCP_IPV6 | NDIS_HASH_UDP_IPV6 | NDIS_HASH_IPV6

If this flag combination is set, the NIC should perform the hash calculation as specified by the transport in the packet. However, if the packet does not contain a TCP or UDP header, the NIC should compute the hash value as specified in the NDIS_HASH_IPV6 case.

For example, if the packet is fragmented, then it may not contain the TCP or UDP header. In that case, the NIC should compute the hash only over the IP header.

IPv6 with extension headers hash type combinations

The valid combinations in the IPv6 with extension headers set are:

- NDIS_HASH_IPV6_EX
- NDIS_HASH_TCP_IPV6_EX
- NDIS_HASH_UDP_IPV6_EX

- NDIS_HASH_TCP_IPV6_EX | NDIS_HASH_IPV6_EX
- NDIS_HASH_UDP_IPV6_EX | NDIS_HASH_IPV6_EX
- NDIS_HASH_TCP_IPV6_EX | NDIS_HASH_UDP_IPV6_EX | NDIS_HASH_IPV6_EX

NDIS_HASH_IPV6_EX

If this flag alone is set, the NIC should compute the hash over the following fields:

- Home address from the home address option in the IPv6 destination options header. If the extension header is not present, use the Source IPv6 Address.
- IPv6 address that is contained in the Routing-Header-Type-2 from the associated extension header. If the extension header is not present, use the Destination IPv6 Address.

NDIS_HASH_TCP_IPV6_EX

If this flag alone is set, the NIC should compute the hash over the following fields:

- Home address from the home address option in the IPv6 destination options header. If the extension header is not present, use the Source IPv6 Address.
- IPv6 address that is contained in the Routing-Header-Type-2 from the associated extension header. If the extension header is not present, use the Destination IPv6 Address.
- Source TCP Port
- Destination TCP Port

NDIS_HASH_UDP_IPV6_EX

If this flag alone is set, the NIC should compute the hash over the following fields:

- Home address from the home address option in the IPv6 destination options header. If the extension header is not present, use the Source IPv6 Address.
- IPv6 address that is contained in the Routing-Header-Type-2 from the associated extension header. If the extension header is not present, use the Destination IPv6 Address.
- Source UDP Port
- Destination UDP Port

NDIS_HASH_TCP_IPV6_EX | NDIS_HASH_IPV6_EX

If this flag combination is set, the NIC should perform the hash calculations as specified for the NDIS_HASH_TCP_IPV6_EX case. However, if the packet does not contain a TCP header, the NIC should compute the hash as specified for the NDIS_HASH_IPV6_EX case.

NDIS_HASH_UDP_IPV6_EX | NDIS_HASH_IPV6_EX

If this flag combination is set, the NIC should perform the hash calculations as specified for the NDIS_HASH_UDP_IPV6_EX case. However, if the packet does not contain a UDP header, the NIC should compute the hash as specified for the NDIS_HASH_IPV6_EX case.

NDIS_HASH_TCP_IPV6_EX | NDIS_HASH_UDP_IPV6_EX | NDIS_HASH_IPV6_EX

If this flag combination is set, the NIC should perform the hash calculations as specified by the packet transport. However, if the packet does not contain a TCP or UDP header, the NIC should compute the hash as specified for the NDIS_HASH_IPV6_EX case.

> ⓘ **Note**
>
> If a miniport driver reports NDIS_RSS_CAPS_HASH_TYPE_TCP_IPV6_EX and/or NDIS_RSS_CAPS_HASH_TYPE_UDP_IPV6_EX capability for a NIC, the NIC must calculate hash values (over fields in the IPv6 extension headers) in accordance with the IPv6 extension hash types that the protocol driver set. The NIC can store either the extension hash type or the regular hash type in the NET_BUFFER_LIST structure of the IPv6 packet for which a hash value is computed.

A miniport driver sets the hash type in a **NET_BUFFER_LIST** structure before indicating the received data. For more information, see Indicating RSS Receive Data.

RSS Hashing Functions

Article • 12/15/2021

Overview

A NIC or its miniport driver uses the RSS hashing function to calculate an RSS hash value.

Overlying drivers set the hash type, function, and table to assign connections to CPUs. For more information, see RSS Configuration.

The hashing function can be one of the following:

- **NdisHashFunctionToeplitz**
- **NdisHashFunctionReserved1**
- **NdisHashFunctionReserved2**
- **NdisHashFunctionReserved3**

> ⓘ Note
>
> Currently, **NdisHashFunctionToeplitz** is the only hashing function available to miniport drivers. The other hashing functions are reserved for NDIS.

A miniport driver should identify the hashing function and value that it uses in each **NET_BUFFER_LIST** structure before the driver indicates received data. For more information, see Indicating RSS Receive Data.

Examples

The following four pseudocode examples show how to calculate the **NdisHashFunctionToeplitz** hash value. These examples represent the four possible hash types that are available for **NdisHashFunctionToeplitz**. For more information about hash types, see RSS Hashing Types.

To simplify the examples, a generalized algorithm that processes an input byte stream is required. Specific formats for the byte streams are defined later in the four examples.

The overlying driver provides a secret key (K) to the miniport driver for use in the hash calculation. The key is 40 bytes (320 bits) long. For more information about the key, see RSS Configuration.

Given an input array that contains *n* bytes, the byte stream is defined as follows:

C++

```
input[0] input[1] input[2] ... input[n-1]
```

The left-most byte is input[0], and the most-significant bit of input[0] is the left-most bit. The right-most byte is input[n-1], and the least-significant bit of input[n-1] is the right-most bit.

Given the preceding definitions, the pseudocode for processing a general input byte stream is defined as follows:

C++

```
ComputeHash(input[], n)

result = 0
For each bit b in input[] from left to right
{
if (b == 1) result ^= (left-most 32 bits of K)
shift K left 1 bit position
}

return result
```

The pseudocode contains entries of the form @n-m. These entries identify the byte range of each element in the TCP packet.

Example Hash Calculation for IPv4 with the TCP Header

Concatenate the SourceAddress, DestinationAddress, SourcePort, and DestinationPort fields of the packet into a byte array, preserving the order in which they occurred in the packet:

C++

```
Input[12] = @12-15, @16-19, @20-21, @22-23
Result = ComputeHash(Input, 12)
```

Example Hash Calculation for IPv4 Only

Concatenate the SourceAddress and DestinationAddress fields of the packet into a byte array.

```c++
Input[8] = @12-15, @16-19
Result = ComputeHash(Input, 8)
```

Example Hash Calculation for IPv6 with the TCP Header

Concatenate the SourceAddress, DestinationAddress, SourcePort, and DestinationPort fields of the packet into a byte array, preserving the order in which they occurred in the packet.

```c++
Input[36] = @8-23, @24-39, @40-41, @42-43
Result = ComputeHash(Input, 36)
```

Example Hash Calculation for IPv6 Only

Concatenate the SourceAddress and DestinationAddress fields of the packet into a byte array.

```c++
Input[32] = @8-23, @24-39
Result = ComputeHash(Input, 32)
```

Verifying the RSS Hash Calculation

Article • 12/15/2021

You should verify your implementation of the RSS hash calculation. To verify your calculations for the **NdisHashFunctionToeplitz** hash function, use the following secret key data:

```syntax
0x6d, 0x5a, 0x56, 0xda, 0x25, 0x5b, 0x0e, 0xc2,
0x41, 0x67, 0x25, 0x3d, 0x43, 0xa3, 0x8f, 0xb0,
0xd0, 0xca, 0x2b, 0xcb, 0xae, 0x7b, 0x30, 0xb4,
0x77, 0xcb, 0x2d, 0xa3, 0x80, 0x30, 0xf2, 0x0c,
0x6a, 0x42, 0xb7, 0x3b, 0xbe, 0xac, 0x01, 0xfa
```

The following table provides verification data for the IPv4 versions of the **NdisHashFunctionToeplitz** hash function. The destination and source columns contain the input data and the IPv4 columns contain the resulting hash value.

Destination Address :Port	Source Address :Port	IPv4 only	IPv4 with TCP
161.142.100.80 :1766	66.9.149.187 :2794	0x323e8fc2	0x51ccc178
65.69.140.83 :4739	199.92.111.2 :14230	0xd718262a	0xc626b0ea
12.22.207.184 :38024	24.19.198.95 :12898	0xd2d0a5de	0x5c2b394a
209.142.163.6 :2217	38.27.205.30 :48228	0x82989176	0xafc7327f
202.188.127.2 :1303	153.39.163.191 :44251	0x5d1809c5	0x10e828a2

The following table contains verification data for the IPv6 versions of the RSS hash algorithm. The destination and source columns contain the input data and the IPv6 columns contain the resulting hash value. Note that the IPv6 addresses are provided for verification of the algorithm only; they might not make sense as actual addresses.

Destination Address (Port)	Source Address (Port)	IPv6 only	IPv6 with TCP
3ffe:2501:200:3::1 (1766)	3ffe:2501:200:1fff::7 (2794)	0x2cc18cd5	0x40207d3d
ff02::1 (4739)	3ffe:501:8::260:97ff:fe40:efab (14230)	0x0f0c461c	0xdde51bbf

Destination Address (Port)	Source Address (Port)	IPv6 only	IPv6 with TCP
fe80::200:f8ff:fe21:67cf (38024)	3ffe:1900:4545:3:200:f8ff:fe21:67cf (44251)	0x4b61e985	0x02d1feef

RSS Configuration

Article • 09/27/2024

To obtain RSS configuration information, an overlying driver can send an OID query of OID_GEN_RECEIVE_SCALE_CAPABILITIES to a miniport driver. NDIS also provides the RSS configuration information to overlying protocol drivers in the **NDIS_BIND_PARAMETERS** structure during initialization.

The overlying driver chooses a hashing function, type, and indirection table. To set these configuration options, the driver sends an OID set request of OID_GEN_RECEIVE_SCALE_PARAMETERS to the miniport driver. Overlying drivers can also query this OID to obtain the current RSS settings. The information buffer for the OID_GEN_RECEIVE_SCALE_PARAMETERS OID contains a pointer to an **NDIS_RECEIVE_SCALE_PARAMETERS** structure.

The overlying driver can disable RSS on the NIC. In this case, the driver sets the NDIS_RSS_PARAM_FLAG_DISABLE_RSS flag in the **Flags** member of the NDIS_RECEIVE_SCALE_PARAMETERS structure. When this flag is set, the miniport driver should ignore all of the other flags and settings and disable RSS on the NIC.

NDIS processes OID_GEN_RECEIVE_SCALE_PARAMETERS before passing it to the miniport driver and updates the miniport adapter's *RSS standardized keyword, if required. For more information about the ***RSS** keyword, see Standardized INF Keywords for RSS.

After receiving an OID_GEN_RECEIVE_SCALE_PARAMETERS set request with the NDIS_RSS_PARAM_FLAG_DISABLE_RSS flag set, the miniport driver should set the RSS state of the NIC to the initial state of the NIC after initialization. Therefore, if the miniport driver receives a subsequent OID_GEN_RECEIVE_SCALE_PARAMETERS set request with the NDIS_RSS_PARAM_FLAG_DISABLE_RSS flag cleared, all of the parameters should have the same values that were set after the miniport driver received the OID_GEN_RECEIVE_SCALE_PARAMETERS set request for the first time after the miniport adapter was initialized.

An overlying driver can use the OID_GEN_RECEIVE_HASH OID to enable and configure hash calculations on received frames without enabling RSS. Overlying drivers can also query this OID to obtain the current receive hash settings.

The information buffer for the OID_GEN_RECEIVE_HASH OID contains a pointer to an **NDIS_RECEIVE_HASH_PARAMETERS** structure. For a set request, the OID specifies the hash parameters that the miniport adapter should use. For a query request, the OID returns the hash parameters that the miniport adapter is using. This OID is optional for drivers that support RSS.

Note If receive hash calculation is enabled, NDIS disables receive hash calculation before it enables RSS. If RSS is enabled, NDIS disables RSS before it enables receive hash calculation.

All of the miniport adapters that the miniport driver supports must provide the same hash configuration settings to all subsequent protocol bindings. This OID also includes the secret key that the miniport driver or NIC must use for hash calculations. The key is 320 bits long (40 bytes) and can contain any data that the overlying driver chooses, for example, a random stream of bytes.

To rebalance the processing load, the overlying driver can set the RSS parameters and modify the indirection table. Normally, all the parameters are unchanged except for the indirection table. However, after RSS is initialized, the overlying driver might change other RSS initialization parameters. If necessary, the miniport driver can reset the NIC hardware to change the hash function, hash secret key, hash type, base CPU number, or the number of bits that are used to index the indirection table.

Note The overlying driver can set these parameters at any time. This can cause out of order receive indications. Miniport drivers that support TCP are not required to purge their receive queues in this instance.

The following figure provides example contents for two instances of the indirection table.

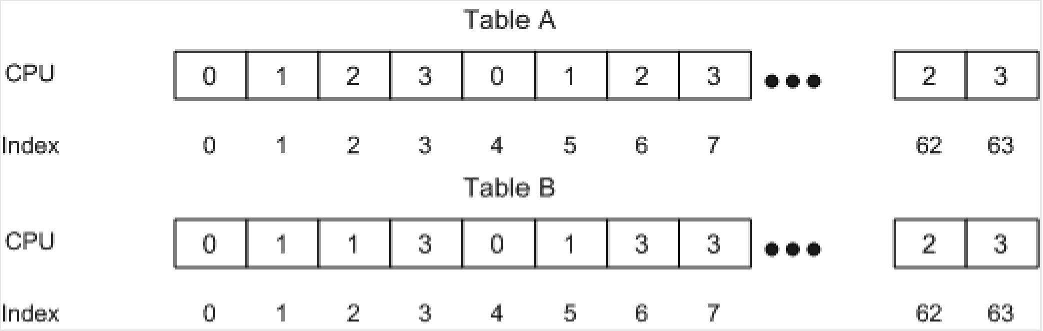


Table A

CPU	0	1	2	3	0	1	2	3	•••	2	3
Index	0	1	2	3	4	5	6	7		62	63

Table B

CPU	0	1	1	3	0	1	3	3	•••	2	3
Index	0	1	2	3	4	5	6	7		62	63

The preceding figure assumes a four processor configuration, and the number of least significant bits used from the hash value is 6 bits. Therefore, the indirection table contains 64 entries.

In the figure, table A lists the values in the indirection table immediately after initialization. Later, as normal traffic load varies, the processor load grows unbalanced. The overlying driver detects the unbalanced condition and attempts to rebalance the load by defining a new indirection table. Table B lists the new indirection table values. In the table B, some of the load from CPU 2 is moved to CPUs 1 and 3.

Note When the indirection table is changed, for a short time (while the current receive descriptor queues are being processed), packets can be processed on the wrong CPU. This is a normal transient condition.

The size of the indirection table is typically two to eight times the number of processors in the system.

When the miniport driver distributes packets to CPUs, if there are far too many CPUs, the effort spent in distributing the load could become prohibitive. In this case, overlying drivers should choose a subset of CPUs on which the processing of network data occurs.

In some cases, the number of available hardware receive queues might be less than the number of CPUs on the system. The miniport driver must examine the indirection table to determine the CPU numbers to associate with hardware queues. If the total number of different CPU numbers that appear in the indirection table is more than the number of hardware queues that the NIC supports, the miniport driver must pick a subset of the CPU numbers from the indirection table. The subset is equal in number to the number of hardware queues. The miniport driver obtained the **IndirectionTableSize** parameter from OID_GEN_RECEIVE_SCALE_PARAMETERS. The miniport driver specified the **NumberOfReceiveQueues** value in response to OID_GEN_RECEIVE_SCALE_CAPABILITIES.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Reserving Processors for Applications

Article • 09/27/2024

The receive side scaling (RSS) interface enables an administrator to reserve a set of processors for applications to use. The administrator can reserve a set of processors starting at logical CPU number 0 and ending at a specified CPU number. The RSS *base CPU number* is the CPU number of the first CPU that RSS can use. RSS cannot use the CPUs that are numbered below the base CPU number. For example, on a quad-core system with hyper-threading turned off, if base CPU number is set to 1, processors 1, 2, and 3 can be used for RSS.

NDIS uses the default value of 0 for base CPU number, but an administrator can change this value. The RSS interface does not permit the administrator to reserve a non-contiguous, arbitrary subset of CPUs for applications to use.

In Microsoft Windows Server 2003 with the Scalable Networking Pack, administrators can set the base CPU number with the **RssBaseCpu** registry keyword in **HKEY_LOCAL_MACHINE\\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters**. The **RssBaseCpu** value is a DWORD type and, if it is not present, NDIS uses the default value of 0.

In Windows Server 2008, administrators can set the base CPU number with the **RssBaseCpu** registry keyword in **HKEY_LOCAL_MACHINE\\SYSTEM\CurrentControlSet\Services\NDIS\Parameters**. The **RssBaseCpu** value is a DWORD type and, if it is not present, NDIS uses the default value of 0. This registry keyword also applies to later versions of Windows Server.

Note Starting in Windows 8 and Windows Server 2012, administrators can control many aspects of Network Adapters by using PowerShell cmdlets. Directly editing the registry is now discouraged.

The PowerShell cmdlet for reserving RSS CPUs is Set-NetAdapterRss. The primary difference between using **Set-NetAdapterRss** and using **RssBaseCpu** is that PowerShell cmdlets operate on a per-Network Adapter basis while **RssBaseCpu** is global, meaning it applies to all Network Adapters. Generally, working with each Network Adapter separately is recommended because it offers more flexibility, granularity, and understandability in giving each Network Adapter its own configuration. However, administrators might still use the global **RssBaseCpu** key if they would like to apply a configuration to all current and all future Network Adapters at the same time.

For a complete list of Network Adapter cmdlets, see Network Adapter Cmdlets in Windows PowerShell.

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

Setting the Number of RSS Processors

Article • 06/28/2024

Administrators should set the number of receive side scaling (RSS) processors to help the overall performance of a computer.

Concurrent deferred procedure calls (DPCs) that are running on multiple CPUs enable distributed receive processing and remove the CPU bottleneck (for example, in high-speed NICs). However, multiple DPCs do create additional overhead. The interrupt and DPC processing overhead increases as more processors are used for RSS. Therefore, when RSS is active, the total CPU utilization across all CPUs increases. An administrator should select the number of CPUs that are used for RSS to avoid a situation where using RSS leaves less processing power for applications to use and does not improve network throughput.

> ⓘ **Note**
>
> Starting in Windows 8 and Windows Server 2012, administrators can control many aspects of Network Adapters by using PowerShell cmdlets. Directly editing the registry is now discouraged.

The PowerShell cmdlet for setting the number of RSS CPUs is Set-NetAdapterRss.

The primary difference between using **Set-NetAdapterRss** and using the **MaxNumRssCpus** registry keyword is that PowerShell cmdlets operate on a per-Network Adapter basis while **MaxNumRssCpus** is global, meaning it applies to all Network Adapters. Generally, working with each Network Adapter separately is recommended because it offers more flexibility, granularity, and understandability in giving each Network Adapter its own configuration. However, administrators might still use the global **MaxNumRssCpus** key if they would like to apply a configuration to all current and all future Network Adapters at the same time.

For a complete list of Network Adapter cmdlets, see Network Adapter Cmdlets in Windows PowerShell.

In Microsoft Windows Server 2003 with the Scalable Networking Pack, administrators can set the maximum number of RSS CPUs with the **MaxNumRssCpus** registry keyword in **HKEY_LOCAL_MACHINE\\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters**. The **MaxNumRssCpus** value is a DWORD type and, if it is not present, NDIS uses the default value of 4.

In Windows Server 2008, administrators can set the maximum number of RSS CPUs with the **MaxNumRssCpus** registry keyword in **HKEY_LOCAL_MACHINE\\SYSTEM\CurrentControlSet\Services\Ndis\Parameters**. The **MaxNumRssCpus** value is a DWORD type and, if it is not present, NDIS uses the default value of 4. This registry keyword also applies to later versions of Windows Server.

To avoid complicated cases (and unrealistic cases that are not implemented in actual hardware) where the number of available hardware receive queues is less than the number of RSS CPUs, administrators must not set the **MaxNumRssCpus** value to a value that is greater than 16.

The actual number of CPUs that are used for RSS is also limited by the total number of core processors that remain after the RSS base CPU number has been configured. For example, if the administrator sets the maximum number of RSS CPUs on a quad-core computer system to 6, the networking driver stack uses, at most, 4 CPUs for RSS. If the administrator also sets the RSS base CPU number to 1, the networking driver stack uses at most 3 CPUs (CPU numbers 1, 2, and 3).

The number of CPUs that the computer uses for RSS is static and does not change at run time. Therefore, any changes to the **MaxNumRssCpus** registry value require a restart to take effect.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Standardized INF Keywords for RSS

Article • 09/27/2024

The RSS interface supports standardized INF keywords that appear in the registry and are specified in INF files.

The following list shows the enumeration standardized INF keywords for RSS:

***RSS**
Enable or disable support for RSS for miniport adapters.

***RSSProfile**
The processor selection and load-balancing profile.

Note: Changes to the ***RSSProfile** setting require an adapter restart.

Note: If ***RSSProfile** is set to **NdisRssProfileBalanced**, you can't configure advanced keywords such as ***RssBaseProcNumber**, ***RssBaseProcGroup**, ***RssMaxProcNumber**, ***RssMaxProcGroup**, or ***NumaNodeId**. You can configure ***MaxRssProcessors** and ***NumRSSQueues**.

NDIS 6.30 added support for ***RSSProfile**.

Enumeration standardized INF keywords have the following attributes:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with SubkeyName.

Value
The enumeration integer value that is associated with each option in the list. This value is stored in NDI\params\ *SubkeyName\Value*. EnumDesc
The display text that is associated with each value that appears in the menu.

Default
The default value for the menu.

The following table describes the possible INF entries for the RSS enumeration keywords.

Expand table

SubkeyName	ParamDesc	Value	EnumDesc
*RSS	Receive Side Scaling	0	Disabled
		1 (Default)	Enabled
*RSSProfile	RSS load balancing	1	**ClosestProcessor**: Default behavior is consistent with that of Windows Server 2008 R2.

SubkeyName	ParamDesc	Value	EnumDesc
	profile		
		2	**ClosestProcessorStatic**: No dynamic load-balancing - Distribute but don't load-balance at runtime.
		3	**NUMAScaling**: Assign RSS CPUs in a round robin basis across every NUMA node to enable applications that are running on NUMA servers to scale well.
		4 (Default)	**NUMAScalingStatic**: RSS processor selection is the same as for NUMA scalability without dynamic load-balancing.
		5	**ConservativeScaling**: RSS uses as few processors as possible to sustain the load. This option helps reduce the number of interrupts.
		6 (Default on heterogeneous CPU systems)	**NdisRssProfileBalanced**: RSS processor selection is based on traffic workload. Only available in NetAdapterCx, starting in WDK preview version 25197.

The following list shows the standardized INF keywords for RSS that can be edited:

***RssBaseProcGroup**
The number of the processor group for the processor number that is specified in the ***RssBaseProcNumber** keyword.

***NumaNodeId**
The preferred NUMA node that is used for the memory allocations of the network adapter. Also, the operating system attempts to use the CPUs from the preferred NUMA node first for RSS.

A driver for a PCI expansion card should not specify the NUMA node ID statically in its INF, since the closest node depends on which PCI slot the card is plugged into. Only specify ***NumaNodeId** if the network adapter is integrated into the system, the NUMA node is known in advance, and the node cannot be determined at runtime by querying ACPI.

Note: If this keyword is present and its value is less than the number of NUMA nodes in the computer, NDIS uses this value in the **PreferredNumaNode** member in the **NDIS_RSS_PROCESSOR_INFO** structure.

Note: In Windows 8 the ***NumaNodeId** value is ignored if the NIC RSS profile is set to **NUMAScaling**(2) or **NUMAScalingStatic**(3).

***RssBaseProcNumber**
The number of the base RSS processor in the specified group.

***MaxRssProcessors**
The maximum number of RSS processors.

***RssMaxProcNumber**
The maximum processor number of the RSS interface. If ***RssMaxProcNumber** is specified, then ***RssMaxProcGroup** should also be specified.

***NumRSSQueues**
The number of RSS queues.

***RssMaxProcGroup** The maximum processor group of the RSS interface.

***RssBaseProcGroup** together with ***RssBaseProcNumber** form a PROCESSOR_NUMBER structure that identifies the smallest processor number that can be used with RSS. ***RssMaxProcGroup** together with ***RssMaxProcNumber** form a PROCESSOR_NUMBER structure that identifies the maximum processor number that can be used with RSS.

For example, suppose ***RssBaseProcGroup** is set to 1, ***RssBaseProcNumber** is set to 16, ***RssMaxProcGroup** is set to 3, and ***RssMaxProcNumber** is set to 8. Using `<group>:<processor>` notation, the base processor is 1:16 and the max processor is 3:8. Then processors 0:0, 0:32, 1:0, and 1:15 will not be considered candidates for RSS, because they are below the base processor number. Processors 1:16, 1:31, 2:0, 2:63, 3:0, and 3:8 will all be considered candidates for RSS, because they fall in the range 1:16 through 3:8. Processors 3:9, 3:31, and 4:0 will not be considered candidates for RSS, because they are beyond the maximum processor number.

NDIS 6.20 added support for the ***RssBaseProcGroup**, ***NumaNodeId**, ***RssBaseProcNumber**, and ***MaxRssProcessors** keywords.

NDIS 6.30 added support for the ***RssMaxProcNumber**, and ***NumRSSQueues** keywords.

Standardized INF keywords that can be edited have the following attributes:

SubkeyName
The name of the keyword that you must specify in the INF file and that appears in the registry.

ParamDesc
The display text that is associated with SubkeyName.

Type
The type of value that can be edited. The value can be either numeric (Int) or text that can be edited (Edit).

Default value
The default value for the integer or text. <IHV defined> indicates that the value is associated with the particular independent hardware vendor (IHV) requirements.

Min
The minimum value that is allowed for an integer. <IHV defined> indicates that the minimum value is associated with the particular IHV requirements.

Max
The maximum value that is allowed for an integer. <IHV defined> indicates that the minimum value is associated with the particular IHV requirements.

The following table describes all of the RSS keywords that can be edited.

Expand table

SubkeyName	ParamDesc	Type	Default value	Min	Max
***RssBaseProcGroup**	RSS Base Processor Group	Int	0	0	MAXIMUM_GROUPS-1
***NumaNodeId**	Preferred NUMA node	Int	65535 (Any node)	0	System specific - cannot exceed 65535
***RssBaseProcNumber**	RSS Base Processor Number	Int	0	0	MAXIMUM_PROC_PER_GROUP-1
***MaxRssProcessors**	Maximum number of RSS Processors	Int	16	1	MAXIMUM_PROC_PER_SYSTEM
***RssMaxProcNumber**	Maximum RSS Processor Number	Int	MAXIMUM_PROC_PER_GROUP-1 (Default)	0	MAXIMUM_PROC_PER_GROUP-1
***NumRSSQueues**	Maximum Number of RSS Queues	Int	16	1	Device-specific
***RSSMaxProcGroup**	RSS Max Processor Group	Int	0	0	MAXIMUM_GROUPS-1

Note: Although the valid range for ***RssBaseProcGroup** is zero to MAXIMUM_GROUPS-1, in Windows 7 it must be zero. Otherwise, the TCP/IP protocol will not use any processors for RSS.

Note: The default value for ***NumaNodeId** (65535) means the network adapter is agnostic to NUMA node, and NDIS should not attempt to prefer any node over another. If the ***NumaNodeId** keyword is not present, then NDIS automatically selects the closest node based on hints from ACPI.

Note: The max value for ***MaxRssProcessors** may be set to the maximum number of processors that the NIC can support. NDIS will automatically cap this value to be the maximum number of processors on the current system.

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

Feedback

Indicating RSS Receive Data

Article • 12/15/2021

A miniport driver indicates received data by calling the **NdisMIndicateReceiveNetBufferLists** function from its *MiniportInterruptDPC* function.

After the NIC computes the RSS hash value successfully, the driver should store the hash type, hashing function, and hash value in the **NET_BUFFER_LIST** structure with the following macros:

NET_BUFFER_LIST_SET_HASH_TYPE

NET_BUFFER_LIST_SET_HASH_FUNCTION

NET_BUFFER_LIST_SET_HASH_VALUE

The hash type identifies the area of the received packet that the hash should be calculated over. For more information about the hash type, see RSS Hashing Types. The hashing function identifies the function that is used to calculate the hash value. For more information about hashing functions, see RSS Hashing Functions. The protocol driver selects the hash type and function at initialization. For more information, see RSS Configuration.

If the NIC fails to identify the area of the packet that the hash type specifies, then it should not do any hash computation or scaling. In this case, the miniport driver or NIC should assign the received data to the default CPU.

If the NIC runs out of receive buffers, each buffer must be returned as soon as the original receive DPC returns. The miniport driver can indicate the received data with a status of NDIS_STATUS_RESOURCES. In this case, the overlying driver has to go through a slow path of copying the buffer descriptors and relinquishing ownership of the original one immediately.

For more information about receiving network data, see Receiving Network Data.

Supporting RSS in Intermediate Drivers or Filter Drivers

Article • 12/15/2021

All intermediate drivers and filter drivers should, at a minimum, pass on OID requests, other requests, and status indications. Intermediate drivers or filter drivers should provide additional driver-specific support for receive side scaling (RSS) if the driver does any of the following:

- Originates send requests.
- Originates receive indications.
- Queues send requests or receive indications for later processing.

Filter drivers that bypass send and receive processing do not have to do anything additional to support RSS. For more information about bypassing send or receive requests in a filter driver, see Data Bypass Mode.

A QoS scheduler is an example of a filter driver that should support RSS. Such a driver queues send packets for sending at an appropriate time. The filter driver should use the same CPU that the protocol driver uses for a given connection.

An intermediate driver or filter driver that does not support RSS can intercept the RSS OID requests and disable RSS by reporting that RSS is not supported.

A filter driver or intermediate driver that supports RSS can use the information from the RSS OIDs to assign connections to the same CPUs the protocol driver and miniport driver are using.

For more information about the RSS OIDs, see RSS Configuration.

Overview of Virtual Machine Multiple Queues (VMMQ)

Article • 03/14/2023

Virtual Machine Multiple Queues (VMMQ) is a NIC offload technology that extends Native RSS (RSSv1) to a Hyper-V virtual environment.

VMMQ provides scalable network traffic processing for virtual ports (VPorts) in the parent partition of a virtualized node. A VPort represents an internal port on the NIC switch of a network adapter that supports single root I/O virtualization (SR-IOV). For an overview of the SR-IOV interface and its components see SR-IOV Architecture. Previously, RSS processing was not available for VPorts. VMMQ extends the native RSS feature to VPorts that are associated with the physical function (PF) of a NIC, including the default VPort.

VMMQ works by efficiently distributing network traffic within the NIC hardware. You can assign multiple hardware queues from the NIC to a single PF VPort. The NIC distributes network traffic across these queues using RSS hashing, placing packets directly onto the assigned processor. Offloading traffic distribution to the NIC improves CPU performance because the software doesn't have to complete this task.

You may want to enable the VMMQ feature to reduce the host CPU consumption and enable higher throughput to the virtual system by spreading the CPU load across multiple processors. You can add VMMQ support to new or existing NDIS 6.60 and later drivers. If an adapter supports VMMQ, the driver is vendor-supplied, and the OS is Windows Server 2019, then VMMQ is enabled by default. If the adapter doesn’t support VMMQ, the driver is system-supplied, or the OS is Windows Server 2016, then VMMQ is disabled by default or not available. If the OS is earlier than Windows Server 2016 then VMMQ is not available.

VMMQ is available for the VPorts exposed in the parent partition regardless of whether the NIC is operating in SR-IOV or Virtual Machine Queue (VMQ) mode.

Expected feature interactions

- Network Virtualization using Generic Routing Encapsulation (NVGRE) and Virtual Extensive Local Area Network (VXLAN): The NIC will calculate the hash for spreading receive queues based on the inner headers of the packets.
- SR-IOV: The NIC can support VMMQ and SR-IOV simultaneously.

In this section

VMMQ send and receive processing

Advertising VMMQ capabilities

Standardized INF keywords for VMMQ

Allocating VPorts for VMMQ

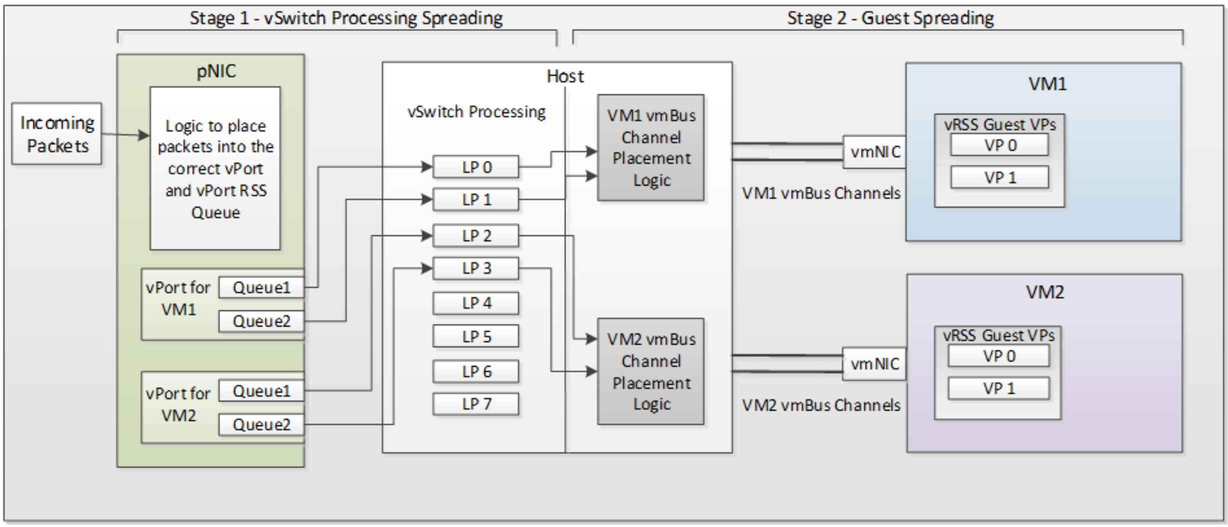
Enabling, disabling, and updating VMMQ on a VPort

VMMQ send and receive processing

Article • 03/14/2023

Virtual Machine Multiple Queues (VMMQ) efficiently distributes the network traffic for physical function virtual ports (PF VPorts) using RSS processing. For more information on the single root I/O virtualization (SR-IOV) interface and its components, see SR-IOV Architecture.

The following figure shows the network packet receive path within the VMMQ interface.



On the receive path, when a packet arrives at a NIC that supports VMMQ the NIC:

1. Matches the destination MAC address to find the target VPort.

2. Uses the RSS parameters of the VPort (the secret key, hash function, and hash type) to calculate the RSS hash value of the packet.

3. Uses the hash value to index the indirection table associated with the VPort. The values in the indirection table are used to assign the received data to a processor.

4. Interrupts the target processor and the received packet is indicated to the host network stack.

When indicating a received NBL, the miniport adapter sets the VPort ID and RSS related out-of-band (OOB) fields to the appropriate values.

On the transmit path, the NIC must use the RSS hash value in the packet (if present) as an index into the RSS indirection table for the VPort. The NIC uses this indirection table value to determine the processor that handles the transmit complete interrupts and DPCs for the packet.

If the NIC cannot calculate the RSS hash value of a received packet or the RSS hash value is not present in a transmit packet, it should use the default RSS processor of the VPort as the target RSS processor. The default RSS processor for a VPort will be specified in the RSS parameters for the VPort. For more information, see Enabling, disabling, and updating VMMQ on a VPort.

The host networking stack can update the RSS parameters of a VPort dynamically at runtime. The NIC should respond to the changes in the RSS parameters of a VPort with minimal interruption in traffic to and from the VPort.

Advertising VMMQ capabilities

Article • 03/14/2023

Miniport drivers register the Virtual Machine Multiple Queues (VMMQ) capability of a NIC during miniport adapter initialization.

> ⓘ **Note**
>
> If the NIC supports VMMQ, the default VPort and at least one non-default VPort must support VMMQ.

During initialization, the miniport driver must examine the ***RssOnHostVPorts** INF keyword in order to determine if it should enable the VMMQ feature on the NIC. For more information on handling RSS keywords for VMMQ, see Standardized INF keywords for VMMQ.

Additionally, the stack can only activate VMMQ on the NIC if the miniport adapter supports creating a NIC switch. NDIS can create a NIC switch on the miniport adapter when either the ***SriovPreferred** INF keyword is set to **one** or ***SriovPreferred** is set to **zero** and ***RssOrVmqPreference** is set to **one**. For more information, see Standardized INF Keywords for SR-IOV and Standardized INF Keywords for VMQ.

When the miniport driver configures the parameters for the NIC switch, it must set the fields of the **NDIS_NIC_SWITCH_PARAMETERS** structure as follows:

1. Set the **Revision** member of **Header** to **NDIS_NIC_SWITCH_PARAMETERS_REVISION_2**.

2. Set **NumQueuePairsForDefaultVPort** to the number of queue pairs assigned to a default VPort.

Miniport drivers advertise the NIC's VMMQ capability through the **NDIS_NIC_SWITCH_CAPABILITIES** structure. The miniport driver must initialize **NDIS_NIC_SWITCH_CAPABILITIES** as follows:

1. Set the **Revision** member of **Header** to **NDIS_NIC_SWITCH_CAPABILITIES_REVISION_3**.

2. Set the **NicSwitchCapabilities** flags as follows:

    - Set NDIS_NIC_SWITCH_CAPS_SINGLE_VPORT_POOL to **one** to indicate that non-default VPorts can be created on the PF. This flag must be set.

- Set NDIS_NIC_SWITCH_CAPS_ASYMMETRIC_QUEUE_PAIRS_FOR_NONDEFAULT_VPORT_SUPPORTED to indicate that NDIS can allocate an arbitrary number of VMMQ queues on each VPort. Otherwise, all non-default VPorts have the same maximum number of VMMQ queues as the **MaxNumQueuePairsPerNonDefaultVPort** field defines.

- Set NDIS_NIC_SWITCH_CAPS_RSS_ON_PF_VPORTS_SUPPORTED to **one** to indicate that the NIC supports VMMQ for PF VPorts.

> ⓘ **Note**
>
> If any of the following four per PF VPort flags are not set, higher level drivers will use the values that are specified when the RSS parameters of the PF VPorts are set (including the default VPort). For more information see **Enabling, disabling, and updating VMMQ on a VPort**.

- Set NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_INDIRECTION_TABLE_SUPPORTED to **one** to indicate that the NIC is able to maintain per PF VPort indirection tables. This flag must be set.

> ⓘ **Note**
>
> The following three flags NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_HASH_FUNCTION_SUPPORTED, NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_HASH_TYPE_SUPPORTED, and NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_HASH_KEY_SUPPORTED must all be set to **zero** or all be set to **one**. If they're all set to **zero**, software will re-calculate the hash.

- Set NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_HASH_FUNCTION_SUPPORTED to **one** if the NIC supports setting a different hash function per PF VPort.

- Set NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_HASH_TYPE_SUPPORTED to **one** if the NIC supports setting a different hash type per PF VPort.

- Set NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_HASH_KEY_SUPPORTED to **one** if the NIC supports setting a different hash secret key per PF VPort.

- Set NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_INDIRECTION_TABLE_SIZE_RESTRICTED to **one** if the NIC has a limitation on indirection table size for PF VPorts. This flag forces the issuer of an RSS OID to use a per-PF VPort indirection table size equal to the number of VPort queues rounded up to the next power of two. This flag can be combined with the NDIS_NIC_SWITCH_CAPS_ASYMMETRIC_QUEUE_PAIRS_FOR_NONDEFAULT_VPORT_SUPPORTED flag (different PF VPorts can have different numbers of queues). This flag prevents VMMQ users from performing fine-grained queue steering.

3. Set **MaxNumVPorts** to specify the maximum number of VPorts.

4. Set **MaxNumQueuePairs** to specify the maximum number of queue pairs that can be assigned to all VPorts. This includes the default VPort that is attached to the PF. This number should reflect the actual hardware capabilities.

5. Set **MaxNumQueuePairsPerNonDefaultVPort** to specify the maximum number of queue pairs that can be assigned to a non-default VPort.

6. Set **MaxNumRssCapableNonDefaultPFVPorts** to specify the maximum number of non-default PF VPorts that can support VMMQ.

7. Set **NumberOfIndirectionTableEntriesForDefaultVPort** to specify the number of indirection table entries for the default VPort.

8. Set **NumberOfIndirectionTableEntriesPerNonDefaultPFVPort** to specify the number of indirection table entries for each non-default PF VPort. The size of indirection table should be the same for all non-default PF VPorts.

9. Set **MaxNumQueuePairsForDefaultVPort** to specify the maximum number of queue pairs that can be assigned to a default VPort during NIC Switch creation.

After the VMMQ capabilities are advertised, NDIS is responsible for handling the OID_GEN_RECEIVE_SCALE_CAPABILITIES OID when it is called on either the default VPort or a non-default VPort. When the miniport driver returns the RSS capabilities in the **NDIS_RECEIVE_SCALE_CAPABILITIES** structure, it should not constrain the **NumberOfInterruptMessages** fields by any of the standard RSS keywords (such as ***MaxRssProcessors**). The upper level driver will incorporate this number into the host CPU allocation algorithm.

Standardized INF keywords for VMMQ

Article • 05/29/2024

The ***RssOnHostVPorts** standardized INF keyword is defined to enable or disable support for the network adapter Virtual Machine Multiple Queues (VMMQ) feature.

The ***RssOnHostVPorts** INF keyword is an enumeration keyword. Enumeration standardized INF keywords have the following attributes:

SubkeyName: The name of the keyword that you must specify in the INF file.

ParamDesc: The display text that is associated with the SubkeyName.

Value: The enumeration integer value that is associated with each SubkeyName in the list.

EnumDesc: The display text that is associated with each value that appears in the menu.

Default: The default value for the menu.

The following table describes the possible INF entries for the ***RssOnHostVPorts** INF keyword.

Expand table

SubkeyName	ParamDesc	Value	EnumDesc
*RssOnHostVPorts	Virtual Switch RSS	0	Disabled
		1 (Default)	Enabled

During miniport adapter initialization, the miniport driver must examine the ***RssOnHostVPorts** keyword to determine if it should enable the VMMQ feature on the NIC.

Handling RSS INF keywords for VMMQ

If a NIC supports VMMQ, all Standardized INF Keywords for RSS should also be supported to provide future compatibility even if the OS does not currently use them all. You should use the keywords as normal for RSS functionality except for:

- ***RSSProfile**: The “ClosestProcessor” profile should be supported and used as a policy for VMMQ.

- ***MaxRssProcessors**: When VMMQ is active, this keyword should not restrict the number of MSIx interrupt messages reported in NDIS_RECEIVE_SCALE_CAPABILITIES.

Allocating VPorts for VMMQ

Article • 03/14/2023

NDIS allocates VPorts when the Virtual Machine Multiple Queues (VMMQ) capability is present in the following way.

NDIS creates a non-default VPort on the miniport adapter by issuing the OID_NIC_SWITCH_CREATE_VPORT OID request. When creating an RSS physical function (PF) VPort, NDIS will initialize the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure as follows:

- NDIS sets the **AttachedFunctionId** field to **NDIS_PF_FUNCTION_ID**.
- If VMMQ is enabled, NDIS sets the **NumQueuePairs** field to the number of VMMQ queue pairs that should be used for this VPort. This number includes the default RSS processor for this VPort. It is guaranteed that total number of processors will not exceed this number. If VMMQ is disabled, NDIS sets this value to **one**.
- If VMMQ is enabled, the **ProcessorAffinity** field defines a bitmask of the potential RSS processors that the miniport adapter must use for this VPort. The processors that the network stack used to populate the indirection table entries for the VPort are a subset of the processors that this bitmask identifies. The mask will be a subset of the RSS processors returned from the call to **NdisGetRssProcessorInformation** and the number of set bits might exceed the number of RSS queues requested for the VPort. If VMMQ is disabled, the miniport adapter must use the lowest processor number specified in this bitmask when setting the affinity of the VPort queue.
- NDIS sets the NDIS_NIC_SWITCH_VPORT_PARAMS_NUM_QUEUE_PAIRS_CHANGED flag to indicate that the **NumQueuePairs** member has been updated after the VPort has been created. When VMMQ is enabled, the number of queues for default and non-default VPorts can be updated.

Enabling, disabling, and updating VMMQ on a VPort

Article • 03/14/2023

After creating a VPort, an upper layer driver can enable, disable, or update the RSS parameters of the VPort.

The driver can update the RSS indirection table of the VPort in order to change the number queues for a VPort. However the RSS hash type, hash function, and hash secret key of a VPort are considered static parameters and are not changed by the overlying drivers during the lifetime of a VPort. If an upper layer driver wishes to change any of the RSS static parameters, it must delete and recreate the VPort.

The upper layer driver enables, disables, or changes the RSS parameters of a VPort by issuing an OID_GEN_RECEIVE_SCALE_PARAMETERS OID request. The upper layer driver sets the **VPortId** field in the **NDIS_OID_REQUEST** structure to the ID of the target VPort of the new configuration.

The upper layer driver also sets the **NDIS_RECEIVE_SCALE_PARAMETERS** structure used in the OID request as follows. Please note that based on the VMMQ capabilities advertised by the underlying miniport adapter, some of the fields may be set to the same value for all PF VPorts.

- Set the **Revision** member of **Header** to **NDIS_RECEIVE_SCALE_PARAMETERS_REVISION_3**.
- Set the NDIS_RSS_PARAM_FLAG_DEFAULT_PROCESSOR_UNCHANGED flag to specify that the **DefaultProcessorNumber** member has not changed.
- Set **BaseCpuNumber** to **zero**.
- Set **DefaultProcessorNumber** to specify the default RSS processor for this VPort. The miniport can assume that default processor is part of RSS processor list, but it cannot assume that the default RSS processor is in the current indirection table.
- Set **HashInformation** to indicate the hash type and hash function that the NIC should use to calculate the hash value of the packets received for this VPort. The upper layer driver may set this field to a different value for each VPort.
- Set **IndirectionTableSize** to specify the size of the indirection table in bytes. Set this field to the same value for all PF VPorts. The upper layer driver must ensure that the number of entries in the indirection table is a power of two.

- Set **IndirectionTableOffset** to specify the offset of the indirection table from the beginning of the NDIS_RECEIVE_SCALE_PARAMETERS structure.

- Set **HashSecretKeySize** to specify the size of the hash secret key in bytes. The upper layer driver may set a different secret key for each VPort if the miniport adapter supports this. For more information, see Advertising VMMQ capabilities.

- Set **HashSecretKeyOffset** to specify the offset of the hash secret key from the beginning of the NDIS_RECEIVE_SCALE_PARAMETERS structure. The upper layer driver may set a different secret key for each VPort if the miniport adapter supports this. For more information, see Advertising VMMQ capabilities.

- Set **ProcessorMaskOffset**, **NumberOfProcessorMasks**, and **ProcessorMasksEntrySize** appropriately.

When a miniport driver receives an OID request to disable VMMQ for a VPort, it should revert to indicating all packets received for that VPort on the processor specified by the **ProcessorAffinity** field in the NDIS_NIC_SWITCH_VPORT_PARAMETERS structure that was used in the OID_NIC_SWITCH_CREATE_VPORT OID request.

Changing the number of queues for a VPort

The number of unique processors used in the indirection table of a VPort cannot exceed the value of the **NumQueuePairs** field of the NDIS_NIC_SWITCH_VPORT_PARAMETERS structure specified in the last issued OID_NIC_SWITCH_CREATE_VPORT OID request. These processors will be a subset of the RSS processor set returned by a call to NdisGetRssProcessorInformation. For more information, see Allocating VPorts for VMMQ. However, the indirection tables on different VPorts could contain the same processor.

To decrease the number of queues for a PF VPort an upper layer driver must:

1. Send an OID_GEN_RECEIVE_SCALE_PARAMETERS OID with the original indirection table size. However, the indirection table at this step can only reference the number of distinct processors up to the new number of queues. If the new indirection table needs to be smaller than the original table due to the NDIS_NIC_SWITCH_CAPS_RSS_PER_PF_VPORT_INDIRECTION_TABLE_SIZE_RESTRICTED flag of the NDIS_NIC_SWITCH_PARAMETERS structure, the issuer must guarantee that the indirection table at this step will contain the new indirection table replicated as many times as needed to satisfy the RESTRICTED flag requirement for the original number of queues.

2. Send an OID_NIC_SWITCH_VPORT_PARAMETERS OID with new number of queues.

3. Send an OID_GEN_RECEIVE_SCALE_PARAMETERS with the new indirection table size if it has changed.

To increase the number of queues for a PF VPort an upper layer driver must:

1. The driver doesn't need to update the current indirection table before step 2 because the table only references the number of distinct processors up to the current number of queues.

2. Send an OID_NIC_SWITCH_VPORT_PARAMETERS OID with new number of queues. If the RESTRICTED flag is set, the miniport driver should internally replicate the original indirection table as many times as needed to match the indirection table size requirement for the new number of queues.

3. Send an OID_GEN_RECEIVE_SCALE_PARAMETERS OID with new indirection table size if it has changed.

TCP/IP Offload Overview

Article • 12/15/2021

To increase its performance, the Microsoft TCP/IP transport can offload tasks or connections to a NIC that has the appropriate TCP/IP-offload capabilities.

Beginning with Windows Vista, the Windows operating system supports the following TCP/IP offload services:

- Checksum tasks
- Applications Internet protocol security (IPsec) offload version 1
- IPsec offload version 2
  - [The IPsec Task Offload feature is deprecated and should not be used.]
- Large send offload version 1
- Large send offload version 2
- Connection offload

Starting in Windows 10, version 2004, Windows also supports UDP Segmentation Offload (USO).

The TCP/IP transport that is provided beginning with Windows Vista supports TCP/IP offload services for both IPv4 and IPv6 packets.

NDIS 6.0 and later miniport drivers support TCP/IP offload services in a multiple-protocol driver environment. Multiple NDIS 6.0 and later protocol drivers that are bound to a TCP/IP offload-capable miniport adapter can configure TCP/IP offload services.

This section includes:

- Accessing TCP/IP Offload NET_BUFFER_LIST Information
- Using the TCP/IP Offload Administrator Interface
- Security Guidelines for Offload-Capable Miniport Drivers
- TCP/IP Task Offload
- Connection Offload

Accessing TCP/IP Offload NET_BUFFER_LIST Information

Article • 12/15/2021

NDIS versions 6.0 and later provide TCP/IP offload out-of-band (OOB) data in the **NetBufferListInfo** member of the NET_BUFFER_LIST structure, which specifies a linked list of NET_BUFFER structures. The **NetBufferListInfo** member is an array of values that contain information that is common to all of the NET_BUFFER structures in the list.

Use the following identifiers with the NET_BUFFER_LIST_INFO macro to set and get the TCP/IP offload OOB data in the **NetBufferListInfo** array:

TcpIpChecksumNetBufferListInfo
Specifies checksum information that is used in offloading checksum tasks from the TCP/IP protocol to a miniport driver. When you specify **TcpIpChecksumNetBufferListInfo**, NET_BUFFER_LIST_INFO returns an NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO structure (not a pointer to the structure). This structure contains a union that enables the checksum information to be accessed as a single PVOID value or as bit fields.

IPsecOffloadV1NetBufferListInfo
Specifies Internet protocol security (IPsec) offload information that is used in offloading IPsec tasks from the TCP/IP protocol to a miniport driver. When you specify **IPsecOffloadV1NetBufferListInfo**, NET_BUFFER_LIST_INFO returns an NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO structure.

TcpLargeSendNetBufferListInfo
Specifies information that is used in offloading the segmentation of a large TCP packet from the TCP/IP protocol to a miniport driver. When you specify **TcpLargeSendNetBufferListInfo**, NET_BUFFER_LIST_INFO returns an NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO structure (not a pointer to the structure). This structure contains a union that enables the information to be accessed as a single PVOID value or as bit fields.

Ieee8021QNetBufferListInfo
Specifies 802.1Q information about a packet. When you specify **Ieee8021QNetBufferListInfo**, NET_BUFFER_LIST_INFO returns the **Value** member of an NDIS_NET_BUFFER_LIST_8021Q_INFO structure. This structure can specify 802.1p priority and virtual LAN (VLAN) identifier information. 802.1p priority information is used to establish packet priority in shared-media 802 networks.

If a miniport driver reports support for the NDIS_ENCAPSULATION_IEEE_802_3_P_AND_Q_IN_OOB encapsulation, it must insert the **Ieee8021QNetBufferListInfo** data into large send offload version 1 (LSOV1) and large send offload version 2 (LSOV2) Ethernet packets.

TcpOffloadBytesTransferred
Specifies the number of data bytes that were transferred in a TCP chimney offload send, receive, or disconnect operation.

TcpReceiveNoPush
Specifies a Boolean value that represents the push mode of a TCP chimney offload receive request. If **TRUE**, the receive request is in non-push mode. Otherwise, the receive request is in push mode.

For LSOV1, LSOV2, checksum, and IPsec offload types, a miniport driver performs task offload based on the type of OOB data and the offload capabilities that it reported. For example, if a protocol driver requires LSOV1 services for an IPv4 packet, each send request that the protocol driver provides includes the information from the **LsoV1Transmit** member in the NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO OOB data. Note that the protocol driver must verify that the miniport driver supports IPv4, with the specified encapsulation type, before making the send request.

The NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO structure contains the maximum segment size (MSS). The **TcpHeaderOffset** member specifies the location of the TCP header so that the miniport driver does not have to parse IP headers, IP options, or IP extension headers.

An NDIS 6.0 and later miniport driver that supports LSOV2 and LSOV1 must check the **Type** member of NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO to determine whether the driver stack is using LSOV2 or LSOV1 and must perform the appropriate offload.

For LSOv1, before a miniport driver completes the send of a large TCP packet that it has segmented into smaller packets by using LSO, the driver writes the number of TCP payload bytes that it sent in the segmented packets in the **TcpPayload** member of NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO.

If a miniport driver specifies the NDIS_ENCAPSULATION_IEEE_802_3_P_AND_Q flag in its capabilities, the driver can perform task offload services for NET_BUFFER_LIST structures that contain the VLAN header in the buffer data. In the case of received data, this flag indicates that the miniport driver will perform the receive checksum calculation and put the VLAN header in the Ethernet packet.

If a miniport driver specifies the NDIS_ENCAPSULATION_IEEE_802_3_P_AND_Q_IN_OOB flag in its capabilities, the driver can perform offload on NET_BUFFER_LIST structures that contain the VLAN header in the **Ieee8021QnetBufferListInfo** OOB data. In the receive checksum offload case, the miniport inserts the VLAN header into the **Ieee8021QnetBufferListInfo** OOB data.

Using the TCP/IP Offload Administrator Interface

Article • 12/15/2021

In NDIS 6.0 and later versions, user-mode applications (or overlying drivers) can enable or disable TCP/IP offload capabilities. A system administrator can access the settings through the Microsoft Windows Management Instrumentation (WMI) interface. There might also be capabilities that are disabled through registry settings that can be enabled if they are supported in the hardware.

In response to an OID_TCP_OFFLOAD_PARAMETERS object identifier (OID) set request, a miniport driver uses the settings in the **NDIS_OFFLOAD_PARAMETERS** structure to set the current offload or connection offload configuration of the miniport adapter.

NDIS retains the requested settings in the registry in the offload standardized keywords. When NDIS restarts the miniport adapter, the miniport driver reads the offload standardized keywords and uses them to set the default offload configuration of the NIC. If the miniport driver also supports non-standard keywords, the miniport driver is responsible for updating the registry when it changes the task offload settings. For more information about the standardized keywords, see Standardized INF Keywords for Network Devices.

The miniport drivers must use the contents of the **NDIS_OFFLOAD_PARAMETERS** structure to update the currently reported offload configuration. The miniport driver must report the current configuration with the task offload **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** or connection offload NDIS_STATUS_OFFLOAD_RESUME status indication. (For information on NDIS_STATUS_OFFLOAD_RESUME, see NDIS 6.0 TCP chimney offload documentation.) The status indication ensures that all of the overlying protocol drivers are updated with the new capabilities information.

Before user-mode applications (or overlying drivers) set OID_TCP_OFFLOAD_PARAMETERS they can use the OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES OID or OID_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES OID to determine what capabilities a miniport adapter's hardware can support. Use the OID_TCP_OFFLOAD_PARAMETERS OID to enable capabilities that the OID_TCP_OFFLOAD_CURRENT_CONFIG OID or OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG OID reports as not currently enabled.

If the hardware capabilities change (for example, because a MUX intermediate driver switches to a difference underlying miniport adapter), the intermediate driver must report any changes in offload hardware capabilities with the **NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES** or **NDIS_STATUS_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES** status indication.

NDIS and overlying drivers can use the OID_OFFLOAD_ENCAPSULATION OID to set or query the task offload encapsulation settings of an underlying miniport adapter. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains an **NDIS_OFFLOAD_ENCAPSULATION** structure.

Security Guidelines for Offload-Capable Miniport Drivers Overview

Article • 12/15/2021

To increase its performance, the Microsoft TCP/IP transport can offload tasks or connections to a network interface card (NIC) that has the appropriate TCP/IP-offload capabilities. Offloaded TCP/IP network communication tasks are handled in the NIC hardware. Miniport drivers advertise the various offload capabilities of the NIC hardware to the operating system and confugure the NIC hardware. The NIC hardware performs the advertised offload tasks on outgoing and incoming packets in the send and receive dispatch handlers. The hardware performs operations such as computing IP header checksum and so on.

To ensure a secure environment, the miniport driver should advertise only those offload capabilities that the NIC hardware can provide and no others. The miniport driver should configure the hardware to offload the advertised tasks on the packets that meet the advertised criteria. On the send path, the operating system does not require a driver to offload a task that the miniport driver did not advertise. On the receive path, the miniport driver and NIC should not perform any tasks that are not included in the capabilities of the NIC hardware that the miniport driver advertised.

If a miniport driver or NIC cannot perform an offload task on a received packet, the miniport driver should indicate such a packet up the driver stack without taking any action. In this case, the overlying drivers handle the packet as a normal packet.

The miniport driver should never advertise capabilities that the NIC hardware does not support. The miniport driver should never use the send or receive dispatch handlers to perform software emulation of the offload operations that the hardware cannot provide. If the miniport driver provides such software emulation, the driver must inspect the packet data in software. If the driver inspects the packet data in software, the computer might be exposed to security attacks.

The following topics provide more information about security attacks and how to avoid security problems in NDIS drivers:

Vulnerability to Security Attacks in NDIS Drivers

Performance Degradation and Denial of Service Attacks in NDIS Drivers

Added Costs for Testing Vulnerable NDIS Drivers

Security Checklist for NDIS Drivers

Vulnerability to Security Attacks in NDIS Drivers

Article • 03/14/2023

If an NDIS driver parses and interprets packet data, the driver and the operating system might be vulnerable to security attacks. Some of these attacks could be started remotely and cause serious problems, including crashing the computer.

For example, consider a network interface card (NIC) that can support IPv4 checksum offload but cannot support IPv6 checksum offload. However, the miniport driver advertises that the NIC can provide IPv6 checksum offload. When the NIC receives IPv4 packets, the NIC hardware computes and verifies the checksum and puts the results in the **NET_BUFFER_LIST** structure out-of-band (OOB) information. If the NIC receives an IPv6 packet with IPv6 extension, the miniport driver computes and verifies the checksum in the receive interrupt handler. In the IPv6 case, it appears to the operating system that the NIC is performing IPv6 offload. However, the interrupt handler would have to parse the received packet and would have to check for error conditions and guard against bad information in the IP header fields in a manner at least as good as the hardware. Such a software implementation must be very robust or it could crash the computer.

Performance Degradation and Denial of Service Attacks in NDIS Drivers

Article • 03/14/2023

If an NDIS driver interrupt handler parses received packets, the interrupt handler implementation might lead to performance degradation and denial of service attacks. For example, a malicious user can target the computer by sending many packets so that the miniport driver is busy computing the checksum on bad packets in the interrupt handler.

Even if you are careful in how your driver handles received packets, the driver would perform receive operations at dispatch IRQL. Instead, you should let the driver stack handle the received packets. In this case, the overlying driver stack might copy the packet and operate on it later at passive IRQL.

Added Costs for Testing Vulnerable NDIS Drivers

Article • 03/14/2023

We recommend that you remove any code that parses the packet payload, particularly in handling offload verification, from your driver's packet handling dispatch routines. To have confidence in such code, you would have to extensively test the drivers to make sure that all potential error conditions are handled safely and correctly. This kind of testing means increased testing costs.

Miniport drivers should avoid parsing the packet data. They should not try to handle offload operations that the hardware cannot handle. In the receive side of the system, be very careful about how your driver inspects packet payload information. The send side of the driver could also be potentially affected with routed/bridged system configurations.

Security Checklist for NDIS Drivers

Article • 03/14/2023

To make sure that your driver follows good security practices, do the following:

- If it is possible, avoid code that parses the packet payload information for any reason. We recommend that you remove any such code, particularly in handling offload verification, from your driver's packet handling dispatch routines.

- Check your driver's send and receive code paths and carefully verify any code that parses the packet payload information for any reason.

- Thoroughly review the driver code for security holes and test your driver before you release the driver. Make sure that you verify all error paths as well as the normal code paths.

- Run random packet generation tests to make sure that your drivers can resist bad packet information. In the future, such tests will be mandatory for device logo certification.

TCP/IP Task Offload Overview

Article • 09/27/2024

To increase its performance, the Microsoft TCP/IP transport can offload tasks to a network interface card (NIC) that has the appropriate task offload capabilities.

Beginning with Windows Vista, the Windows operating system supports the following task offload services:

Checksum tasks

The TCP/IP transport can offload the calculation and validation of IP and TCP checksums.

Large send offload version 1 (LSOV1)

The TCP/IP transport supports large send offload version 1 (LSOV1). With LSOV1, the TCP/IP transport can offload the segmentation of large (up to 64 KB including the IP header) TCP packets for IPv4.

Large send offload version 2 (LSOV2)

The large send offload version 2 (LSOV2) interface is an enhanced version of LSOV1. LSOV2 supports IPv6, IPv4, and segmentation for large TCP packets that are larger than 64K. For more information about offloading the segmentation of large packets, see Offloading the Segmentation of Large TCP Packets.

Beginning with Windows 8 and Windows Server 2012, the Windows operating system supports the following additional task overload services:

Receive Segment Coalescing (RSC)

Receive segment coalescing (RSC) enables network card miniport drivers to coalesce multiple TCP segments and indicate them as a single coalesced unit (SCU) to the operating system's networking subsystem.

Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload

Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload makes it possible to use Generic Routing Encapsulation (GRE)-encapsulated packets with:

- Large Send Offload (LSO)
- Receive Side Scaling (RSS)
- Virtual Machine Queue (VMQ)

UDP Segmentation Offload (USO)

Beginning with Windows 10, version 2004, Windows supports UDP Segmentation Offload (USO). USO enables network cards to offload the segmentation of UDP datagrams that are larger than the maximum transmission unit (MTU) size of the network medium.

This section includes:

- Determining Task Offload Capabilities
- Enabling and Disabling Task Offload Services
- Determining the Current Task Offload Settings
- Combining Types of Task Offloads
- Using Registry Values to Enable and Disable Task Offloading
- Offloading Checksum Tasks
- Offloading the Segmentation of Large TCP Packets
- UDP Segmentation Offload (USO)

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Determining Task Offload Capabilities

Article • 12/15/2021

NDIS supports task offload services that are enhanced forms of the NDIS 5.1 and earlier task offload services. For more information about how to determine connection offload capabilities, see Determining Connection Offload Capabilities.

NDIS provides the offload hardware capabilities and the current configuration of the underlying miniport adapter to protocol drivers in the **NDIS_BIND_PARAMETERS** structure. NDIS provides the task offload hardware capabilities and current configuration of the underlying miniport adapter to filter drivers in the **NDIS_FILTER_ATTACH_PARAMETERS** structure.

Administrative applications use object identifier (OID) queries to obtain task offload capabilities of a miniport adapter. However, overlying drivers should avoid using OID queries. Protocol drivers must handle changes in the task offload capabilities that underlying drivers report. Miniport drivers can report changes in task offload capabilities in status indications. For a list of status indications, see NDIS 6.0 TCP/IP Offload Status Indications.

Administrative applications (or overlying drivers) can determine the current task offload configuration of a network interface card (NIC) by querying the OID_TCP_OFFLOAD_CURRENT_CONFIG OID.

The **NDIS_OFFLOAD** structure that is associated with OID_TCP_OFFLOAD_CURRENT_CONFIG specifies the following:

- The header information, which includes the task offload version that the TCP/IP transport supports.
- The checksum offload information, in an **NDIS_TCP_IP_CHECKSUM_OFFLOAD** structure.
- The large send offload version 1 (LSOV1) information, in an **NDIS_TCP_LARGE_SEND_OFFLOAD_V1** structure.
- The Internet protocol security (IPsec) information, in an **NDIS_IPSEC_OFFLOAD_V1** structure.
- The large send offload version 2 (LSOV2) information, in an **NDIS_TCP_LARGE_SEND_OFFLOAD_V2** structure.

- The Internet protocol security (IPsecvOV) information in an **NDIS_IPSEC_OFFLOAD_V2** structure.

The following topics contain specific information for each type of offload service:

- Reporting a NIC's Checksum Capabilities
- Reporting a NIC's LSOV1 TCP-Packet-Segmentation Capabilities
- Reporting a NIC's LSOV2 TCP-Packet-Segmentation Capabilities
- Reporting a NIC's IPsec Capabilities
  - [The IPsec Task Offload feature is deprecated and should not be used.]

Reporting a NIC's Checksum Capabilities

Article • 12/15/2021

An NDIS miniport driver reports whether a NIC is currently configured to calculate and validate IP, TCP, and UDP checksums in an **NDIS_TCP_IP_CHECKSUM_OFFLOAD** structure. Miniport drivers must include the current checksum offload configuration in the **NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES** structure. Miniport drivers call the **NdisMSetMiniportAttributes** function from the *MiniportInitializeEx* function and pass in the information in NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES.

Miniport drivers must report changes in the current checksum offload configuration, if any, in the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication.

In response to a query of OID_TCP_OFFLOAD_CURRENT_CONFIG, NDIS includes the NDIS_TCP_IP_CHECKSUM_OFFLOAD structure in the **NDIS_OFFLOAD** structure that NDIS returns in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. NDIS uses the information that the miniport driver provided.

A miniport driver indicates the following checksum information for IPv4 and IPv6 send and receive packets:

- The types of checksums (IP, TCP, or UDP) that a NIC can calculate for send packets and can validate for receive packets.

- Encapsulation settings, in the **Encapsulation** member. For more information about this member, see the Remarks section in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**.

- Whether the NIC can calculate or validate (or calculate and validate) checksums for a packet whose IP headers contain IPv4 options.

- Whether the NIC can calculate or validate (or calculate and validate) checksums for an IPv6 packet whose IP headers contain IPv6 extension headers.

- Whether the NIC can calculate or validate (or calculate and validate) checksums for a packet whose TCP header contain TCP options.

Related topics

Determining Task Offload Capabilities

Reporting a NIC's LSOV1 TCP-Packet-Segmentation Capabilities

Article • 12/15/2021

An NDIS miniport driver specifies the current large send offload version 1 (LSOV1)-TCP-packet-segmentation configuration of a NIC in an **NDIS_TCP_LARGE_SEND_OFFLOAD_V1** structure.Miniport drivers must include the current LSOV1 offload configuration in the **NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES** structure. Miniport drivers call the **NdisMSetMiniportAttributes** function from the *MiniportInitializeEx* function and pass in the information in NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES.

Miniport drivers must report changes in the LSOV1 configuration, if any, in the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication.

In response to a query of OID_TCP_OFFLOAD_CURRENT_CONFIG, NDIS includes the NDIS_TCP_LARGE_SEND_OFFLOAD_V1 structure in the **NDIS_OFFLOAD** structure that NDIS returns in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. NDIS uses the information that the miniport driver provided.

NDIS supports large send offload version 2 (LSOV2), which is an enhanced version of LSO. For more information about LSOV2 capabilities, see Reporting a NIC's LSOV2 TCP-Packet-Segmentation Capabilities.

The miniport driver must specify the following information in the NDIS_TCP_LARGE_SEND_OFFLOAD_V1 structure:

- Encapsulation settings, in the **Encapsulation** member. For more information about this member, see the Remarks section in **NDIS_TCP_LARGE_SEND_OFFLOAD_V1**.

- The maximum bytes of user data that the TCP/IP transport can pass to the miniport driver in a large TCP packet, in the **MaxOffLoadSize** member. The maximum size cannot exceed 64K bytes.

- The minimum number of segments that a large TCP packet must be divisible by before the TCP/IP transport can offload it to a NIC for segmentation, in the **MinSegmentCount** member.

- Whether a NIC can segment a large TCP packet that contains TCP options.

- Whether a NIC can segment a large TCP packet that contains IPv4 options.

Related topics

Determining Task Offload Capabilities

Reporting a NIC's LSOV2 TCP-Packet-Segmentation Capabilities

Article • 12/15/2021

An NDIS miniport driver specifies the current large send offload version 2 (LSOV2) TCP-packet-segmentation configuration of a NIC in an **NDIS_TCP_LARGE_SEND_OFFLOAD_V2** structure.Miniport drivers must include the current LSOV2 configuration in the **NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES** structure. Miniport drivers call the **NdisMSetMiniportAttributes** function from the *MiniportInitializeEx* function and pass in the information in NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES.

Miniport drivers must report changes in the LSOV2 configuration, if any, in the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication.

In response to a query of OID_TCP_OFFLOAD_CURRENT_CONFIG, NDIS includes the NDIS_TCP_LARGE_SEND_OFFLOAD_V2 structure in the **NDIS_OFFLOAD** structure that NDIS returns in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. NDIS uses the information that the miniport driver provided.

We recommend that a miniport driver that supports LSOV2 hardware should also support LSOV1. This support will enable the TCP/IP transport to use LSOV1 if an NDIS 5.*x* intermediate driver is installed over a miniport adapter. For more information about LSOV1 capabilities, see Reporting a NIC's LSOV1 TCP-Packet-Segmentation Capabilities.

LSOV2 supports IPv4 and IPv6 packets. The miniport driver must specify the following information for both IPv4 and IPv6 in the **NDIS_TCP_LARGE_SEND_OFFLOAD_V2** structure:

- Encapsulation settings, in the **Encapsulation** member. For more information about this member, see the Remarks section in **NDIS_TCP_LARGE_SEND_OFFLOAD_V2**.

- The maximum bytes of user data that the TCP/IP transport can pass to the miniport driver in a large TCP packet, in the **MaxOffLoadSize** member.

- The minimum number of segments that a large TCP packet must be divisible by before the TCP/IP transport can offload it to a NIC for segmentation, in the **MinSegmentCount** member.

Related topics

Determining Task Offload Capabilities

Reporting a NIC's IPsec Capabilities

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

An NDIS miniport driver specifies the current Internet protocol security (IPsec) offload configuration of a NIC in an **NDIS_IPSEC_OFFLOAD_V1** structure.Miniport drivers must include the current IPsec offload configuration in the **NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES** structure. Miniport drivers call the **NdisMSetMiniportAttributes** function from the *MiniportInitializeEx* function and pass in the information in NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES.

Miniport drivers must report changes in the IPsec offload capabilities, if any, in the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication.

In response to a query of OID_TCP_OFFLOAD_CURRENT_CONFIG, NDIS includes the NDIS_IPSEC_OFFLOAD_V1 structure in the **NDIS_OFFLOAD** structure that NDIS returns in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. NDIS uses the information that the miniport driver provided.

A miniport driver indicates the following information in the NDIS_IPSEC_OFFLOAD_V1 structure:

- Encapsulation settings, in the **Encapsulation** member. For more information about this member, see the Remarks section in **NDIS_IPSEC_OFFLOAD_V1**.

- Whether a NIC can perform combined IPsec operations on a packet--that is, whether the NIC can process a packet that contains both an authentication header (AH) and an encapsulating security payload (ESP) in a packet with the following format:

  [IP][AH][ESP][rest of packet]

- Whether a NIC can perform IP security processing on both the transport-mode portion and the tunnel-mode portion of send and receive packets. The transport-mode portion of a packet pertains to an end-to-end security association, and the tunnel-mode portion of a packet pertains to a tunnel security association.

- Whether a NIC can perform IP security operations on packets if the packet's IP headers contain IP options.

A miniport driver specifies the following capabilities of a NIC to calculate or validate (or calculate and validate) encrypted checksums for AH payloads and authentication information:

- The integrity algorithms (MD5 or SHA 1) that the NIC can use

- Whether the NIC can process AH security payloads for:
  - The transport-mode portion of a packet
  - The tunnel-mode portion of a packet
  - Send packets
  - Receive packets

A miniport driver specifies the following capabilities of a NIC to process ESP payloads:

- The confidentiality algorithms (DES, triple DES, or both) that the NIC can use

- Whether the NIC supports null encryption (that is, the ESP payload without encryption but with authentication hashes)

- Whether the NIC can do ESP processing for:
  - The transport-mode portion of a packet
  - The tunnel-mode portion of a packet
  - Send packets
  - Receive packets

Related topics

Determining Task Offload Capabilities

Enabling and Disabling Task Offload Services

Article • 12/15/2021

A protocol driver can enable or disable task offload services for an underlying miniport adapter by issuing an OID_OFFLOAD_ENCAPSULATION OID set request. This OID request sets the required encapsulation type and tells the miniport driver to activate all of the available task offload services.

Before issuing the OID_OFFLOAD_ENCAPSULATION OID set request, the protocol driver should make sure that the underlying miniport adapter supports the required encapsulation type. There are two ways to do this:

- Check the **NDIS_BIND_PARAMETERS** structure that the protocol driver received in its *ProtocolBindAdapterEx* function.
- Issue an OID_TCP_OFFLOAD_CURRENT_CONFIG query request.

If the miniport driver supports any task offload type that supports the requested encapsulation type, the miniport driver must return NDIS_STATUS_SUCCESS in response to the OID_OFFLOAD_ENCAPSULATION set request. Otherwise, the miniport driver should return NDIS_STATUS_INVALID_PARAMETER.

Determining the Current Task Offload Settings

Article • 12/15/2021

A protocol driver can determine the current task offload encapsulation settings of an underlying miniport adapter by issuing an OID_OFFLOAD_ENCAPSULATION OID query request.

For more information about issuing an OID request, see Generating OID Requests from an NDIS Protocol Driver.

Combining Types of Task Offloads

Article • 12/15/2021

The following restrictions determine which combinations of NDIS 6.0 and later task offload services can be active on the system:

- A task offload-capable miniport adapter can support checksum offload alone.
- A large send offload version 1 (LSOV1)-capable network interface card (NIC) must support V4 TCP and IPv4 checksum transmit offload services. If an LSOV1-capable miniport adapter also supports Internet protocol security (IPsec) offload, NDIS will configure the adapter to offload either IPsec or LSOV1, but not both.
- A large send offload version 2 (LSOV2)-capable miniport adapter must support TCP and IP Checksum transmit offload. If a LSOV2-capable miniport adapter also supports IPsec offload, NDIS will configure it to offload IPsec or LSOV2, but not both.

Miniport drivers are not required to support both IPv4 and IPv6. All NDIS 6.0 and later miniport drivers must support Ethernet 802.3 encapsulations with the ability to support Ethernet 802.1Q tags. The following table describes the hardware requirements when the miniport driver reports support for various offload capabilities.

Type of offload	IPv4	IpV6
Checksum Offload		
UDP Checksum	Optional	Optional
TCP Checksum	Optional	Optional
TCP Options	Optional	Required (for TCP Checksum)
IP Checksum	Optional	Not Applicable
IP Options	Required (for TCP checksum)	Not Applicable
IP Extension Header	Not Applicable	Required (128 bytes)
Large Send Offload version 1 (LSOv1)		
Max Offload Size	<= 64K	Not Applicable
TCP Options	Required	Not Applicable
IP Options	Required	Not Applicable

Type of offload	IPv4	IpV6
Large Send Offload version 2 (LSOv2)		
Max Offload Size	Unlimited	Unlimited
IP Options	Required	Required (128 bytes)
IP ID Support	0x0000 to 0xffff	0x0000 to 0x7fff reserved for segmentation offload

Using Registry Values to Enable and Disable Task Offloading

Article • 09/27/2024

When you debug a driver's task offload functionality, you might find it useful to enable or disable task offload services with a registry key setting. There are standardized keywords that you can define in INF files and in the registry. For more information about standardized keywords, see Standardized INF Keywords for Network Devices.

Task offload keywords belong to one of two groups: granular keywords or grouped keywords. *Granular keywords* provide keywords per offload capability--Transport Layer differentiation, IP protocol differentiation. *Grouped keywords* provide combined keywords capability at the transport layer.

Granular keywords

The granular keywords are defined as follows:

Expand table

Keyword	Description
***IPChecksumOffloadIPv4**	Describes whether the device enabled or disabled the calculation of IPv4 checksums.
***TCPChecksumOffloadIPv4**	Describes whether the device enabled or disabled the calculation of TCP Checksum over IPv4 packets.
***TCPChecksumOffloadIPv6**	Describes whether the device enabled or disabled the calculation of TCP checksum over IPv6 packets.
***UDPChecksumOffloadIPv4**	Describes whether the device enabled or disabled the calculation of UDP Checksum over IPv4 packets.
***UDPChecksumOffloadIPv6**	Describes whether the device enabled or disabled the calculation of UDP Checksum over IPv6 packets.
***LsoV1IPv4**	Describes whether the device enabled or disabled the segmentation of large TCP packets over IPv4 for large send offload version 1 (LSOv1).
***LsoV2IPv4**	Describes whether the device enabled or disabled the segmentation of large TCP packets over IPv4 for large send offload version 2 (LSOv2).

Keyword	Description
***LsoV2IPv6**	Describes whether the device enabled or disabled the segmentation of large TCP packets over IPv6 for large send offload version 2 (LSOv2).
***IPsecOffloadV1IPv4**	Describes whether the device enabled or disabled the calculation of IPsec headers over IPv4.
***IPsecOffloadV2**	Describes whether the device enabled or disabled IPsec offload version 2 (IPsecOV2). IPsecOV2 provides support for additional crypto-algorithms, IPv6, and co-existence with large send offload version 2 (LSOv2).
***IPsecOffloadV2IPv4**	Describes whether the device enabled or disabled IPsecOV2 for IPv4 only.

The following table describes the granular keywords that you can use to configure offload services.

Expand table

SubkeyName	ParamDesc	Value	EnumDesc
IPChecksumOffloadIPv4	IPv4 Checksum Offload	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
TCPChecksumOffloadIPv4	TCP Checksum Offload (IPv4)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
TCPChecksumOffloadIPv6	TCP Checksum Offload (IPv6)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled

SubkeyName	ParamDesc	Value	EnumDesc
UDPChecksumOffloadIPv4	UDP Checksum Offload (IPv4)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
UDPChecksumOffloadIPv6	UDP Checksum Offload (IPv6)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Rx & Tx Enabled
LsoV1IPv4	Large Send Offload Version 1 (IPv4)	0	Disabled
		1 (Default)	Enabled
LsoV2IPv4	Large Send Offload V2 (IPv4)	0	Disabled
		1 (Default)	Enabled
LsoV2IPv6	Large Send Offload V2 (IPv6)	0	Disabled
		1 (Default)	Enabled
IPsecOffloadV1IPv4	IPsec Offload Version 1 (IPv4)	0	Disabled
		1	Auth Header Enabled
		2	ESP Enabled
		3 (Default)	Auth Header & ESP Enabled
IPsecOffloadV2	IPsec Offload	0	Disabled
		1	Auth Header Enabled
		2	ESP Enabled

SubkeyName	ParamDesc	Value	EnumDesc
		3 (Default)	Auth Header & ESP Enabled
*IPsecOffloadV2IPv4	IPsec Offload (IPv4 only)	0	Disabled
		1	Auth Header Enabled
		2	ESP Enabled
		3 (Default)	Auth Header & ESP Enabled

ⓘ Note

The INF file can support granular keywords that are displayed in the Advanced Property page of the UI. The miniport driver must read all of the granular settings from the registry at initialization, including settings that are not displayed, to register NDIS offload capabilities.

Grouped keywords

The grouped keywords are defined as follows:

Expand table

Keyword	Description
***TCPUDPChecksumOffloadIPv4**	Describes whether the device enabled or disabled the calculation of IP, TCP, and UDP checksum over IPv4.
***TCPUDPChecksumOffloadIPv6**	Describes whether the device enabled or disabled the calculation of TCP and UDP checksum over IPv6.

The following table describes the grouped keywords that you can use to configure offload services.

Expand table

SubkeyName	ParamDesc	Value	EnumDesc
TCPUDPChecksumOffloadIPv4	TCP/UDP Checksum Offload (IPv4)	0	Disabled

SubkeyName	ParamDesc	Value	EnumDesc
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Tx & Rx Enabled
TCPUDPChecksumOffloadIPv6	TCP/UDP Checksum Offload (IPv6)	0	Disabled
		1	Tx Enabled
		2	Rx Enabled
		3 (Default)	Tx & Rx Enabled

There are restrictions on the combinations of offloads that can be enabled. For example, if a miniport adapter supports LSOV1 or LSOV2, the miniport adapter also calculates the IP and TCP checksums. For more information about valid combinations of offloads, see Combining Types of Task Offloads.

If task offload services are disabled with a registry key setting, protocol drivers must not issue the OID_OFFLOAD_ENCAPSULATION object identifier (OID).

You can use the following registry values to enable or disable task offloading for the TCP/IP protocol:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TCPIP\Parameters\DisableTaskOffload
Setting this value to one disables all of the task offloads from the TCP/IP transport. Setting this value to zero enables all of the task offloads.

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Ipsec\EnabledOffload
Setting this value to zero disables Internet protocol security (IPsec) offloads from the TCP/IP transport. The offloading of TCP/IP checksum tasks, large send offload version 1 (LSOV1), and large send offload version 2 (LSOV2) are not affected. Setting this value to one enables IPsec offloads.

Feedback

Was this page helpful? Yes No

Provide product feedback ↗ | Get help at Microsoft Q&A

Offloading Checksum Tasks

Article • 12/15/2021

NDIS supports offloading TCP/IP checksum tasks at run time.

> ⓘ **Note**
>
> Checksum offload out-of-band (OOB) data is stored in the **NET_BUFFER_LIST** information array. For more information about OOB data, see **Accessing TCP/IP Offload NET_BUFFER_LIST Information**.

Before passing to the miniport driver a NET_BUFFER_LIST structure for a packet on which the miniport driver will perform checksum tasks, the TCP/IP transport specifies the checksum information that is associated with the NET_BUFFER_LIST structure. This information is specified by an **NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO** structure, which is part of the NET_BUFFER_LIST information (out-of-band data) that is associated with the NET_BUFFER_LIST structure.

Before offloading the checksum calculation for a TCP packet, the TCP/IP transport calculates the one's complement sum for the TCP pseudoheader. The TCP/IP transport calculates the one's complement sum across all fields in the pseudoheader, including Source IP Address, Destination IP Address, Protocol, and the TCP length for TCP packets. The TCP/IP transport enters the one's complement sum for the pseudoheader in the Checksum field of the TCP header.

The one's complement sum for the pseudoheader provided by the TCP/IP transport gives the NIC an early start in calculating the real TCP checksum for the send packet. To calculate the actual TCP checksum, the NIC calculates the variable part of the TCP checksum (for the TCP header and payload), adds this checksum to the one's complement sum for the pseudoheader calculated by the TCP/IP transport, and calculates the 16-bit one's complement for the checksum. For more information about calculating such checksums, see RFC 793 and RFC 1122.

> ⓘ **Note**
>
> The TCP/IP transport computes the one's complement sum for the pseudoheader of a UDP packet using the same steps as it does for a TCP packet, and stores the value in the Checksum field of the UDP header.

Note that the TCP/IP transport always ensures that the checksum field in the IP header of a packet is set to zero before passing the packet to an underlying miniport driver. The miniport driver should ignore the checksum field in an IP header. The miniport driver does not need to verify that the checksum field is set to zero and does not need to set this field to zero.

After it receives the NET_BUFFER_LIST structure in its *MiniportSendNetBufferLists* or **MiniportCoSendNetBufferLists** function, a miniport driver typically does the following checksum processing:

1. The miniport driver calls the **NET_BUFFER_LIST_INFO** macro with an *_Id* of **TcpIpChecksumNetBufferListInfo** to obtain an **NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO** structure.

2. The miniport driver tests the **IsIPv4** and **IsIPv6** flags in the NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO structure. If both the **IsIPv4** and **IsIPv6** flags are not set, the NIC should not perform any checksum operations on the packet.

3. If the **IsIPv4** or **IsIPv6** flag is set, the miniport driver tests the **TcpChecksum**, **UdpChecksum**, and **IpHeaderChecksum** flags to determine which checksums the NIC should calculate for the packet.

4. The miniport driver passes the packet to the NIC, which calculates the appropriate checksums for the packet. If a packet has both a tunnel IP header and a transport IP header, a NIC that supports IP checksum offloads performs IP checksum tasks only on the tunnel header. The TCP/IP transport performs IP checksum tasks on the transport IP header.

Before indicating a **NET_BUFFER_LIST** structure for a receive packet on which it performs checksum tasks, the miniport driver validates the appropriate checksums and sets the appropriate *Xxx***ChecksumFailed** or *Xxx***ChecksumSucceeded** flags in the NDIS_TCP_IP_CHECKSUM_NET_BUFFER_LIST_INFO structure.

Turning off Address Checksum Offloads when Large Send Offload (LSO) is enabled does not prevent the miniport driver from computing and inserting checksums in the packets generated by the LSO feature. To disable Address Checksum Offloads in this case the user must also disable LSO.

Background Reading on IPsec

Article • 01/29/2022

[The IPsec Task Offload feature is deprecated and should not be used.]

To comprehend this section, you must understand Internet protocol security (IPsec) as specified in the following RFCs and drafts published by the IP Security Working Group of the Internet Engineering Task Force (IETF):

- Security Architecture for the Internet Protocol (RFC 2401)

Authentication header (AH):

- IP Authentication Header (RFC 2402)
- The Use of HMAC-MD5-96 within ESP and AH (RFC 2403)
- The Use of HMAC-SHA-1-96 within ESP and AH (RFC 2404)
- HMAC-MD5 IP Authentication with Replay Prevention (RFC 2085)

Encapsulating security payload (ESP):

- IP Encapsulating Security Payload (ESP) (RFC 2406)
- The ESP CBC-Mode Cipher Algorithms (RFC 2451)
- The ESP DES-CBC Cipher Algorithm with Explicit IV (RFC 2405)
- The NULL Encryption Algorithm and Its Use with IPsec (RFC 2410)

Requirements and Restrictions That Apply to IPsec Offloads

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

The following requirements and restrictions apply to Internet protocol security (IPsec) offloads:

- The NIC must maintain the security association (SA) tables. This improves performance by eliminating the need to include keys or other information that is required for AH and ESP processing in send packets.

- A NIC might be able to process both AH and ESP payloads for a single packet. In this case, the NIC must support the following possible combinations of integrity (authentication) algorithms for AH and ESP:

AH	ESP
MD5	MD5
SHA 1	SHA 1
MD5	SHA 1
SHA 1	MD5
MD5	Null (only if the NIC supports null encryption)
SHA 1	Null (only if the NIC supports null encryption)

- A NIC that supports DES algorithms must generate the initialization vector (IV) that these algorithms require.

- The only IPsec tasks that a NIC performs are processing encrypted AH checksums or ESP checksums (or both) and encrypting and decrypting ESP payloads. For send packets, the TCP/IP transport creates all headers, padding, and replay numbers and chooses SPI values that are unique to destination address/IPsec protocol pairs. For receive packets, the TCP/IP transport performs inbound policy checks, handles replay detection and prevention, and handles audit events.

- For a send packet, the TCP/IP transport does not provide explicit offsets (such as indicating the start of encrypted data) because the offload driver can easily determine this information from the particular security association (SA) that it uses to process the packet.

- A packet with IPsec protocols must have authentication information in an authentication header (AH) or the encapsulating security payload (ESP) header (or both). It is not permissible for a IPsec packet to have no authentication.

- IPsec tasks are not offloaded for send packets that require IP fragmentation or for receive packets that require reassembly from IP fragmentation.

- IPsec tasks are not offloaded for send and receive packets that pass through a load-balancing miniport driver. For more information about load balancing, see Load Balancing and Failover.

Adding a Security Association to a NIC

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

After the TCP/IP transport determines that a NIC can perform Internet protocol security (IPsec) operations (see Reporting a NIC's IPsec Capabilities), the transport must request the NIC's miniport driver to add one or more inbound and outbound security associations (SAs) to the NIC before the transport can offload IPsec tasks to the NIC. To request that a miniport driver add one or more SAs to the NIC, the TCP/IP transport sets OID_TCP_TASK_IPSEC_ADD_SA.

Deleting a Security Association from a NIC

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

If necessary, the TCP/IP transport can set OID_TCP_TASK_IPSEC_DELETE_SA to request that the miniport driver delete a security association (SA) from the NIC.

To create space for another SA on the NIC, the miniport driver can set the **SaDeleteReq** flag in the **NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO** structure for a receive packet. The TCP/IP transport subsequently issues OID_TCP_TASK_IPSEC_DELETE_SA one time to delete the inbound security association (SA) over which the packet was received and another time to delete the outbound SA that corresponds to the deleted inbound SA. A NIC must not remove either of these SAs before it receives the corresponding OID_TCP_TASK_IPSEC_DELETE_SA request. The miniport driver can set **SaDeleteReq** independently of the **CryptoDone** flag.

Offloading IPsec Tasks in the Send Path

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

Before the TCP/IP transport passes to the miniport driver a **NET_BUFFER_LIST** structure for a packet on which a NIC will perform Internet protocol security (IPsec) tasks, it updates the IPsec information that is associated with the NET_BUFFER_LIST structure. The TCP/IP transport specifies this information in an **NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO** structure, which is part of the NET_BUFFER_LIST information (out-of-band data) that is associated with the NET_BUFFER_LIST structure.

The TCP/IP transport supplies *OffloadHandle*, which specifies the handle to the outbound SA for the transport (end-to-end connection) portion of the send packet. If the packet will be transmitted through a tunnel, the TCP/IP transport also supplies *NextOffloadHandle*, which specifies the handle to the outbound SA for the tunnel portion of the send packet.

After a miniport driver receives the **NET_BUFFER_LIST** structure in its *MiniportSendNetBufferLists* or **MiniportCoSendNetBufferLists** function, it can call the **NET_BUFFER_LIST_INFO** macro with an *_Id* of **IPsecOffloadV1NetBufferListInfo** to obtain the NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO structure that is associated with the NET_BUFFER_LIST structure.

When the NIC performs IPsec processing on a send packet, it calculates the AH or ESP encryption checksums (or both) for the packet and, if the packet contains an ESP payload, encrypts the packet. The TCP/IP transport has already framed the packet, padded it (if necessary), and assigned it a sequence number and SPI.

Offloading IPsec Tasks in the Receive Path

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

When a NIC performs Internet protocol security (IPsec) processing on a receive packet, it decrypts the packet if the packet contains an ESP payload and calculates the AH or ESP encryption checksums (or both) for the packet. Before indicating the NET_BUFFER_LIST structure for the packet up to the TCP/IP transport, the miniport driver calls the NET_BUFFER_LIST_INFO macro with an *_Id* of **IPsecOffloadV1NetBufferListInfo** to obtain the NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO structure that is associated with a packet.

The miniport driver sets the **CryptoDone** flag in the NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO structure to indicate that the NIC performed IPsec checking on at least one IPsec payload in the receive packet. If a NIC performed IPsec checking on both the tunnel and transport portions of a receive packet, the miniport driver also sets the **NextCryptoDone** flag in the NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO structure. The miniport driver sets **NextCryptoDone** only if a packet has both tunnel and transport IPsec payloads. Otherwise, the miniport driver sets **NextCryptoDone** to zero. To indicate the results of the IPsec checks, the miniport driver must also supply a value for the **CryptoStatus** member in the NDIS_IPSEC_OFFLOAD_V1_NET_BUFFER_LIST_INFO structure. If the NIC detects a checksum failure or a decryption failure, the miniport driver must indicate a NET_BUFFER_LIST structure for the receive packet in whatever form it is and specify the appropriate **CryptoStatus** value.

Note that, if the miniport driver is not decrypting an incoming packet, it clears both the **CryptoDone** and the **NextCryptoDone** flags. The miniport driver does this for all receive packets that it does not decrypt, regardless of whether the packet is AH-protected or ESP-protected. The miniport driver sets **CryptoStatus** to CRYPTO_SUCCESS for all packets that it does not decrypt.

After the miniport driver indicates the NET_BUFFER_LIST structure to the TCP/IP transport, the transport examines the results of the IPsec checks that the NIC performed, checks the sequence numbers for the packet, and determines what to do with a packet that fails the checksum or sequencing tests.

Impact of Network Interface Changes on IPsec Offloads

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

The following events in the network interface affect the offloading of Internet protocol security (IPsec) tasks:

- A NIC is removed.

  Before a NIC to which tasks are being offloaded is removed from the system, its miniport driver should delete all security associations (SAs) from the NIC. The miniport driver does not have to request that the TCP/IP transport delete the SAs.

- A routing interface is changed.

  When network traffic is routed through a new interface, the TCP/IP stack temporarily performs IPsec tasks until it has added the appropriate SAs to the NIC that is used in the new interface. The TCP/IP stack adds an SA to a NIC by issuing OID_TCP_TASK_IPSEC_ADD_SA. After the SAs on the NIC that is used for the old interface expire, the TCP/IP transport issues OID_TCP_TASK_IPSEC_DELETE_SA as many times as necessary to request that the NIC's miniport driver delete the SAs from the NIC.

Traversing NATs and NAPTs with UDP-Encapsulated ESP Packets

Article • 09/06/2024

[The IPsec Task Offload feature is deprecated and should not be used.]

Network address translators (NATs) and network address port translators (NAPTs) convert multiple private network addresses into one routeable IP public address and vice versa, thereby allowing many systems to share a single IP address. In this way, NATs and NAPTs help to alleviate the shortage of routeable IPv4 addresses.

However, NATs and NAPTs can cause problems with Internet protocol security (IPsec). Because NATs and NAPTs modify the IP header of a packet, they cause AH-protected packets to fail checksum validation. NAPTs, which modify TCP and UDP ports, cannot modify the ports in the encrypted TCP header of an ESP-protected packet.

UDP encapsulation solves this problem. In practice, UDP encapsulation is used only on ESP packets. A NAT or NAPT can modify the unencrypted IP and UDP headers of a UDP-encapsulated ESP packet without breaking ESP authentication and without being stymied by ESP encryption.

Microsoft supports UDP encapsulation of ESP packets on port 4500. After IKE peers initiate negotiation on port 500, detect support for NAT-traversal, and detect a NAT or NAPT along the path, they can negotiate to "float" IKE and UDP-ESP traffic to port 4500.

Floating to port 4500 for NAT traversal provides the following benefits:

- It bypasses "IPsec-aware" NATs or NAPTs that break UDP-ESP encapsulation on port 500.
- It improves performance. The UDP encapsulation of ESP data packets is more efficient on port 4500 than on port 500. For more information, see UDP-ESP Encapsulation Types.

To support UDP-ESP encapsulation, a miniport driver or the NIC (or both) must:

- Be able to process ESP packets in the receive path, as described in Offloading IPsec Tasks in the Receive Path.
- Maintain a list of parser entries. A parser entry contains information that a NIC requires to parse incoming UDP-ESP packets on one or more security associations (SAs). For more information about parser entries, see UDP-ESP SAs and Parser Entries.

- Maintain a list of SAs that the transport has offloaded to the NIC.
- Support the following OIDs:
  - OID_TCP_TASK_IPSEC_ADD_UDPESP_SA
  - OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA

Feedback

Was this page helpful? Yes No

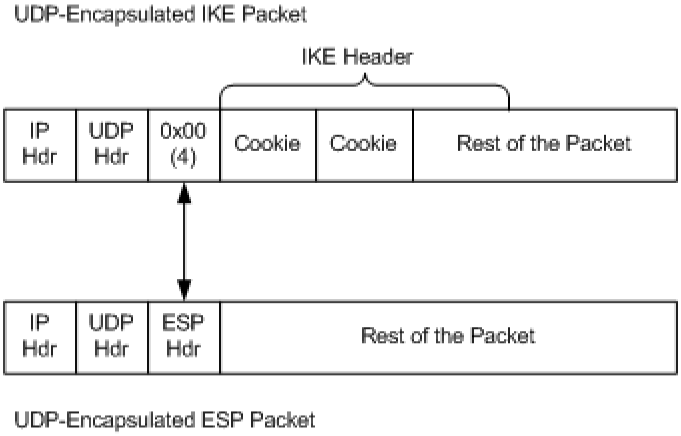
Provide product feedback | Get help at Microsoft Q&A

UDP-ESP Encapsulation Types

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

The following figure shows the UDP encapsulation of Internet Key Exchange (IKE) packets and ESP-protected data packets that are received on port 4500.



Note the four bytes of zeros that follow the UDP header in IKE packets. This field of zeros differentiates IKE packets from UDP-encapsulated ESP packets on port 4500. Instead of zeros, ESP headers have a nonzero ESP header at this location in the packet.

UDP-ESP Encapsulation Subtypes

ESP packets on port 4500 can be formatted according to one of the following UDP-ESP encapsulation subtypes:

- UDP-encapsulated transport.

  An ESP-encapsulated transport-mode packet is encapsulated by UDP.

- UDP-encapsulated tunnel.

  The tunnel-mode portion of a packet is UDP-encapsulated. The transport-mode portion of the packet is not UDP-encapsulated and is not ESP-protected.

- Transport over UDP-encapsulated tunnel.

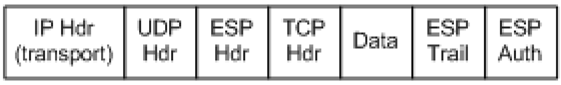
  The tunnel-mode portion of a packet is UDP-encapsulated. The transport-mode portion of a packet is not UDP-encapsulated, but is ESP-protected.

- UDP-encapsulated transport over tunnel.

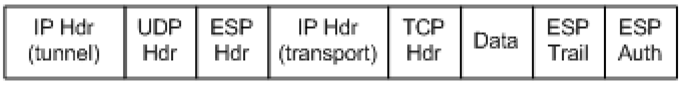
The tunnel-mode portion of a packet is not UDP-encapsulated. The transport-mode portion of a packet is UDP-encapsulated and ESP-protected.

Note that a UDP-encapsulated transport over a UDP-encapsulated tunnel is not a supported encapsulation subtype.

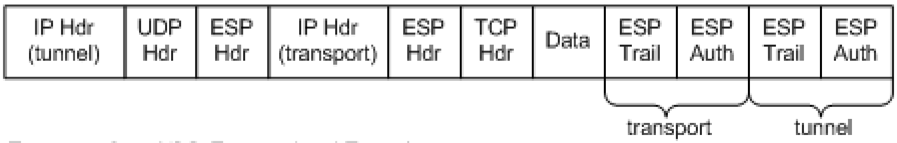
The following figure shows the UDP-ESP encapsulation subtypes for port 4500.



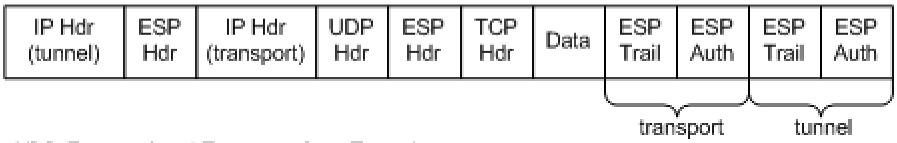
UDP-Encapsulated Transport



UDP-Encapsulated Tunnel



Transport Over UDP-Encapsulated Tunnel



UDP-Encapsulated Transport Over Tunnel

Reporting, Enabling, and Disabling a NIC's Ability to Parse UDP-ESP Packets

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

A miniport driver specifies a NIC's Internet protocol security (IPsec) capabilities in an **NDIS_IPSEC_OFFLOAD_V1** structure. For more information, see Reporting a NIC's IPsec Capabilities.

The miniport reports the NIC's ability to parse incoming UDP-encapsulated ESP packets by setting one or more flags in the **Supported** . **Reserved** member of the NDIS_IPSEC_OFFLOAD_V1 structure. The miniport driver can specify any or all of the four UDP-ESP encapsulation subtypes that are described in UDP-ESP Encapsulation Types.

For information about enabling and disableing UDP ESP parsing capabilities, see Enabling and Disabling TCP/IP Offload Services.

UDP-ESP SAs and Parser Entries

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

A miniport driver that supports UDP-ESP encapsulation must maintain a list of parser entries. A parser entry contains information that a NIC requires to parse incoming UDP-ESP packets on offloaded security associations (SAs).

A parser entry contains the following information:

- The UDP-ESP encapsulation type.

  Currently, only one encapsulation type is supported. For a description of the basic UDP-ESP encapsulation types, see UDP-ESP Encapsulation Types.

- The destination encapsulation port.

  The NIC should look for the destination port in the UDP header of inbound UDP-encapsulated packets that it processes on the offloaded SAs. Currently, UDP encapsulation of ESP packets is supported only on port 4500.

The TCP/IP transport maintains its own list of parser entries that it has offloaded to the miniport driver. When adding or deleting a UDP-ESP SA, the transport and miniport driver use a handle to identify a particular parser entry.

Note that parser entries allow UDP-ESP functionality to be extended, if necessary, to accommodate different encapsulation types and more than one port for each encapsulation type.

Adding a UDP-ESP SA and Parser Entry

The TCP/IP transport requests a miniport driver to add one or more UDP-ESP SAs, and the parser entry for these SAs, by issuing an OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request. The **EncapTypeEntry** member of the OFFLOAD_IPSEC_ADD_UDPESP_SA structure contains the parser entry information.

Before issuing an OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request, the TCP/IP transport determines whether the parser entry for the SAs that is being offloaded is in its parser entry list for the specified IP interface.

- If the parser entry is not in the transport's list, the transport creates its own copy of the entry and sets the **EncapTypeEntryOffloadHandle** member of the

OFFLOAD_IPSEC_ADD_UDPESP_SA structure to **NULL**. The transport then issues the OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request. After receiving the request, the miniport driver determines whether the parser entry that the **EncapTypeEntry** specified is in the NIC's parser entry list.

  - If the specified parser entry is not in the NIC's parser entry list, the miniport driver creates the parser entry by using the encapsulation type and destination port specified in **EncapTypeEntry** and adds the parser entry to the NIC's parser entry list. The miniport driver then offloads the SAs specified in the OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request. After successfully completing the OID request, the miniport driver returns a handle in **EncapTypeEntryOffloadHandle** that identifies the newly created parser entry. The miniport driver also returns a handle that identifies the offloaded SAs in the **OffloadHandle** member of the OFFLOAD_IPSEC_ADD_UDPESP_SA structure.
  - If the specified parser entry is already in the NIC's parser entry list, the miniport driver simply returns the handle in **EncapTypeEntryOffloadHandle** for the existing parser entry. The miniport driver also returns a handle that identifies the offloaded SAs in the **OffloadHandle** member of the OFFLOAD_IPSEC_ADD_UDPESP_SA structure.

If the miniport driver completes the OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request successfully, the transport adds its copy of the new parser entry to its own parser entry list for the given IP interface. In addition, the transport increments the reference count for the parser entry by one. The transport uses this reference count to enumerate how many offloaded UDP-ESP SAs are associated with the parser entry.

If the miniport driver fails the OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request, the transport discards its copy of the parser entry. If the miniport driver fails such a request, it must ensure that it has not, in fact, added the parser entry and offloaded the SAs.

- If the parser entry is already in the transport's parser entry list, the miniport driver has already added the parser entry in response to a previous OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request. In this case, the transport increments the reference count for the parser entry by one and sets the **EncapTypeEntryOffloadHandle** to the value that the miniport driver previously returned. The transport then issues an OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request This requests the miniport driver to use an existing parser entry for the additional SAs that are being offloaded. In this case, the miniport driver should simply return an **OffloadHandle** that identifies the offloaded SAs. If the OID_TCP_TASK_IPSEC_ADD_UDPESP_SA request fails, the transport decrements the reference count for the parser entry.

Deleting a UDP-ESP SA and Parser Entry

The TCP/IP transport requests a miniport driver to delete one or more SAs and possibly the parser entry for these SAs by issuing an OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA request.

Before issuing this request, the TCP/IP transport decrements the reference count for the parser entry that is associated with the SAs to be deleted. The transport then tests whether the reference count is zero.

- If the reference count is not zero, the parser entry is associated with one or more other SAs that are currently offloaded to the NIC. In this case, the transport sets the **EncapTypeEntryOffldHandle** member of the OFFLOAD_IPSEC_DELETE_UDPESP_SA structure to **NULL**. After it receives the OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA request, the miniport driver simply deletes the SAs that are specified in the OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA request.

- If the reference count is zero, the parser entry is not associated with any other SAs that have been offloaded to the NIC. In this case, the transport sets the **EncapTypeEntryOffldHandle** member to the value of the parser entry handle that the miniport driver previously returned. The miniport driver deletes both the specified parser entry and the specified SAs.

If the miniport driver fails the OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA request, it should mark the specified SAs and, if appropriate, the specified parser entry for deletion and perform the deletion later. To process incoming packets, the miniport driver must not use a parser entry or SA that is marked for deletion.

Note that a transport could request a miniport driver to delete an SA or a parser entry (or both) before the miniport driver completes adding that SA or parser entry (or both). The miniport driver must therefore serialize the deletion operation with the addition operation.

Processing UDP-Encapsulated ESP Packets

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

When a NIC receives a UDP-encapsulated packet on port 4500, it checks whether the packet is an IKE (control) packet or an ESP (data) packet. For a description of the UDP encapsulation types for IKE and ESP packets, see UDP-ESP Encapsulation Types.

- If the packet is an IKE packet, the NIC passes the packet to the miniport driver without further IPsec-related processing.

- If the packet is an ESP packet, the NIC checks whether the packet's inbound SA (or SAs in the case of a transport-over-tunnel packet) is currently offloaded to the NIC.
  - If the inbound SAs are not currently offloaded to the NIC, the NIC passes the packet to the miniport driver without further IPsec-related processing.
  - If the inbound SAs are currently offloaded to the NIC, the NIC parses the packet by using the encapsulation type specified by the parser entry that is associated with the SAs. The NIC then processes the ESP payloads in the packet, as described in Offloading IPsec Tasks in the Receive Path.

If the incoming ESP packet is a UDP-encapsulated transport-over-tunnel packet, as described in UDP-ESP Encapsulation Types, the NIC first decrypts the ESP payload of tunnel-mode portion of the packet, which is not UDP-encapsulated. Then the NIC processes the UDP-encapsulated tunnel-mode portion of the packet.

Introduction to IPsec Offload Version 2

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

IPsec offload version 2 (IPsecOV2) extends services that are provided in IPsec offload version 1 (IPsecOV1). For more information about IPsecOV1 offload and IPsec, see IPsec Offload Version 1.

An NDIS 6.1 and later miniport driver reports the IPsecOV2 offload capabilities of a miniport adapter to NDIS. To report IPsec capabilities:

- During initialization, a miniport driver reports the task offload default configuration and the hardware capabilities of a miniport adapter in the **NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES** structure.

- If the configured capabilities change, the miniport driver reports the current configuration with the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication. The configuration can change if the OID_TCP_OFFLOAD_PARAMETERS OID sets the current task offload configuration of a miniport adapter. Also, if the hardware configuration under a MUX intermediate driver changes, the MUX intermediate driver must report the hardware configuration changes with the **NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES** status indication.

NDIS reports the default configuration of the offload capabilities of a miniport adapter to overlying protocol drivers in the **NDIS_BIND_PARAMETERS** structure. Overlying protocol drivers can choose IPsecOV2 task offload services from the services that are supported in the current configuration. The **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication ensures that all of the overlying protocol drivers are updated with the new capabilities information.

When reporting hardware capabilities during initialization, the miniport driver must read the standardized keywords from the registry. For more information about IPsecOV2 offload capabilities, see Reporting a NIC's IPsec Offload Version 2 Capabilities.

Note NDIS provides a direct OID request interface for NDIS 6.1 and later drivers. The direct OID request path supports OID requests that are queried or set frequently.

IPsecOV2 provides the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA, OID_TCP_TASK_IPSEC_OFFLOAD_V2_UPDATE_SA, and OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA direct OID requests to enable protocol drivers to add, update, and delete security associations (SAs). For more information about SAs, see Managing Security Associations in IPsec Offload Version 2.

A NIC can perform IPsec offload tasks on the send and receive paths. NDIS drivers use the **NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO**, **NDIS_IPSEC_OFFLOAD_V2_HEADER_NET_BUFFER_LIST_INFO**, and **NDIS_IPSEC_OFFLOAD_V2_TUNNEL_NET_BUFFER_LIST_INFO** structures to access the IPsec out-of-band (OOB) information.

On the send path, the overlying drivers set the handle to the outbound SA and IPsec header information in OOB information in the **NET_BUFFER_LIST** structure to specify that the NIC should perform IPsecOV2 offload tasks.

On the receive path, after the SA is offloaded, the NIC must perform the IPsec processing on all the received packets that match the capabilities that the miniport driver reported to NDIS. The miniport driver sets the appropriate flags in OOB information in the **NET_BUFFER_LIST** structure to specify specific offload tasks that the NIC performed and the result of those operations.

For more information about send and receive processing in IPsecOV2, see Sending Network Data with IPsec Offload Version 2 and Receiving Network Data with IPsec Offload Version 2.

Reporting a NIC's IPsec Offload Version 2 Capabilities

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

To specify IPsec offload version 2 (IPsecOV2) capabilities, an NDIS 6.1 and later miniport driver specifies the current or default configuration of a NIC in an **NDIS_IPSEC_OFFLOAD_V2** structure. Miniport drivers must include the default IPsecOV2 configuration in the **NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES** structure. Miniport drivers call the **NdisMSetMiniportAttributes** function from the *MiniportInitializeEx* function and pass in the information in NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES.

Miniport drivers must report changes in the IPsecOV2 capabilities, if any, in the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication.

Note NDIS provides a direct OID request interface for NDIS 6.1 and later drivers. The direct OID request path supports OID requests that are queried or set frequently.

In response to a query of OID_TCP_OFFLOAD_CURRENT_CONFIG, NDIS includes the NDIS_IPSEC_OFFLOAD_V2 structure in the **NDIS_OFFLOAD** structure that NDIS returns in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. NDIS uses the information that the miniport driver provided.

Accessing NET_BUFFER_LIST Information in IPsec Offload Version 2

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

NDIS provides out-of-band (OOB) data in the **NetBufferListInfo** member of the NET_BUFFER_LIST structure, which specifies a linked list of NET_BUFFER structures. The **NetBufferListInfo** member is an array of values that contain information that is common to all of the NET_BUFFER structures in the list.

Use the following identifiers with the NET_BUFFER_LIST_INFO macro to set and get the IPsec offload version 2 (IPsecOV2) OOB data in the **NetBufferListInfo** array:

IPsecOffloadV2NetBufferListInfo
Specifies IPsecOV2 information that is used in offloading IPsec tasks from the TCP/IP protocol to a NIC. When you specify **IPsecOffloadV2NetBufferListInfo**, NET_BUFFER_LIST_INFO returns an NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO structure.

IPsecOffloadV2TunnelNetBufferListInfo
Specifies IPsecOV2 tunnel information that is used in offloading IPsec tasks from the TCP/IP protocol to a NIC. When you specify **IPsecOffloadV2TunnelNetBufferListInfo**, NET_BUFFER_LIST_INFO returns an NDIS_IPSEC_OFFLOAD_V2_TUNNEL_NET_BUFFER_LIST_INFO structure.

IPsecOffloadV2HeaderNetBufferListInfo
Specifies the header offsets for IPsec headers in the NET_BUFFER_LIST and the values for the next header and pad length. When you specify **IPsecOffloadV2HeaderNetBufferListInfo**, NET_BUFFER_LIST_INFO returns an NDIS_IPSEC_OFFLOAD_V2_HEADER_NET_BUFFER_LIST_INFO structure.

Managing Security Associations in IPsec Offload Version 2

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

After the TCP/IP transport determines that a NIC can perform IPsec offload version 2 (IPsecOV2) operations (see Reporting a NIC's IPsec Offload Version 2 Capabilities), the transport requests that the NIC's miniport driver add one or more security associations (SAs) to the NIC before the transport can offload IPsec tasks to the NIC. After adding SAs, the TCP/IP transport can also delete or update them. The IPsecOV2 interface requires the NDIS direct OID interface for add, delete, and update OIDs.

Note NDIS provides a direct OID request interface for NDIS 6.1 and later drivers. The direct OID request path supports OID requests that are queried or set frequently.

To request that a miniport driver add one or more SAs to a NIC, the TCP/IP transport sets the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID. The miniport driver receives an **IPSEC_OFFLOAD_V2_ADD_SA** structure and configures the NIC for IPsecOV2 processing on an SA. With a successful set to OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA, the miniport driver initializes a handle that identifies the offloaded SA in the IPSEC_OFFLOAD_V2_ADD_SA structure. The transport uses this handle in subsequent requests to the miniport driver (that is, on the send path or in the calls to modify or delete the SA). For more information about using the SA handle in the send path, see Sending Network Data with IPsec Offload Version 2.

The miniport driver reports the number of SAs that a NIC can support in the **SaOffloadCapacity** member of the **NDIS_IPSEC_OFFLOAD_V2** structure.

The miniport driver can set the **SaDeleteReq** flag in the **NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO** structure for a receive packet. The TCP/IP transport subsequently issues OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA one time to delete the inbound SA that the packet was received over and one time again to delete the outbound SA that corresponds to the deleted inbound SA.

The TCP/IP transport issues OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA to delete an inbound SAs over which a packet was received and to delete the outbound SAs that correspond to the deleted inbound SAs. A NIC must not remove these SAs before it receives the corresponding OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA request.

The TCP/IP transport sets the OID_TCP_TASK_IPSEC_OFFLOAD_V2_UPDATE_SA OID to request that a miniport driver update a NIC with the higher order bits for an SA with extended sequence numbers (ESN). For NICs that support ESN, when the miniport driver receives this request, the driver should update the sequence number of the specified SA in the NIC in accordance with the **IPSEC_OFFLOAD_V2_OPERATION** enumeration value that is specified in the **Operation** member of the **IPSEC_OFFLOAD_V2_UPDATE_SA** structure.

Sending Network Data with IPsec Offload Version 2

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

The TCP/IP transport provides IPsec Offload Version 2 (IPsecOV2) information for one or more SAs with the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID. Before the miniport driver returns a successful result for OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA, the miniport driver initializes an offload handle. The TCP/IP transport requests the miniport driver to offload the processing of a **NET_BUFFER_LIST** structure by specifying IPsecOV2 information in the **NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO** and **NDIS_IPSEC_OFFLOAD_V2_HEADER_NET_BUFFER_LIST_INFO** structures, which are part of the **NET_BUFFER_LIST** out-of-band (OOB) information.

The TCP/IP transport supplies an offload handle in the **OffloadHandle** member of **NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO** that specifies the handle to the outbound security association (SA) for the transport (end-to-end connection) portion of the send packet.

The TCP/IP transport supplies the following header information in the **NDIS_IPSEC_OFFLOAD_V2_HEADER_NET_BUFFER_LIST_INFO** structure:

- Header offsets for an AH header, ESP header, or both.
- The next protocol value (identical to the one that is contained in the ESP trailer).
- The pad length that is used for a combined large send offload (LSO) and IPsec offload.

Also, if the send packet will be transmitted through a tunnel, the TCP/IP transport supplies an **NDIS_IPSEC_OFFLOAD_V2_TUNNEL_NET_BUFFER_LIST_INFO** structure. This structure specifies the offload handle to the outbound SA for the tunnel portion of the send packet. For more information about accessing OOB information, see Accessing NET_BUFFER_LIST Information in IPsec Offload Version 2.

The miniport driver provided the offload handles in response to an OID set request of OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA. For more information about SAs, see Managing Security Associations in IPsec Offload Version 2.

When a miniport driver handles a send request in the *MiniportSendNetBufferLists* function, the miniport driver:

- Verifies that the hardware is configured to handle IPsec offload services. If the hardware is not configured to handle IPsec offload services, the miniport driver should handle the send request without providing the offload services.

- Verifies the handles in the NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO and NDIS_IPSEC_OFFLOAD_V2_TUNNEL_NET_BUFFER_LIST_INFO structures to determine if IPsec cryptographic processing is required. An offload handle value of zero indicates that no IPsec task offload should be performed for the NET_BUFFER_LIST. If the miniport driver cannot find the offloaded SA that corresponds to the specified offload handle, the send packet should fail with an **NDIS_STATUS_FAILURE** value.

- Verifies the handles in the NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO structures to determine if segmentation offload should be performed for the NET_BUFFER_LIST.

- Completes the required AH and ESP processing for all of the send packets in the NET_BUFFER_LIST. When the NIC performs IPsec processing on a send packet, it performs the cryptographic operations on the packet data. The TCP/IP transport has already framed the packet, padded it (if necessary), and assigned it a sequence number and security parameters index (SPI). For a combined LSO and IPsec offload, the NET_BUFFER might have padding that will be discarded while the NIC segments the large packet. The amount of padding is specified in the **PadLength** member of the NDIS_IPSEC_OFFLOAD_V2_HEADER_NET_BUFFER_LIST_INFO structure. Segmented packets might require padding to support IPsec operations.

When a protocol driver transmits a packet that requests both LSO and IPsecOV2, it will not frame the ESP trailer. This is because the information in the ESP trailer, such as the padding length, will not be accurate for the last segment that was generated by the NIC.

Receiving Network Data with IPsec Offload Version 2

Article • 12/15/2021

[The IPsec Task Offload feature is deprecated and should not be used.]

A NIC performs IPsec offload version 2 (IPsecOV2) processing on a receive packet as specified in a security association (SA) that was offloaded from the transport.

The miniport driver sets the IPsecOV2 out-of-band (OOB) information before indicating the received data to overlying drivers. For more information about accessing OOB information, see Accessing NET_BUFFER_LIST Information in IPsec Offload Version 2.

Note A miniport driver should indicate all received packets to overlying drivers even if an error occurs while processing the IPsec data in the NIC. The driver must indicate packets with errors to enable the driver stack to monitor and troubleshoot the network traffic.

Before the miniport driver indicates the received data packet up the driver stack, the miniport driver:

- Verifies that the hardware is configured to handle IPsec offload tasks. If not, the miniport driver does a receive indication with no additional IPsec offload processing.

- Looks at the security parameters index (SPI) to determine if a matching offloaded SA exists. The miniport driver confirms the destination address on the packet is same as the one specified in offloaded SA. If there is no matching SA, the NIC indicates the received data without setting the IPsecOV2 OOB information.

- Verifies that it can process the packet based on the capabilities that the miniport driver reported to the transport or it makes a receive indication without further IPsec processing. For example, the packet might have IP options where the NIC does not support IPsec offload processing for such packets and the miniport driver does the IPsec processing.

- Sets the **CryptoDone** flag in the NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO structure to indicate that a NIC performed IPsec checking on at least one IPsec payload in the received packet.

- Sets the **NextCryptoDone** flag in the NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO structure to indicate that a NIC performed IPsec checking on both the tunnel and transport portions of a receive packet. The miniport driver sets this flag only if a packet has both tunnel and transport payloads; otherwise, this flag should be zero.

- Sets the correct **CryptoStatus** value of the NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO structure to indicate the results of the IPsec checks.

If the NIC did not perform offload processing on an incoming packet, the miniport driver clears both the **CryptoDone** and the **NextCryptoDone** flags. The miniport driver clears these flags for all receive packets where a NIC does not decrypt, regardless of whether the packet is AH-protected or ESP-protected.

A miniport driver can set **SaDeleteReq**, in the **NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO** structure for a receive **NET_BUFFER_LIST**. The TCP/IP transport subsequently issues OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA once to delete the inbound SA that the packet was received over and once again to delete the outbound SA that corresponds to the deleted inbound SA. For more information about adding and deleting SAs, see Managing Security Associations in IPsec Offload Version 2.

After the miniport driver indicates the NET_BUFFER_LIST structure to the TCP/IP transport, the TCP/IP transport examines the results of the IPsec checks that the NIC performed on the packet, checks the sequence numbers for the packet, and determines what to do with a packet that fails the checksum or sequencing tests.

Offloading the Segmentation of Large TCP Packets

Article • 12/15/2021

NDIS miniport drivers can offload the segmentation of large TCP packets that are larger than the maximum transmission unit (MTU) of the network medium. A NIC that supports the segmentation of large TCP packets must also be able to:

- Calculate IP checksums for send packets that contain IP options.
- Calculate TCP checksums for send packets that contain TCP options.

NDIS versions 6.0 and later support large send offload version 1 (LSOv1), which is similar to large send offload (LSO) in NDIS 5.*x*. NDIS versions 6.0 and later also support large send offload version 2 (LSOv2), which provides enhanced large packet segmentation services, including support for IPv6.

A miniport driver that supports LSOv2 and LSOv1 must determine the offload type from the **NET_BUFFER_LIST** structure OOB information. The driver can use the **Type** member of the **NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO** structure to determine whether the driver stack is using LSOv2 or LSOv1 and perform the appropriate offload services. Any NET_BUFFER_LIST structure that contains the LSOv1 or LSOv2 OOB data also contains a single **NET_BUFFER** structure. For more information about NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO, see Accessing TCP/IP Offload NET_BUFFER_LIST Information.

However, in a case where the miniport has received **OID_TCP_OFFLOAD_PARAMETERS** to turn off LSO feature on the miniport and after the miniport has completed the OID successfully, the miniport shall drop all **NET_BUFFER_LIST** which contain any non-zero LSOv1 or LSOv2 OOB data(**NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO**).

The TCP/IP transport offloads only those large TCP packets that meet the following criteria:

- The packet is a TCP packet. The TCP/IP transport does not offload large UDP packets for segmentation.
- The packet must be divisible by at least the minimum number of segments specified by the miniport driver. For more information, see Reporting a NIC's LSOv1 TCP-Packet-Segmentation Capabilities and Reporting a NIC's LSOv2 TCP-Packet-Segmentation Capabilities.

- The packet is not a loopback packet.
- The packet will not be sent through a tunnel.

Before offloading a large TCP packet for segmentation, the TCP/IP transport:

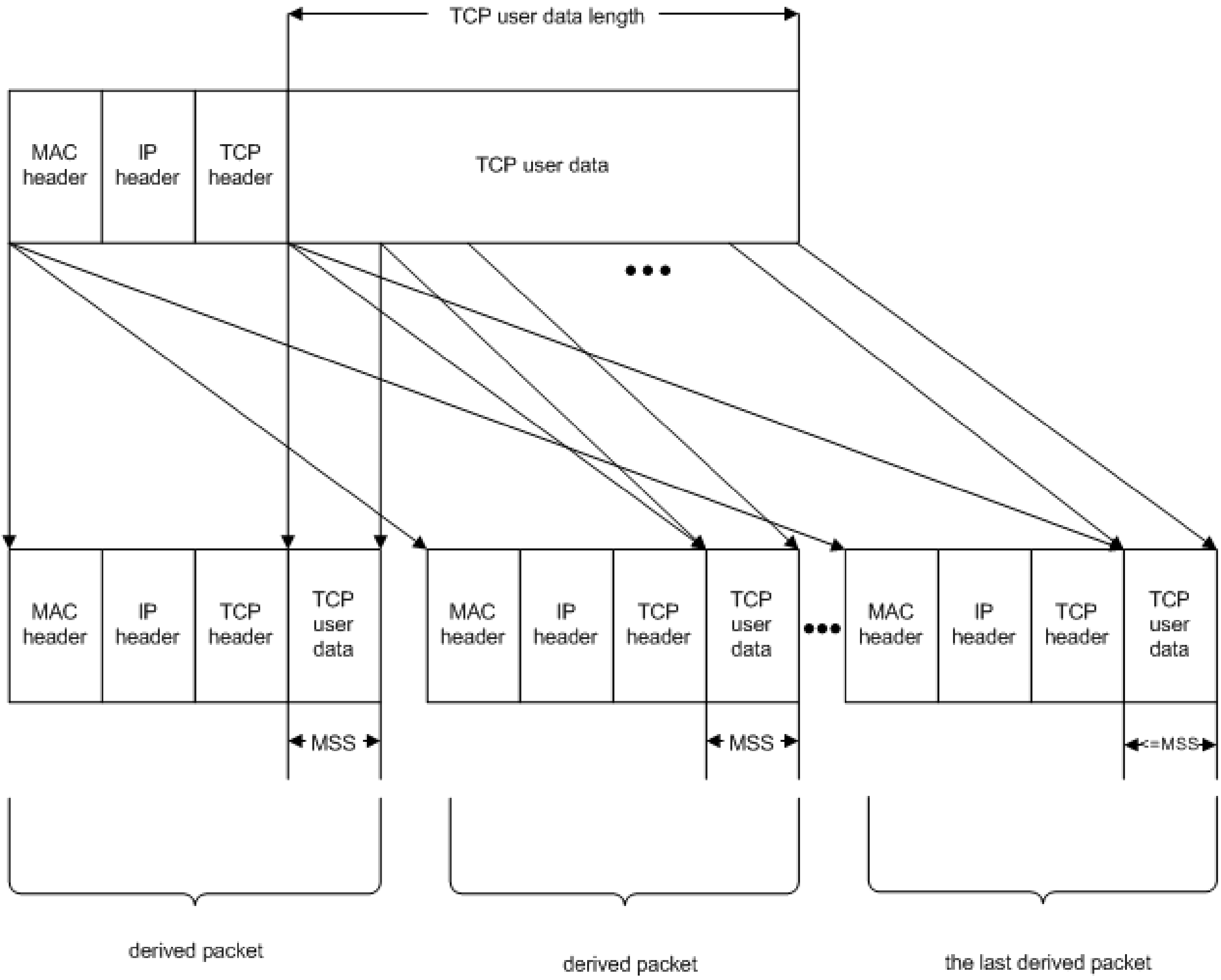
- Updates the large-packet segmentation information that is associated with the NET_BUFFER_LIST structure. This information is an NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO structure that is part of the **NET_BUFFER_LIST** information that is associated with the **NET_BUFFER_LIST** structure. For more information about **NET_BUFFER_LIST** information, see Accessing TCP/IP Offload NET_BUFFER_LIST Information. The TCP/IP transport sets the **MSS** value to the maximum segment size (MSS).
- For LSOv1, writes the total length of the large TCP packet to the Total Length field of the packet's IP header. The total length includes the length of the IP header, the length of the IP options if they are present, the length of the TCP header, the length of the TCP options if they are present, and the length of the TCP payload. For LSOv2, sets the Total Length field of the packet's IP header to 0. Miniport drivers should determine the length of the packet from the length of the first NET_BUFFER structure in the NET_BUFFER_LIST structure.
- Calculates a one's complement sum for the TCP pseudoheader and writes this sum to the Checksum field of the TCP header. The TCP/IP transport calculates the one's complement sum over the following fields in the pseudoheader: Source IP Address, Destination IP Address, and Protocol. The one's complement sum for the pseudoheader provided by the TCP/IP transport gives the NIC an early start in calculating the real TCP checksum for each packet that the NIC derives from the large TCP packet without having to examine the IP header. Note that RFC 793 stipulates that the pseudo-header checksum is calculated over the Source IP Address, Destination IP Address, Protocol, and TCP Length. (The TCP Length is the length of the TCP header plus the length of the TCP payload. The TCP Length does not include the length of the pseudo-header.) However, because the underlying miniport driver and NIC generate TCP segments from the large packet that is passed down by the TCP/IP transport, the transport does not know the size of the TCP payload for each TCP segment and therefore cannot include the TCP Length in the pseudo-header. Instead, as described below, the NIC extends the pseudo-header checksum that was supplied by the TCP/IP transport to cover the TCP Length of each generated TCP segment.
- Writes the correct sequence number to the Sequence Number field of the TCP header. The sequence number identifies the first byte of the TCP payload.

After the miniport driver obtains the **NET_BUFFER_LIST** structure in its *MiniportSendNetBufferLists* or **MiniportCoSendNetBufferLists** function, it can call the **NET_BUFFER_LIST_INFO** macro with an *_Id* of **TcpLargeSendNetBufferListInfo** to obtain the MSS value written by the TCP/IP transport.

The miniport driver obtains the total length of the large packet from the packet's IP header and uses the MSS value to divide the large TCP packet into smaller packets. Each of the smaller packets contains MSS or less user data bytes. Note that only the last packet that was created from the segmented large packet should contain less than MSS user data bytes. All other packets that were created from the segmented packet should contain MSS user data bytes. If you do not follow this rule, the creation and transmission of unnecessary extra packets could degrade performance.

The miniport driver affixes MAC, IP, and TCP headers to each segment that is derived from the large packet. The miniport driver must calculate the IP and TCP checksums for these derived packets. To calculate the TCP checksum for each packet that was derived from the large TCP packet, the NIC calculates the variable part of the TCP checksum (for the TCP header and TCP payload), adds this checksum to the one's complement sum for the pseudoheader calculated by the TCP/IP transport, and then calculates the 16-bit one's complement for the checksum. For more information about calculating such checksums, see RFC 793 and RFC 1122.

The following figure shows the segmentation of a large packet.



The length of the TCP user data in the large TCP packet should be equal to or less than the value that the miniport driver assigns to the **MaxOffLoadSize** value. For more information about the **MaxOffLoadSize** value, see Reporting a NIC's LSOv1 TCP-Packet-Segmentation Capabilities and Reporting a NIC's LSOv2 TCP-Packet-Segmentation Capabilities.

After a driver issues a status indication to indicate a change to the **MaxOffLoadSize** value, the driver must not crash if it receives an LSO send request that uses the previous **MaxOffLoadSize** value. Instead, the driver can fail the send request.

An intermediate driver that independently issues status indications that report a change in the **MaxOffLoadSize** value must ensure that the underlying miniport adapter that has not issued a status indication does not get any packets that are larger in size than the **MaxOffLoadSize** value that the miniport adapter reported.

A miniport-intermediate driver that responds to OID_TCP_OFFLOAD_PARAMETERS to turn off LSO services must be prepared for a small window of time where LSO send requests could still reach the miniport driver.

The length of the TCP user data in a segment packet must be less than or equal to the MSS. The MSS is the ULONG value that the TCP transport passes down by using the LSO

NET_BUFFER_LIST information that is associated with the **NET_BUFFER_LIST** structure. Note that only the last packet that was created from the segmented large packet should contain less than MSS user data bytes. All other packets that were created from the segmented packet should contain MSS user data bytes. If you do not follow this rule, the creation and transmission of unnecessary extra packets could degrade performance.

The number of segment packets that were derived from the large TCP packet must be equal to or greater than the **MinSegmentCount** value that is specified by the miniport driver. For more information about the **MinSegmentCount** value, see Reporting a NIC's LSOv1 TCP-Packet-Segmentation Capabilities and Reporting a NIC's LSOv2 TCP-Packet-Segmentation Capabilities.

The following assumptions and restrictions apply to processing IP and TCP headers for any LSO-capable miniport driver regardless of version:

- The MF bit in the IP header of the large TCP packet that the TCP/IP transport offloaded will not be set, and the Fragment Offset in the IP header will be zero.
- The URG, RST, and SYN flags in the TCP header of the large TCP packet will not be set, and the urgent offset (pointer) in the TCP header will be zero.
- If the FIN bit in the TCP header of the large packet is set, the miniport driver must set this bit in the TCP header of the last packet that it creates from the large TCP packet.
- If the PSH bit in the TCP header of the large TCP packet is set, the miniport driver must set this bit in the TCP header of the last packet that it creates from the large TCP packet.
- If the CWR bit in the TCP header of the large TCP packet is set, the miniport driver must set this bit in the TCP header of the first packet that it creates from the large TCP packet. The miniport driver may choose to set this bit in the TCP header of the last packet that it creates from the large TCP packet, although this is less desirable.
- If the large TCP packet contains IP options or TCP options (or both), the miniport driver copies these options, unaltered, to each packet that it derived from the large TCP packet. Specifically, the NIC will not increment the Time Stamp option.
- All packet headers (Ethernet, IP, TCP) will be in the first MDL of the packet. The headers will not be split across multiple MDLs.

  > **Tip**

> This assumption is valid when LSO is enabled. Otherwise, when LSO is not enabled, miniport drivers cannot assume that IP headers are in the same MDL as Ethernet headers.

The miniport driver must send the packets in NET_BUFFER_LIST structures in the order that it receives the NET_BUFFER_LIST structures from the TCP/IP transport.

When processing a large TCP packet, the miniport adapter is responsible only for segmenting the packet and affixing MAC, IP, and TCP headers to the packets that are derived from the large TCP packet. The TCP/IP transport performs all other tasks (such as adjusting the send window size based on the remote host's receive window size).

Before completing the send operation for the large packet (such as with NdisMSendNetBufferListsComplete or NdisMCoSendNetBufferListsComplete), the miniport driver writes the NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO value (NET_BUFFER_LIST information for large-send offloads) with the total number of TCP user data bytes that are sent successfully in all packets that were created from the large TCP packet.

In addition to the previous LSO requirements, LSOv2-capable miniport drivers must also:

- Support IPv4 or IPv6 or both IPv4 and IPv6.

- Support replication of the IPv4 options, from the large packet, in each segment packet that the network interface card (NIC) generates.

- Support replication of the IPv6 extension header, from the large TCP packet, in each TCP segment packet.

- Support replication of TCP options in each TCP segment packet that the miniport driver generates.

- Use the IP and TCP header in the NET_BUFFER_LIST structure as a template to generate TCP/IP headers for each segment packet.

- Use IP identification (IP ID) values in the range from 0x0000 to 0x7FFF. (The range from 0x8000 to 0xFFFF is reserved for TCP chimney offload-capable devices.) For example, if the template IP header starts with an Identification field value of 0x7FFE, the first TCP segment packet must have an IP ID value of 0x7FFE, followed by 0x7FFF, 0x0000, 0x0001, and so on.

- Use the byte offset in the **TcpHeaderOffset** member of NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO to determine the location of the TCP header, starting from the first byte of the packet.

- Limit the number of NET_BUFFER structures that are associated with each LSOv2 NET_BUFFER_LIST structure to one.

  > ⓘ **Note**
  >
  > This is a new requirement for LSOv2-capable miniport drivers. This rule is not enforced for LSOv1 miniport drivers explicitly, although it is recommended.

- Determine the total length of the packet from the length of the first NET_BUFFER structure in the NET_BUFFER_LIST structure. This is different from the method drivers use for LSOv1.

- Support TCP options, IP options, and IP extension headers.

- When a send operation is complete, the miniport driver must set the **LsoV2TransmitComplete.Reserved** member of the NDIS_TCP_LARGE_SEND_OFFLOAD_NET_BUFFER_LIST_INFO structure to zero and the **LsoV2TransmitComplete.Type** member to NDIS_TCP_LARGE_SEND_OFFLOAD_V2_TYPE.

UDP Segmentation Offload (USO)

Article • 12/15/2021

UDP Segmentation Offload (USO), supported in Windows 10, version 2004 and later, is a feature that enables network interface cards (NICs) to offload the segmentation of UDP datagrams that are larger than the maximum transmission unit (MTU) of the network medium. By doing so, Windows reduces CPU utilization associated with per-packet TCP/IP processing. Requirements for USO are similar to LSOv2, which is for the TCP transport protocol.

Requirements for USO

This section refers primarily to NDIS protocol and miniport drivers. NDIS lightweight filter drivers (LWFs) must follow protocol driver requirements when modifying or sending packets, and can also assume that any packets provided to its *FilterSendNetBufferLists* handler meet the protocol driver requirements.

Miniport drivers can offload the segmentation of large UDP packets that are larger than the MTU of the network medium. A NIC that supports the segmentation of large UDP packets must also be able to do the following:

- Calculate IP checksums for sent packets that contain IPv4 options
- Calculate UDP checksums for sent packets

A miniport driver that supports USO must determine the offload type from the **NET_BUFFER_LIST** structure's out of band (OOB) information. If the value of the **NDIS_UDP_SEGMENTATION_OFFLOAD_NET_BUFFER_LIST_INFO** structure is non-zero, then the miniport driver must perform USO. Any **NET_BUFFER_LIST** that contains USO OOB data also contains a single **NET_BUFFER** structure. However, in the case where the miniport driver has received OID_TCP_OFFLOAD_PARAMETERS to turn off USO, after the miniport driver has completed the OID successfully it should reject and return any **NET_BUFFER_LIST** that has the USO OOB field set.

The TCP/IP transport offloads only those UDP packets that meet the following criteria:

- The packet is a UDP packet.
- The packet length must be greater than maximum segment size **(MSS)** * **(MinSegmentCount - 1)**.
- If the miniport driver does not set the **SubMssFinalSegmentSupported** capability, then each large UDP packet offloaded by the transport must have **Length % MSS == 0**. That is, the large packet is divisible into **N** packets with each packet segment

containing exactly **MSS** user bytes. If the miniport driver sets the **SubMssFinalSegmentSupported** capability, then this packet length divisibility condition on the transport does not apply. In other words, the final segment can be less than **MSS**.
- The packet is not a loopback packet.
- The **MF** bit in the IP header of the large UDP packet that the TCP/IP transport offloaded will not be set, and the **Fragment Offset** in the IP header will be zero.
- The application has specified UDP_SEND_MSG_SIZE/WSASetUdpSendMessageSize.

Before offloading a large UDP packet for segmentation, the TCP/IP transport does the following:

- Updates the large packet segmentation information that is associated with the **NET_BUFFER_LIST** structure. This information is an **NDIS_UDP_SEGMENTATION_OFFLOAD_NET_BUFFER_LIST_INFO** structure that is part of the **NET_BUFFER_LIST** structure's OOB information. The TCP/IP transport sets the MSS value to the desired MSS.
- Calculates a one's complement sum for the UDP pseudoheader and writes this sum to the **Checksum** field of the UDP header. The TCP/IP transport calculates the one's complements sum over the following fields in the pseudoheader: Source IP Address, Destination IP Address, and Protocol.

The one's complement sum for the pseudoheader provided by the TCP/IP transport gives the NIC an early start in calculating the real UDP checksum for each packet that the NIC derives from the large UDP packet, without having to examine the IP header.

Note that RFC 768 and RFC 2460 stipulate that the pseudoheader is calculated over the Source IP Address, the Destination IP Address, Protocol, and UDP Length (the length of the UDP header plus the length of the UDP payload, not including the length of the pseudoheader). However, because the underlying miniport driver and NIC generate UDP datagrams from the large packet that is passed down by the TCP/IP transport, the transport does not know the size of the UDP payload for each UDP datagram and thus cannot include the UDP Length in the pseudoheader calculation. Instead, as described in the following section, the NIC extends the pseudoheader checksum that was supplied by the TCP/IP transport to cover the UDP Length of each generated UDP datagram.

ⓘ **Important**

If the UDP header checksum field provided by the TCP/IP transport is zero, the NIC should not perform UDP checksum calculation.

Sending packets with USO

After the miniport driver obtains the **NET_BUFFER_LIST** in its *MiniportSendNetBufferLists* callback function, it can call the **NET_BUFFER_LIST_INFO** macro with an **_Id** of **UdpSegmentationOffloadInfo** to obtain the MSS value and IP protocol.

The miniport driver obtains the total length of the large packet from the length of the first **NET_BUFFER** structure and uses the **MSS** value to divide the large UDP packet into smaller UDP packets. Each of the smaller packets contains **MSS** or fewer user data bytes. Note that only the last packet that was created from the segmented large packet should contain less than **MSS** user data bytes. All other packets that were created from the segmented packet must contain **MSS** user data bytes. If a miniport driver does not adhere to this rule, the UDP datagrams will be incorrectly delivered. If the miniport driver does not set the **SubMssFinalSegmentSupported** capability, then the packet length divides by **MSS** and each of the segmented packets contains **MSS** user bytes.

The miniport driver affixes MAC, IP, and UDP headers to each segment that is derived from the large packet. The miniport driver must calculate the IP and UDP checksums for these derived packets. To calculate the UDP checksum for each packet that was derived from the large UDP packet, the NIC calculates the variable part of the UDP checksum (for the UDP header and UDP payload), adds this checksum to the one's complement sum for the pseudoheader that was calculated by the TCP/IP transport, then calculates the 16-bit one's complement for the checksum. For more information about calculating such checksums, see RFC 768 and RFC 2460.

The length of the UDP user data in the large UDP packet must be less than or equal to the value that the miniport driver assigns to the **MaxOffLoadSize** value.

After a driver issues a status indication to indicate a change to the **MaxOffLoadSize** value, the driver must not cause a bug check if it receives an LSO send request that uses the previous **MaxOffLoadSize** value. Instead, the driver must fail the send request. Drivers **must** fail any send request they can't perform, for any reason (including size, minimum segment count, IP options, etc.). Drivers must send a status indication as soon as possible if their capabilities change.

An intermediate driver that independently issues status indications that report a change in the **MaxOffLoadSize** value must ensure that the underlying miniport adapter that has not issued a status indication does not get any packets that are larger than the **MaxOffLoadSize** value that the miniport adapter reported.

A miniport-intermediate driver that responds to OID_TCP_OFFLOAD_PARAMETERS to turn off USO services must be prepared for a small window of time where USO requests could still reach the miniport driver.

The number of segmentation packets that were dervied from the large UDP packet must be equal to or greater than the **MinSegmentCount** value that is specified by the miniport driver.

When processing a large UDP packet, the miniport driver is responsible only for segmenting the packet and affixing MAC, IP, and UDP headers to the packets that are derived from the large UDP packet. If the miniport fails to send at least one segmented packet, the NBL must eventually be completed with a failure status. The miniport can continue sending subsequent packets but is not required to do so. The NBL cannot be completed back to NDIS until all segmented packets have transmitted or failed.

USO-capable miniport drivers must also do the following:

- Support both IPv4 and IPv6.
- Support replication of IPv4 options from the large packet in each segmented packet that the NIC generates.
- Use the IP and UDP header in the NET_BUFFER_LIST structure as a template to generate UDP and IP headers for each segmented packet.
- Use IP identification (IP ID) values in the range from **0x0000** to **0xFFFF**. For example, if the template IP header starts with an Identification field value of **0xFFFE**, the first UDP datagram packet must have a value of **0xFFFE**, followed by **0xFFFF**, **0x0000**, **0x0001**, and so on.
- If the large UDP packet contains IP options, the miniport driver copies these options, unaltered, to each packet that is derived from the large UDP packet.
- Use the byte offset in the **UdpHeaderOffset** member of NDIS_UDP_SEGMENTATION_OFFLOAD_NET_BUFFER_LIST_INFO to determine the location of the UDP header, starting from the first byte of the packet.
- Increment transmit statistics based on the segmented packets. For example, include the count of Ethernet, IP, and UDP header bytes for each packet segment, and the packet count is the number of **MSS**-sized segments, not **1**.
- Set the UDP total length and IP length fields based on each segmented datagram size.

NDIS interface changes

This section describes the changes in NDIS 6.83 that enable the host TCP/IP driver stack to harness the USO capabilities exposed by miniport drivers.

NDIS and the miniport driver perform the following:

- Advertise that the NIC supports USO capability
- Enable or disable USO

- Get the current USO functionality state

Advertising USO capability

Miniport drivers advertise USO capability by filling in the **UdpSegmentation** field of the NDIS_OFFLOAD structure, which is passed in the parameters of NdisMSetMiniportAttributes. The **Header.Revision** field in the **NDIS_OFFLOAD** structure must be set to **NDIS_OFFLOAD_REVISION_6** and the **Header.Size** field must be set to **NDIS_SIZEOF_NDIS_OFFLOAD_REVISION_6**.

Querying USO state

The current USO state can be queried with OID_TCP_OFFLOAD_CURRENT_CONFIG. NDIS handles this OID and does not pass it down to the miniport driver.

Changing USO state

USO can be enabled or disabled using OID_TCP_OFFLOAD_PARAMETERS. After the miniport driver processes the OID, it must send an NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG status indication with the updated offload state.

USO keywords

The USO enumeration keywords are as follows:

- ***UsoIPv4**
- ***UsoIPv6**

These values describe whether USO is enabled or disabled for that particular IP protocol. The USO settings are not dependent on the NDIS_TCP_IP_CHECKSUM_OFFLOAD configuration. For example, disabling ***UDPChecksumOffloadIPv4** does not implicitly disable ***UsoIPv4**.

Subkey name	Parameter description	Value	Enum description
***UsoIPv4**	UDP Segmentation Offload (IPv4)	0	Disabled
		1	Enabled
***UsoIPv6**	UDP Segmentation Offload (IPV6)	0	Disabled
		1	Enabled

Connection Offload Overview

Article • 12/15/2021

To increase its performance, the Microsoft TCP/IP transport can offload connections to a NIC that has the appropriate TCP/IP-connection offload capabilities.

The NDIS connection offload interface provides hooks to enable configuration of connection offload services such as TCP chimney offload. For more information about connection offload services in NDIS, see Offloading TCP/IP Connections.

TCP chimney offload services are supported in NDIS 6.0 and later.

This section includes:

- Determining Connection Offload Capabilities
- Reporting a NIC's Connection Offload Capabilities
- Enabling and Disabling Connection Offload Services
- Determining the Current Connection Offload Settings
- Using Registry Values to Enable and Disable Connection Offloading
- Offloading TCP/IP Connections

Determining Connection Offload Capabilities

Article • 12/15/2021

NDIS supports two categories of offload services: TCP/IP offload services that are enhanced forms of the NDIS 5.1 and earlier task offload services and connection offload services.

NDIS provides the offload hardware capabilities and the current configuration of the underlying miniport adapter to protocol drivers in the **NDIS_BIND_PARAMETERS** structure. NDIS provides the task offload hardware capabilities and current configuration of the underlying miniport adapter to filter drivers in the **NDIS_FILTER_ATTACH_PARAMETERS** structure.

Administrative applications use object identifier (OID) queries to obtain TCP/IP offload capabilities of a miniport adapter. However, overlying drivers should avoid using OID queries. Protocol drivers must handle changes in the TCP/IP offload capabilities that underlying drivers report. Miniport drivers can report changes in task offload capabilities in status indications. For a list of status indications, see NDIS TCP/IP Offload Status Indications.

Administrative applications (or overlying drivers) can determine the current connection offload configuration of a NIC by querying the OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG OID. The **NDIS_TCP_CONNECTION_OFFLOAD** structure that is associated with OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG specifies the miniport adapter's current connection-offload configuration.

Reporting a NIC's Connection Offload Capabilities

Article • 12/15/2021

An NDIS miniport driver specifies the current connection offload configuration of a NIC in an **NDIS_TCP_CONNECTION_OFFLOAD** structure. Miniport drivers must include the current connection offload configuration in the **NDIS_MINIPORT_ADAPTER_OFFLOAD_ATTRIBUTES** structure. Miniport drivers call the **NdisMSetMiniportAttributes** function from the *MiniportInitializeEx* function and pass in the information in NDIS_MINIPORT_TCP_CONNECTION_OFFLOAD_ATTRIBUTES.

Miniport drivers must report changes in the connection offload capabilities. The drivers request the stack to pause and upload all of the connections by issuing an status indication. (For information on NDIS_STATUS_OFFLOAD_PAUSE, see Full TCP Offload.) After any configuration changes are complete, the drivers request the stack to restart and re-query the miniport adapter's offload capabilities by issuing an status indication. (For information on NDIS_STATUS_OFFLOAD_RESUME, see Full TCP Offload.)

In response to a query of OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG, NDIS returns the **NDIS_TCP_CONNECTION_OFFLOAD** structure in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. NDIS uses the information that the miniport driver provided.

For more information about specifying connection offload capabilities, see Initializing an Offload Target in the NDIS 6.0 TCP chimney offload documentation.

Enabling and Disabling Connection Offload Services

Article • 12/15/2021

Protocol drivers enable connection offload services with an object identifier (OID) request.

Note Enabling or disabling task offload services is different than enabling or disabling connection offload services. Miniport drivers activate all of the available task offload services after a protocol driver specifies an encapsulation type.

The TCP/IP transport enables or disables the connection offload capabilities of a network interface card (NIC) by setting the OID_TCP_CONNECTION_OFFLOAD_PARAMETERS OID. In this set operation, the TCP/IP transport passes the NDIS_TCP_CONNECTION_OFFLOAD_PARAMETERS structure in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. (For information on NDIS_TCP_CONNECTION_OFFLOAD_PARAMETERS, see NDIS 6.0 TCP chimney offload documentation.)

For more information about configuring connection offload services, see Initializing an Offload Target in the NDIS 6.0 TCP chimney offload documentation.

Determining the Current Connection Offload Settings

Article • 12/15/2021

Protocol drivers can obtain the connection offload services with an object identifier (OID) request.

To obtain the current connection offload settings of a network interface card (NIC), protocol drivers can query the OID_TCP_CONNECTION_OFFLOAD_PARAMETERS OID.

Using Registry Values to Enable and Disable Connection Offloading

Article • 12/15/2021

When you debug a driver's connection offload functionality, you might find it useful to enable or disable connection offload services with a registry key setting. There are standardized keywords that you can define in INF files and in the registry. For more information about standardized keywords, see Standardized INF Keywords for Network Devices.

The connection offload keywords are defined as follows:

***TCPConnectionOffloadIPv4**
Describes whether the device enabled or disabled the offload of TCP connections over IPv4.

***TCPConnectionOffloadIPv6**
Describes whether the device enabled or disabled the offload of TCP connections over IPv6.

The following table describes the grouped keywords that you can use to configure offload services.

SubkeyName	ParamDesc	Value	EnumDesc
TCPConnectionOffloadIPv4	TCP Connection Offload (IPv4)	0	Disabled
		1 (Default)	Enabled
TCPConnectionOffloadIPv6	TCP Connection Offload (IPv6)	0	Disabled
		1 (Default)	Enabled

Offloading TCP/IP Connections

Article • 12/15/2021

Offloading TCP/IP connections is supported in NDIS 6.0 and later.

The NDIS TCP/IP connection offload interface enables services such as TCP chimney offload.

For more information about offloading TCP/IP connections, see Overview of TCP Chimney Offload in the Full TCP Offload.

Task offload OIDs

Article • 12/15/2021

The following table summarizes the OIDs that support TCP/IP task offload operations. For more info about such operations, see TCP/IP Task Offload.-ndis-status-dot11-wfd-group-operating-channel.md

In this table, M indicates an OID is mandatory, while O indicates it is optional.

Length	Query	Set	Name
Arr		M	OID_TCP_TASK_IPSEC_ADD_SA
Arr		M	OID_TCP_TASK_IPSEC_ADD_UDPESP_SA
4		M	OID_TCP_TASK_IPSEC_DELETE_SA
4		M	OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA
Arr	M	M	OID_TCP_TASK_OFFLOAD

NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication to notify NDIS and overlying drivers that there has been a change in the task offload configuration of a NIC.

Remarks

Miniport drivers must report the current capabilities with the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication when current capabilities change. This status indication ensures that all of the overlying protocol drivers are updated with the new capabilities information. Miniport drivers are required to issue this status indication in the following cases:

1. When a miniport driver receives an OID_TCP_OFFLOAD_PARAMETERS set request, it must use the contents of the **NDIS_OFFLOAD_PARAMETERS** structure to update the currently-enabled task offload capabilities.
2. When a miniport driver receives an OID_OFFLOAD_ENCAPSULATION set request, it must use the contents of the **NDIS_OFFLOAD_ENCAPSULATION** structure to update the currently-enabled task offload capabilities.

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains an **NDIS_OFFLOAD** structure. When issuing the **NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG** status indication, a miniport driver must use the **NDIS_OFFLOAD** structure to report the current task offload configuration of the NIC.

Note The contents of the **NDIS_OFFLOAD** structure reflect only the NIC's current task offload configuration, not its actual hardware capabilities.

For more information about the current task offload configuration, see OID_TCP_OFFLOAD_CURRENT_CONFIG.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_OFFLOAD

NDIS_OFFLOAD_ENCAPSULATION

NDIS_OFFLOAD_PARAMETERS

NDIS_STATUS_INDICATION

NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES

OID_OFFLOAD_ENCAPSULATION

OID_TCP_OFFLOAD_CURRENT_CONFIG

OID_TCP_OFFLOAD_PARAMETERS

NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES

Article • 03/14/2023

NDIS miniport drivers and MUX intermediate drivers use the **NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES** status indication to notify NDIS and overlying drivers that there has been change in the task offload hardware capabilities of the underlying NIC.

Remarks

If an underlying NIC is added or deleted, the overall set of hardware capabilities that is associated with a miniport driver or MUX intermediate driver can change. For example, if a miniport driver issues the **NDIS_STATUS_TASK_OFFLOAD_HARDWARE_CAPABILITIES** status indication, specifying that it cannot support Large Send Offload (LSO), the NIC can no longer be configured to support LSO.

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains an **NDIS_OFFLOAD** structure. This structure specifies the task offload hardware capabilities.

For more information about task offload hardware capabilities, see OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_OFFLOAD

NDIS_STATUS_INDICATION

NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG

OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES

NDIS_STATUS_OFFLOAD_ENCASPULATION_CHANGE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_OFFLOAD_ENCASPULATION_CHANGE status indication to notify NDIS and overlying drivers that there has been change in the encapsulation settings.

Remarks

The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure contains an NDIS_OFFLOAD_ENCAPSULATION structure. NDIS_OFFLOAD_ENCAPSULATION specifies the encapsulation settings.

For more information about encapsulation settings, see OID_OFFLOAD_ENCAPSULATION.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_OFFLOAD_ENCAPSULATION

NDIS_STATUS_INDICATION

OID_OFFLOAD_ENCAPSULATION

NDIS_STATUS_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES

Article • 03/14/2023

MUX intermediate drivers use the NDIS_STATUS_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES CAPABILITIES status indication to notify NDIS and overlying drivers that there has been change in the connection offload characteristics of the underlying hardware.

Remarks

If an underlying NIC is added or deleted, the overall set of hardware capabilities that is associated with a MUX intermediate driver can change.

The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure contains an **NDIS_TCP_CONNECTION_OFFLOAD** structure. NDIS_TCP_CONNECTION_OFFLOAD specifies the task offload hardware capabilities.

For more information about task offload hardware capabilities, see OID_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NDIS_TCP_CONNECTION_OFFLOAD

OID_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES

NDIS_STATUS_PD_CURRENT_CONFIG

Article • 03/14/2023

This status indication is a notification that the **NDIS_PD_CONFIG** structure has changed.

A PacketDirect-capable miniport driver must make an NDIS_STATUS_PD_CURRENT_CONFIG status indication after an OID_PD_CLOSE_PROVIDER or OID_PD_OPEN_PROVIDER request.

The miniport driver calls **NdisMIndicateStatusEx** to make the status indication, and must pass a pointer to an **NDIS_STATUS_INDICATION** structure through the *StatusIndication* parameter. When making this indication, the miniport driver must set the following members of the **NDIS_STATUS_INDICATION** structure:

- **SourceHandle** must be set to the handle that the miniport received in the *MiniportAdapterHandle* parameter of the *MiniportInitializeEx* function.
- **StatusCode** must be set to NDIS_STATUS_PD_CURRENT_CONFIG.
- **StatusBuffer** must be set to the address of a ULONG variable, which stores the appropriate NDIS_STATUS_xxxx code for the result of the scan operation.
- **StatusBufferSize** must be set to **sizeof**(ULONG).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NdisMIndicateStatusEx

OID_PD_CLOSE_PROVIDER

OID_PD_OPEN_PROVIDER

Virtualized Networking Topics

Article • 07/07/2022

Virtualized networking refers to the NDIS technologies for packet transfer and management within a Hyper-V virtual environment.

This section describes the following components of virtualized networking:

Virtualized Networking Concepts and Terms

Overview of Virtualized Networking

Single Root I/O Virtualization (SR-IOV)

Virtual Machine Queue (VMQ)

Hyper-V Extensible Switch

Virtualized Networking Concepts and Terms

Article • 07/07/2022

The following list gives definitions of key concepts and terms that are used in the Virtualized Networking section. We recommend that you become familiar with these terms before you read the other topics in this section:

Child Partition
In Hyper-V, the child partition is a software-based virtual machine (VM) that has unprivileged access to the physical resources of the host computer.

Each child partition is created through the parent partition. There can be one or more child partitions that run under Hyper-V on the host computer. Each child partition hosts a guest operating system.

In general, child partitions do not have direct access to the physical hardware resources and are presented a virtual view of the resources as virtual devices. Requests to the virtual devices are redirected, either through the VM bus (VMBus) or the hypervisor, to the parent partition where these requests are handled. In addition, child partitions cannot create other partitions.

Note Starting with Windows Server 2012, child partitions do have direct access to the resources of a physical network adapter that supports single root I/O virtualization (SR-IOV).

Emulated Network Adapter
A Hyper-V extensible switch Ethernet adapter that is exposed in the guest operating system that runs in a Hyper-V child partition. An emulated network adapter is a type of VM network adapter. The emulated network adapter mimics an Intel network adapter and uses hardware emulation to forward packets to and from the extensible switch port.

This adapter is exposed in a guest operating system that is Windows XP, Windows Vista, or later versions of Windows. This adapter is also exposed in a guest operating system that is a non-Windows operating system.

External Extensible Switch

A virtual Ethernet switch over which packets are routed between the Hyper-V parent partition, one or more Hyper-V child partitions, and the physical networking interface of the host. This type of switch allows packets to be sent or received between all Hyper-V partitions and the physical network interface on the host.

Also, applications and drivers that run in the management operating system can send or receive packets through this type of switch.

External Network Adapter
A Hyper-V extensible switch Ethernet adapter that is exposed in the management operating system that runs in the Hyper-V parent partition. The external network adapter is bound to one or more physical network adapters on the host.

The external network adapter routes packets between the Hyper-V partitions and the physical network interface on the host.

Note Each instance of an extensible switch supports no more than one external network adapter.

Extensible Switch Team
This is a configuration in which the extensible switch external network adapter is bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX intermediate driver is bound to a team of one or more physical networks on the host.

In this configuration, the extensible switch extensions are exposed to every network adapter in the team. This allows the forwarding extension in the extensible switch driver stack to manage the configuration and use of individual network adapters in the team. For example, the extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. Such an extension is known as a *teaming provider*.

For more information, see NDIS MUX Intermediate Drivers.

Guest Operating System
The operating system that runs in a Hyper-V child partition. Each child partition can host only one operating system. However, many different operating systems can be hosted in child partitions. This includes different versions of Windows and Linux.

Hypervisor
In Hyper-V, the hypervisor is a layer of software that runs between the physical hardware and one or more operating systems that run in Hyper-V partitions.

The hypervisor's main purpose is to provide isolated execution environments called *partitions*. The hypervisor provides each partition with a set of hardware resources, such memory, devices, and CPU cycles. The hypervisor controls and arbitrates access from each partition to the underlying hardware.

Hyper-V Extensible Switch
A virtual Ethernet switch that runs in the management operating system. Each instance

of the extensible switch routes packets between ports that are connected to the Hyper-V extensible switch network adapters.

For more information, see Hyper-V Extensible Switch.

Note The Hyper-V extensible switch is supported in NDIS 6.30 and later versions of NDIS.

Hyper-V Extensible Switch Extension
A Hyper-V extensible switch extension is an NDIS filter driver that attaches to the extensible switch driver stack. Once attached, the extension can capture, filter, or forward network packets and NDIS OIDs. Packets and OIDs can be forwarded to network adapters that are connected to extensible switch ports.

Hyper-V extensible switch extensions are supported in NDIS 6.30 and later versions of NDIS.

Note The Windows Filtering Platform (WFP) provides an in-box extensible switch filtering extension (Wfplwfs.sys). This extension allows WFP filters or callout drivers to intercept packets along the Hyper-V extensible switch data path. This allows the filters or callout drivers to perform packet inspection or modification by using the WFP management and system functions. For an overview of WFP, see Windows Filtering Platform.

Hyper-V Extensible Switch Network Adapter
A network adapter that is managed by the Hyper-V extensible switch. These network adapters connect to ports on the extensible switch, and consist of the following adapter types:

- The external and internal network adapters that are exposed in the management operating system that runs in the Hyper-V parent partition.
- The synthetic or emulated VM network adapters that are exposed in the guest operating system that runs in a Hyper-V child partition.

Internal Extensible Switch
A virtual Ethernet switch over which packets are routed between the Hyper-V parent partition and one or more Hyper-V child partitions. This type of switch excludes packet traffic from the physical network interface on the host.

Also, applications and drivers that run in the management operating system can send or receive packets through this type of switch.

Internal Network Adapter
A Hyper-V extensible switch Ethernet adapter that is exposed in the management

operating system that runs in the Hyper-V parent partition. The internal network adapter sends or receives packets between all Hyper-V partitions. However, the internal network adapter is not bound to a physical networking interface of the host.

I/O Memory Management Unit (IOMMU)
An IOMMU is used to remap physical memory addresses to the addresses that are used by the child partitions. The IOMMU operates independently of the memory management hardware that is used by the processor.

Load Balancing Failover (LBFO) Team
This is a configuration in which the extensible switch external network adapter is bound to the virtual miniport edge of an LBFO provider. The LBFO provider itself can bind to a team of one or more physical network adapters.

In this configuration, the extensible switch extensions are exposed to only the underlying virtual miniport edge as a network adapter. This allows the provider to support an LBFO solution by binding to multiple physical network adapters. These adapters are not managed by a forwarding extension that runs in the extensible switch driver stack.

Management Operating System
The operating system that runs in the Hyper-V parent partition. The parent partition runs the operating system that is running on the host computer. For Hyper-V, the host computer must run x64 versions of Windows Server 2008 or later versions of Windows Server.

Network Virtual Service Client (NetVSC) Driver
An NDIS driver that runs in the guest operating system of a Hyper-V child partition. The NetVSC exposes a virtualized network adapter that is known as a *VM network adapter*.

The NetVSC accesses the Hyper-V extensible switch to forward packets over the network interface managed by the switch. The NetVSC does this by passing messages over the VMBus to the associated NetVSP driver. This driver runs in the management operating system of the Hyper-V parent partition.

In most cases, the NetVSC sends and receives packets by connecting to a port on the Hyper-V extensible switch. However, the NetVSC could be configured to connect to a Virtual Function (VF) of a physical network adapter that supports the SR-IOV interface. In this case, the NetVSC sends and receives packets directly over the underlying physical adapter.

Network Virtual Service Producer (NetVSP) Driver
An NDIS driver that runs in the management operating system of the Hyper-V parent

partition. This driver provides services to support networking access by the Hyper-V child partitions.

NIC Switch
The NIC switch is a hardware component of a network adapter that supports single root I/O virtualization (SR-IOV). This switch bridges network traffic between the adapter's physical network interface and the Physical Function (PF) and one or more VFs on the adapter.

Partition
A partition is managed by the hypervisor. Each partition represents a logical unit of isolated processor and memory resources. This allows multiple isolated operating systems to share a single hardware platform.

The hypervisor also manages policies for memory and device access on the host computer. These policies are different for parent and child partitions.

Parent Partition
In Hyper-V, the parent partition is the first partition on the host computer. This partition has privileged access to the physical resources of the host computer, such as access to memory and devices. In addition, the parent partition is responsible for starting the hypervisor and creating child partitions.

There is only one parent partition that runs under Hyper-V on the host computer. The parent partition hosts the management operating system.

Note The parent partition is also known as the *root* partition.

Physical Function (PF)
A PCI Express (PCIe) function that supports the single root I/O virtualization (SR-IOV) interface. SR-IOV extends the PCIe interface to enable multiple VMs to share the same PCIe physical hardware resources. The PF contains the PCIe SR-IOV Extended Capability structure in its PCI configuration space.

PF/VF Backchannel
A private software-based communication interface between the miniport drivers of a PCIe Virtual Function (VF) and the PCIe Physical Function (PF). Each VF miniport driver can issue requests over the backchannel to the PF miniport driver. The PF miniport driver can issue status notifications over the backchannel to individual VF miniport drivers.

Data exchanged between the PF and VF miniport drivers over the backchannel interface involves the use of a *VF configuration block*. Each VF configuration block is similar in concept to an interprocess communication (IPC) message, in which each block has a proprietary format, length, and block identifier. The independent hardware vendor (IHV) can define one or more VF configuration blocks for the PF and VF miniport drivers.

Private Extensible Switch
A virtual Ethernet switch over which packets are routed between one or more Hyper-V child partitions. This type of switch excludes packet traffic from the Hyper-V parent partition and the physical network interface on the host.

Note Applications and drivers that run in the management operating system cannot send or receive packets through this type of switch.

Single Root I/O Virtualization (SR-IOV)
SR-IOV is a method by which a PCIe network adapter can be partitioned into one Physical Function (PF) and one or more virtual functions (VF). Each function on the adapter is assigned a unique PCIe requester ID. This enables the adapter to apply memory and interrupt translations so that different network traffic streams can be delivered directly to the appropriate PF or VF. By avoiding the routing of network traffic through the Hyper-V extensible switch component, SR-IOV reduces the I/O overhead in the virtualized networking environment.

For more information, see Single Root I/O Virtualization (SR-IOV).

Note SR-IOV is supported in NDIS 6.30 and later versions of NDIS.

Synthetic Data Path
The networking data path between a VM network adapter exposed in a guest operating system and the Hyper-V extensible switch component in the management operating system.

Synthetic Network Adapter
A Hyper-V extensible switch Ethernet adapter that is exposed in the guest operating system that runs in a Hyper-V child partition. A synthetic network adapter is a type of VM network adapter. The network virtual service client (NetVSC) that runs in the VM exposes this synthetic network adapter. NetVSC forwards packets to and from the extensible switch port over the VM bus (VMBus) to the associated NetVSP driver.

This network adapter is exposed in a guest operating system that is Windows Vista or a later version of Windows.

Virtual Function (VF)
A PCIe function that is associated with a PF on a network adapter that supports SR-IOV. A VF shares one or more physical resources on the adapter, such as the physical Ethernet port, with the PF and other VFs that are associated with the same PF.

VF Data Path
The networking data path between a VM network adapter exposed in a guest operating system and the VF on an SR-IOV network adapter. In this data path, the VM network adapter is teamed with the VF network adapter in the guest operating system. The VF miniport driver forwards packets to or from the VM network adapter to the VF. The NIC switch on the SR-IOV network adapter then forwards packets to or from the VF to the physical network interface on the adapter.

VF Network Adapter
The virtual network adapter that is exposed in the guest operating system for the VF. When resources are allocated for the VF and it becomes attached to a child partition, the VPCI bus driver in the guest operating system of that partition exposes the VF network adapter. The VPCI bus driver also loads the VF miniport driver for this adapter.

Virtual Machine (VM)
A virtual guest computer that is implemented in software and is hosted within a physical host computer. A virtual machine emulates a complete hardware system, from processor to network adapter, in a self-contained, isolated software environment. This enables concurrent operation of otherwise incompatible operating systems.

Each guest operating system runs in its own isolated software virtual machine.

Note In Hyper-V, a child partition is also known as a VM.

Virtual Machine Bus (VMBus)
A virtual communications bus that passes control and data messages between the Hyper-V parent and child partitions. Access to the physical resources on the host computer by child partitions is made through messages that are passed over the VMBus between Virtual Service Client (VSC) and Virtual Service Provider (VSP) components.

Virtual Machine (VM) Network Adapter
A Hyper-V extensible switch virtual network adapter that is exposed in the guest operating system of a Hyper-V child partition.

The VM network adapter supports the following virtualization types:

- The VM network adapter could be a synthetic virtualization of a network adapter (*synthetic network adapter*). In this case, the network virtual service client (NetVSC) that runs in the VM exposes this virtual network adapter. NetVSC forwards packets to and from the extensible switch port over the VM bus (VMBus).

- The VM network adapter could be an emulated virtualization of a physical network adapter (*emulated network adapter*). In this case, the VM network adapter mimics an Intel network adapter and uses hardware emulation to forward packets to and from the extensible switch port.

A VM network adapter can be configured to access either the Hyper-V external, internal, or private network interfaces.

Virtual Machine Queue (VMQ)
A VMQ-capable network adapter uses DMA to transfer all incoming frames directly to VM memory. VMQ also improves network throughput by distributing the processing of network traffic for multiple VMs among multiple processors.

For more information, see Virtual Machine Queue (VMQ).

Note VMQ is supported in NDIS 6.20 and later versions of NDIS.

Virtual PCI (VPCI) Driver
The PCI bus driver that runs in the guest operating system of a Hyper-V child partition. This driver exposes the VF as a virtual network adapter in the guest operating system.

The VPCI driver is a Hyper-V VSC and communicates with the VPCI VSP that runs in the management operating system in the Hyper-V parent partition. Communication between the VPCI VSP and VSC components occurs over VMBUS.

For more information about the VPCI interface, see GUID_PCI_VIRTUALIZATION_INTERFACE.

Virtualization Stack
A collection of software components that manages the creation and execution of child partitions under Hyper-V. The virtualization stack manages the access by child partitions to the hardware resources on the host computer. The virtualization stack runs in the Hyper-V parent partition.

Overview of Hyper-V

Article • 12/15/2021

Hyper-V is a hypervisor-based virtualization technology for x64 versions of Windows Server 2008 and later versions of Windows Server. The hypervisor is the processor-specific virtualization platform that allows multiple isolated operating systems to share a single hardware platform.

Hyper-V supports isolation through separate *partitions*. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute. The virtualization stack runs in the management operating system of the Hyper-V parent partition, and has direct access to the hardware devices. The management operating system then creates the Hyper-V child partitions and starts the guest operating systems within them.

Partitions do not have access to the physical processor, nor do they handle the processor interrupts. Instead, they have a virtual view of the processor and run in a virtual memory address region that is private to each guest partition. The hypervisor handles the interrupts to the processor, and redirects them to the respective partition. Hyper-V can also hardware accelerate the address translation between various guest virtual address spaces by using an I/O memory management unit (IOMMU) which operates independently of the memory management hardware used by the processor. An IOMMU is used to remap physical memory addresses to the addresses that are used by the child partitions.

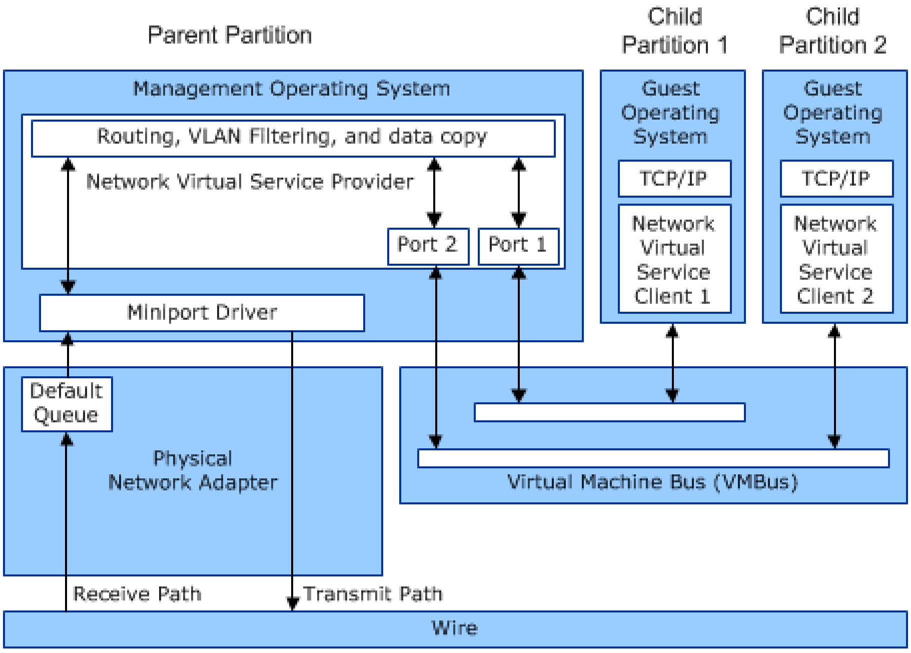
Child partitions also do not have direct access to other hardware resources. Instead, child partitions are presented a virtual view of the resources, known as *virtual devices*. Requests to the virtual devices are redirected either through the virtual machine bus (VMBus) or the hypervisor to the management operating system in the parent partition, which handles the device requests. The VMBus is a logical inter-partition communication channel, with separate channels allocated for communication between the parent partition and a child partition.

The management operating system hosts virtual service providers (VSPs) that communicate over the VMBus to handle device access requests from child partitions. The guest operating system on a child partition hosts virtual service clients (VSCs) that redirect device requests to VSPs in the management operating system by using the VMBus.

For network access to child partitions, a Network VSC (NetVSC) runs in a guest operating system. Networking requests and packets are sent between each NetVSC and the Network VSP that runs in the management operating system. The NetVSC also exposes a virtualized view of the physical network adapter on the host computer. This virtualized network adapter is known as a *synthetic network adapter*.

Note Hyper-V also supports another less-efficient virtualized network adapter that is known as an *emulated network adapter*. An emulated network adapter mimics an Intel network adapter and uses hardware emulation to forward packets to and from the NetVSP.

The following figure shows the networking data paths in Hyper-V over synthetic network adapters.



These data paths are extended by using NDIS virtualized networking interfaces, such as the virtual machine queue (VMQ), single root I/O virtualization (SR-IOV), or Hyper-V extensible switch interfaces. For example, the NetVSC could be configured to connect to a Virtual Function (VF) of a physical network adapter that supports the SR-IOV interface. In this case, the NetVSC sends and receives packets directly over the underlying physical adapter and not over the VMBus.

For more information about Hyper-V, see Hyper-V.

Single Root I/O Virtualization (SR-IOV) Interface

Article • 12/15/2021

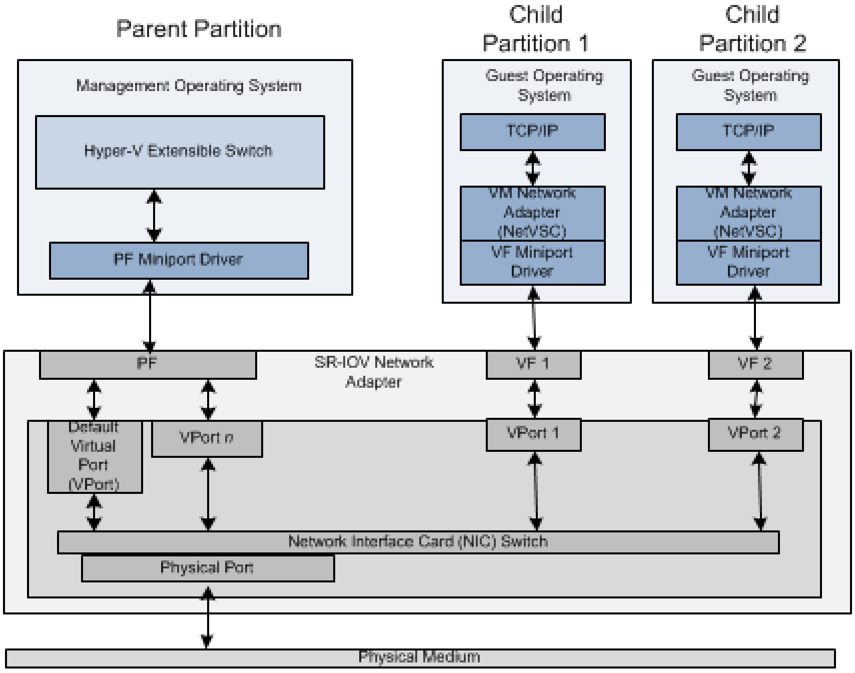
The SR-IOV interface allows for the partitioning of the hardware resources on a PCI Express (PCIe) network adapter into one or more virtual interfaces, known as *virtual functions (VFs)*. This allows the adapter resources to be shared in a virtual environment. SR-IOV enables network traffic to bypass the virtual software switch layer by assigning a VF to the Hyper-V child partition directly. By doing this, the I/O overhead in the software emulation layer is diminished and network throughput achieves nearly the same performance as in nonvirtualized environments.

Each PCIe VF is assigned a unique Requester ID, which allows an I/O memory management unit (IOMMU) to do the following:

- Distinguish between different traffic streams on each PCIe function of the network adapter. This allows the IOMMU to apply memory and interrupt translations so that these traffic streams can be delivered directly to the appropriate child or parent partition.

- Isolate traffic flows between partitions. This guarantees that traffic flow from a partition does not affect other partitions on the device.

The following figure shows the VF data path within the SR-IOV interface.



The use of the VF data path provides the following benefits:

- All data packets flow directly between the protocol stacks in the guest operating system and the VF. This eliminates the overhead of the synthetic data path in which data packets flow between the Hyper-V child and parent partitions. Once forwarded to the parent partition, the Hyper-V extensible switch module forwards these packets to other child partitions or to the physical network interface on the underlying SR-IOV physical adapter.

- The VF data path bypasses any involvement by the management operating system in packet traffic from a Hyper-V child partition. The VF provides independent memory space, interrupts and DMA streams for the child partition to which it is attached. This achieves networking performance that is almost compatible with nonvirtualized environments.

- The routing of packets over the VF data path is performed by the NIC switch on the SR-IOV network adapter. Packets are sent or received over the external network through the physical port of the adapter. Packets are also forwarded to or from other child partitions to which a VF is attached.

  Note Packets to or from child partitions to which no VF is attached are forwarded by the NIC switch to the extensible switch module. This module runs in the Hyper-V parent partition and delivers these packets to the child partition by using the synthetic data path.

For more information about the SR-IOV interface, see Single Root I/O Virtualization (SR-IOV).

Virtual Machine Queue (VMQ) Interface

Article • 09/27/2024

A network adapter that supports the VMQ interface includes hardware that routes packets to receive queues. This requires parsing of the packet header and configuration of the queues on the network adapter.

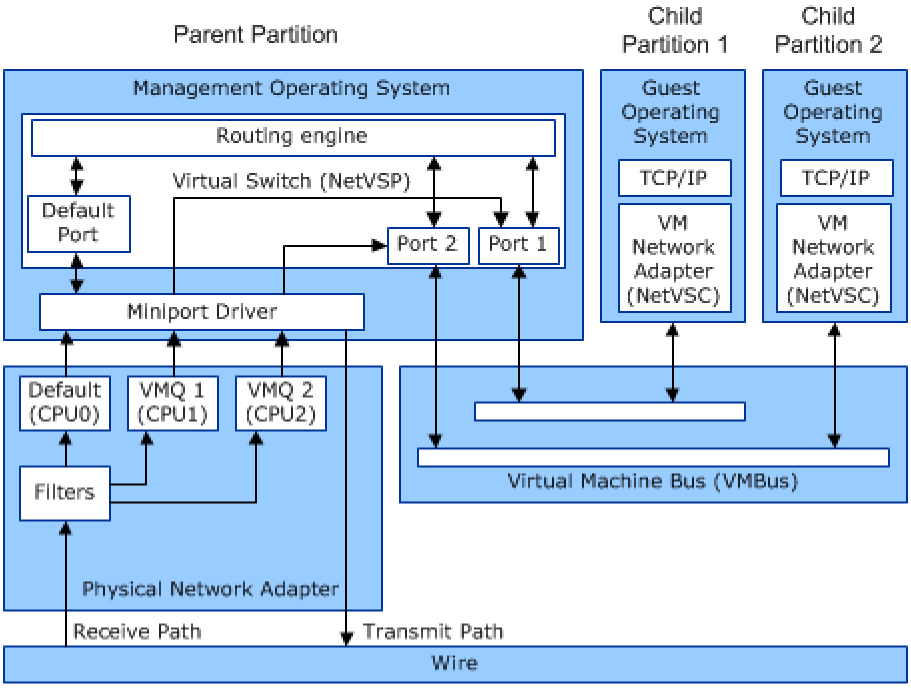
When a miniport driver makes a receive indication, all of the packets are for the same VM queue.

As an option, the network adapter can provide VLAN filtering in hardware for a specified media access control (MAC) address.

Routing the packets to queues and indicating all the packets on a queue to a VM allows concurrent receive processing for multiple VMs. Every queue is serviced by a different processor.

Routing to queues in the network adapter prevents a copy step to copy data from the network adapter receive buffers to the VM address space.

The following figure shows the synthetic data paths within the VMQ interface.



In the figure, the miniport driver for the physical network adapter indicates received data up to the Hyper-V extensible switch component. This component acts as a network virtual service provider (NetVSP) and provides services to support networking access by the Hyper-V child partitions.

The services that the extensible switch provides includes routing packets to and from the virtual machine (VM) network adapters in the guest operating systems. The VM network adapter is exposed by the network virtual service client (NetVSC) that runs in the guest operating system.

Under VMQ, the physical network adapter transfers the data that matches a receive filter test for a VMQ directly to that queue. This prevents software processing in the extensible switch. Data that does not pass any filter tests goes to the default queue where the extensible switch must process the data. In addition to preventing the routing and copying in the extensible switch, the receive interrupts for VM queues are assigned to different processors.

For more information about the VMQ interface, see Virtual Machine Queue (VMQ).

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Hyper-V Extensible Switch Interface

Article • 12/15/2021

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

A Hyper-V extensible switch is a virtual Ethernet switch that runs in the management operating system of the Hyper-V parent partition. Each instance of the extensible switch routes packets between the physical network interface in the host and the virtual network interfaces that are configured for the Hyper-V child partitions. These virtual network interfaces include Hyper-V external, internal, and private network interfaces.

Starting with NDIS 6.30 in Windows Server 2012, the extensible switch module supports an interface that allows NDIS filter drivers (known as *extensible switch extensions*) to bind within the extensible switch driver stack. This allows extensions to monitor, modify, and forward packets to extensible switch ports. This also allows extensions to inspect and inject packets in the virtual network interfaces that are used by the various Hyper-V partitions.

Extensions can be configured with switch and port policies to apply to packets that are routed through the extensible switch data path. This allows the driver to allow or deny a packet from being sent or received over a port.

In the extensible switch interface, the filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

The extensible switch interface supports the following types of extensions:

Capturing Extension
An extension that captures and monitors packet traffic. This type of extension cannot modify packets or packet destinations through the extensible switch. However, capturing extensions can originate packet traffic, such as packets that contain traffic statistics that the extension sends to a host application.

For more information, see Capturing Extensions.

Filtering Extension
An extension that captures and monitors packet traffic. This type of extension can also inspect and reject packet delivery based on custom port or switch policy settings.

For more information, see Filtering Extensions.

Forwarding Extension
An extension that has the same capabilities as a filtering extension. This type of extension can determine the extensible switch destination ports that a packet is delivered to, as well as inject packet traffic to any extensible switch port. This type of extension also inspects and rejects packet delivery based on standard port policy settings.

For more information, see Forwarding Extensions.

Note In NDIS 6.40 (Windows Server 2012 R2) and later, forwarding extensions must support Hybrid Forwarding.

Note If a forwarding extension is not installed and enabled in the extensible switch, the switch determines a packet's destination ports as well as filters packets based on standard port settings.

For more information about the extensible switch interface, see Hyper-V Extensible Switch.

Potential Performance Bottlenecks in an NDIS Virtualized Networking Environment

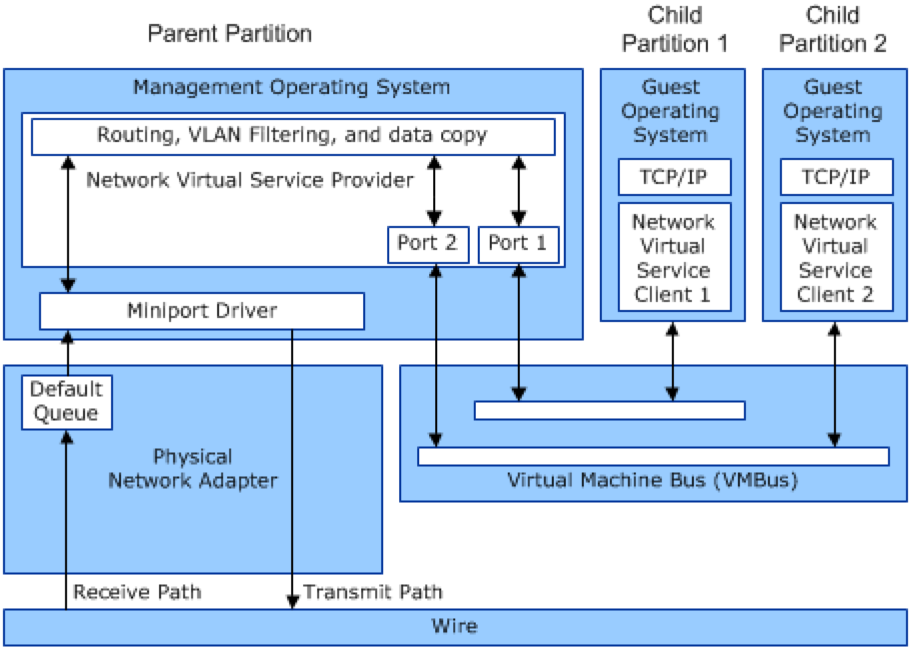
Article • 07/07/2022

In a networking environment that supports device sharing, a virtual interface with its own media access control (MAC) address is exposed in each Hyper-V child partition. These virtual interfaces use the underlying virtual machine bus (VMBus) to connect to a port on the Hyper-V extensible switch module that runs in the management operating system of the Hyper-V parent partition. The extensible switch transmits all the outgoing frames from the different partitions by issuing send requests to the underlying shared network adapter.

The physical network adapter indicates all incoming frames that it receives up to the extensible switch in the management operating system. The extensible switch uses the destination MAC address to assign the indicated incoming frames to virtual network adapters. For each child partition, the incoming frames must be copied from network adapter buffers in the management operating system to secondary buffers that were preallocated from the associated child partition. A notification is sent to each virtual interface that has pending frames. The virtual interface in each child partition indicates the incoming frames to overlying transport drivers. After identifying the application, transport drivers copy the data payload from the secondary buffer to an application buffer.

Therefore, in the virtualization environment, incoming received frames are copied twice, first from the network adapter buffer to a temporary buffer that is allocated from the target memory address space of a child partition, and then again from this temporary buffer to an application buffer. The extensible switch in the management operating system must use CPU cycles to parse incoming frames and place them in separate queues based on their destination MAC address.

The following figure shows the performance bottlenecks for receive processing in a virtualized environment.



The performance issues in the previous figure include the following:

- Each incoming packet must be examined to identify the target virtual network adapter.
- Received data must be copied from the parent partition's memory address space to the child partition's memory address space.
- Lack of concurrency for interrupts and DPCs.

Overcoming Performance Bottlenecks with VMQ

To address performance issues, the virtual machine queue (VMQ) interface allows:

- A network adapter to determine the target child partition by implementing MAC address filtering in hardware.
- A network adapter to use DMA to transfer received packets directly to a child partition's memory address space.
- A miniport driver to provide interrupt and DPC concurrency by indicating received packets for different child partitions on different CPUs.

Note Packets that are received from the external network still have to be forwarded by the management operating system to the guest operating system over the VMBus.

For more information about the VMQ interface, see Virtual Machine Queue (VMQ).

Overcoming Performance Bottlenecks with SR-IOV

The single root I/O virtualization (SR-IOV) interface provides a standards-based foundation for efficiently sharing a PCI Express (PCIe) device among multiple child partitions. Physical I/O resources are virtualized within the PCIe device, so each device presents multiple virtual I/O interfaces called *virtual functions* (VFs). The management operating system can configure each VF on the device and assign them to particular child partitions.

A VF is exposed as a hardware device to the child partition. All data packets flow directly between the guest operating system and the VF. This eliminates the software path between the management operating system and the child partition for data traffic. It bypasses the involvement by the management operating system in data movement by providing independent memory space, interrupts, and DMA streams for each child partition. Frames are sent to the external network by using the physical port of the device or to another child partition by using the internal port connected to the VF.

In all cases, the SR-IOV interface eliminates the need for any involvement of the management operating system in the data path. As a result, the SR-IOV interface provides the following:

- Improved I/O throughput and reduced CPU utilization.
- Lower latency.
- Improved scalability.

For more information about the SR-IOV interface, see Single Root I/O Virtualization (SR-IOV).

Introduction to Single Root I/O Virtualization (SR-IOV)

Article • 12/15/2021

This section describes the NDIS single root I/O virtualization (SR-IOV) interface. Starting with NDIS 6.30, the SR-IOV interface supports Microsoft Hyper-V performance improvements for virtualized networks on Windows Server 2012 and later versions of Windows Server.

The SR-IOV specification from PCI-SIG defines the extensions to the PCI Express (PCIe) specification suite that enable multiple virtual machines (VMs) to share the same PCIe physical hardware resources. This section describes the NDIS SR-IOV interface and describes the techniques for writing an NDIS miniport driver for an SR-IOV capable network adapter that implements the PCIe SR-IOV specification.

This section includes the following topics:

Overview of Single Root I/O Virtualization (SR-IOV)

Writing SR-IOV PF Miniport Drivers

Writing SR-IOV VF Miniport Drivers

SR-IOV PF/VF Backchannel Communication

SR-IOV OIDs

For more information on SR-IOV, refer to the PCI-SIG Single Root I/O Virtualization and Sharing 1.1 specification.

Overview of Single Root I/O Virtualization (SR-IOV)

Article • 09/27/2024

The single root I/O virtualization (SR-IOV) interface is an extension to the PCI Express (PCIe) specification. SR-IOV allows a device, such as a network adapter, to separate access to its resources among various PCIe hardware functions. These functions consist of the following types:

- A PCIe Physical Function (PF). This function is the primary function of the device and advertises the device's SR-IOV capabilities. The PF is associated with the Hyper-V parent partition in a virtualized environment.

- One or more PCIe Virtual Functions (VFs). Each VF is associated with the device's PF. A VF shares one or more physical resources of the device, such as a memory and a network port, with the PF and other VFs on the device. Each VF is associated with a Hyper-V child partition in a virtualized environment.

Each PF and VF is assigned a unique PCI Express Requester ID (RID) that allows an I/O memory management unit (IOMMU) to differentiate between different traffic streams and apply memory and interrupt translations between the PF and VFs. This allows traffic streams to be delivered directly to the appropriate Hyper-V parent or child partition. As a result, nonprivileged data traffic flows from the PF to VF without affecting other VFs.

SR-IOV enables network traffic to bypass the software switch layer of the Hyper-V virtualization stack. Because the VF is assigned to a child partition, the network traffic flows directly between the VF and child partition. As a result, the I/O overhead in the software emulation layer is diminished and achieves network performance that is nearly the same performance as in nonvirtualized environments.

For more information, see the following topics:

SR-IOV Architecture

SR-IOV Data Paths

Feedback

Was this page helpful? Yes No

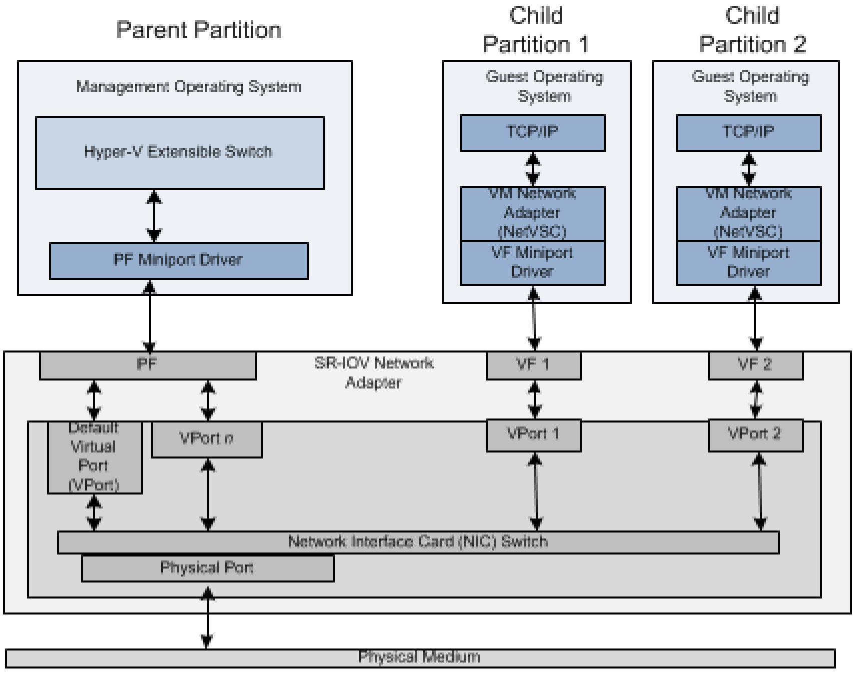
Provide product feedback ↗ | Get help at Microsoft Q&A

SR-IOV Architecture

Article • 06/28/2024

This section provides a brief overview of the single root I/O virtualization (SR-IOV) interface and its components.

The following figure shows the components of the SR-IOV starting with NDIS 6.30 in Windows Server 2012.



The SR-IOV interface consists of the following components:

Hyper-V Extensible Switch Module
The extensible switch module that configures the NIC switch on the SR-IOV network adapter to provide network connectivity to the Hyper-V child partitions.

Note Hyper-V child partitions are known as *virtual machines (VMs)*.

If the child partitions are connected to a PCI Express (PCIe) Virtual Function (VF), the extensible switch module does not participate in data traffic between the VM and the network adapter. Instead, data traffic is passed directly between the VM and the VF to which it is attached.

For more information about the extensible switch, see Hyper-V Extensible Switch.

Physical Function (PF)
The PF is a PCI Express (PCIe) function of a network adapter that supports the SR-IOV interface. The PF includes the SR-IOV Extended Capability in the PCIe Configuration

space. The capability is used to configure and manage the SR-IOV functionality of the network adapter, such as enabling virtualization and exposing VFs.

For more information, see SR-IOV Physical Function (PF).

PF Miniport Driver
The PF miniport driver is responsible for managing resources on the network adapter that are used by one or more VFs. Because of this, the PF miniport driver is loaded in the management operating system before any resources are allocated for a VF. The PF miniport driver is halted after all resources that were allocated for VFs are freed.

For more information, see Writing SR-IOV PF Miniport Drivers.

Virtual Function (VF)
A VF is a lightweight PCIe function on a network adapter that supports the SR-IOV interface. The VF is associated with the VF on the network adapter, and represents a virtualized instance of the network adapter. Each VF has its own PCI Configuration space. Each VF also shares one or more physical resources on the network adapter, such as an external network port, with the PF and other VFs.

For more information, see SR-IOV Virtual Functions (VFs).

VF Miniport Driver
The VF miniport driver is installed in the VM to manage the VF. Any operation that is performed by the VF miniport driver must not affect any other VF or the PF on the same network adapter.

For more information, see Writing SR-IOV VF Miniport Drivers.

Network Interface Card (NIC) Switch
The NIC switch is a hardware component of the network adapter that supports the SR-IOV interface. The NIC switch forwards network traffic between the physical port on the adapter and internal virtual ports (VPorts). Each VPort is attached to either the PF or a VF.

For more information, see NIC Switches.

Virtual Ports (VPorts)
A VPort is a data object that represents an internal port on the NIC switch of a network adapter that supports the SR-IOV interface. Similar to a port on a physical switch, a VPort on the NIC switch delivers packets to and from a PF or VF to which the port is attached.

For more information, see NIC Switches.

Physical Port
The physical port is a hardware component of the network adapter that supports the SR-IOV interface. The physical port provides the interface on the adapter to the external networking medium.

SR-IOV Physical Function (PF)

Article • 09/27/2024

The Physical Function (PF) is a PCI Express (PCIe) function of a network adapter that supports the single root I/O virtualization (SR-IOV) interface. The PF includes the SR-IOV Extended Capability in the PCIe Configuration space. The capability is used to configure and manage the SR-IOV functionality of the network adapter, such as enabling virtualization and exposing PCIe Virtual Functions (VFs).

The PF is exposed as a virtual network adapter in the management operating system of the Hyper-V parent partition. The PF miniport driver is an NDIS miniport driver that manages the PF in the management operating system. The configuration and provisioning of the VFs, together with other hardware and software resources for the support of VFs, is performed through the PF miniport driver. The PF miniport driver uses the traditional NDIS miniport driver functionality to provide the access to the networking I/O resources to the management operating system. The PF driver is also used as a way to manage the resources allocated on the adapter for the VFs.

The PF supports the SR-IOV Extended Capability structure in its PCIe configuration space. This structure is defined in the PCI-SIG Single Root I/O Virtualization and Sharing 1.1 specification. This structure includes the following members:

TotalVFs
A read-only field that specifies the maximum number of VFs that can be associated with the PF.

NumVFs
A read-write field that specifies the current number of VFs that are available on the SR-IOV network adapter.

SR-IOV Control
A read-write field that specifies various control bits that enable or disable SR-IOV functionality on the network adapter. For example, if the **VF Enable** bit is set to one, VFs can be associated with the PF on the adapter. If this bit is set to zero, VFs are disabled and not visible on the adapter.

The PF also provides the mechanism for the management operating system to communicate with the external physical network. The PF provides network connectivity to the all virtual network adapters that are connected to the Hyper-V extensible switch module. This includes the following:

- Virtual network adapters that provide network connectivity to the Hyper-V parent partition.

- Virtual network adapters that provide network connectivity to the Hyper-V child partitions that do not have VFs allocated to them.

The PF miniport driver is responsible for managing resources on the network adapter that are used by one or more VFs. Because of this, the PF miniport driver is loaded in the management operating system before any resources are allocated for a VF. The PF miniport driver is halted after all resources that were allocated for VFs are freed.

SR-IOV Virtual Functions (VFs)

Article • 09/27/2024

A PCI Express (PCIe) Virtual Function (VF) is a lightweight PCIe function on a network adapter that supports single root I/O virtualization (SR-IOV).

The VF is associated with the PCIe Physical Function (PF) on the network adapter, and represents a virtualized instance of the network adapter. Each VF has its own PCI Configuration space. Each VF also shares one or more physical resources on the network adapter, such as an external network port, with the PF and other VFs.

A VF is not a full-fledged PCIe device. However, it provides a basic mechanism for directly transferring data between a Hyper-V child partition and the underlying SR-IOV network adapter. Software resources associated for data transfer are directly available to the VF and are isolated from use by the other VFs or the PF. However, the configuration of most of these resources is performed by the PF miniport driver that runs in the management operating system of the Hyper-V parent partition.

A VF is exposed as a virtual network adapter (*VF network adapter*) in the guest operating system that runs in a Hyper-V child partition. After the VF is associated with a virtual port (VPort) on the NIC switch of the SR-IOV network adapter, the virtual PCI (VPCI) driver that runs in the VM exposes the VF network adapter. Once exposed, the PnP manager in the guest operating system loads the VF miniport driver.

> ⓘ **Note**
>
> A Hyper-V child partition is also known as a *virtual machine (VM)*.

The VF miniport driver is an NDIS miniport driver that is installed in the VM to manage the VF. Any operation that is performed by the VF miniport driver must not affect any other VF or the PF on the same network adapter.

The VF miniport driver can function like any PCI device driver. It can read and write to the VF's PCI configuration space. However, access to the virtual PCI device is a privileged operation and is managed by the PF miniport driver in the following way:

- When the VF miniport driver calls NdisMGetBusData to read data from the PCI configuration space of the VF network adapter, the virtualization stack is notified. This stack runs in the management operating system of the Hyper-V parent partition. When the stack is notified of the read request, it issues an object identifier (OID) method request of OID_SRIOV_READ_VF_CONFIG_SPACE to the PF miniport driver. The data to be read is specified in an **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure that is contained in the OID request.

The driver reads the requested data from the VF PCI configuration space and returns the data by completing the OID request. This data is then returned to the VF miniport driver when the call to **NdisMGetBusData** completes.

- When the VF miniport driver calls **NdisMSetBusData** to write data to the PCI configuration space of the VF network adapter, the virtualization stack is notified of the write request. It issues an OID method request of OID_SRIOV_WRITE_VF_CONFIG_SPACE to the PF miniport driver. The data to be written is specified in an **NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS** structure that is contained in the OID request.

  The driver writes the data to the VF PCI configuration space and returns the status of the request when it completes the OID request. This status is returned to the VF miniport driver after the call to **NdisMSetBusData** completes.

The VF miniport driver may also communicate with the PF miniport driver. This communication path is over a backchannel interface. For more information, see SR-IOV PF/VF Backchannel Communication.

Note The VF miniport driver must be aware that it is running in a virtualized environment so that it can communicate with the PF miniport driver for certain operations. For more information on how the driver does this, see Initializing a VF Miniport Driver.

Feedback

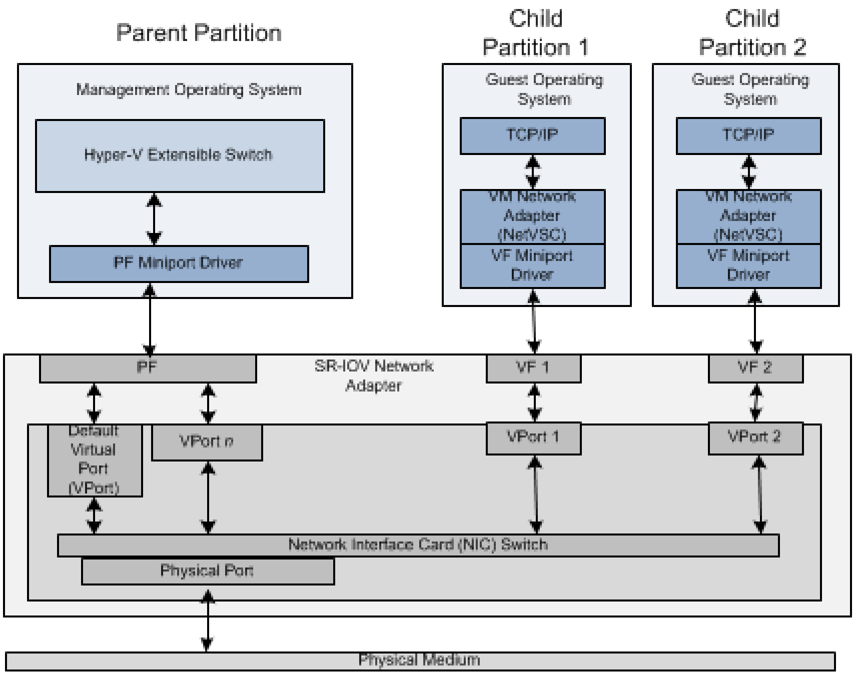
Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

NIC Switches

Article • 12/15/2021

A network adapter that supports single root I/O virtualization (SR-IOV) must implement a hardware bridge that forwards network traffic between the physical port on the adapter and internal virtual ports (VPorts). This bridge is known as the *NIC switch* and is shown in the following figure.



Each NIC switch contains the following components:

- One external, or *physical*, port that provides network connectivity to the external physical network.

- One internal port that provides the PCI Express (PCIe) Physical Function (PF) on the network adapter with access to the external physical network. An internal port is known as a *virtual port (VPort)*.

  The PF always has a VPort that is created and assigned to it. This VPort is known as the *default VPort*, and is referenced by the DEFAULT_VPORT_ID identifier.

  For more information about VPorts, see Virtual Ports (VPorts).

- One or more VPorts that provide a PCIe Virtual Function (VF) on the network adapter with access to the external physical network.

  Note Additional VPorts can be created and allocated to the PF for network access.

Note Starting with NDIS 6.30 in Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC* *switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

The hardware resources for the NIC switch are managed by the PF miniport driver for the SR-IOV network adapter. The driver creates and configures the NIC switch through one of the following methods:

- Static creation based on standardized SR-IOV and NIC switch INF keywords. For more information on these keywords, see Standardized INF Keywords for SR-IOV.

- Dynamic creation based on object identifier (OID) method requests of OID_NIC_SWITCH_CREATE_SWITCH. NDIS or the Hyper-V extensible switch module issues these OID requests to create NIC switches on the SR-IOV network adapter.

For more information on how NIC switches are created, configured, and managed, see Managing NIC Switches.

Virtual Ports (VPorts)

Article • 03/09/2022

A virtual port (VPort) is a data object that represents an internal port on the NIC switch of a network adapter that supports single root I/O virtualization (SR-IOV). Each NIC switch has the following ports for network connectivity:

- One external physical port for connectivity to the external physical network.

- One or more internal VPorts which are connected to the PCI Express Physical Function (PF) or virtual functions (VFs).

  The PF is attached to the Hyper-V parent partition and is exposed as a virtual network adapter in the management operating system that runs in that partition.

  A VF is attached to the Hyper-V child partition and is exposed as a virtual network adapter in the guest operating system that runs in that partition.

The NIC switch bridges network traffic from the physical port to one or more VPorts. This provides virtualized access to the underlying physical network interface.

Each VPort has a unique identifier (*VPortId*) that is unique for the NIC switch on the network adapter. A default VPort always exists on the default NIC switch and can never be deleted. The default VPort has the VPortId of NDIS_DEFAULT_VPORT_ID.

When the PF miniport driver handles an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_SWITCH, it creates the NIC switch and the default VPort for that switch. The default VPort is always attached to the PF and is always in an operational state.

Nondefault VPorts are created through OID method requests of OID_NIC_SWITCH_CREATE_VPORT. Only one nondefault VPort can be attached to a VF. Once attached, the default is in an operational state. One or more nondefault VPorts can also be created and attached to the PF. These VPorts are nonoperational when created and can become operational through an OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS.

> ⓘ **Note**
>
> After a VPort becomes operational, it can only become nonoperational when it is deleted through an OID request of **OID_NIC_SWITCH_DELETE_VPORT**.

Each VPort has one or more hardware queue pairs associated with it for receiving and transmitting packets. The default queue pair on the network adapter is reserved for use by the default VPort. Queue pairs for nondefault VPorts are allocated and assigned when the VPort is created through the OID_NIC_SWITCH_CREATE_VPORT request.

Nondefault VPorts are created and configured through OID method requests of OID_NIC_SWITCH_CREATE_VPORT. The default VPort and nondefault VPorts are reconfigured through OID set requests of OID_NIC_SWITCH_VPORT_PARAMETERS. Each OID request contains an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure that specifies the following configuration parameters:

- The PCIe function to which the VPort is attached.

  Each VPort can be either attached to the PF or with a VF at any time. After the VPort is created and attached to a PCIe function, the attachment cannot be dynamically changed to another PCIe function.

  > ⓘ **Note**
  >
  > The default VPort is always attached to the PF on the network adapter.

Starting with NDIS 6.30 in Windows Server 2012, only one nondefault VPort can be attached to a VF. However, multiple nondefault VPorts along with the default VPort can be attached to the PF.

- The number of hardware queue pairs that are assigned to a VPort.

  Each VPort has a set of hardware queue pairs that are available to it. Each queue pair consists of a separate transmit and receive queue on the network adapter.

  Queue pairs are limited resources on the network adapter. The total number of queue pairs reserved for use by the default and nondefault VPorts is specified when the NIC switch is created. This allows the number of queue pairs that are assigned to the default VPort to differ from the nondefault VPorts.

  Each nondefault VPort can be configured to have a different number of queue pairs. This is known as *asymmetric allocation* of queue pairs. If the NIC does not allow for such an asymmetric allocation, each nondefault VPort is configured to have equal number of queue pairs. This is known as *symmetric allocation* of queue pairs. For more information, see Symmetric and Asymmetric Assignment of Queue Pairs.

  > ⓘ **Note**

The PF miniport driver reports on whether it supports asymmetric allocation of queue pairs during ***MiniportInitializeEx***. For more information, see **Initializing a PF Miniport Driver**.

The number of queue pairs assigned to each VPort is not changed dynamically. The number of queue pairs assigned to a VPort cannot be changed after the VPort has been created.

ⓘ **Note**

One or more queue pairs assigned to the nondefault VPorts can be used for receive side scaling (RSS) by the VF miniport driver that runs in the guest operating system.

- Interrupt moderation parameters for the VPort.

  Different interrupt moderation types can be specified for different VPorts. This allows the virtualization stack to control the number of interrupts generated by a particular VPort.

In addition to configuration parameters, overlying drivers can configure receive filters for each VPort by issuing OID method requests of OID_RECEIVE_FILTER_SET_FILTER. The NIC switch performs the specified receive filtering on a VPort basis.

Receive filters parameters for VPorts include packet filtering conditions, such as a list of media access control (MAC) addresses and the virtual LAN (VLAN) identifiers. Filters for MAC addresses and VLAN identifiers are always specified together in the **NDIS_RECEIVE_FILTER_PARAMETERS** associated with the OID_RECEIVE_FILTER_SET_FILTER request. The NIC switch must filter incoming packets to the switch whose destination MAC address and VLAN identifier matches any receive filter condition that was set on the VPort. The NIC switch filters packets received from either another VPort or from the external physical port. If the packet matches a filter, the NIC switch must forward it to the VPort.

Multiple MAC address and VLAN identifier pairs may be set on the VPort. If only a MAC address is set, the receive filter specifies that the VPort should receive packets that match the following condition:

- The packet's destination MAC address matches the filter's MAC address.

- The packet has a VLAN tag or (if a VLAN tag is present) a VLAN identifier of zero.

Nondefault VPorts are deleted through OID set requests of OID_NIC_SWITCH_DELETE_VPORT. The default VPort is only deleted when the NIC switch is deleted through an OID set request of OID_NIC_SWITCH_DELETE_SWITCH.

SR-IOV Data Paths

Article • 12/15/2021

This section describes the possible data paths between a network adapter that supports single root I/O virtualization (SR-IOV) and the Hyper-V parent and child partitions.

This section includes the following topics:

Overview of SR-IOV Data Paths

Article • 12/15/2021

When a Hyper-V child partition is started and the guest operating system is running, the virtualization stack starts the Network Virtual Service Client (NetVSC). NetVSC exposes a virtual machine (VM) network adapter by providing a miniport driver edge to the protocol stacks that run in the guest operating system. In addition, NetVSC provides a protocol driver edge that allows it to bind to the underlying miniport drivers.

NetVSC also communicates with the Hyper-V extensible switch that runs in the management operating system of the Hyper-V parent partition. The extensible switch component operates as a Network Virtual Service Provider (NetVSP). The interface between the NetVSC and NetVSP provides a software data path that is known as the *synthetic data path*. For more information about this data path, see SR-IOV Synthetic Data Path.

If the physical network adapter supports the single root I/O virtualization (SR-IOV) interface, it can enable one or more PCI Express (PCIe) Virtual Functions (VFs). Each VF can be attached to a Hyper-V child partition. When this happens, the virtualization stack performs the following steps:

1. The virtualization stack exposes a network adapter for the VF in the guest operating system. This causes the PCI driver that runs in the guest operating system to start the VF miniport driver. This driver is provided by the independent hardware vendor (IHV) for the SR-IOV network adapter.

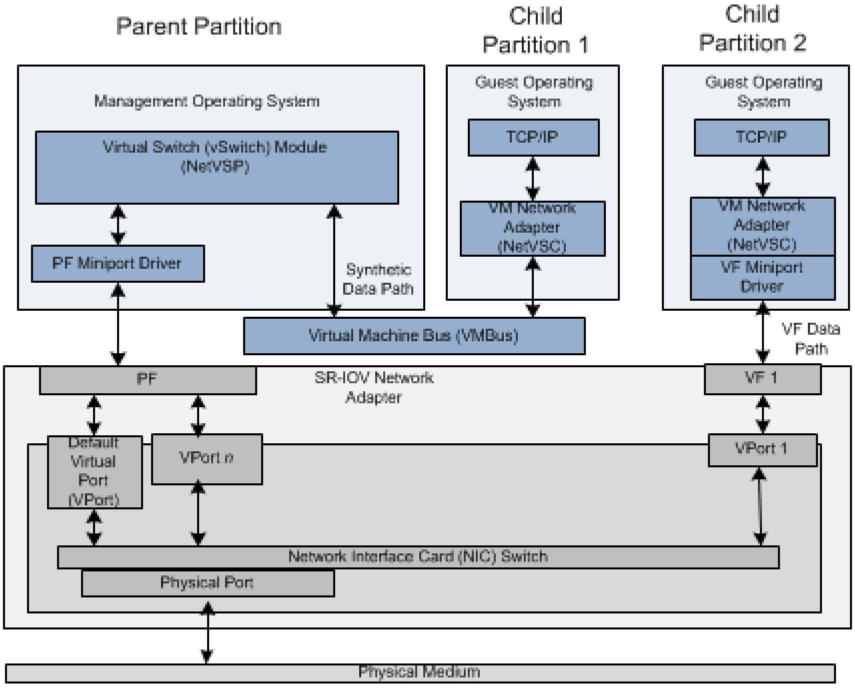
2. After the VF miniport driver is loaded and initialized, NDIS binds the protocol edge of the NetVSC in the guest operating system to the driver.

   Note NetVSC only binds to the VF miniport driver. No other protocol stacks in the guest operating system can bind to the VF miniport driver.

After the NetVSC successfully binds to the driver, network traffic in the guest operating system occurs over the *VF data path*. Packets are sent or received over the underlying VF of the network adapter instead of the synthetic data path.

For more information about the VF data path, see SR-IOV VF Data Path.

The following figure shows the various data paths that are supported over an SR-IOV network adapter.



After the Hyper-V child partition is started and before the VF data path is established, network traffic flows over the synthetic data path. After the VF data path is established, network traffic can revert to the synthetic data path if the following conditions are true:

- The VF becomes unattached to the Hyper-V child partition. For example, the virtualization stack could detach a VF from one child partition and attach it to another child partition. This might occur when there are more Hyper-V child partitions that are running than there are VF resources on the underlying SR-IOV network adapter.

  The process of failing over to the synthetic data path from the VF data path is known as *VF failover*.

- The Hyper-V child partition is being live migrated to a different host.

For more information about VF failover and live migration, see SR-IOV VF Failover and Live Migration.

SR-IOV VF Data Path

Article • 12/15/2021

If the physical network adapter supports the single root I/O virtualization (SR-IOV) interface, it can enable one or more PCI Express (PCIe) Virtual Functions (VFs). Each VF can be attached to a Hyper-V child partition. When this happens, the virtualization stack performs the following steps:

1. Once resources for the VF are allocated, the virtualization stack exposes a network adapter for the VF in the guest operating system. This causes the PCI driver that runs in the guest operating system to start the VF miniport driver. This driver is provided by the independent hardware vendor (IHV) for the SR-IOV network adapter.

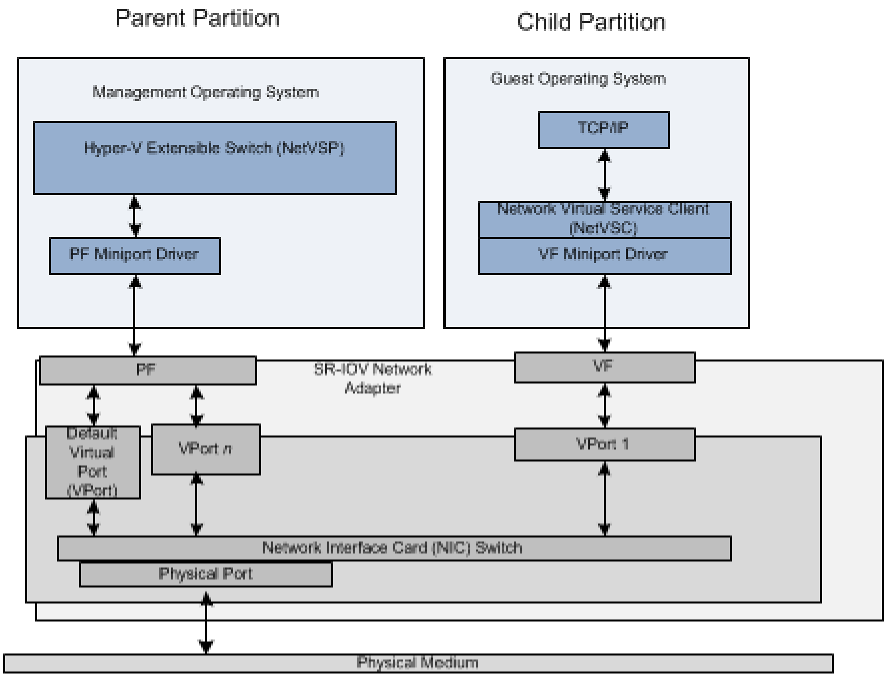
   Note Resources for the VF must be allocated by the miniport driver for the PCIe Physical Function (PF) before the VF can be attached to the Hyper-V child partition. VF resources include assigning a virtual port (VPort) on the NIC switch to the VF. For more information, see SR-IOV Virtual Functions.

2. After the VF miniport driver is loaded and initialized, NDIS binds the protocol edge of the Network Virtual Service Client (NetVSC) in the guest operating system to the driver.

   Note NetVSC only binds to the VF miniport driver. No other protocol stacks in the guest operating system can bind to the VF miniport driver.

After the NetVSC successfully binds to the driver, network traffic in the guest operating system occurs over the *VF data path*. Packets are sent or received over the underlying VF of the network adapter instead of the software-based synthetic data path. For more information about the synthetic data path, see SR-IOV Synthetic Data Path.

The following diagram shows the components of the VF data path over an SR-IOV network adapter.


Parent Partition
Child Partition
Management Operating System
Hyper-V Extensible Switch (NetVSP)
PF Miniport Driver
Guest Operating System
TCP/IP
Network Virtual Service Client (NetVSC)
VF Miniport Driver
PF
SR-IOV Network Adapter
VF
Default Virtual Port (VPort)
VPort n
VPort 1
Network Interface Card (NIC) Switch
Physical Port
Physical Medium

The use of the VF data path provides the following benefits:

- All data packets flow directly between the networking components in the guest operating system and the VF. This eliminates the overhead of the synthetic data path in which data packets flow between the Hyper-V child and parent partitions.

  For more information about the synthetic data path, see SR-IOV Synthetic Data Path.

- The VF data path bypasses any involvement by the management operating system in packet traffic from a Hyper-V child partition. The VF provides independent memory space, interrupts, and DMA streams for the child partition to which it is attached. This achieves networking performance that is almost compatible with nonvirtualized environments.

- The routing of packets over the VF data path is performed by the NIC switch on the SR-IOV network adapter. Packets are sent or received over the external network through the physical port of the adapter. Packets are also forwarded to or from other child partitions to which a VF is attached.

  Note Packets to or from child partitions to which no VF is attached are forwarded by the NIC switch to the Hyper-V extensible switch module. This module runs in the Hyper-V parent partition and delivers these packets to the child partition by using the synthetic data path.

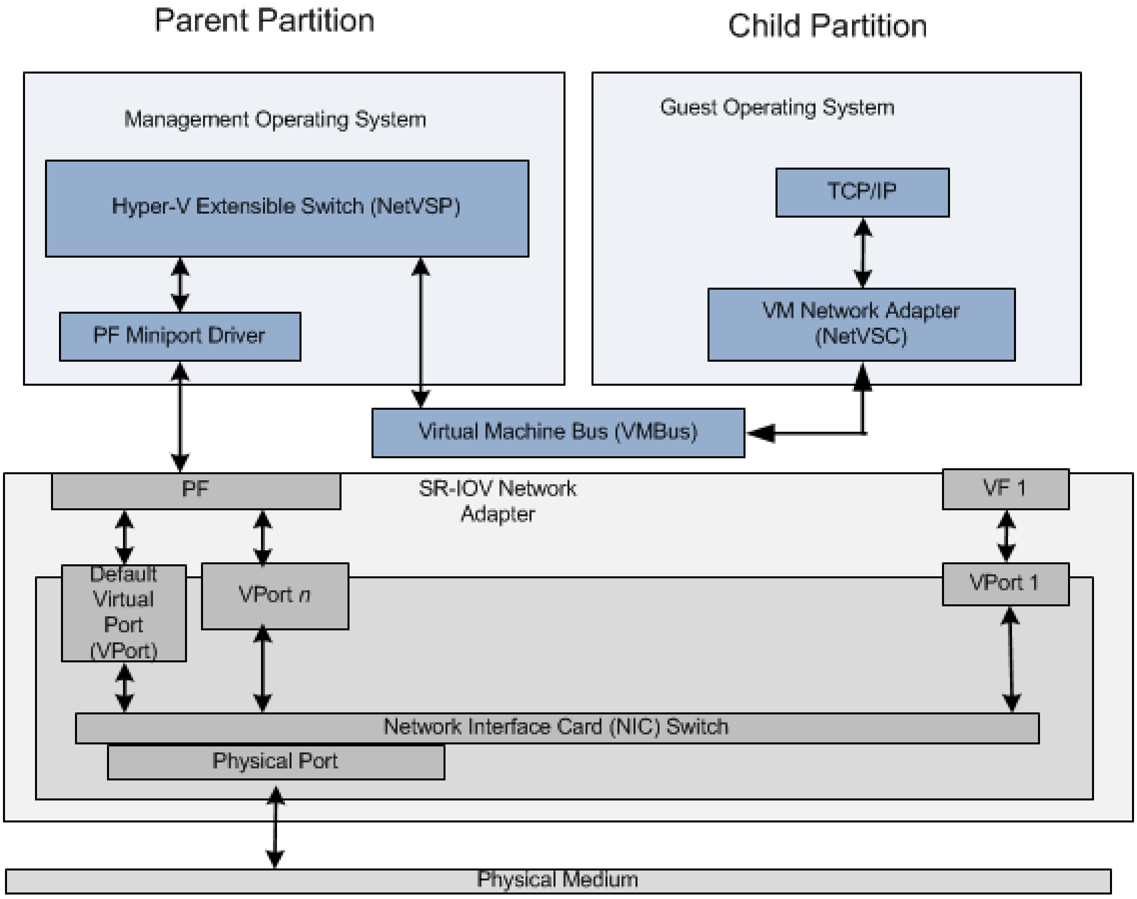
SR-IOV Synthetic Data Path

Article • 12/15/2021

When a Hyper-V child partition is started and the guest operating system is running, the virtualization stack starts the Network Virtual Service Client (NetVSC). NetVSC exposes a virtual machine (VM) network adapter that provides a miniport driver edge to the protocol stacks that run in the guest operating system.

NetVSC also communicates with the Hyper-V extensible switch that runs in the management operating system of the Hyper-V parent partition. The extensible switch component operates as a Network Virtual Service Provider (NetVSP). The interface between the NetVSC and NetVSP provides a software data path that is known as the *synthetic data path*.

The following diagram shows the components of the synthetic data path over an SR-IOV network adapter.


Parent Partition
Child Partition
Management Operating System
Hyper-V Extensible Switch (NetVSP)
PF Miniport Driver
Guest Operating System
TCP/IP
VM Network Adapter (NetVSC)
Virtual Machine Bus (VMBus)
PF
SR-IOV Network Adapter
VF 1
Default Virtual Port (VPort)
VPort n
VPort 1
Network Interface Card (NIC) Switch
Physical Port
Physical Medium

If the underlying SR-IOV network adapter allocates resources for PCI Express (PCIe) Virtual Functions (VFs), the virtualization stack will attach a VF to a Hyper-V child partition. Once attached, packet traffic within the child partition will occur over the hardware-optimized VF data path instead of the synthesized data path. For more information on the VF data path, see SR-IOV Data Path.

The virtualization stack may still enable the synthetic data path for a Hyper-V child partition if one of the following conditions is true:

- The SR-IOV network adapter has insufficient VF resources to accommodate all of the Hyper-V child partitions that were started. After all VFs on the network adapter are attached to child partitions, the remaining partitions use the synthetic data path.

  The process of failing over to the synthetic data path from the VF data path is known as *VF failover*.

- A VF was attached to a Hyper-V child partition but becomes detached. For example, the virtualization stack could detach a VF from one child partition and attach it to another child partition. This might occur when there are more Hyper-V child partitions that are running than there are VF resources on the underlying SR-IOV network adapter.

- The Hyper-V child partition is being live migrated to a different host.

Although the synthetic data path over an SR-IOV network adapter is not as efficient as the VF data path, it can still be hardware optimized. For example, if one or more virtual ports (VPorts) are configured and attached to the PCIe Physical Function (PF), the data path can provide the offload capabilities that resemble the virtual machine queue (VMQ) interface. For more information, see Nondefault Virtual Ports and VMQ.

SR-IOV VF Failover and Live Migration Support

Article • 12/15/2021

After the Hyper-V child partition is started, network traffic flows over the synthetic data path. If the physical network adapter supports the Single Root I/O Virtualization (SR-IOV) interface, it can enable one or more PCI Express (PCIe) Virtual Functions (VFs). Each VF can be attached to a Hyper-V child partition. When this happens, network traffic flows over the hardware-optimized SR-IOV VF Data Path.

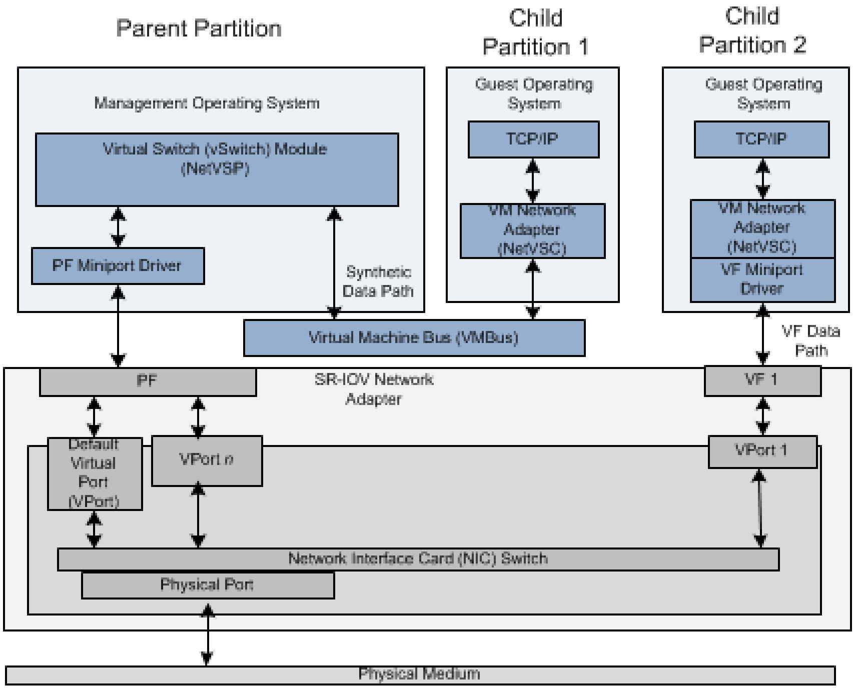
After the VF data path is established, network traffic can revert to the synthetic data path if any of the following conditions is true:

- A VF was attached to a Hyper-V child partition but becomes detached. For example, the virtualization stack could detach a VF from one child partition and attach it to another child partition. This might occur when there are more Hyper-V child partitions that are running than there are VF resources on the underlying SR-IOV network adapter.

  The process of failing over to the synthetic data path from the VF data path is known as *VF failover*.

- The Hyper-V child partition is being live migrated to a different host.

The following figure shows the various data paths that are supported over an SR-IOV network adapter.



The NetVSC exposes a Virtual Machine (VM) network adapter which is bound to the VF miniport driver to support the VF data path. During the transition to the synthetic data path, the VF network adapter is gracefully removed if possible from the guest operating system. If the VF cannot be removed gracefully and times out, it will be surprise removed. Then the VF miniport driver is halted, and the Network Virtual Service Client (NetVSC) is unbound from the VF miniport driver.

The transition between the VF and synthetic data paths occurs with minimum loss of packets and prevents the loss of TCP connections. Before the transition to the synthetic data path is complete, the virtualization stacks follows these steps:

1. The virtualization stack moves the Media Access Control (MAC) and Virtual LAN (VLAN) filters for the VM network adapter to the default Virtual Port (VPort) that is attached to the PCIe Physical Function (PF). The VM network adapter is exposed in the guest operating system of the child partition.

   After the filters are moved to the default VPort, the synthetic data path is fully operational for network traffic to and from the networking components that run in the guest operating system. The PF miniport driver indicates received packets on the default PF VPort which uses the synthetic data path to indicate the packets to the guest operating system. Similarly, all transmitted packets from the guest operating system are routed through the synthetic data path and transmitted through the default PF VPort.

   For more information about VPorts, see Virtual Ports (VPorts).

2. The virtualization stack deletes the VPort that is attached to the VF by issuing an Object Identifier (OID) set request of OID_NIC_SWITCH_DELETE_VPORT to the PF miniport driver. The miniport driver frees any hardware and software resources associated with the VPort and completes the OID request.

   For more information, see Deleting a Virtual Port.

3. The virtualization stack requests a PCIe Function Level Reset (FLR) of the VF before its resources are deallocated. The stack does this by issuing an OID set request of OID_SRIOV_RESET_VFto the PF miniport driver. The FLR brings the VF on the SR-IOV network adapter into a quiescent state and clears any pending interrupt events for the VF.

4. After the VF has been reset, the virtualization stack requests a deallocation of the VF resources by issuing an OID set request of OID_NIC_SWITCH_FREE_VF to the PF miniport driver. This causes the miniport driver to free the hardware resources associated with the VF.

For more information about this process, see Virtual Function Teardown Sequence.

Writing SR-IOV PF Miniport Drivers Overview

Article • 12/15/2021

This section discusses the requirements and guidelines for writing an NDIS miniport driver for the PCI Express (PCIe) Physical Function (PF) of a single root I/O virtualization (SR-IOV) network adapter.

This section includes the following topics:

Initializing a PF Miniport Driver

Managing NIC Switches

Managing Virtual Ports

Managing Virtual Functions

Halting a PF Miniport Driver

INF Requirements for PF Miniport Drivers

Note For information on how to write a miniport driver for a PCIe Virtual Function (VF) of the SR-IOV network adapter, see Writing SR-IOV VF Miniport Drivers.

Initializing a PF Miniport Driver Topics

Article • 12/15/2021

This section discusses the requirements and guidelines for initializing an NDIS miniport driver for the PCI Express (PCIe) Physical Function (PF) of a single root I/O virtualization (SR-IOV) network adapter.

This section includes the following topics:

Determining SR-IOV Capabilities

Determining NIC Switch Capabilities

Determining Receive Filtering Capabilities

Initialization Sequence for PF Miniport Drivers

Determining SR-IOV Capabilities

Article • 12/15/2021

This topic describes how NDIS and overlying drivers determine the single root I/O virtualization (SR-IOV) capabilities of a network adapter. This topic contains the following information:

Reporting SR-IOV Capabilities during *MiniportInitializeEx*

Querying SR-IOV Capabilities by Overlying Drivers

Reporting SR-IOV Capabilities during *MiniportInitializeEx*

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver provides the following SR-IOV capabilities:

- The complete set of SR-IOV hardware capabilities that the network adapter can support.
- The SR-IOV capabilities that are currently enabled on the network adapter.
- Whether the miniport driver is managing the PCI Express (PCIe) Physical Function (PF) or Virtual Function (VF) on the network adapter.

The miniport driver reports the SR-IOV hardware capabilities of the underlying network adapter through an **NDIS_SRIOV_CAPABILITIES** structure that is initialized in the following way:

1. The miniport driver initializes the **Header** member. The driver sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT.

   Starting with NDIS 6.30, the miniport driver sets the **Revision** member of **Header** to NDIS_SRIOV_CAPABILITIES _REVISION_1 and the **Size** member to NDIS_SIZEOF_SRIOV_CAPABILITIES_REVISION_1.

2. The miniport driver sets the appropriate flags in the **SriovCapabilities** member to report SR-IOV capabilities.

   If the network adapter supports SR-IOV, the miniport driver for the PCI Express (PCIe) Physical Function of the adapter must set the following flags:

   - NDIS_SRIOV_CAPS_SRIOV_SUPPORTED

- NDIS_SRIOV_CAPS_PF_MINIPORT

> ⓘ **Note**
>
> The miniport driver for a PCIe Virtual Function (VF) of the network adapter must set both the NDIS_SRIOV_CAPS_VF_MINIPORT flag and the NDIS_SRIOV_CAPS_SRIOV_SUPPORTED flag.

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver registers the SR-IOV capabilities of the network adapter by following these steps:

1. The miniport driver initializes an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

   The miniport driver sets the **HardwareSriovCapabilities** member to a pointer to the previously initialized **NDIS_SRIOV_CAPABILITIES** structure.

   If the registry setting for the ***SRIOV** INF keyword has a value of one, the SR-IOV capabilities are currently enabled on the network adapter. The miniport driver sets the **CurrentSriovCapabilities** members to a pointer to the same **NDIS_SRIOV_CAPABILITIES** structure.

   If the registry setting for the ***SRIOV** INF keyword has a value of zero, the SR-IOV capabilities are currently disabled on the network adapter. The miniport driver must set the **CurrentSriovCapabilities** member to NULL.

   For more information about the ***SRIOV** INF keyword, see Standardized INF Keywords for SR-IOV.

2. The driver calls **NdisMSetMiniportAttributes** and sets the *MiniportAttributes* parameter to a pointer to the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

For more information about the adapter initialization process, see Initializing a Miniport Adapter.

Querying SR-IOV Capabilities by Overlying Drivers

NDIS passes the network adapter's currently enabled SR-IOV capabilities to overlying drivers that bind to the network adapter in the following way:

- When NDIS calls an overlying filter driver's *FilterAttach* function, NDIS passes the network adapter's SR-IOV capabilities through the *AttachParameters* parameter. This parameter contains a pointer to an **NDIS_FILTER_ATTACH_PARAMETERS** structure. The **SriovCapabilities** member of this structure contains a pointer to an **NDIS_SRIOV_CAPABILITIES** structure.

- When NDIS calls an overlying protocol driver's *ProtocolBindAdapterEx* function, NDIS passes the network adapter's SR-IOV capabilities through the *BindParameters* parameter. This parameter contains a pointer to an **NDIS_FILTER_ATTACH_PARAMETERS** structure. The **SriovCapabilities** member of this structure contains a pointer to an **NDIS_SRIOV_CAPABILITIES** structure.

NDIS also returns the **NDIS_SRIOV_CAPABILITIES** structure when it handles object identifier (OID) query requests of OID_SRIOV_HARDWARE_CAPABILITIES and OID_SRIOV_CURRENT_CAPABILITIES that are issued by overlying protocol or filter drivers.

Determining NIC Switch Capabilities

Article • 12/15/2021

This topic describes how NDIS and overlying drivers determine the NIC switch capabilities of a network adapter that supports single root I/O virtualization (SR-IOV). This topic contains the following information:

Reporting NIC Switch Capabilities during *MiniportInitializeEx*

Querying NIC Switch Capabilities by Overlying Drivers

Note Only the miniport driver for the PCI Express (PCIe) Physical Function (PF) of an SR-IOV network adapter can report NIC switch capabilities. Miniport drivers for PCIe Virtual Functions (VFs) must not report the NIC switch capabilities of the SR-IOV adapter.

For more information on NIC switches, see NIC Switches.

Reporting NIC Switch Capabilities during *MiniportInitializeEx*

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver provides the following NIC switch capabilities:

- The complete set of hardware capabilities for a NIC switch that the network adapter can support.

  Note Starting with NDIS 6.30, only one NIC switch is created on the network adapter. This switch is known as the *default NIC switch*.

- The NIC switch capabilities that are currently enabled on the network adapter.

The miniport driver reports the NIC switch hardware capabilities of the underlying network adapter through an **NDIS_NIC_SWITCH_CAPABILITIES** structure that is initialized in the following way:

1. The miniport driver initializes the **Header** member. The driver sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT.

   Starting with NDIS 6.30, the miniport driver sets the **Revision** member of **Header** to NDIS_NIC_SWITCH_CAPABILITIES_REVISION_2 and the **Size** member to NDIS_SIZEOF_NIC_SWITCH_CAPABILITIES_REVISION_2.

2. The miniport driver sets appropriate flags in the **NicSwitchCapabilities** member of the **NDIS_NIC_SWITCH_CAPABILITIES** structure to the NIC switch capabilities of the SR-IOV network adapter. For example, the miniport driver sets the NDIS_NIC_SWITCH_CAPS_PER_VPORT_INTERRUPT_MODERATION_SUPPORTED flag if the NIC switch supports interrupt moderation on each virtual port (VPort) that is created on the switch.

3. The miniport driver sets the other members of the **NDIS_NIC_SWITCH_CAPABILITIES** structure to the range of values for the NIC switch capabilities of the SR-IOV network adapter. For example, the miniport driver sets the **MaxNumVFs** and **MaxNumVPorts** members to the maximum number of VFs and VPorts that the adapter can support.

   Note Depending on the available hardware resources on the network adapter, the miniport driver can set the **MaxNumVFs** member to a value that is less than its ***NumVFs** keyword. For more information about this keyword, see Standardized INF Keywords for SR-IOV.

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver registers the NIC switch capabilities of the network adapter by following these steps:

1. The miniport driver initializes an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

   The miniport driver sets the **HardwareNicSwitchCapabilities** member to a pointer to a previously initialized **NDIS_NIC_SWITCH_CAPABILITIES** structure.

   If the registry setting for the ***SRIOV** INF keyword has a value of one, the network adapter is currently enabled for NIC switch creation and configuration. The miniport driver sets the **CurrentNicSwitchCapabilities** members to a pointer to the same **NDIS_NIC_SWITCH_CAPABILITIES** structure.

   If the registry setting for the ***SRIOV** INF keyword has a value of zero, the network adapter is not currently enabled for NIC switch creation and configuration. The miniport driver must set the **CurrentNicSwitchCapabilities** member to NULL.

   For more information about the ***SRIOV** INF keyword, see Standardized INF Keywords for SR-IOV.

2. The driver calls **NdisMSetMiniportAttributes** and sets the *MiniportAttributes* parameter to a pointer to the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

For more information about the adapter initialization process, see Initializing a Miniport Adapter.

Creating a NIC switch without SR-IOV

Miniport drivers in NDIS 6.60 and later must adhere to the following requirements for the coexistence of a NIC switch and VMQ capabilities when SR-IOV is not enabled. When SR-IOV is enabled, the miniport driver should adhere to the existing requirements in the previous section.

- The miniport driver advertises both NIC switch and VMQ capabilities.
- The NIC can toggle between NIC switch and VMQ mode without a NIC restart.
  - When the NIC starts initially, it is ready to be in either mode (either creating a NIC switch or creating VMQ queues).
    - If a NIC switch is created, the miniport fails any subsequent VMQ queue allocation callbacks.
    - If a VMQ queue is created first, the miniport driver succeeds the VMQ queue allocation and fails any NIC switch allocation calls.
  - When the NIC switch is deleted or all VMQ queues are deleted, the miniport driver returns to the initial state and is ready to go into either of these modes again.

To advertise that a NIC switch can be created without the use of SR-IOV, the miniport driver sets the **NDIS_NIC_SWITCH_CAPS_NIC_SWITCH_WITHOUT_IOV_SUPPORTED** flag in the **NicSwitchCapabilities** member of the NDIS_NIC_SWITCH_CAPABILITIES structure.

Querying NIC Switch Capabilities by Overlying Drivers

NDIS passes the network adapter's currently enabled NIC switch capabilities to overlying drivers that bind to the network adapter in the following way:

- When NDIS calls an overlying filter driver's *FilterAttach* function, NDIS passes the network adapter's NIC switch capabilities through the *AttachParameters* parameter. This parameter contains a pointer to an NDIS_FILTER_ATTACH_PARAMETERS structure. The **NicSwitchCapabilities** member of this structure contains a pointer to an NDIS_NIC_SWITCH_CAPABILITIES structure.

- When NDIS calls an overlying protocol driver's *ProtocolBindAdapterEx* function, NDIS passes the network adapter's NIC switch capabilities through the *BindParameters* parameter. This parameter contains a pointer to an

NDIS_FILTER_ATTACH_PARAMETERS structure. The **NicSwitchCapabilities** member of this structure contains a pointer to an **NDIS_NIC_SWITCH_CAPABILITIES** structure.

NDIS also returns the **NDIS_NIC_SWITCH_CAPABILITIES** structure when it handles object identifier (OID) query requests of OID_NIC_SWITCH_HARDWARE_CAPABILITIES and OID_NIC_SWITCH_CURRENT_CAPABILITIES that are issued by overlying protocol or filter drivers.

Determining Receive Filtering Capabilities

Article • 12/15/2021

This topic describes how NDIS and overlying drivers determine the receive filtering capabilities of a network adapter that supports single root I/O virtualization (SR-IOV). This topic contains the following information:

Reporting Receive Filtering Capabilities during *MiniportInitializeEx*

Querying Receive Filtering Capabilities by Overlying Drivers

Note Only the miniport driver for the PCI Express (PCIe) Physical Function (PF) of an SR-IOV network adapter can report receive filtering capabilities. Miniport drivers for PCIe Virtual Functions (VFs) must not report the receive filtering capabilities of the SR-IOV adapter.

Reporting Receive Filtering Capabilities during *MiniportInitializeEx*

When NDIS calls the PF miniport driver's *MiniportInitializeEx* function, the driver provides the following receive filtering capabilities:

- The complete hardware receive filtering capabilities that the network adapter can support.
- The receive filtering capabilities for the interfaces that are currently enabled on the network adapter.

The miniport driver reports the complete hardware receive filtering capabilities of the underlying network adapter through an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure that is initialized in the following way:

1. The miniport driver initializes the **Header** member. The driver sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT.

   Starting with NDIS 6.30, the miniport driver sets the **Revision** member of **Header** to NDIS_RECEIVE_FILTER_CAPABILITIES_REVISION_2 and the **Size** member to NDIS_SIZEOF_RECEIVE_FILTER_CAPABILITIES_REVISION_2.

2. The miniport driver sets the other members of the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure to the range of values for the receive filtering capabilities of the SR-IOV network adapter. For example, the miniport driver sets the appropriate flags in the **SupportedFilterTests** to specify filter test operations that the miniport driver supports.

3. Besides SR-IOV, receive filtering is also used in the following interfaces:

   - NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.

   - Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

   If the miniport driver supports any of these interfaces, it must also set the members of the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure to the range of receive filtering capability values that are specific to the interface. For example, if the driver supports NDIS packet coalescing and SR-IOV, it must set the NDIS_RECEIVE_FILTER_PACKET_COALESCING_SUPPORTED_ON_DEFAULT_QUEUE flag in the **SupportedQueueProperties** member.

The miniport driver reports the currently-enabled receive filtering capabilities of the underlying network adapter through an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure that is initialized in the following way:

1. The miniport driver initializes the **Header** member. The driver sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT.

   Starting with NDIS 6.30, the miniport driver sets the **Revision** member of **Header** to NDIS_RECEIVE_FILTER_CAPABILITIES_REVISION_2 and the **Size** member to NDIS_SIZEOF_RECEIVE_FILTER_CAPABILITIES_REVISION_2.

2. The miniport driver sets the other members of the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure to the range of values for the receive filtering capabilities of the interfaces that are currently enabled. For example, if NDIS packet coalescing is enabled, the driver must only set the members that are specific to this technology.

   Interfaces that use receive filtering are enabled or disabled through standardized INF keywords. For more information on how NDIS packet coalescing is enabled, see Standardized INF Keywords for Packet Coalescing. For more information on how SR-IOV and VMQ are enabled, see Handling SR-IOV, VMQ, and RSS Standardized INF Keywords.

When NDIS calls the miniport driver's *MiniportInitializeEx* function, the driver registers the receive filtering capabilities of the network adapter by following these steps:

1. The miniport driver initializes an NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

   The miniport driver sets the **HardwareReceiveFilterCapabilities** member to the address of an NDIS_RECEIVE_FILTER_CAPABILITIES structure. This structure was previously-initialized with the complete hardware receive filtering capabilities of the network adapter.

2. If the VMQ, SR-IOV, and NDIS packet coalescing are all currently disabled on the network adapter, the miniport driver sets the **CurrentReceiveFilterCapabilities** member to NULL.

3. If either VMQ, SR-IOV, or NDIS packet coalescing are currently enabled on the network adapter, the miniport driver must do the following:

   - The miniport driver must initialize another NDIS_RECEIVE_FILTER_CAPABILITIES structure with the current receive filtering capabilities for the interfaces that are currently enabled on the network adapter.

     If the SR-IOV interface is enabled, there are situations in which the miniport driver must set the members of the NDIS_RECEIVE_FILTER_CAPABILITIES structure to the same or different values. This is because the SR-IOV interface provides a similar queuing mechanism to VMQ, but uses virtual ports (VPorts) instead of VM receive queues.

     For example, the miniport driver must set the NDIS_RECEIVE_FILTER_VMQ_FILTERS_ENABLED flag in the **EnabledFilterTypes** member if either the VMQ or SR-IOV interface is enabled. However, the miniport driver must set the **NumQueues** member to zero if the SR-IOV interface is enabled and a nonzero value if the VMQ interface is enabled.

   - The miniport driver sets the **CurrentReceiveFilterCapabilities** member to the address of the NDIS_RECEIVE_FILTER_CAPABILITIES structure that contains the current receive filtering capabilities for the currently-enabled interface.

4. If either VMQ, SR-IOV, or NDIS packet coalescing are currently enabled on the network adapter, the miniport driver sets the **HardwareReceiveFilterCapabilities** member to the address of an NDIS_RECEIVE_FILTER_CAPABILITIES structure. This structure was previously-initialized with the currently-enabled receive filtering capabilities of the network adapter.

5. The driver calls **NdisMSetMiniportAttributes** and sets the *MiniportAttributes* parameter to a pointer to the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

For more information about the adapter initialization process, see Initializing a Miniport Adapter.

Querying Receive Filtering Capabilities by Overlying Drivers

NDIS passes the network adapter's currently-enabled receive filtering capabilities to overlying drivers that bind to the network adapter in the following way:

- When NDIS calls an overlying filter driver's *FilterAttach* function, NDIS passes the network adapter's NIC switch capabilities through the *AttachParameters* parameter. This parameter contains a pointer to an **NDIS_FILTER_ATTACH_PARAMETERS** structure. The **ReceiveFilterCapabilities** member of this structure contains a pointer to an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure.

- When NDIS calls an overlying protocol driver's *ProtocolBindAdapterEx* function, NDIS passes the network adapter's NIC switch capabilities through the *BindParameters* parameter. This parameter contains a pointer to an **NDIS_FILTER_ATTACH_PARAMETERS** structure. The **ReceiveFilterCapabilities** member of this structure contains a pointer to an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure.

NDIS also returns the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure when it handles object identifier (OID) query requests of OID_RECEIVE_FILTER_CURRENT_CAPABILITIES and OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES that are issued by overlying protocol or filter drivers.

Initialization Sequence for PF Miniport Drivers

Article • 12/15/2021

This section describes the requirements and guidelines for the initialization sequence of a miniport driver for the PCI Express (PCIe) Physical Function (PF). The PF is a component of a network adapter that supports single root I/O virtualization (SR-IOV).

This section includes the following topics:

DriverEntry Guidelines for PF Miniport Drivers

MiniportAddDevice Guidelines for PF Miniport Drivers

MiniportInitializeEx Guidelines for PF Miniport Drivers

Note For information on initializing miniport drivers for a PCIe Virtual Function (VF) on the SR-IOV network adapter, see Initializing a VF Miniport Driver.

DriverEntry Guidelines for PF Miniport Drivers

Article • 12/15/2021

This topic describes the guidelines for writing a **DriverEntry** function for the miniport driver of the PCI Express (PCIe) Physical Function (PF). The PF is a component of a network adapter that supports single root I/O virtualization (SR-IOV).

Note These guidelines only apply to PF miniport drivers. For initialization guidelines for the miniport driver of a PCIe Virtual Function (VF) of the adapter, see Initializing a VF Miniport Driver.

The SR-IOV network adapter must implement a hardware bridge that forwards network traffic over the physical port on the adapter and internal virtual ports (VPorts). This bridge is known as the *NIC switch*. For more information, see NIC Switches.

If the PF miniport driver supports the static creation of the NIC switch on the SR-IOV network adapter, it may need to allocate switch resources when the functional device object (FDO) is created for the network adapter in the device stack. In this case, the driver must allocate those resources before NDIS calls *MiniportInitializeEx*. To do this, the driver must register optional Plug-and-Play (PnP) handlers so that it can participate in the process when the adapter's FDO is added or removed from the device stack.

The miniport driver must provide a *MiniportSetOptions* function to register these PnP handler functions. To do this, the driver follows these steps from the context of the call to its **DriverEntry** function:

1. The miniport driver initializes an **NDIS_MINIPORT_DRIVER_CHARACTERISTICS** structure with the entry points of the *MiniportXxx* functions. In particular, the driver sets the **SetOptionsHandler** member to the entry point of the driver's *MiniportSetOptions* function.

2. The miniport driver calls the **NdisMRegisterMiniportDriver** function to register its entry points. From the context of this call, NDIS calls the driver's *MiniportSetOptions* function

3. When NDIS calls *MiniportSetOptions*, the miniport driver calls the **NdisSetOptionalHandlers** function and specifies an **NDIS_MINIPORT_PNP_CHARACTERISTICS** structure. This structure defines the entry points for the *MiniportAddDevice*, *MiniportRemoveDevice*, *MiniportStartDevice*, and *MiniportFilterResourceRequirements* functions. NDIS calls

these handler functions when it handles PnP I/O request packets (IRPs) issued by the PCI bus driver.

If the PF miniport driver must allocate additional software resources for the NIC switch before NDIS calls the driver's *MiniportInitializeEx* function, the driver must register a *MiniportAddDevice* function. When NDIS calls the *MiniportAddDevice* function, the PF miniport driver can call **NdisReadConfiguration** to read the NIC switch configuration keyword settings from the registry. For more information about these keywords, see Standardized INF Keywords for SR-IOV.

For more information about guidelines for the *MiniportAddDevice* function, see *MiniportAddDevice* Guidelines for PF Miniport Drivers.

For more information on how NIC switches are created, see Creating a NIC Switch.

MiniportAddDevice Guidelines for PF Miniport Drivers

Article • 12/15/2021

This topic describes the guidelines for writing a *MiniportAddDevice* function for the miniport driver of the PCI Express (PCIe) Physical Function (PF). The PF is a component of a network adapter that supports single root I/O virtualization (SR-IOV).

Note These guidelines only apply to PF miniport drivers. For initialization guidelines for the miniport driver of a PCIe Virtual Function (VF) of the adapter, see Initializing a VF Miniport Driver.

The Plug and Play (PnP) Manager calls the NDIS *AddDevice* function to create the functional device object (FDO) for the network adapter. If the PF miniport driver registered a *MiniportAddDevice* entry point when it called **NdisMRegisterMiniportDriver**, NDIS calls the driver's *MiniportAddDevice* function.

When *MiniportAddDevice* is called, the PF miniport driver can allocate additional software resources for the SR-IOV and the network interface card (NIC) switch. Typically, these are resources that must be allocated before NDIS calls the driver's *MiniportInitializeEx* function.

The driver can do the following within the context of the call to *MiniportAddDevice*:

- The PF miniport driver can call **NdisReadConfiguration** to read the SR-IOV and NIC switch configuration settings from the registry. These configuration settings are defined through the standardized SR-IOV keywords. For more information about these keywords, see Standardized INF Keywords for SR-IOV.

- Based on these configuration settings, the PF miniport driver allocates the additional software resources for the SR-IOV network adapter.

Note The actual allocation of hardware resources and the enabling of SR-IOV in the PCI configuration space must only be done within the context of the call to *MiniportInitializeEx*. Because the network adapter's memory-mapped I/O (MMIO) space is uninitialized when *MiniportAddDevice* is called, the miniport driver must not read or write to the adapter until *MiniportInitializeEx* is called.

MiniportInitializeEx Guidelines for PF Miniport Drivers

Article • 12/15/2021

This topic describes the guidelines for writing a *MiniportInitializeEx* function for the miniport driver of the PCI Express (PCIe) Physical Function (PF). The PF is a component of a network adapter that supports single root I/O virtualization (SR-IOV).

Note These guidelines only apply to PF miniport drivers. For initialization guidelines for the miniport driver of a PCIe Virtual Function (VF) of the adapter, see Initializing a VF Miniport Driver.

The PF miniport driver follows the same steps as any NDIS miniport driver when its *MiniportInitializeEx* function. For more information about these steps, see Initializing a Miniport Driver.

In addition to these steps, the PF miniport driver must follow these additional steps when NDIS calls the driver's *MiniportInitializeEx* function:

1. The PF miniport driver calls the **NdisGetHypervisorInfo** function to verify that it is running in the Hyper-V parent partition. This function returns an **NDIS_HYPERVISOR_INFO** structure that defines the partition type. If the partition type is reported as **NdisHypervisorPartitionMsHvParent**, the miniport driver is running in the Hyper-V parent partition that is attached to the PF on the adapter.

   Note If the partition type is reported as **NdisHypervisorPartitionMsHvChild**, the miniport driver is running in the Hyper-V child partition that is attached to a VF on the adapter. In this case, the miniport driver must not initialize as a PF driver. If possible, the driver must initialize as a VF driver as described in Initializing a VF Miniport Driver.

2. The PF miniport driver must read the SR-IOV standardized keywords to determine whether SR-IOV is enabled and obtain the NIC switch configuration settings. For more information about these keywords, see Standardized INF Keywords for SR-IOV.

   Note If the PF miniport driver registered an entry point to a *MiniportSetOptions* function, the driver may have previously obtained these settings from the registry when NDIS called *MiniportSetOptions*.

3. If the network adapter supports SR-IOV, virtual machine queue (VMQ), or RSS, the miniport driver must determine which feature to enable on the network adapter.

For more information on how to determine this, see Handling SR-IOV, VMQ, and RSS Standardized INF Keywords.

4. Along with RSS and VMQ hardware capabilities (if supported), the miniport driver must report its full set of hardware SR-IOV capabilities. These capabilities must be advertised regardless of the SR-IOV standardized keyword settings in the registry.

   If SR-IOV is enabled on the network adapter, the miniport driver must also report the currentlyenabled SR-IOV settings on the adapter.

   For more information on reporting the SR-IOV capabilities, see Determining SR-IOV Capabilities.

5. The miniport driver must report its full set of hardware NIC switch capabilities. These capabilities must be advertised regardless of the SR-IOV standardized keyword settings in the registry.

   If SR-IOV is enabled on the network adapter, the miniport driver must also report the currentlyenabled NIC switch settings on the adapter.

   For more information on reporting the NIC switch capabilities, see Determining NIC Switch Capabilities.

6. The miniport driver must report its full set of hardware receive filtering capabilities. These capabilities must be advertised regardless of the SR-IOV standardized keyword settings in the registry.

   If SR-IOV is enabled on the network adapter, the miniport driver must also report the currentlyenabled receive filtering settings on the adapter.

   For more information on reporting the receive filtering capabilities, see Determining Receive Filtering Capabilities.

7. If the miniport driver supports static NIC switch creation, it must do the following in the context of the call to *MiniportInitializeEx*.

   - The driver configures the adapter hardware based on the NIC switch standardized keyword settings. Based on these settings, the driver allocates the necessary hardware and software resources for the NIC switch.

   - The miniport driver calls **NdisMEnableVirtualization** to enable SR-IOV and set the number of VFs on the network adapter. This function configures the SR-IOV Extended Capability in the adapter's PCI configuration space. If this function returns NDIS_STATUS_SUCCESS, SR-IOV is enabled and the VFs are exposed over the PCIe interface.

For more information, see Static Creation of a NIC Switch.

Note If the miniport driver supports dynamic NIC switch creation, it creates the switch and enables virtualization when it handles an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_SWITCH. For more information, see Dynamic Creation of a NIC Switch.

Managing NIC Switches

Article • 12/15/2021

This section describes the requirements and guidelines for managing the NIC switch of a network adapter that supports single root I/O virtualization (SR-IOV). The miniport driver for the PCI Express (PCIe) Physical Function (PF) of the SR-IOV network adapter manages the NIC switch on the adapter.

This section includes the following topics:

Creating a NIC Switch

Deleting a NIC Switch

Enumerating NIC Switches on a Network Adapter

Querying the Parameters of a NIC Switch

Setting the Parameters of a NIC Switch

For more information on NIC switches for SR-IOV network adapters, see NIC Switches.

Note Only the PF miniport driver can configure the network adapter's hardware resources, such as the NIC switch. The miniport driver for a PCIe Virtual Function (VF) on the SR-IOV network adapter cannot directly access most of the adapter's hardware resources. For more information, see Writing SR-IOV VF Miniport Drivers.

Creating a NIC Switch

Article • 12/15/2021

This section describes the requirements and guidelines for creating the NIC switch of a network adapter that supports single root I/O virtualization (SR-IOV). The miniport driver for the PCI Express (PCIe) Physical Function (PF) of the SR-IOV network adapter creates and configures the NIC switch on the adapter.

A NIC switch can be created through one of the following methods:

Static Creation
The NIC switch is statically created on the SR-IOV network adapter by using a set of switch parameters defined by registry settings. After the NIC switch is created, its parameters cannot be changed while the driver is running.

The PF miniport driver statically creates the NIC switch within the context of the call to the driver's *MiniportInitializeEx* function. However, the NIC switch cannot be used until NDIS issues an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_SWITCH. Even though the NIC switch was previously created, the PF miniport driver enabled the NIC switch for use when it handled this OID request.

For more information about this method, see Static Creation of a NIC Switch.

Dynamic Creation
The NIC switch is dynamically created on the SR-IOV network adapter through the OID method request of OID_NIC_SWITCH_CREATE_SWITCH. This OID request defines the NIC switch parameters through the **NDIS_NIC_SWITCH_PARAMETERS** structure. These parameters are also based on the staticallydefined registry settings but could change dynamically while the miniport driver is running.

For more information about this method, see Dynamic Creation of a NIC Switch.

For more information on how to handle the OID_NIC_SWITCH_CREATE_SWITCH request, see Handling the OID_NIC_SWITCH_CREATE_SWITCH Request.

For more information on NIC switches for SR-IOV network adapters, see NIC Switches.

Note The miniport driver for a PCIe Virtual Function (VF) on the SR-IOV network adapter does not create or configure the network adapter's hardware resources, such as the NIC switch. For more information, see Writing SR-IOV VF Miniport Drivers.

Static Creation of a NIC Switch

Article • 12/15/2021

A network adapter that supports single root I/O virtualization (SR-IOV) must be able to create a NIC switch. For some adapters, the NIC switch can be created statically within the context of the call to *MiniportInitializeEx*.

Only the miniport driver for the PCI Express (PCIe) Physical Function (PF) of the SR-IOV adapter can create a NIC switch on the adapter.

Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

The parameters for the default NIC switch are defined through standardized keyword settings in the registry. For more information on these keywords, see Standardized INF Keywords for SR-IOV.

The PF miniport driver statically creates the NIC switch when NDIS calls the driver's *MiniportInitializeEx* function. Typically, the driver creates and configures the NIC switch as part of its initialization sequence before it enables SR-IOV on the network adapter.

The PF miniport driver follows these steps when it statically creates the NIC switch and enables SR-IOV on the network adapter in the context of the call to *MiniportInitializeEx*:

1. The PF miniport driver must read the SR-IOV standardized keywords to determine whether SR-IOV is enabled and obtain the NIC switch configuration parameters.

   Note If the PF miniport driver registered an entry point to a *MiniportSetOptions* function, the driver may have previously obtained these parameters from the registry when NDIS called *MiniportSetOptions*.

2. If SR-IOV is enabled, the PF miniport driver configures the network adapter with the NIC switch parameters from the registry. The driver must verify that the parameters are valid before it configures the network adapter. For example, the miniport driver must verify that the maximum number of PCIe Virtual Functions (VFs) assigned to the NIC switch does not exceed the number of VFs supported by the network adapter.

3. The miniport driver calls **NdisMEnableVirtualization** to enable SR-IOV and set the number of VFs on the network adapter. This function configures the SR-IOV Extended Capability in adapter's PCI configuration space. If this function returns

NDIS_STATUS_SUCCESS, SR-IOV is enabled and the VFs are exposed over the PCIe interface.

Note If the PF miniport driver statically creates the NIC switch, the switch cannot be used until NDIS issues an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_SWITCH. If the PF miniport driver statically created the NIC switch, it must verify that the switch parameters are specified in the OID request. These parameters, as contained within the **NDIS_NIC_SWITCH_PARAMETERS** structure associated with the OID request, must be identical to the parameters the driver used to create the switch.

For more information on how to handle the OID_NIC_SWITCH_CREATE_SWITCH request, see Handling the OID_NIC_SWITCH_CREATE_SWITCH Request.

For more information on the initialization sequence and requirements for PF miniport drivers, see Initializing a PF Miniport Driver.

Dynamic Creation of a NIC Switch

Article • 12/15/2021

A network adapter that supports single root I/O virtualization (SR-IOV) must be able to create a NIC switch. For some adapters, the NIC switch can be created dynamically after the miniport driver has returned from the call to *MiniportInitializeEx*.

Only the miniport driver for the PCI Express (PCIe) Physical Function (PF) of the SR-IOV adapter can create a NIC switch on the adapter.

Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

NDIS issues an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_SWITCH to create a NIC switch on the SR-IOV network adapter. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to the **NDIS_NIC_SWITCH_PARAMETERS** structure that contains the parameters for the switch.

If the PF miniport driver supports dynamic NIC switch creation, it must follow these steps when it handles this OID request:

1. The PF miniport driver allocates the necessary hardware and software resources for the NIC switch based on these parameters. The driver also configures the network adapter with these parameters.

   Note PF miniport drivers that support dynamic NIC switch creation do not have to read the switch parameters through the standardized SR-IOV keyword settings in the registry. NDIS reads these keywords to initialize the **NDIS_NIC_SWITCH_PARAMETERS** structure before it issues the OID_NIC_SWITCH_CREATE_SWITCH request. For more information on these keywords, see Standardized INF Keywords for SR-IOV.

2. The miniport driver calls **NdisMEnableVirtualization** to enable SR-IOV and set the number of VFs on the network adapter. This function configures the SR-IOV Extended Capability in adapter's PCI configuration space. If this function returns NDIS_STATUS_SUCCESS, SR-IOV is enabled and the VFs are exposed over the PCIe interface.

For more information on how to handle the OID_NIC_SWITCH_CREATE_SWITCH request, see Handling the OID_NIC_SWITCH_CREATE_SWITCH Request.

Handling the OID_NIC_SWITCH_CREATE_SWITCH Request

Article • 12/15/2021

NDIS issues an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_SWITCH to do the following:

- Enable a NIC switch on a network adapter that was statically created by the miniport driver for the PCI Express (PCIe) Physical Function (PF). The PF is a hardware component of the network adapter that supports single root I/O virtualization (SR-IOV).

  A NIC switch is statically created by the PF miniport driver from within the context to the call to *MiniportInitializeEx*. The driver allocates the resources and creates the switch based on parameters read from registry settings.

- Dynamically create a NIC switch on a network adapter.

  If the PF miniport driver does not support static NIC switch creation, the miniport driver allocates the resources and creates the switch based on parameters that are specified in the OID request.

The PF miniport driver advertises its support of the SR-IOV interface when NDIS calls the driver's *MiniportInitializeEx* function. If the PF miniport driver supports SR-IOV, NDIS reads the NIC switch configuration from the registry. Before NDIS issues an OID method request of OID_NIC_SWITCH_CREATE_SWITCH to the PF miniport driver, NDIS formats an **NDIS_NIC_SWITCH_PARAMETERS** structure with the registry information in the following way:

- NDIS sets the **SwitchType** member to the type of the NIC switch.

  Starting with Windows Server 2012, Windows only supports a switch type of **NdisNicSwitchTypeExternal**. An external switch specifies that the virtual ports (VPorts) that are connected to this type of switch can access the external network through the physical port on the network adapter.

  For more information about the NIC switch, see SR-IOV Architecture.

- NDIS sets the **SwitchId** member to an identifier value for the NIC switch. The switch identifier is an integer between zero and the number of switches that the network adapter supports. An NDIS_DEFAULT_SWITCH_ID value indicates the default NIC switch.

Note Starting with Windows Server 2012, the SR-IOV interface only supports the default NIC switch on the network adapter.

- NDIS sets the **NumVFs** member that specifies the number of PCIe Virtual Function (VFs) that can be allocated on the NIC switch.

When it receives the OID method request of OID_NIC_SWITCH_CREATE_SWITCH, the PF miniport driver must do the following:

1. If the PF miniport driver supports static switch creation and configuration, it creates the NIC switch when NDIS calls *MiniportInitializeEx*. When the driver handles this OID request, it must verify the configuration parameters in the **NDIS_NIC_SWITCH_PARAMETERS** structure. The parameters must be the same as those used by the driver to create the switch during the call to *MiniportInitializeEx*. If this is not true, the driver must fail the OID request.

   For more information, see Static Creation of a NIC Switch.

2. If the PF miniport driver supports dynamic switch creation and configuration, the driver must validate the configuration values of the **NDIS_NIC_SWITCH_PARAMETERS** structure and create the NIC switch based on these values.

   For more information, see Dynamic Creation of a NIC Switch.

3. The PF miniport driver must allocate the necessary hardware and software resources for the default VPort on the NIC switch.

   Note The default VPort is always created through an OID request of OID_NIC_SWITCH_CREATE_SWITCH and deleted through an OID request of OID_NIC_SWITCH_DELETE_SWITCH. OID requests of OID_NIC_SWITCH_CREATE_VPORT and OID_NIC_SWITCH_DELETE_VPORT are used for the creation and deletion of nondefault VPorts on the NIC switch.

4. The PF miniport driver that supports dynamic switch creation and configuration must enable SR-IOV virtualization on the switch by calling **NdisMEnableVirtualization**. This call configures the **NumVFs** member and the **VF Enable** bit in the SR-IOV Extended Capability structure of the adapter's PCI Express (PCIe) configuration space.

   For more information about the SR-IOV configuration space, see the PCI-SIG Single Root I/O Virtualization and Sharing 1.1 ↗ specification.

Note If the PF miniport driver supports static switch creation, it enables SR-IOV virtualization after it creates the switch when *MiniportInitializeEx* is called.

If the PF miniport driver successfully completesthe OID method request of OID_NIC_SWITCH_CREATE_SWITCH, it allows the following to occur:

- VFs can be allocated on the NIC switch through OID method requests of OID_NIC_SWITCH_ALLOCATE_VF.
- Nondefault VPorts can be created on the NIC switch through OID method requests of OID_NIC_SWITCH_CREATE_VPORT.

  The miniport driver is responsible for managing its pool of nondefault VPorts. The driver specifies the number of nondefault VPorts in its pool through the **NumVPorts** member of the **NDIS_NIC_SWITCH_INFO** structure. The driver returns this structure through an OID query request of OID_NIC_SWITCH_ENUM_SWITCHES.

  Note The network adapter must always create a default VPort from its pool for the PF.

Deleting a NIC Switch

Article • 12/15/2021

A network adapter that supports single root I/O virtualization (SR-IOV) must be able to delete a NIC switch. Only the miniport driver for the PCI Express (PCIe) Physical Function (PF) of the SR-IOV adapter can delete a NIC switch on the adapter.

Note Starting with NDIS 6.30 in Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

Prior to halting the PF miniport driver, NDIS deletes the NIC switch by issuing an object identifier (OID) set request of OID_NIC_SWITCH_DELETE_SWITCH. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_DELETE_SWITCH_PARAMETERS** structure that specifies the identifier of the switch being deleted.

NDIS enforces the following policies before issuing the OID set request of OID_NIC_SWITCH_DELETE_SWITCH to the PF miniport driver:

- NDIS guarantees that all receive filters have been cleared from the default and nondefault virtual ports (VPorts) on the NIC switch. Receive filters are cleared through an OID set request of OID_RECEIVE_FILTER_CLEAR_FILTER.

- NDIS guarantees that all nondefault virtual ports (VPorts) created on the switch have been previously deleted. VPorts are deleted through an OID set request of OID_NIC_SWITCH_DELETE_VPORT.

- NDIS guarantees that all the resources for PCIe Virtual Functions (VFs) attached to the NIC switch have been previously freed. VFs are freed through an OID set request of OID_NIC_SWITCH_FREE_VF.

When it receives the OID method request of OID_NIC_SWITCH_DELETE_SWITCH, the PF miniport driver must do the following:

1. If the PF miniport driver supports static creation and configuration of NIC switches, it must free the software resources associated with the specified NIC switch. However, the driver can only free the hardware resources for the NIC switch when *MiniportHaltEx* is called.

   For more information about static NIC switch creation, see Static Creation of a NIC Switch.

2. If the PF miniport driver supports the dynamic creation and configuration of NIC switches, it must free the hardware and software resources associated with the specified NIC switch.

   For more information about dynamic NIC switch creation, see Dynamic Creation of a NIC Switch.

3. If the PF miniport driver supports the dynamic creation of NIC switches and all the NIC switches have been deleted on the network adapter, the driver must disable virtualization on the adapter by calling **NdisMEnableVirtualization**. To disable virtualization, the network adapter must set the *EnableVirtualization* parameter to FALSE and the *NumVFs* parameter to zero.

   NdisMEnableVirtualization clears the **NumVFs** member and the **VF Enable** bit in the SR-IOV Extended Capability structure in the PCIe configuration space of the network adapter's PF.

   Note If the PF miniport driver supports static creation and configuration of NIC switches, it must only call **NdisMEnableVirtualization** when *MiniportHaltEx* is called.

Enumerating NIC Switches on a Network Adapter

Article • 12/15/2021

An overlying driver or user application can obtain a list of all NIC switches that have been created on a network adapter that supports single root I/O virtualization (SR-IOV). The driver or application issues an object identifier (OID) query request of OID_NIC_SWITCH_ENUM_SWITCHES to obtain this list.

After a successful return from this OID request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer that contains the following:

- An **NDIS_NIC_SWITCH_INFO_ARRAY** structure that defines the number of elements within the array.

- An array of **NDIS_NIC_SWITCH_INFO** structures. Each of these structures contains the information about a single NIC switch created on the network adapter.

  Note If the network adapter has no NIC switches, the driver sets the **NumElements** member of the **NDIS_NIC_SWITCH_INFO_ARRAY** structure to zero and no **NDIS_NIC_SWITCH_INFO** structures are returned.

Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

NDIS handles the OID_NIC_SWITCH_ENUM_SWITCHES request for miniport drivers. NDIS returns the information from an internal cache of the data that it maintains from the following sources:

- The standardized SR-IOV keyword settings in the registry. For more information on these keywords, see Standardized INF Keywords for SR-IOV.

- OID requests of OID_NIC_SWITCH_CREATE_SWITCH and OID_NIC_SWITCH_PARAMETERS.

Note NDIS also provides the enumeration of the switches in the **NicSwitchArray** member in the **NDIS_BIND_PARAMETERS** and **NDIS_FILTER_ATTACH_PARAMETERS** structures. Therefore, the overlying protocol and filter drivers do not have to issue OID_NIC_SWITCH_ENUM_SWITCHES requests to obtain this information.

Querying the Parameters of a NIC Switch

Article • 12/15/2021

An overlying driver or user application can obtain the parameters for a NIC switch that has been created on a network adapter that supports single root I/O virtualization (SR-IOV). The driver or application issues an object identifier (OID) method request of OID_NIC_SWITCH_PARAMETERS to obtain these parameters.

Before the overlying driver or user application issues this OID method request, it must initialize an **NDIS_NIC_SWITCH_PARAMETERS** structure. The driver or application must set the **SwitchId** member to the identifier of the NIC switch for which parameters are to be returned.

Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

After a successful return from this OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_PARAMETERS** structure. This structure contains the parameters for the specified switch.

NDIS handles the OID_NIC_SWITCH_PARAMETERS request for miniport drivers. NDIS returns the information from an internal cache of the data that it maintains from the following sources:

- The standardized SR-IOV keyword settings in the registry. For more information on these keywords, see Standardized INF Keywords for SR-IOV.

- OID requests of OID_NIC_SWITCH_CREATE_SWITCH and OID_NIC_SWITCH_PARAMETERS.

Setting the Parameters of a NIC Switch

Article • 12/15/2021

An overlying driver can change the parameters for a NIC switch that has been created on a network adapter that supports single root I/O virtualization (SR-IOV). The driver issues an object identifier (OID) set request of OID_NIC_SWITCH_PARAMETERS to change these parameters. Only the miniport driver for the PCI Express (PCIe) Physical Function (PF) of the SR-IOV adapter handles this OID set request.

Before the overlying driver issues this OID set request, it must initialize an **NDIS_NIC_SWITCH_PARAMETERS** structure with the parameters to be changed on the NIC switch. The driver then initializes an **NDIS_OID_REQUEST** structure for the OID request, and sets the **InformationBuffer** member to a pointer of the **NDIS_NIC_SWITCH_PARAMETERS** structure.

Only a limited subset of configuration parameters for a NIC switch can be changed. The overlying driver specifies the parameter to change by setting the following members of the **NDIS_NIC_SWITCH_PARAMETERS** structure:

- The **SwitchId** member is set to the identifier of the NIC switch whose parameters will be changed.

  Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*. The **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

- The appropriate NDIS_NIC_SWITCH_PARAMETERS_*Xxx*_CHANGED flags are set in the **Flags** member. Members of the **NDIS_NIC_SWITCH_PARAMETERS** structure can only be changed if a corresponding NDIS_NIC_SWITCH_PARAMETERS_*Xxx*_CHANGED flag is defined in Ntddndis.h.

- The members of the **NDIS_NIC_SWITCH_PARAMETERS** structure, which correspond to the NDIS_NIC_SWITCH_PARAMETERS_*Xxx*_CHANGED flags set in the **Flags** member, are set with the NIC switch configuration parameters that are to be changed.

  Note Starting with Windows Server 2012, only the **SwitchName** member of the **NDIS_NIC_SWITCH_PARAMETERS** structure can be changed through an OID set request of OID_NIC_SWITCH_PARAMETERS.

The PF miniport driver must follow these guidelines when it receives the OID set request of OID_NIC_SWITCH_PARAMETERS

- If the PF miniport driver can apply the changes without requiring a reinitialization of the network adapter, the driver applies the changes to the hardware and completes the OID request with NDIS_STATUS_SUCCESS.

  If this status code is returned, NDIS updates the NIC switch configuration information in the registry.

- If the PF miniport driver requires a reinitialization of the network adapter to apply the changes, the driver completes the OID request with NDIS_STATUS_REINIT_REQUIRED.

  If this status code is returned, NDIS also updates the NIC switch configuration information in the registry. However, the overlying driver that issued the OID set request must reinitialize the network adapter so that the changes can take effect.

  Note PF miniport drivers that support static NIC creation and configuration can return NDIS_STATUS_REINIT_REQUIRED to make sure that the adapter is reinitialized for the new parameters to take effect.

- If the PF miniport driver cannot apply the changes requested in the OID, it must fail the OID and return the appropriate NDIS_STATUS_*Xxx* code.

  In this case, NDIS does not update the NIC switch configuration information in the registry.

Managing Virtual Ports

Article • 12/15/2021

This section describes the requirements and guidelines for managing the virtual ports (VPorts) on a NIC switch. This switch is provided by a network adapter that supports single root I/O virtualization (SR-IOV).

This section includes the following topics:

Creating a Virtual Port

Deleting a Virtual Port

Enumerating Virtual Ports on a Network Adapter

Querying the Parameters of a Virtual Port

Setting the Parameters of a Virtual Port

Managing the Receive Filters for a Virtual Port

Symmetric and Asymmetric Assignment of Queue Pairs

Packet Flow over a Virtual Port

Nondefault Virtual Ports and VMQ

For more information on VPorts, see Virtual Ports (VPorts).

For more information on NIC switches, see NIC Switches.

Note Only the miniport driver for the PCI Express (PCIe) Physical Function (PF) can configure the network adapter's hardware resources, such as the VPorts. The miniport driver for the PCIe Virtual Function (VF) cannot directly access most of the SR-IOV adapter's hardware resources. For more information, see Writing SR-IOV VF Miniport Drivers.

Creating a Virtual Port

Article • 12/15/2021

A virtual port (VPort) is a data object that represents an internal port on the NIC switch of a network adapter that supports single root I/O virtualization (SR-IOV). Each NIC switch has the following ports for network connectivity:

- One external physical port for connectivity to the external physical network.
- One or more internal VPorts which are connected to the PCI Express (PCIe) Physical Function (PF) or Virtual Functions (VFs).

  The PF is attached to the Hyper-V parent partition and is exposed as a virtual network adapter in the management operating system that runs in that partition.

  A VF is attached to the Hyper-V child partition and is exposed as a virtual network adapter in the guest operating system that runs in that partition.

There are two types of VPorts:

Default VPort
The default VPort provides network connectivity to the networking components that run in the management operating system. The default VPort has an identifier of NDIS_DEFAULT_VPORT_ID.

When the PF miniport driver creates and configures the default NIC switch, the driver implicitly creates the default VPort and attaches it to the PF. The default VPort cannot be attached to a VF.

The default VPort is always in an activated state and cannot be explicitly deleted. The PF miniport driver implicitly deletes the default VPort only when it deletes the default NIC switch.

For more information on how to create a NIC switch and the default VPort on the switch, see Creating a NIC Switch.

Nondefault VPort
Nondefault VPorts are not created implicitly when the NIC switch is created. An overlying driver, such as the virtualization stack, explicitly creates these ports by issuing OID method requests of OID_NIC_SWITCH_CREATE_VPORT. Nondefault VPorts may be attached to the PF or to a VF, and can only be created after the NIC switch has been created.

A nondefault VPort that is attached to a VF provides network connectivity to the networking components that run in the guest operating system. After it is created and attached to the VF, the nondefault VPort is in an activated state.

A nondefault VPort that is attached to the PF provides additional network offload capabilities to the networking components that run in the management operating system. For example, nondefault VPorts on the PF could be used to provide offload capabilities similar to the virtual machine queue (VMQ) interface.

Note Nondefault VPorts can only be created after the NIC switch has been created.

An overlying driver issues an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_VPORT to create a nondefault VPort on a specified NIC switch. This OID request also attaches the created VPort to the network adapter's PF or a previously allocated VF.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to the**NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure. After a successful return from the OID_NIC_SWITCH_CREATE_VPORT request, the **VPortId** member of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure has a VPort identifier that is unique across the VPorts on the NIC switch.

The overlying driver initializes the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure with the configuration information about the nondefault VPort to be created. The configuration information includes the PCIe function to which the nondefault VPort is attached and the number of queue pairs for the nondefault VPort.

When it initializes the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure, the overlying driver must do the following:

- The **SwitchId** member must be set to the identifier of a NIC switch that was previously created on the network adapter through an OID method request of OID_NIC_SWITCH_CREATE_SWITCH.

  Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*. When creating a nondefault VPort, the overlying driver must set the **SwitchId** member to the NDIS_DEFAULT_SWITCH_ID identifier.

- The **VPortId** member must be set to NDIS_DEFAULT_VPORT_ID.

- The **AttachedFunctionId** member must be set to the identifier of the VF or PF on which the nondefault VPort is to be attached.

A value of NDIS_PF_FUNCTION_ID specifies the PF. Otherwise, the value must be set to the identifier of a VF whose resources were previously allocated through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

Note The attachment of a nondefault VPort to a VF or PF cannot be changed after the nondefault VPort has been created.

The overlying driver can also specify the number of queue pairs assigned to the VPort. A queue pair is a transmit and receive queue on the network adapter that is assigned to the VPort. If the network adapter supports asymmetric queue pairs for nondefault VPorts, the overlying driver may specify a different number of queue pairs for each VPort that the driver creates. For more information, see Symmetric and Asymmetric Assignment of Queue Pairs.

The overlying driver calls **NdisOidRequest** to issue the OID_NIC_SWITCH_CREATE_VPORT request to the underlying PF miniport driver. Before NDIS forwards the OID method request to the miniport driver, it does the following:

1. NDIS validates the parameters within the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure. If the parameters are in error, NDIS fails the OID method request and does not pass the request to the PF miniport driver.

2. NDIS assigns an identifier for the nondefault VPort within the range from one to (**NumVPorts**– 1), where **NumVPorts** is the number of VPorts that the miniport driver has configured on the network adapter. The driver specifies this number in the **NumVPorts** member of the **NDIS_NIC_SWITCH_INFO** structure. The driver returns this structure through an OID query request of OID_NIC_SWITCH_ENUM_SWITCHES.

   Note A VPort identifier of NDIS_DEFAULT_VPORT_ID is reserved for the default VPort that is attached to the PF on the default NIC switch.

The assigned VPort identifier uniquely identifies the nondefault VPort on the NIC switch of the network adapter.

3. NDIS sets the **VPortId** member of the NDIS_NIC_SWITCH_VPORT_PARAMETERS structure with the assigned VPort identifier.

When the PF miniport driver is issued the OID request, the driver allocates the hardware and software resources associated with the specified nondefault VPort. After all of the resources are successfully allocated, the PF miniport driver completes the OID successfully by returning NDIS_STATUS_SUCCESS from *MiniportOidRequest*.

If the OID_NIC_SWITCH_CREATE_VPORT request is completed successfully, the PF miniport driver and the overlying driver must retain the **VPortId** value of the nondefault VPort for successive operations. The **VPortId** value is used during these operations:

- NDIS and the overlying drivers use the **VPortId** value to identify the nondefault VPort in successive OID requests related to this VPort, such as OID_NIC_SWITCH_VPORT_PARAMETERS and OID_NIC_SWITCH_DELETE_VPORT.

- During send operations, NDIS specifies the **VPortId** value to identify the VPort from which a packet was sent. This value is specified within the out-of-band (OOB) **NDIS_NET_BUFFER_LIST_FILTERING_INFO** data of the NET_BUFFER_LIST structure.

- During receive operations, the PF miniport driver specifies the **VPortId** value to which a packet is to be forwarded. This value is also specified in the OOB **NDIS_NET_BUFFER_LIST_FILTERING_INFO** data of the NET_BUFFER_LIST structure.

The following points apply to the creation of nondefault VPorts:

- Receive filters for media access control (MAC) and virtual LAN (VLAN) identifiers are configured on the VPort after it has been created. Overlying drivers dynamically set these receive filters by issuing OID method requests of OID_RECEIVE_FILTER_SET_FILTER. Receive filters can also be moved from one VPort to another through OID set requests of OID_RECEIVE_FILTER_MOVE_FILTER.

- A nondefault VPort attached to the VF is in an activated state when it is created. The VPort cannot be deactivated if it is attached to the VF.

  A nondefault VPort attached to the PF is in a deactivated state when it is created. An overlying driver, such as the Hyper-V extensible switch module, explicitly activates the nondefault VPort attached to the PF after the VPort has been created successfully. This is done by issuing an OID method request of OID_NIC_SWITCH_VPORT_PARAMETERS to the PF miniport driver.

  When the overlying driver issues this OID request, it passes an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure with the **VPortState** member set to **NdisNicSwitchVPortStateActivated**.

  After a nondefault VPort is in an activated state, the PF miniport driver can allocate shared memory for the VPort by calling **NdisAllocateSharedMemory**. The driver must set the **VPortId** member in the **NDIS_SHARED_MEMORY_PARAMETERS** structure to the VPort's identifier value.

Note When a nondefault VPort is in an activated state, it is only set to a deactivated state when it is deleted through an OID set request of OID_NIC_SWITCH_DELETE_VPORT.

Deleting a Virtual Port

Article • 12/15/2021

An overlying driver issues an object identifier (OID) set request of OID_NIC_SWITCH_DELETE_VPORT to delete a nondefault virtual port (VPort) on a network adapter's NIC switch. The overlying driver can only delete a VPort that it has previously created by issuing an OID method request of OID_NIC_SWITCH_CREATE_VPORT.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to the **NDIS_NIC_SWITCH_DELETE_VPORT_PARAMETERS** structure.

An overlying driver, such as the virtualization stack, can delete a nondefault VPort that it has previously created. The overlying driver creates a VPort by issuing an OID method request of OID_NIC_SWITCH_CREATE_VPORT.

Before it issues the OID set request of OID_NIC_SWITCH_DELETE_VPORT, the overlying driver must do the following:

- The overlying drivers must clear or move all receive filters that the driver previously set on the VPort before deleting the VPort. Receive filters are set through OID requests of OID_RECEIVE_FILTER_SET_FILTER and are moved through OID requests of OID_RECEIVE_FILTER_MOVE_FILTER.

- The overlying driver sets the **VPortId** member of the **NDIS_NIC_SWITCH_DELETE_VPORT_PARAMETERS** structure to the identifier of the nondefault VPort to be deleted.

  Note The overlying driver must not set the **VPortId** member to **NDIS_DEFAULT_PORT_NUMBER**. This VPort identifier is reserved for the default VPort that is attached to the PCI Express (PCIe) Physical Function (PF) on the network adapter. The default VPort always exists and is not deleted explicitly though an OID set request of OID_NIC_SWITCH_DELETE_VPORT.

The overlying driver calls **NdisOidRequest** to issue the OID_NIC_SWITCH_DELETE_VPORT request to the underlying PF miniport driver. When the miniport driver receives the OID_NIC_SWITCH_DELETE_VPORT request, the driver must do the following:

- The driver must free the hardware and software resources that were allocated for the specified VPort.

- The driver must detach the specified VPort from the PF or a PCIe Virtual Function (VF).

If the VPort is attached to a VF, the virtualization stack ensures that the VF miniport driver that runs in the guest operating system has been previously paused and halted. As a result, all previouslyindicated receive packets from the VPort should have been returned to the VF miniport driver.

If the VPort is attached to the PF, the PF miniport driver must stop any additional DMA to the shared memory associated with the VPort. The PF miniport driver must make sure that all previouslyindicated receive packets from the VPort are returned to the miniport. The PF miniport driver must not make any additional receive indications to NDIS that specify the VPort's identifier in the packet's **NET_BUFFER_LIST** structure. After all of the indicated receive packets from the VPort are returned to the PF miniport driver, it must free the shared memory associated with the VPort by calling **NdisFreeSharedMemory**.

The following points apply to the deletion of VPorts:

- The overlying protocol driver must delete all nondefault VPorts that it created before it calls **NdisCloseAdapterEx**.

- The overlying filter driver must delete all nondefault VPorts that it created within its *FilterDetach* function.

- Before NDIS issues a set request of OID_NIC_SWITCH_DELETE_SWITCH to delete a NIC switch on the network adapter, it guarantees that all nondefault VPorts are deleted from that switch.

- Only nondefault VPorts can be explicitly deleted through OID requests of OID_NIC_SWITCH_DELETE_SWITCH. The default VPort is implicitly deleted when the PF miniport driver deletes the default NIC switch. For more information, see Deleting a NIC Switch.

Enumerating Virtual Ports on a Network Adapter

Article • 12/15/2021

An overlying driver or user application can obtain a list of all virtual ports (VPorts) on a NIC switch of a network adapter that supports single root I/O virtualization (SR-IOV). The driver or application issues an object identifier (OID) method request of OID_NIC_SWITCH_ENUM_VPORTS to obtain this list.

After a successful return from this OID query request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer that contains the following:

- An **NDIS_NIC_SWITCH_VPORT_INFO_ARRAY** structure that defines the number of elements within the array.

- An array of **NDIS_NIC_SWITCH_VPORT_INFO** structures. Each of these structures contains information about a VPort on the network adapter's NIC switch.

  Note If no VPorts have been created on the network adapter, the driver sets the **NumElements** member of the **NDIS_NIC_SWITCH_VPORT_INFO_ARRAY** structure to zero and no **NDIS_NIC_SWITCH_VPORT_INFO** structures are returned.

Before the overlying driver or user application issues the OID_NIC_SWITCH_ENUM_VPORTS request, it must initialize an **NDIS_NIC_SWITCH_VPORT_INFO_ARRAY** structure that is passed along with the request. The driver or application must follow these guidelines when initializing the **NDIS_NIC_SWITCH_VPORT_INFO_ARRAY** structure:

- If the NDIS_NIC_SWITCH_VPORT_INFO_ARRAY_ENUM_ON_SPECIFIC_SWITCH flag is set in the **Flags** member, information is returned for all VPorts created on a specified NIC switch. The NIC switch is specified by the **SwitchId** member of that structure.

  Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier. Regardless of the flags that are set in the **Flags** member, the **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

- If the NDIS_NIC_SWITCH_VPORT_INFO_ARRAY_ENUM_ON_SPECIFIC_FUNCTION flag is set in the **Flags** member, information is returned for all VPorts attached to a specified PCI Express (PCIe) Physical Function (PF) or Virtual Function (VF) on the network adapter. The PF or VF is specified by the **AttachedFunctionId** member of that structure.

  If the **AttachedFunctionId** member is set to NDIS_PF_FUNCTION_ID, information is returned for all VPorts. This includes the default VPort that is attached to the PF. If the **AttachedFunctionId** member is set to a valid VF identifier, information is returned for all VPorts attached to the specified VF.

  Note Starting with Windows Server 2012, only one nondefault VPort can be attached to a VF. However, multiple VPorts (including the default VPort) can be attached to the PF.

- If the **Flags** member is set to zero, information is returned for all VPorts attached to the PF or VF on the network adapter. In this case, the values of the **SwitchId** and **AttachedFunctionId** are ignored.

NDIS handles the OID_NIC_SWITCH_ENUM_VPORTS request for miniport drivers. NDIS returns the information from an internal cache of the data that it maintains from inspecting the following sources:

- OID method requests of OID_NIC_SWITCH_CREATE_VPORT.

- OID set requests of OID_NIC_SWITCH_VPORT_PARAMETERS.

Querying the Parameters of a Virtual Port

Article • 12/15/2021

An overlying driver can obtain the parameters for a virtual port (VPort) on a NIC switch on a network adapter that supports single root I/O virtualization (SR-IOV). The driver issues an object identifier (OID) method request of OID_NIC_SWITCH_VPORT_PARAMETERS to obtain these parameters.

Before the overlying driver issues this OID method request, it must initialize an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure. The driver must set the members of this structure in the following way:

- The **SwitchId** member must be set to the identifier of the NIC switch for which parameters are to be returned.

  Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*. The **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

- The **VPortId** member must be set to the identifier associated with the VPort. The overlying driver obtains the VPort identifier through one of the following ways:

  - From a previous OID method request of OID_NIC_SWITCH_CREATE_VPORT.

  - From a previous OID method request of OID_NIC_SWITCH_ENUM_VPORTS.

After a successful return from this OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure. This structure contains the parameters for the specified VPort.

NDIS handles the OID_NIC_SWITCH_VPORT_PARAMETERS request for miniport drivers. NDIS returns the information from an internal cache of the data that it maintains from inspecting the following sources:

- OID method requests of OID_NIC_SWITCH_CREATE_VPORT.

- OID set requests of OID_NIC_SWITCH_VPORT_PARAMETERS.

Setting the Parameters of a Virtual Port

Article • 12/15/2021

An overlying driver can change the parameters for a virtual port (VPort) on a NIC switch on a network adapter that supports single root I/O virtualization (SR-IOV). The driver issues an object identifier (OID) set request of OID_NIC_SWITCH_VPORT_PARAMETERS to change these parameters.

Before the overlying driver issues this OID set request, it must initialize an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure with the parameters to be changed on the VPort. The driver then initializes an **NDIS_OID_REQUEST** structure for the OID request, and sets the **InformationBuffer** member to a pointer to the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure.

Only a limited subset of configuration parameters for a VPort can be changed. The overlying driver specifies the parameter to change by setting the following members of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure:

- The **SwitchId** member must be set to the identifier of the NIC switch for which parameters are to be returned.

  Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*. The **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

- The **VPortId** member must be set to the identifier associated with the VPort. The overlying driver obtains the VPort identifier through one of the following ways:

  - From a previous OID method request of OID_NIC_SWITCH_CREATE_VPORT.

  - From a previous OID method request of OID_NIC_SWITCH_ENUM_VPORTS.

- The appropriate NDIS_NIC_SWITCH_VPORT_PARAMS_*Xxx*_CHANGED flags must be set in the **Flags** member. Members of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure can only be changed if a corresponding NDIS_NIC_SWITCH_VPORT_PARAMS_*Xxx*_CHANGED flag is defined in Ntddndis.h.

- The members of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure, which correspond to the NDIS_NIC_SWITCH_VPORT_PARAMS_*Xxx*_CHANGED flags set in the **Flags** member, are set with the VPort configuration parameters that are to be changed.

Starting with Windows Server 2012, only the following members of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure can be changed through an OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS:

PortName
This member contains a user-friendly description of the VPort.

InterruptModeration
This member specifies the interrupt moderation setting of the VPort.

ProcessorAffinity
This member specifies the group number and a bitmap of the CPUs that this VPort can be associated with.

The overlying driver must follow these guidelines for changing the **ProcessorAffinity** member for a VPort:

- This member is valid only for the VPorts attached to the PF. This field is not valid for nondefault VPorts that are attached to a VF.

- For nondefault VPorts that are attached to the PF, at least one processor must be specified when the VPort is created. The processor affinity associated with the nondefault VPort can be changed after VPort creation.

  Note Nondefault VPorts are created through OID method requests of OID_NIC_SWITCH_CREATE_VPORT.

VPortState
This member specifies the current state of the VPort.

The overlying driver must follow these guidelines for changing the **VPortState** member for a VPort:

- For a nondefault VPort that is attached to a VF, the **VPortState** member must always be set to **NdisNicSwitchVPortStateActivated**.

- For a nondefault VPort that is attached to the PF, the **VPortState** member must be set to **NdisNicSwitchVPortStateDeactivated** when the VPort is created. The PF VPort is activated only after an OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS is issued by the overlying drivers to change the VPortState to an activated state.

  When the nondefault VPort is activated, the PF miniport driver can allocate resources for the VPort, such as shared memory that is allocated through **NdisAllocateSharedMemory**. The PF miniport driver may allocate resources for

VPort any time after it is activated until the driver deletes the VPort through an OID set request of OID_NIC_SWITCH_DELETE_VPORT.

- The default VPort is always in an activated state. The value of the **VPortState** member must always be set to **NdisNicSwitchVPortStateActivated** for the default VPort.

- When a VPort is in an activated state, it cannot be deactivated. A PF miniport driver can receive and transmit packets from a VPort only if it is in an activated state and the corresponding MAC filters are set on the VPort. However, after the VPort is deleted through an OID set request of OID_NIC_SWITCH_DELETE_VPORT, the driver must free the resources that were allocated for the VPort. The driver must also cancel all pending transmit or receive operations for packets on the VPort.

After the PF miniport driver receives the OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS, the driver configures the hardware with the configuration parameters. The driver can only change those configuration parameters identified by NDIS_NIC_SWITCH_VPORT_PARAMETERS_*Xxx*_CHANGED flags in the **Flags** member of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure.

Managing the Receive Filters for a Virtual Port

Article • 12/15/2021

Receive filters can be set, cleared, moved, and enumerated on a virtual port (VPort) after the VPort has been created on the NIC switch.

This section includes the following topics:

Setting a Receive Filter on a Virtual Port

Clearing a Receive Filter on a Virtual Port

Moving a Receive Filter to a Virtual Port

Enumerating Receive Filters on a Virtual Port

For information about how to create VPorts, see Creating a Virtual Port.

Setting a Receive Filter on a Virtual Port

Article • 12/15/2021

After a virtual port (VPort) is created on the NIC switch of the network adapter, overlying drivers can set receive filters on the VPort. Only the driver that created the VPort can set a receive filter on that VPort

This topic contains the following information:

Setting a Receive Filter on a VPort

Using the NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO Flag

Using the Filter Identifier

Handling Receive Filters on a VPort

For more information on how to create a VPort, see Creating a Virtual Port.

Note Because the default VPort always exists and is never explicitly created, any overlying driver can set a receive filter on the default VPort. Overlying drivers do not own the default VPort. Therefore, all protocol drivers that are bound to a network adapter can use the default VPort. The default VPort has an identifier value of NDIS_DEFAULT_VPORT_ID.

Setting a Receive Filter on a VPort

To set and configure a filter on a VPort, an overlying driver issues an object identifier (OID) method request of OID_RECEIVE_FILTER_SET_FILTER. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure.

Before the overlying driver issues this OID method request, it must initialize an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. The driver must set the members of this structure in the following way:

- The **FilterType** member must be set to an **NDIS_RECEIVE_FILTER_TYPE** enumeration value.

  Note Starting with NDIS 6.30, only **NdisReceiveFilterTypeVMQueue** filter types are supported for the single root I/O virtualization (SR-IOV) interface.

- The **QueueId** member must be set to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

- The **VPortId** member must be set to the identifier associated with the VPort. The overlying driver obtains the VPort identifier through one of the following ways:

  - From a previous OID method request of OID_NIC_SWITCH_CREATE_VPORT.

  - From a previous OID method request of OID_NIC_SWITCH_ENUM_VPORTS.

- The **FilterId** member must be set to NDIS_DEFAULT_RECEIVE_FILTER_ID.

  Note NDIS assigns a unique filter identifier in this member before it forwards the OID request to the miniport driver for processing.

- The **FieldParametersArrayOffset**, **FieldParametersArrayNumElements**, and **FieldParametersArrayElementSize** members of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure must be set appropriately to define an array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures. Each **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure in the array sets the filter test criterion for one field in a network header.

  For the SR-IOV interface, the following field test parameters are defined:

  - The destination media access control (MAC) address in the packet equals the specified MAC address.

  - The virtual LAN (VLAN) identifier in the packet equals the specified VLAN identifier.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure with a new filter identifier.

Using the NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO Flag

The **Flags** member of the NDIS_RECEIVE_FILTER_FIELD_PARAMETERS structure specify actions to be performed for the receive filter. The following points apply to the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag:

- If the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is set in the **Flags** member, the network adapter must indicate only received packets that match all of the following test criteria:

  - A packet with a matching MAC address.

  - A packet that has no VLAN tag or has a VLAN identifier of zero.

  If the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is set, the network adapter must not indicate packets that have a matching MAC address and a nonzero VLAN identifier.

  Note If the virtualization stack sets the MAC address filter and no VLAN identifier filter is configured by the OID_RECEIVE_FILTER_SET_FILTER set request, the switch also sets the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag.

- Starting with NDIS 6.30, if the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is not set and there is no VLAN identifier filter configured by the OID_RECEIVE_FILTER_SET_FILTER method request, the miniport driver must do either one of the following:

  - The miniport driver must return a failed status for the OID_RECEIVE_FILTER_SET_FILTER method request.

  - The miniport driver must configure the network adapter to inspect and filter the specified MAC address fields. If a VLAN tag is present in the received packet, the network adapter must remove it from the packet data. The miniport driver must put the VLAN tag in an **NDIS_NET_BUFFER_LIST_8021Q_INFO** that is associated with the packet's **NET_BUFFER_LIST** structure.

- If a protocol driver sets a MAC address filter and a VLAN identifier filter with the OID_RECEIVE_FILTER_SET_FILTER method request, it does not set the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag in either of the filter fields. In this case, the miniport driver should indicate packets that match both the specified MAC address and the VLAN identifier. That is, the miniport driver should not indicate packets with a matching MAC address that have a zero VLAN identifier or are untagged packets.

Using the Filter Identifier

NDIS assigns a filter identifier in the **FilterId** member of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure and passes the OID method request of OID_RECEIVE_FILTER_SET_FILTER to the underlying miniport driver. Each filter that is set on a VPort has a unique filter identifier for a network adapter. That is, the filter identifiers are not duplicated on different queues that the network adapter manages.

The overlying driver must use the filter identifier that NDIS provides in later OID requests to change the filter parameters or to free a filter.

When NDIS receives an OID request to set a filter on a VPort, it verifies the filter parameters. After NDIS allocates the necessary resources and the filter identifier, it submits the OID request to the underlying network adapter. If the network adapter can successfully allocate the necessary software and hardware resources for the filter, it completes the OID request with **NDIS_STATUS_SUCCESS**.

The miniport driver must retain the filter identifiers for the allocated receive filters. NDIS uses the filter identifier of a filter with later OID requests to change the receive filter parameters or clear the receive filter. For more information about how to change parameters and clear filters, see Obtaining and Updating VM Queue Parameters and Clearing a VMQ Filter.

Handling Receive Filters on a VPort

The miniport driver programs the network adapter based on the filters in the following way:

- All field test parameters for a particular filter must match to assign a packet to the VPort.
- Multiple filters can be set on a VPort.
- Packets must be assigned to the VPort if any of the filters pass.

The network adapter combines the results from all the field tests with a logical **AND** operation. That is, if any field test that is included in the array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures fails, the network packet does not meet the specified filter criterion.

When a network adapter tests a received packet against these filter criteria, it must ignore all fields in the packet that have no test criteria that were specified.

Clearing a Receive Filter on a Virtual Port

Article • 12/15/2021

To clear a receive filter from a virtual port (VPort) on the NIC switch, an overlying driver issues an object identifier (OID) set request of OID_RECEIVE_FILTER_CLEAR_FILTER. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS** structure.

Before the overlying driver issues the OID_RECEIVE_FILTER_CLEAR_FILTER request, it must initialize the **NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS** structure and set the members in the following way:

- The **QueueId** member must be set to NDIS_DEFAULT_RECEIVE_QUEUE_ID.
- The **FilterId** member must be set to the filter identifier value that the driver obtained from an earlier OID_RECEIVE_FILTER_SET_FILTER method OID request. For more information about how to set receive filters, see Setting a Receive Filter on a Virtual Port.

An overlying driver must clear all of the filters that it set on a VPort before it frees the VPort. An overlying driver must also clear all of the filters that it set on the default VPort before it closes its binding to or detached from the network adapter.

Moving a Receive Filter to a Virtual Port

Article • 12/15/2021

The overlying driver issues an object identifier (OID) set request of OID_RECEIVE_FILTER_MOVE_FILTER to move a receive filter from a virtual port (VPort) to another VPort on the NIC switch. Typically, the overlying driver, such as the virtualization stack, issues this OID request if any of the following conditions are true:

- The virtualization stack sets a receive filter on the default VPort. This filter contains the media access control (MAC) address and virtual LAN (VLAN) parameters for the virtual machine (VM) network adapter that is exposed in the Hyper-V child partition. This allows packets to be forwarded between the VM network adapter and the underlying network adapter over the software-based synthetic data path.

  After resources for a PCI Express (PCIe) Virtual Function (VF) are allocated and the VF is attached to a child partition, the virtualization stack creates a nondefault VPort on the VF. The virtualization stack then moves the receive filter for the VM network adapter from the default VPort to the nondefault VPort attached to the VF. This allows packets to be forwarded between the VM network adapter and the underlying network adapter over the hardware-based VF data path.

  For more information about these data paths, see SR-IOV Data Paths.

- A VF has been detached from a Hyper-V child partition in which the guest operating system is still running. In this case, the overlying driver issues the OID set request to move the receive filter for the VM network adapter from the nondefault VPort to the default VPort attached to the PF. When this happens, packet traffic reverts to the synthetic data path.

To move a receive filter from one VPort to another VPort, an overlying driver issues an OID set request of OID_RECEIVE_FILTER_MOVE_FILTER. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS** structure.

Before the overlying driver issues the OID_RECEIVE_FILTER_MOVE_FILTER request, it must initialize an **NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS** structure in the following way:

- The driver sets the **FilterId** member to the identifier of the identifier of the previously allocated receive filter.

  Note The overlying driver obtained the filter identifier from an earlier OID method request of OID_RECEIVE_FILTER_SET_FILTER or OID_RECEIVE_FILTER_ENUM_FILTERS.

- The driver sets the **SourceQueueId** member to NDIS_DEFAULT_RECEIVE_QUEUE_ID.
- The driver sets the **SourceVPortId** member to the identifier of the VPort on which this filter was previously set.
- The driver sets the **DestQueueId** member to NDIS_DEFAULT_RECEIVE_QUEUE_ID.
- The driver sets the **DestVPortId** member to the identifier of the VPort on which this filter is to be moved.

NDIS validates the members of the NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS before it forwards the OID set request to the PF miniport driver.

When the PF miniport driver handles this OID set request, it must move the receive filter in an atomic operation. The driver must be able to configure the network adapter to simultaneously remove the filter from a receive queue and VPort and set it on a different receive queue and VPort.

Enumerating Receive Filters on a Virtual Port

Article • 12/15/2021

After a virtual port (VPort) is created on the NIC switch of the network adapter, overlying drivers and user applications can do the following:

- Enumerate the parameters for receive filters on a VPort.

  For more information, see Enumerating Receive Filters.

- Query the parameters for a specific receive filter.

  For more information, see Querying a Specific Receive Filter.

For more information on how to create a VPort, see Creating a Virtual Port.

Enumerating Receive Filters

To obtain a list of all receive filters that are set on a virtual port (VPort) of a NIC switch, overlying drivers and applications can issue object identifier (OID) method requests of OID_RECEIVE_FILTER_ENUM_FILTERS.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure.

Before the overlying driver or user application issues this OID method request, it must initialize an **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure and set the members of this structure in the following way:

- The **QueueId** member must be set to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

- The **VPortId** member must be set to the identifier associated with the VPort. The overlying driver obtains the VPort identifier through one of the following ways:

  - From a previous OID method request of OID_NIC_SWITCH_CREATE_VPORT.

  - From a previous OID method request of OID_NIC_SWITCH_ENUM_VPORTS.

  Note This member is only valid if the driver or application sets the NDIS_RECEIVE_FILTER_INFO_ARRAY_VPORT_ID_SPECIFIED flag in the **Flags** member. If this flag is not set, receive filters are returned that were set on every VPort on the NIC switch.

After a successful return from the OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an updated **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure that is followed by one or more **NDIS_RECEIVE_FILTER_INFO** structures. Each **NDIS_RECEIVE_FILTER_INFO** structure specifies the unique identifier for the receive filter that is set on the specified VPort.

Querying a Specific Receive Filter

Overlying drivers or applications can issue an OID method request of OID_RECEIVE_FILTER_PARAMETERS to obtain the parameters of a specific filter on a VPort.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure.

Before the overlying driver or user application issues this OID method request, it must initialize an **NDIS_RECEIVE_FILTER_PARAMETERS** structure and set the members of this structure in the following way:

- The **FilterId** member must be set to the nonzero identifier value of the filter whose parameters are to be returned.

  Note The overlying driver obtained the filter identifier from an earlier OID method request of OID_RECEIVE_FILTER_SET_FILTER or OID_RECEIVE_FILTER_ENUM_FILTERS. The application can obtain the filter identifier only from an earlier OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS.

- The **QueueId** member must be set to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

- The **VPortId** member must be set to the identifier associated with the VPort. The overlying driver obtains the VPort identifier through one of the following ways:

  - From a previous OID method request of OID_NIC_SWITCH_CREATE_VPORT.

  - From a previous OID method request of OID_NIC_SWITCH_ENUM_VPORTS.

NDIS handles the OID_RECEIVE_FILTER_ENUM_FILTERS and OID_RECEIVE_FILTER_PARAMETERS method OID requests for miniport drivers. NDIS obtained the information from an internal cache of the data that it received from the OID_RECEIVE_FILTER_SET_FILTER OID request.

Symmetric and Asymmetric Assignment of Queue Pairs

Article • 12/15/2021

A queue pair consists of a separate transmit and receive queue on the network adapter. Queue pairs are configured on a virtual port (VPort) when the VPort is created. Queue pairs associated with the default VPort are configured at the time of switch creation through an OID method request of OID_NIC_SWITCH_CREATE_SWITCH. One or more queue pairs are configured on a nondefault VPort through an OID method request of OID_NIC_SWITCH_CREATE_VPORT.

Each nondefault VPort can be configured to have a different number of queue pairs. This is known as *asymmetric allocation* of queue pairs. If the miniport driver does not support asymmetric allocations, each nondefault VPort is configured to have an equal number of queue pairs. This is known as *symmetric allocation* of queue pairs.

The miniport driver advertises its VPort and queue pair capabilities during *MiniportInitializeEx* by using an **NDIS_NIC_SWITCH_CAPABILITIES** structure. The driver advertises its support for asymmetric allocation of queue pairs by setting the NDIS_NIC_SWITCH_CAPS_ASYMMETRIC_QUEUE_PAIRS_FOR_NONDEFAULT_VPORT_SUPPORTED flag in the **NicSwitchCapabilities** member of this structure.

If the miniport driver supports asymmetric queue pair allocation, the virtualization stack configures each nondefault VPort with a different number of queue pairs. If the miniport driver supports symmetric queue pair allocation, the virtualization stack configures each VPort with the same number of queue pairs.

Note A miniport driver that supports either symmetric or asymmetric queue pair allocation on nondefault VPorts must support a different number of queue pairs to be allocated on the default VPort. The default VPort is always attached to the network adapter's PF.

The queue pair configuration is specified when the nondefault VPort is created or updated through OID requests of OID_NIC_SWITCH_CREATE_VPORT and OID_NIC_SWITCH_VPORT_PARAMETERS. The configuration parameters are specified in an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure that is associated with both OID requests.

For example, assume that the miniport driver advertises the configuration for VPorts and queue pairs on the NIC switch by setting the following members of the **NDIS_NIC_SWITCH_CAPABILITIES** structure:

- **MaxNumQueuePairs** is set to 128.
- **MaxNumVPorts** is set to 64.
- **MaxNumQueuePairsPerNonDefaultPort** is set to 4.

If the miniport driver does not support asymmetric configuration of queue pairs on nondefault VPorts, the virtualization stack can specify the following queue pair configuration when VPorts are created:

- 63 nondefault VF VPorts with two queue pairs each, together with the default PF VPort with one queue pair.
- 31 nondefault VF VPorts with four queue pairs each, together with the default PF VPort with one queue pair.

Note Starting with Windows Server 2012, only one default VPort is supported and is always attached to the network adapter's PF.

Packet Flow over a Virtual Port

Article • 12/15/2021

The default NIC switch is a component of a network adapter that supports the single root I/O virtualization (SR-IOV) interface. The switch always attaches the default virtual port (VPort) to the PCI Express (PCIe) Physical Function (PF). The switch can attach one or more nondefault VPorts to the PF. For more information, see Creating a Virtual Port.

The following points apply to packets that are sent or received on a VPort that is attached to the PF:

- Packets sent or received over the default VPort are specified with a VPort identifier value of **DEFAULT_VPORT_ID**.

  Packets sent or received over nondefault VPorts are specified with the VPort identifier that was returned when the VPort was created through an OID method request of OID_NIC_SWITCH_CREATE_VPORT. When the driver handles this OID request, it obtains the VPort identifier from the **VPortId** member of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure that is associated with the OID request.

  Note When a VPort is deleted, it is possible for the miniport driver to receive an NBL that contains an invalid **VPortId** value. If this happens, the miniport should ignore the invalid VPort ID and use **DEFAULT_VPORT_ID** instead. The **VPortId** is found in the **NetBufferListFilteringInfo** portion of the NBL's OOB data, and is retrieved by using the **NET_BUFFER_LIST_RECEIVE_FILTER_VPORT_ID** macro.

- The PF miniport driver calls **NdisMIndicateReceiveNetBufferLists** to indicate packets received from a VPort. Before the PF miniport driver calls **NdisMIndicateReceiveNetBufferLists**, it must set the VPort identifier in the out-of-band (OOB) data in the **NET_BUFFER_LIST** structure for the packet. The driver does this by using the **NET_BUFFER_LIST_RECEIVE_FILTER_VPORT_ID** macro.

- The virtualization stack calls **NdisSendNetBufferLists** to transmit packets to a VPort. Before the virtualization stack calls **NdisSendNetBufferLists**, it sets the VPort identifier in the OOB data in the **NET_BUFFER_LIST** structure for the packet.

  The miniport driver obtains the VPort identifier by using the **NET_BUFFER_LIST_RECEIVE_FILTER_VPORT_ID** macro.

  The miniport driver must queue the transmit packet on the hardware transmit queue of the specified VPort.

Note The miniport driver for the PCIe Virtual Function (VF) does not set or query the VPort identifier in the OOB data of the **NET_BUFFER_LIST** structure for a packet. When the VF miniport driver sends a packet, it queues the packet on the hardware transmit queue for the single nondefault VPort that is attached to the VF.

Nondefault Virtual Ports and VMQ

Article • 07/07/2022

The default NIC switch is a component of a network adapter that supports the single root I/O virtualization (SR-IOV) interface. The switch always attaches the default virtual port (VPort) to the PCI Express (PCIe) Physical Function (PF). The switch can attach one or more nondefault VPorts to the PF. For more information, see Creating a Virtual Port.

The virtualization stack runs in the management operating system of the Hyper-V parent partition. This stack creates VPorts by issuing object identifier (OID) method requests of OID_NIC_SWITCH_CREATE_VPORT. However, the stack can create more VPorts than the number of active PCIe Virtual Functions (VFs) for which resources have been allocated through OID method requests of OID_NIC_SWITCH_ALLOCATE_VF.

If SR-IOV is enabled on a network adapter, full VMQ functionality must be disabled. However, nondefault VPorts that are attached to the PF and not attached to a VF can provide the same functionality as the virtual machine queue (VMQ) interface. The following points discuss how VPorts can provide hardware-accelerated data paths for packet transfer that is similar to VMQ:

- VMQ determines the target VM by media access control (MAC) filtering in hardware. This avoids the overhead of a determining the target VM in the virtualization stack.

  Starting with Windows Server 2012, the virtualization stack configures the receive filters on the VPort by issuing OID method requests of OID_RECEIVE_FILTER_SET_FILTER. For this OID request, the virtualization stack passes an **NDIS_RECEIVE_FILTER_PARAMETERS** structure that specifies the MAC address and virtual LAN (VLAN) identifier that is associated with the virtual network adapter. Similar to VMQ, it can configure multiple MAC address and VLAN ID pairs on the VPort. The virtualization stack also specifies the target VPort to which the receive filter will be set.

  The SR-IOV network adapter performs similar hardware filtering based on the filtering criteria that is specified through the OID_RECEIVE_FILTER_SET_FILTER request. When a packet is received on the hardware receive queue of a VPort, the miniport driver specifies the source VPort identifier in the out-of-band (OOB) data of a **NET_BUFFER_LIST** structure for the packet. Based on the VPort identifier, the virtualization stack determines the target VM and indicates the packets to the network stack that runs in the VM.

Similarly, the virtualization stack specifies the target VPort identifier in the OOB data of a **NET_BUFFER_LIST** structure for a transmit packet. When the driver handles the send request for the packet, it places the packet in the hardware transmit queue of the specified VPort.

The VPort identifier can be obtained from the packet's OOB data by using the **NET_BUFFER_LIST_RECEIVE_FILTER_VPORT_ID** macro.

For more information about this process, see Packet Flow over a Virtual Port.

For more information about the receive filtering requirements for an SR-IOV network adapter, see Determining Receive Filtering Capabilities.

- VMQ provides interrupt and DPC concurrency.

  Starting with NDIS 6.30 and Windows Server 2012, a VPort attached to the PF can be configured to have a specific CPU affinity. The virtualization stack configures the CPU affinity and interrupt moderation parameters for a VPort by using OID method requests of OID_NIC_SWITCH_CREATE_VPORT or OID_NIC_SWITCH_VPORT_PARAMETERS. By doing this, the virtualization stack configures interrupt-based parameters similar to VMQ for interrupt and DPC concurrency.

  For example, when the SR-IOV network adapter receives packets on a VPort that is configured to have a specific CPU affinity, the adapter generates the interrupts on the specified CPU. The miniport driver indicates the received packets to NDIS and the virtualization stack for that CPU.

The PF miniport driver advertises its SR-IOV capabilities within the context of the call to *MiniportInitializeEx*. The driver initializes an **NDIS_SRIOV_CAPABILITIES** structure with its capabilities and calls **NdisMSetMiniportAttributes** to register its capabilities. For more information, see Determining SR-IOV Capabilities.

The following members of the **NDIS_NIC_SWITCH_CAPABILITIES** structure affect the way that VPorts are allocated:

- **MaxNumVPorts**, which specifies the maximum number of VPorts that can be created on the network adapter.

- **MaxNumVFs**, which specifies the maximum number of VFs that can be allocated on the network adapter.

Starting with NDIS 6.30, when the miniport driver initializes the **NDIS_NIC_SWITCH_CAPABILITIES** structure, it can set the NDIS_NIC_SWITCH_CAPS_SINGLE_VPORT_POOL flag in the **NicSwitchCapabilities** member. This flag specifies that the nondefault VPorts can be created in a nonreserved manner from the VPort pool on the network adapter. This allows available nondefault VPorts to be created and assigned on an as-needed basis to the PF and allocated VFs. If the network adapter supports the VMQ interface, nondefault VPorts that are assigned to the PF can also be used for VM receive queues.

If the NDIS_NIC_SWITCH_CAPS_SINGLE_VPORT_POOL flag is set, available nondefault VPorts are created and assigned to the PF and allocated VFs. The maximum number of VPorts that can be created and assigned to the PF is the same value that the driver reports in the **MaxNumVPorts** member. The miniport driver must reserve one VPort to be used as the default VPort that is assigned to the PF. As a result, the maximum number of nondefault VPorts that can be assigned to the PF and used for VM receive queues is (**MaxNumVPorts**– 1).

> ⓘ Note
>
> If this flag is set, the creation and assignment of nondefault VPorts are not reserved for VF allocation. As a result, situations may occur where a VF may not be assigned a VPort if the pool has been exhausted of available VPorts.

If the NDIS_NIC_SWITCH_CAPS_SINGLE_VPORT_POOL flag is not set, the creation and assignment of nondefault VPorts is reserved for VF assignment. The maximum number of additional nondefault VPorts that can be created and assigned to the PF and used for VM receive queues is (**MaxNumVPorts**–**MaxNumVFs**).

For more information about VMQ, see Virtual Machine Queue (VMQ).

Managing Virtual Functions

Article • 12/15/2021

This section describes the requirements and guidelines for managing the PCI Express (PCIe) Virtual Functions (VFs) on a network adapter that supports single root I/O virtualization (SR-IOV).

This section includes the following topics:

Overview of Virtual Function Initialization and Teardown

Allocating Resources for a Virtual Function

Freeing Resources for a Virtual Function

Enumerating Virtual Functions on a Network Adapter

Querying the Parameters of a Virtual Function

Accessing the PCI Configuration Space of a Virtual Function

Setting the Power State of a Virtual Function

Resetting a Virtual Function

For more information on VFs for SR-IOV network adapters, see SR-IOV Virtual Functions (VFs).

Note Only the PF miniport driver can configure the network adapter's hardware resources, such as the VFs. The VF miniport driver cannot directly access most of the SR-IOV adapter's hardware resources. For more information, see Writing SR-IOV VF Miniport Drivers.

Overview of Virtual Function Initialization and Teardown

Article • 12/15/2021

This section provides an overview of the initialization and teardown sequence for PCI Express (PCIe) Virtual Functions (VFs). VFs are provided by a network adapter that supports single root I/O virtualization (SR-IOV).

This section includes the following topics:

Virtual Function Initialization Sequence

Virtual Function Teardown Sequence

For more information on VFs for SR-IOV network adapters, see SR-IOV Virtual Functions (VFs).

Note Only the PF miniport driver can configure the network adapter's hardware resources, such as the VFs. The VF miniport driver cannot directly access most of the SR-IOV adapter's hardware resources. For more information, see Writing SR-IOV VF Miniport Drivers.

Virtual Function Initialization Sequence

Article • 12/15/2021

A network adapter that supports single root I/O virtualization (SR-IOV) must be able to support the following hardware components:

- One PCI Express (PCIe) Physical Function (PF). The PF always exists on the network adapter and is attached to the Hyper-V parent partition.

  For more information on this hardware component, see SR-IOV Physical Function (PF).

- One or more PCIe Virtual Functions (VF). Each VF must be initialized and attached to a Hyper-V child partition before the networking components of the guest operating system can send or receive packets over the VF.

  For more information on this hardware component, see SR-IOV Virtual Functions (VFs).

The PF miniport driver, which runs in the management operating system of the Hyper-V parent partition, initializes and allocates resources for a VF on the SR-IOV network adapter. After NDIS calls the PF miniport driver's *MiniportInitializeEx* function, NDIS and the virtualization stack can issue object identifier (OID) requests to the PF miniport driver to do the following:

- Create a NIC switch on the network adapter. The NIC switch bridges network traffic between the VFs, PF, and the physical network port.

  For more information, see NIC Switches.

  Note Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

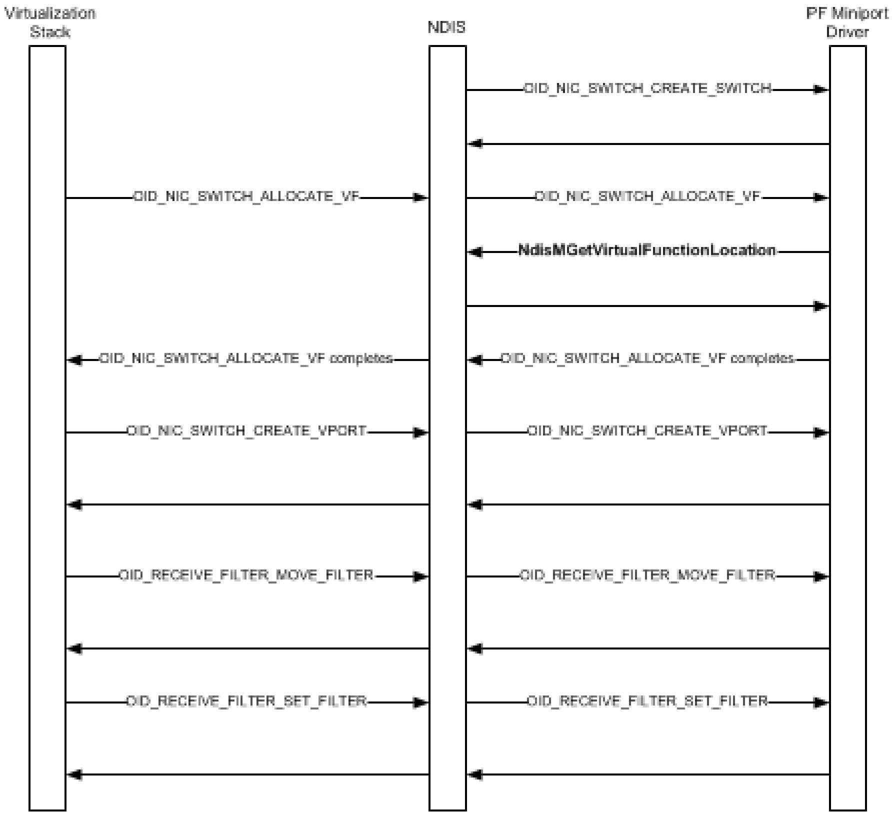
- Request the PF miniport driver to initialize and allocate resources for a VF on the network adapter.

  For more information, see SR-IOV Virtual Functions (VFs).

- Create a virtual port (VPort) on the NIC switch and attach it to the VF.

  For more information, see Virtual Ports (VPorts).

The following diagram shows the steps that are involved with VF initialization.


Virtualization Stack
NDIS
PF Miniport Driver
OID_NIC_SWITCH_CREATE_SWITCH
OID_NIC_SWITCH_ALLOCATE_VF
OID_NIC_SWITCH_ALLOCATE_VF
NdisMGetVirtualFunctionLocation
OID_NIC_SWITCH_ALLOCATE_VF completes
OID_NIC_SWITCH_ALLOCATE_VF completes
OID_NIC_SWITCH_CREATE_VPORT
OID_NIC_SWITCH_CREATE_VPORT
OID_RECEIVE_FILTER_MOVE_FILTER
OID_RECEIVE_FILTER_MOVE_FILTER
OID_RECEIVE_FILTER_SET_FILTER
OID_RECEIVE_FILTER_SET_FILTER

NDIS, the virtualization stack, and the PF miniport driver follow these steps during the VF initialization sequence:

1. NDIS reads the default switch configuration from the registry and issues an OID method request of OID_NIC_SWITCH_CREATE_SWITCH to provision the switch in the network adapter. The parameters that are passed in this OID request include information on how to configure important hardware resources such as VFs and VPorts. It also includes information on how to distribute the resources among the nondefault VPorts and the default VPort that are attached to the PF.

   After the OID has been successfully completed by the PF miniport driver, the NIC switch is ready to be used to create VPorts and allocate VFs on it.

   For more information on how to create a NIC switch, see Creating a NIC Switch.

2. A VF is treated as an offload mechanism for the virtual machine (VM) network adapter. This adapter is exposed in the guest operating system that runs in the Hyper-V child partition. By default, the networking components in the guest operating system send and receive packets over the software-based synthetic data path. However, if a child partition is enabled for VF offload, the virtualization stack issues OID requests to the PF miniport driver for the resource allocation and initialization of a VF. After the VF is attached to the child partition and a VPort on the NIC switch, the networking components send and receive packets over the VF data path. For more information about these data paths, see SR-IOV Data Paths.

If a Hyper-V child partition has been enabled for VF offload, the virtualization stack issues an OID method request of OID_NIC_SWITCH_ALLOCATE_VF to the PF miniport driver. The parameters that are passed in this OID request include the identifier of the NIC switch on which the VF is allocated. Other parameters include identifiers for the child partition to which the VF will be attached.

The PF miniport driver allocates the necessary hardware and software resources for the VF. The PF miniport driver also determines the PCIe Requestor Identifier (RID) for the VF by calling **NdisMGetVirtualFunctionLocation**. The RID is used for DMA and interrupt remapping when DMA requests and interrupts are generated by the VF.

The RID along with the VF identifier are returned by the PF miniport driver when it successfully completes the OID_NIC_SWITCH_ALLOCATE_VF request.

For more information about resource allocation for a VF, see Allocating Resources for a Virtual Function.

3. The virtualization stack creates a VPort on the NIC switch by issuing an OID method request of OID_NIC_SWITCH_CREATE_VPORT to the PF miniport driver. The parameters that are passed in this OID request include the identifier of the NIC switch on which the VPort is to be created. Other parameters include the identifier of the VF to which the VPort will be attached.

   Note The default VPort on the NIC switch always exists and is attached to the PF. Only a single nondefault VPort can be created and attached to a VF.

   Before NDIS forwards the OID request to the PF miniport driver, it allocates a valid VPort identifier that is unique over the network adapter.

   When the PF miniport driver handles the OID request, it allocates the hardware resources required for the VPort and retains the identifier for the VPort. This identifier is used in later OID requests and SR-IOV function calls.

   For more information about how to create a VPort, see Creating a Virtual Port.

4. The Hyper-V child partition may be started long before a VF and VPort are allocated. During this time, the networking components in the guest operating system send and receive packets over the synthetic data path. This involves packet traffic over the default VPort that is attached to the PF. To bridge traffic to the child partition, the virtualization stack configures the default VPort with the media access control (MAC) and virtual LAN (VLAN) filters for the VM network adapter of the child partition.

After resources for the VF and VPort are allocated, the virtualization stack issues an OID method request of OID_RECEIVE_FILTER_MOVE_FILTER to the PF miniport driver. This OID request moves the MAC and VLAN filters for the VM network adapter from the default VPort to the VPort that is attached to the VF. This causes packets that match these filters to be forwarded to the VF VPort over the VF data path.

Note Existing receive filters may be moved from the default VPort to the VF VPort by using OID_RECEIVE_FILTER_MOVE_FILTER. Also, new filters may be set on the VF VPort by using OID_RECEIVE_FILTER_SET_FILTER.

After the VF and the VPort are created successfully and the MAC filters have been set on the VPort, the virtualization stack notifies the Virtual PCI (VPCI) virtual service provider (VSP). This VSP runs in the management operating system of the Hyper-V parent partition. The notification informs the VPCI VSP that the VF that has been successfully allocated and attached to a child partition. The VPCI VSP sends messages over the virtual machine bus (VMBus) to the VPCI virtual service client (VSC) that runs in the guest operating system of the child partition. The VPCI VSC is a bus driver that exposes a PCI device for the VF network adapter.

After the VF network adapter is exposed, the PnP subsystem that runs in the guest operating system detects the adapter and loads the VF miniport driver. This driver registers with NDIS. After the VF miniport driver has been initialized and the appropriate packet filters are configured on the VF network adapter, the VF data path is fully operational. As a result, packet traffic in the guest operating system switched to this data path from the synthetic data path.

Virtual Function Teardown Sequence

Article • 12/15/2021

A network adapter that supports single root I/O virtualization (SR-IOV) must be able to support the following hardware components:

- One PCI Express (PCIe) Physical Function (PF). The PF always exists on the network adapter and is attached to the Hyper-V parent partition.

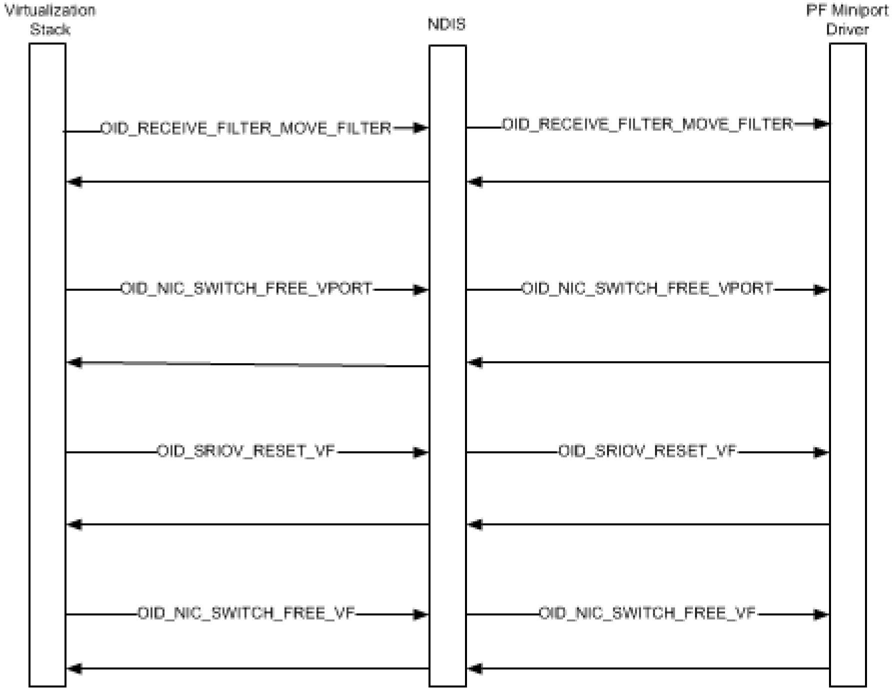
  For more information on this hardware component, see SR-IOV Physical Function (PF).

- One or more PCIe Virtual Functions (VF). Each VF must be initialized and attached to a Hyper-V child partition before the networking components of the guest operating system can send or receive packets over the VF.

  For more information on this hardware component, see SR-IOV Virtual Functions (VFs).

Before the VF is torn down and its resources freed, the virtualization stack notifies the Virtual PCI (VPCI) virtual service provider (VSP). This VSP runs in the management operating system of the Hyper-V parent partition. The notification informs the VPCI VSP that the VF will be torn down and detached from the child partition. The VPCI VSP sends messages over the virtual machine bus (VMBus) to the VPCI virtual service client (VSC) that runs in the guest operating system of the child partition. These messages request the VPCI VSC to gracefully remove the VF network adapter that was exposed when the VF was attached to the child partition. This causes the NetVSC to unbind from the VF miniport driver and the driver to be halted. At this point, packet traffic in the child partition migrates from the VF data path to the software-based synthetic data path. For more information about these data paths, see SR-IOV Data Paths.

After the failover to the synthetic data path is complete, the VF is torn down and its resources freed. The following diagram shows the steps that are involved with VF teardown.



NDIS, the virtualization stack, and the PF miniport driver follow these steps during the VF teardown sequence:

1. The virtualization stack moves the media access control (MAC) and virtual LAN (VLAN) filters for the virtual machine (VM) network adapter to the default virtual port (VPort) that is attached to the PF. The VM network adapter is exposed in the guest operating system of the child partition.

   Aftet the filters are moved to the default VPort, the synthetic data path is fully operational for network traffic to and from the networking components that run in the guest operating system. The PF miniport driver indicates received packets on the default PF VPort which uses the synthetic data path to indicate the packets to the guest operating system. Similarly, all transmitted packets from the guest operating system are routed through the synthetic data path and transmitted through the default PF VPort.

2. The virtualization stack deletes the VPort that is attached to the VF by issuing an object identifier (OID) set request of OID_NIC_SWITCH_DELETE_VPORT to the PF miniport driver. The miniport driver frees any hardware or software resources associated with the VPort and completes the OID request.

   For more information, see Deleting a Virtual Port.

3. The virtualization stack requests a PCIe function level reset (FLR) of the VF before its resources are deallocated. The stack does this by issuing an OID set request of OID_SRIOV_RESET_VFto the PF miniport driver. The FLR brings the VF on the SR-IOV network adapter into a quiescent state and clears any pending interrupt events for the VF.

4. After the VF has been reset, the virtualization stack requests a deallocation of the VF resources by issuing an OID set request of OID_NIC_SWITCH_FREE_VF to the PF miniport driver. This causes the miniport driver to free the hardware resources associated with the VF.

Allocating Resources for a Virtual Function

Article • 12/15/2021

A network adapter that supports single root I/O virtualization (SR-IOV) must be able to support the following hardware components:

- One PCI Express (PCIe) Physical Function (PF). The PF always exists on the network adapter and is attached to the Hyper-V parent partition.

  For more information on this hardware component, see SR-IOV Physical Function (PF).

- One or more PCIe Virtual Functions (VF). Each VF must be initialized and attached to a Hyper-V child partition before the networking components of the guest operating system can send or receive packets over the VF.

  For more information on this hardware component, see SR-IOV Virtual Functions (VFs).

The PF miniport driver, which runs in the management operating system of the Hyper-V parent partition, allocates resources for the PF and each VF on the SR-IOV network adapter. This driver allocates resources for the PF as it would for any network adapter. However, the driver allocates resources for each VF in the following way:

- The PF miniport driver allocates hardware resources for each VF when the driver creates the network interface card (NIC) on the network adapter. The driver completes the hardware resource allocation for the VFs by calling **NdisMEnableVirtualization**. For more information on this process, see Creating a NIC Switch.

- The PF miniport driver allocates software resources for a VF when the driver handles an object identifier (OID) method request of OID_NIC_SWITCH_ALLOCATE_VF. Even though the hardware resources have been allocated for a VF, it is considered nonoperational until the PF miniport driver successfully completes the OID_NIC_SWITCH_ALLOCATE_VF.

The overlying driver can request the allocation of software resources for a VF by issuing an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for the OID request contains a pointer to an **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

After a successful return from the OID request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_VF_PARAMETERS** structure. This structure has an adapter-unique VF identifier and PCI Requestor Identifier (RID). These identifiers are used in the following ways:

- The overlying driver uses the VF identifier in actions related to the VF, such as the following:
    - Obtaining the current VF parameters through an OID method request of OID_NIC_SWITCH_VF_PARAMETERS.
    - Freeing previously allocated resources for the VF through an OID set request of OID_NIC_SWITCH_FREE_VF.
    - Issuing a PCI reset to the VF through an OID set request of OID_SRIOV_RESET_VF.
- The RID is used by the virtualization stack for remapping DMA and interrupts between the PF and VF. The RID also enables the hardware input/output memory management unit (IOMMU) to convert guest physical addresses to host physical addresses.

For more information on how the overlying driver issues OID_NIC_SWITCH_ALLOCATE_VF method requests, see Issuing OID_NIC_SWITCH_ALLOCATE_VF Requests.

For more information on how the PF miniport driver handles OID_NIC_SWITCH_ALLOCATE_VF method requests, see Handling OID_NIC_SWITCH_ALLOCATE_VF Requests.

Note After resources for a VF have been allocated through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF, the resource parameters for the VF cannot be changed dynamically.

Issuing OID_NIC_SWITCH_ALLOCATE_VF Requests

Article • 12/15/2021

Before it issues the object identifier (OID) method request of OID_NIC_SWITCH_ALLOCATE_VF to the miniport driver for the PCI Express (PCIe) Physical Function (PF), the overlying driver formats an **NDIS_NIC_SWITCH_VF_PARAMETERS** structure. This structure contains the configuration parameters for the resources to be allocated for a PCIe Virtual Function (VF) on the network adapter. The overlying driver must set the members of this structure in the following way:

- The **SwitchId** member must be set to the identifier of a NIC switch that was previously created on the network adapter. A NIC switch is created through an OID method request of OID_NIC_SWITCH_CREATE_SWITCH.

  When it handles the OID method request of OID_NIC_SWITCH_ALLOCATE_VF, the miniport driver for the PCIe Physical Function (PF) allocates resources for the VF. If resources are allocated successfully, the PF miniport driver assigns the VF to the specified NIC switch.

  Note Starting with NDIS 6.30 in Windows Server 2012, the SR-IOV interface only supports the default NIC switch on the network adapter. The value of the **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

  For more information on a NIC switch, see NIC Switches.

- The **VFId** member must be set to NDIS_INVALID_VF_FUNCTION_ID.

- The **RequestorId** member must be set to NDIS_INVALID_RID.

- The **VMFriendlyName** and **VMName** members must be set to the parameters of a Hyper-V child partition. The PF miniport driver uses these members only for informational purposes.

  Note The Hyper-V child partition is also known as a *virtual machine (VM)*.

  The VF is associated with the specified VM before the overlying driver issues the OID_NIC_SWITCH_CREATE_SWITCH request.

- The **NicName** member must be set to the identifier of the virtual machine (VM) network adapter. This virtual adapter is exposed in the guest operating system that runs in the VM. The PF miniport driver uses this member only for informational purposes.

When resources are allocated for the VF and it is attached to the child partition, a VF network adapter is exposed in the guest operating system. The VM network adapter teams with the VF network adapter for packet transfer over the hardware-based VF data path.

However, the VF could be detached from the child partition, such as during Live Migration. When this happens, the packet transfer occurs over the software-based synthetic data path. For more information on these data paths, see SR-IOV Data Paths.

- The **PermanentMacAddress** and **CurrentMacAddress** members must be set to the media access control (MAC) addresses for the virtual network adapter of the VF. These addresses are exposed to the network stack that runs in the guest operating system of the Hyper-V child partition.

The overlying driver issues the OID method request of OID_NIC_SWITCH_ALLOCATE_VF by following these steps:

1. The overlying driver initializes an **NDIS_OID_REQUEST** structure for the OID method request. The driver sets the **InformationBuffer** member to a pointer to an initialized **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

2. The overlying driver calls **NdisOidRequest** to issue the OID request to the underlying PF miniport driver.

   Note When the overlying driver calls **NdisOidRequest**, NDIS intercepts the OID request and verifies the VF parameters specified in the **NDIS_NIC_SWITCH_VF_PARAMETERS** structure. If the parameters are verified successfully, NDIS forwards the OID to the PF miniport driver. Otherwise, NDIS fails the OID request with NDIS_STATUS_INVALID_PARAMETER.

After an overlying driver requests resource allocation for a VF, that driver is the only component that can request the freeing of the resources for the same VF. The overlying driver must issue an OID set request of OID_NIC_SWITCH_FREE_VF to free the VF resources. Before the overlying driver can be halted, it must free the resources for each VF that was allocated by the driver's OID_NIC_SWITCH_ALLOCATE_VF request.

Handling OID_NIC_SWITCH_ALLOCATE_VF Requests

Article • 12/15/2021

When the miniport driver for the PCI Express (PCIe) Physical Function (PF) on the network adapter handles the object identifier (OID) method request of OID_NIC_SWITCH_ALLOCATE_VF, it does the following:

- The PF miniport driver allocates the software resources for a PCIe Virtual Function (VF) on the network adapter. These resources are configured based on the parameters that are specified in the **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

- The PF miniport driver assigns the VF to a NIC switch on the network adapter. The NIC switch is identified by the **SwitchId** member of the **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

  For more information on a NIC switch, see NIC Switches.

- The PF miniport driver updates the **VFId** member with a VF identifier. This identifier is a zero-based index and must be unique across all VFs that are allocated on the NIC switch by the PF miniport driver.

  The overlying driver uses the value of the **VFId** member in successive OID requests of OID_NIC_SWITCH_FREE_VF or OID_NIC_SWITCH_VF_PARAMETERS.

- The PF miniport driver updates the **RequestorId** member with a PCIe Requestor Identifier (RID) for the VF.

  The miniport driver calls **NdisMGetVirtualFunctionLocation** to get the RID information that corresponds to the VF. The driver then creates the RID by using the **NDIS_MAKE_RID** macro based on the information returned by the call to **NdisMGetVirtualFunctionLocation**.

  The RID is used by the virtualization stack for remapping DMA and interrupts between the PF and VF. The RID also enables the hardware input/output memory management unit (IOMMU) to convert guest physical addresses to host physical addresses.

- The PF miniport driver initializes and exposes the VF. This makes the VF ready for use by the virtualization stack.

If the PF miniport driver can successfully allocate the necessary software resources and initialize the VF, the driver completes the OID request with NDIS_STATUS_SUCCESS. The PF miniport driver must keep the VF IDs for each allocated VF. NDIS and the overlying drivers use the VF identifier in successive OID requests to the PF miniport driver for various actions, such as resetting or freeing the VF.

Note When resources for the VF are allocated, the VF is in an unattached state because a virtual port (VPort) is not attached to the VF. The overlying driver can issue an OID request of OID_NIC_SWITCH_CREATE_VPORT to create and attach a VPort to the VF. For more information, see Creating a Virtual Port.

Freeing Resources for a Virtual Function

Article • 12/15/2021

The overlying driver requests resource allocation for a PCI Express (PCIe) Virtual Function (VF) through object identifier (OID) method requests of OID_NIC_SWITCH_ALLOCATE_VF. After the VF resources are successfully allocated, the overlying driver frees the resources through an OID set request of OID_NIC_SWITCH_FREE_VF.

This section includes the following topics:

Issuing OID_NIC_SWITCH_FREE_VF Requests

Handling OID_NIC_SWITCH_FREE_VF Requests

Issuing OID_NIC_SWITCH_FREE_VF Requests

Article • 12/15/2021

An overlying driver issues an object identifier (OID) set request of OID_NIC_SWITCH_FREE_VF to free resources for a PCI Express (PCIe) Virtual Function (VF). These resources were previously allocated through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

The overlying driver issues the OID_NIC_SWITCH_FREE_VF set request to the miniport driver for the PCIe Physical Function (PF). Before it issues this OID request, the overlying driver must do the following:

1. The overlying driver must make sure that the VF is not attached to any virtual port (VPort) on the NIC switch of the network adapter. The overlying driver must issue OID set requests of OID_NIC_SWITCH_DELETE_VPORT to delete all VPorts that are attached to the VF. For more information, see Deleting a Virtual Port.

2. The overlying driver initializes an **NDIS_NIC_SWITCH_FREE_VF_PARAMETERS** structure. The driver must set the **VFId** member to the VF identifier that was returned in the OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

The overlying driver issues the OID set request of OID_NIC_SWITCH_FREE_VF by following these steps:

1. The overlying driver initializes an **NDIS_OID_REQUEST** structure for the OID method request. The driver sets the **InformationBuffer** member to a pointer to an initialized **NDIS_NIC_SWITCH_FREE_VF_PARAMETERS** structure.

2. The overlying driver calls **NdisOidRequest** to issue the OID request to the underlying PF miniport driver.

After an overlying driver requests resource allocation for a VF, that driver is the only component that can request the freeing of the resources for the same VF. The overlying driver must issue an OID set request of OID_NIC_SWITCH_FREE_VF to free the VF resources. Before the overlying driver can be halted, it must free the resources for each VF that was allocated by the driver's OID_NIC_SWITCH_ALLOCATE_VF request.

Note If an overlying driver issues an OID method request of OID_NIC_SWITCH_ALLOCATE_VF to allocate resources for a VF, that driver is the only component that can request the freeing of the resources for the same VF. The overlying driver must issue an OID set request of OID_NIC_SWITCH_FREE_VF to free the VF resources. Before the overlying driver can be halted, it must free the resources for each VF that was allocated by the driver's OID_NIC_SWITCH_ALLOCATE_VF request.

Handling OID_NIC_SWITCH_FREE_VF Requests

Article • 12/15/2021

When the miniport driver for the PCI Express (PCIe) Physical Function (PF) on the network adapter handles the object identifier (OID) set request of OID_NIC_SWITCH_FREE_VF, it does the following:

- The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for the OID request contains a pointer to an **NDIS_NIC_SWITCH_FREE_VF_PARAMETERS** structure. The PF miniport driver must verify that the identifier of the PCIe Virtual Function (VF), which is specified by the **VFId** member, is valid. If this is not true, the driver must fail the OID set request by returning NDIS_STATUS_INVALID_PARAMETER.

- The PF miniport driver must verify that resources for the VF have been previously allocated through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If this is not true, the driver must fail the OID set request by returning NDIS_STATUS_INVALID_PARAMETER.

- The PF miniport driver must verify that no virtual ports (VPorts) are currently attached to the VF. If this is not true, the driver must fail the set request by returning NDIS_STATUS_INVALID_PARAMETER.

- The PF miniport driver must free all software resources that were allocated for the specified VF.

- The PF miniport driver must detach the VF from the NIC switch on the network adapter.

If the PF miniport driver can successfully free the allocated software resources and detach the VF from the NIC switch, the driver completes the OID request with NDIS_STATUS_SUCCESS.

Note NDIS guarantees that all the VFs allocated on the miniport are freed before NDIS issues an OID set request of OID_NIC_SWITCH_DELETE_SWITCH to the PF miniport driver. When it handles this OID, the driver deletes a NIC switch on the network adapter.

Enumerating Virtual Functions on a Network Adapter

Article • 12/15/2021

An overlying driver or user application can obtain a list of all PCI Express (PCIe) Virtual Functions (VFs) on a network adapter that supports single root I/O virtualization (SR-IOV). The driver or application issues an object identifier (OID) method request of OID_NIC_SWITCH_ENUM_VFS to obtain this list.

Before the driver or application issues the OID request, it must initialize an **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure that is passed along with the request. The driver or application must follow these guidelines when initializing the **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure:

- If the NDIS_NIC_SWITCH_VF_INFO_ARRAY_ENUM_ON_SPECIFIC_SWITCH flag is set in the **Flags** member, the overlying driver or application must set the **SwitchId** member to the identifier of a NIC switch on the SR-IOV network adapter. By setting these members in this manner, VF information is returned only for the specified NIC switch on the SR-IOV network adapter.

  Note The overlying driver and user-mode application can obtain the NIC switch identifiers by issuing an OID query request of OID_NIC_SWITCH_ENUM_SWITCHES.

- If the **Flags** member is set to zero, the driver or application must set the **SwitchId** member to zero. By setting these members in this manner, VF information is returned for all NIC switches on the SR-IOV network adapter.

  Note Starting with Windows Server 2012, Windows supports only the default NIC switch on the network adapter. Regardless of the flags set in the **Flags** member, the **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

After a successful return from this OID query request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer that contains the following:

- An **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure that defines the number of elements within the array.
- An array of **NDIS_NIC_SWITCH_VF_INFO** structures. Each of these structures contains information about a single VF on a NIC switch of the network adapter. A VF is attached to a NIC switch through OID method requests of OID_NIC_SWITCH_ALLOCATE_VF.

> **Note** If no VFs are attached to a NIC switch on the network adapter, the **NumElements** member of the **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure is set to zero and no **NDIS_NIC_SWITCH_VF_INFO** structures are returned.
>
> For more information on NIC switches, see NIC Switches.

NDIS handles the OID_NIC_SWITCH_ENUM_VFS request for miniport drivers. NDIS returns the information from an internal cache of the data that it maintains from inspecting the following sources:

- OID method requests of OID_NIC_SWITCH_ALLOCATE_VF.
- OID set requests of OID_NIC_SWITCH_VF_PARAMETERS.

Querying the Parameters of a Virtual Function

Article • 12/15/2021

An overlying driver or a user-mode application can obtain the current parameters for a PCI Express (PCIe) Virtual Function (VF) on a network adapter that supports single root I/O virtualization (SR-IOV). The driver or application issues an object identifier (OID) method request of OID_NIC_SWITCH_VF_PARAMETERS to obtain these parameters.

Before the overlying driver issues this OID method request, it must initialize an **NDIS_NIC_SWITCH_VF_PARAMETERS** structure. The driver or application must set the **VFId** member to the identifier of the VF for which parameters are to be returned. The VF identifier can be obtained in the following ways:

- By issuing an OID method request of OID_NIC_SWITCH_ENUM_VFS.

  If this OID request is completed successfully, the overlying driver or user-mode application receives a list of all VFs allocated on the network adapter. Each element within the list is an **NDIS_NIC_SWITCH_VF_INFO** structure, with the VF identifier specified by the **VFId** member.

- By issuing an OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

  If this OID request is completed successfully, the overlying driver receives the identifier of the newly created VF in the **VFId** member of the returned **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

  Note Only overlying drivers can obtain the VF identifier in this manner.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_VF_PARAMETERS** structure. This structure contains the configuration parameters for the specified VF.

NDIS handles the OID_NIC_SWITCH_VF_PARAMETERS request for miniport drivers. NDIS returns the information from an internal cache of the data that it maintains from inspecting the following sources:

- OID method requests of OID_NIC_SWITCH_ALLOCATE_VF.

- OID set requests of OID_NIC_SWITCH_VF_PARAMETERS.

Accessing the PCI Configuration Space of a Virtual Function

Article • 12/15/2021

This section describes guidelines for accessing the data from the PCI configuration space of PCI Express (PCIe) Virtual Functions (VFs). VFs are a hardware function of a network adapter that supports single root I/O virtualization (SR-IOV).

This section includes the following topics:

Querying the PCI Configuration Data of a Virtual Function

Setting the PCI Configuration Data of a Virtual Function

For more information on VFs for SR-IOV network adapters, see SR-IOV Virtual Functions (VFs).

Note Only the PF miniport driver can configure the PCI configuration space for a VF. The VF miniport driver cannot directly access most of the SR-IOV adapter's hardware resources, such as the PCI configuration space. For more information, see Writing SR-IOV VF Miniport Drivers.

Querying the PCI Configuration Space for a Virtual Function

Article • 12/15/2021

Note This method can only be used by overlying drivers that run in the management operating system of the Hyper-V parent partition.

The miniport driver for a PCI Express (PCIe) Virtual Function (VF) runs in the guest operating system of a Hyper-V child partition. Because of this, the VF miniport driver cannot directly access hardware resources, such as the VF's PCIe configuration space. Only the miniport driver for the PCIe Physical Function (PF) can access the PCIe configuration space for a VF. The PF miniport driver runs in the management operating system of a Hyper-V parent partition and has privileged access to the VF resources.

An overlying driver that runs in the management operating system issues an object identifier (OID) method request of OID_SRIOV_READ_VF_CONFIG_SPACE to read data from the PCIe configuration space for a specified VF on the network adapter.

For example, the virtualization stack that runs in the management operating system issues the OID method request of OID_SRIOV_READ_VF_CONFIG_SPACE when the VF miniport driver calls **NdisMGetBusData** to read from its VF PCIe configuration space.

Before it issues this OID method request, the overlying driver must set the members of the **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure in the following way:

- The **VFId** member must be set to the identifier of the VF from which the information is to be read.

- The **Offset** member must be set to the offset within the PCIe configuration space of the VF in which data will be read.

- The **Length** member must be set to the number of bytes to read from the VF's PCIe configuration space.

- The **BufferOffset** member must be set to the offset within the buffer (referenced by the **InformationBuffer** member) that will contain the data that is read from the specified VF's PCI configuration space. This offset is specified in units of bytes from the beginning of the **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure.

When it handles the OID method request of OID_SRIOV_READ_VF_CONFIG_SPACE, the PF miniport driver must follow these guidelines:

- The miniport driver must verify that the VF, specified by the **VFId** member of the **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure, has resources that have been previously allocated. The miniport driver allocates resources for a VF through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.

- The miniport driver must verify that the buffer (referenced by the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure) is large enough to return the requested PCIe configuration space data. If this is not true, the driver must fail the OID request.

- The miniport driver typically calls **NdisMGetVirtualFunctionBusData** to query the requested PCIe configuration space. However, the miniport driver can also return PCIe configuration space data for the VF that the driver has cached from previous read or write operations of the PCIe configuration space.

  Note If an independent hardware vendor (IHV) provides a virtual bus driver (VBD) as part of its SR-IOV driver package, its miniport driver must not call **NdisMGetVirtualFunctionBusData**. Instead, the driver must interface with the VBD through a private communication channel, and request that the VBD call *ReadVfConfigBlock*. This function is exposed from the GUID_VPCI_INTERFACE_STANDARD interface that is supported by the underlying virtual PCI (VPCI) bus driver.

After a successful return from this OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure that contains the parameters for a read operation of the PCIe configuration space of a VF.

- Additional buffer space for the data to be read from the PCIe configuration space. The driver copies the data to the buffer at the offset specified by the**BufferOffset** member of the **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure.

Querying the PCI Vendor and Device Identifiers for a Virtual Function

Article • 12/15/2021

Note This method can only be used by overlying drivers that run in the management operating system of the Hyper-V parent partition.

An overlying driver issues an object identifier (OID) method request of OID_SRIOV_VF_VENDOR_DEVICE_ID to query the PCI Express (PCIe) vendor identifier (*VendorID*) and device identifier (*DeviceID*). This data is read from the PCIe configuration space for the PCIe Virtual Function (VF) on the physical network adapter.

Overlying drivers issue this OID method request to the miniport driver of the PCI Express (PCIe) Physical Function (PF) of the network adapter. This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The guest operating system, which runs in a Hyper-V child partition, uses the VendorID and the DeviceID of the VF for generic Plug and Play (PnP) IDs for device enumeration. Starting with Windows Server 2012, the PF miniport driver can provide the following set of identifiers for the VF network adapter that is exposed in the child partition:

- The VendorID and DeviceID of the physical network adapter. This allows compatible drivers to be loaded in the guest operating system, which runs in the Hyper-V child partition, and the management operating system, which runs in the Hyper-V parent partition.

- A VendorID and DeviceID that differ from the identifiers of the physical network adapter. This allows a driver to be loaded in the guest operating system that is more appropriate for its use. For example, the PF miniport driver may return a VendorID and DeviceID for a VF network adapter so that a driver is loaded that disables certain feature sets, such as power management or protocol task offloads.

Before it issues this OID method request, the overlying driver must initialize an **NDIS_SRIOV_VF_VENDOR_DEVICE_ID_INFO** structure. The driver must set the **VFId** member to the identifier of the VF from which the information is to be read.

When it handles this OID request, the PF miniport driver must verify that the specified VF has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.

Querying the PCI Base Address Registers of a Virtual Function

Article • 12/15/2021

Note This method can only be used by overlying drivers that run in the management operating system of the Hyper-V parent partition.

The PCI bus driver, which runs in the management operating system of the Hyper-V parent partition, queries the memory or I/O address space requirements of each PCI Base Address Register (BAR) of the network adapter. The PCI bus driver performs this query when it first detects the adapter on the bus.

Through this PCI BAR query, the PCI bus driver determines the following:

- Whether a PCI BAR is supported by the network adapter.
- If a BAR is supported, how much memory or I/O address space is required for the BAR.

The PCI driver performs this PCI BAR query in following way:

1. The PCI driver first writes all ones to a BAR.
2. The PCI driver then reads the BAR to determine the required memory or address space that is required by the BAR. A value of zero indicates that the BAR is not supported by the network adapter.

The virtual PCI (VPCI) bus driver runs in the guest operating system of a Hyper-V child partition. When a PCI Express (PCIe) Virtual Function (VF) is attached to the child partition, the VPCI bus driver exposes a virtual network adapter for the VF (*VF network adapter*). Before it does this, the VPCI bus driver must perform a PCI BAR query to determine the required memory or address space that is required by the VF network adapter.

Because access to the PCI configuration space is a privileged operation, it can only be performed by components that run in the management operating system of a Hyper-V parent partition. When the VPCI bus driver queries the PCI BARs, NDIS issues an object identifier (OID) query request of OID_SRIOV_PROBED_BARS to the PF miniport driver. The results returned by this OID query request are forwarded to the VPCI bus driver so that it can determine how much memory address space would be needed by the VF network adapter.

Note OID requests of OID_SRIOV_BAR_RESOURCES can only be issued by NDIS. The OID request must not be issued by overlying drivers, such as protocol or filter drivers.

The OID_SRIOV_PROBED_BARS query request contains an NDIS_SRIOV_PROBED_BARS_INFO structure. When the PF miniport driver handles this OID, the driver must return the PCI BAR values within the array referenced by the **BaseRegisterValuesOffset** member of the **NDIS_SRIOV_PROBED_BARS_INFO** structure. For each offset within the array, the PF miniport driver must set the array element to the ULONG value of the BAR at the same offset within the physical network adapter's PCI configuration space.

Each BAR value returned by the driver must be the same value that would follow a PCI BAR query as performed by the PCI driver that runs in the management operating system. The PF miniport driver can call NdisMQueryProbedBars to determine this information.

For more information about the base address registers of a PCI device, see the *PCI Local Bus Specification*.

Setting the PCI Configuration Data of a Virtual Function

Article • 12/15/2021

The miniport driver for a PCI Express (PCIe) Virtual Function (VF) runs in the guest operating system of a Hyper-V child partition. Because of this, the VF miniport driver cannot directly access hardware resources, such as the VF's PCI configuration space. Only the miniport driver for the PCIe Physical Function (PF) can access the PCI configuration space for a VF. The PF miniport driver runs in the management operating system of a Hyper-V parent partition and has privileged access to the VF resources.

The overlying driver, such as the virtualization stack, issues the OID set request of OID_SRIOV_WRITE_VF_CONFIG_SPACE when the VF miniport driver calls **NdisMSetBusData** to write to its PCI configuration space.

Before it issues this OID set request, the overlying driver must set the members of the**NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS** structure in the following way:

- Set the **VFId** member to the identifier of the VF for which the information is to be written.

- Set the **Offset** member to the offset within the PCI configuration space of the VF in which data will be written.

- Set the **Length** member to the number of bytes to write to the VF's PCI configuration space.

- Set the **BufferOffset** member to the offset within the buffer (referenced by the**InformationBuffer** member) that will contain the data that is written to the specified VF's PCI configuration space. This offset is specified in units of bytes from the beginning of the **NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS** structure.

When it handles the OID method request of OID_SRIOV_WRITE_VF_CONFIG_SPACE, the PF miniport driver must follow these guidelines:

- The PF miniport driver must verify that the VF, specified by the **VFId** member of the **NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS** structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

If resources for the specified VF have not been allocated, the driver must fail the OID request.

- The PF miniport driver calls **NdisMSetVirtualFunctionBusData** to write to the requested PCI configuration space. However, the PF miniport driver can also return PCI configuration space data for the VF that the driver has cached from previous read or write operations of the PCI configuration space.

  Note If an independent hardware vendor (IHV) provides a virtual bus driver (VBD) as part of its SR-IOV driver package, its PF miniport driver must not call **NdisMSetVirtualFunctionBusData**. Instead, the driver must interface with the VBD through a private communication channel, and request that the VBD call *SetVirtualFunctionData*. This function is exposed from the GUID_VPCI_INTERFACE_STANDARD interface that is supported by the underlying virtual PCI (VPCI) bus driver.

If the PF miniport driver can successfully complete the OID request, the driver must copy the requested PCI configuration space data to the buffer referenced by the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. The driver copies the data to the buffer at the offset specified by the**BufferOffset** member of the **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure.

Setting the Power State of a Virtual Function

Article • 12/15/2021

An overlying driver issues an object identifier (OID) set request of OID_SRIOV_SET_VF_POWER_STATE to change the power state of a specified PCI Express (PCIe) Virtual Function (VF) on the network adapter. Because changing the power state is a privileged operation, overlying drivers issue this OID set request to the miniport driver of the PCIe Physical Function (PF) on the network adapter. The PF miniport driver then sets the specified power state on the VF.

For example, the virtualization stack manages the power state of the Hyper-V child partition that is attached to the VF. The stack changes the power state by issuing the OID_SRIOV_SET_VF_POWER_STATE to the PF miniport driver.

Before it issues the OID set request of OID_SRIOV_SET_VF_POWER_STATE, the overlying driver must set the members of **NDIS_SRIOV_SET_VF_POWER_STATE_PARAMETERS** structure in the following way:

- The **VFId** member must be set to the identifier of the VF from which the information is to be read.
- The **PowerState** member must be set to the power state that the VF should transition to.
- If the network adapter must have its WAKE# signal (on the PCI Express bus) or PME# signal (on the PCI bus) asserted as it goes into the low-power state, the **WakeEnable** member must be set to TRUE. Otherwise, this member must be set to FALSE.

When the PF miniport driver is issued this OID set request, it must follow these guidelines:

- The PF miniport driver must verify that the VF, specified by the **VFId** member of the **NDIS_SRIOV_SET_VF_POWER_STATE_PARAMETERS** structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If the specified VF is not in an allocated state, the driver must fail the OID request.
- The power state operation must only affect the specified VF. The operation must not affect other VFs or the PF on the same network adapter.

Resetting a Virtual Function

Article • 12/15/2021

An overlying driver issues an object identifier (OID) set request of OID_SRIOV_RESET_VF to reset a specified PCI Express (PCIe) Virtual Function (VF). The VF is a hardware component of a network adapter that supports single root I/O virtualization. Overlying drivers issue this OID set request to the miniport driver of the PCI Express (PCIe) Physical Function (PF).

For example, the virtualization stack runs in the management operating system of the Hyper-V parent partition. Before the stack detaches a VF from a Hyper-V child partition, it requests a Function Level Reset (FLR) on the VF. Because the FLR is a privileged operation, it can be performed only by the PF miniport driver that also runs in the management operating system. To request an FLR of a specified VF, the virtualization stack issues the OID_SRIOV_RESET_VFrequest to the PF miniport driver.

Before it issues this OID set request, the overlying driver must initialize an **NDIS_SRIOV_RESET_VF_PARAMETERS** structure. The driver must set the **VFId** member to the identifier of the VF to be reset.

When it handles this OID request, the PF miniport driver must follow these guidelines:

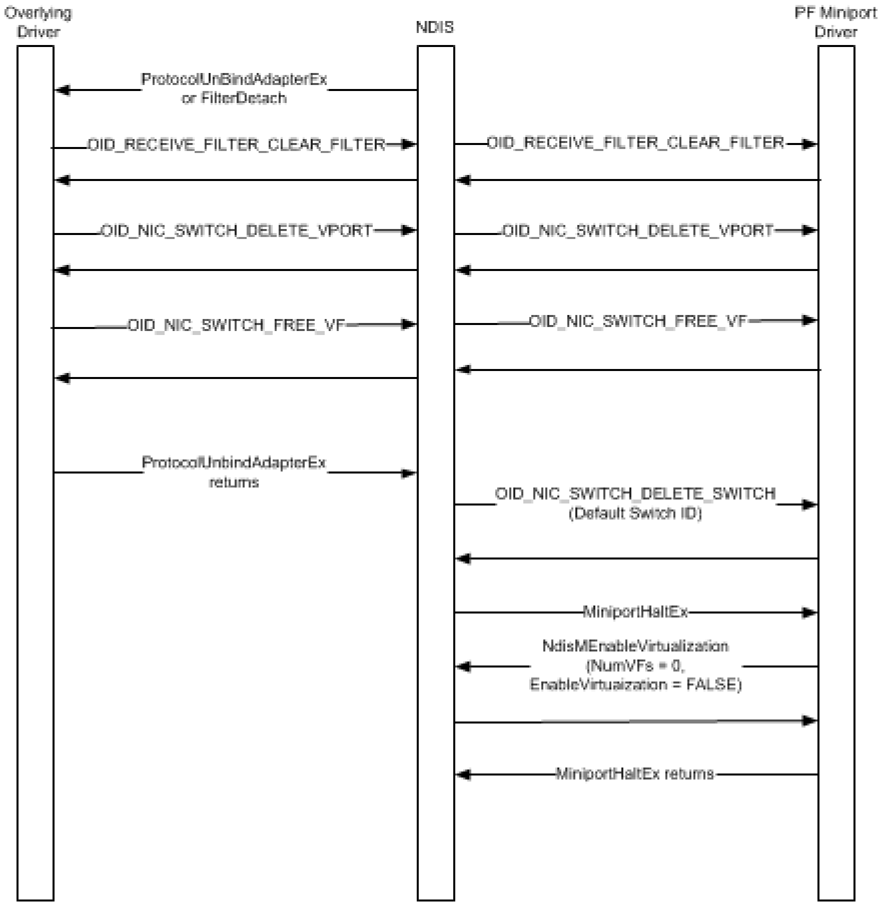
- The PF miniport driver must verify that the VF, specified by the **VFId** member of the **NDIS_SRIOV_RESET_VF_PARAMETERS** structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.

- The reset operation must only affect the specified VF. The operation must not affect other VFs or the PF on the same network adapter.

Halting a PF Miniport Driver

Article • 12/15/2021

This topic discusses the steps that are involved with halting the miniport driver for a PCI Express (PCIe) Physical Function (PF) on an adapter that supports single root I/O virtualization (SR-IOV). These steps are shown in the following figure.


Overlying Driver
NDIS
PF Miniport Driver
ProtocolUnBindAdapterEx or FilterDetach
OID_RECEIVE_FILTER_CLEAR_FILTER
OID_RECEIVE_FILTER_CLEAR_FILTER
OID_NIC_SWITCH_DELETE_VPORT
OID_NIC_SWITCH_DELETE_VPORT
OID_NIC_SWITCH_FREE_VF
OID_NIC_SWITCH_FREE_VF
ProtocolUnbindAdapterEx returns
OID_NIC_SWITCH_DELETE_SWITCH (Default Switch ID)
MiniportHaltEx
NdisMEnableVirtualization (NumVFs = 0, EnableVirtuaization = FALSE)
MiniportHaltEx returns

This topic contains the following information:

- Actions Performed by NDIS and Overlying Drivers Before *MiniportHaltEx* is Called
- Actions Performed by the PF Miniport Driver When *MiniportHaltEx* is Called

Actions Performed by NDIS and Overlying Drivers Before *MiniportHaltEx* is Called

Before NDIS calls the PF miniport driver's *MiniportHaltEx* function, it first does the following:

- NDIS unbinds all protocol drivers that have previously bound to the underlying PF miniport driver. NDIS does this by calling the protocol driver's *ProtocolUnbindAdapterEx* function.

- NDIS detaches all filter drivers that have previously bound to the underlying PF miniport driver. NDIS does this by calling the filter driver's *FilterDetach* function.

When an overlying protocol or filter driver is being unbound or detached from the PF miniport driver, it must follow these steps:

1. The driver must issue an object identifier (OID) set request of OID_RECEIVE_FILTER_CLEAR_FILTER to clear any receive filters that it previously set. The driver sets these filters on the default virtual port (VPort) or any nondefault VPorts of the NIC switch on the network adapter. The driver sets these filters by issuing OID method requests of OID_RECEIVE_FILTER_SET_FILTER to the PF miniport driver.

2. The driver must issue an OID set request of OID_NIC_SWITCH_DELETE_VPORT to delete any nondefault VPorts that it previously created on the NIC switch. The driver sets these VPorts by issuing OID method requests of OID_NIC_SWITCH_CREATE_VPORT to the PF miniport driver.

3. The driver must issue an OID set request of OID_NIC_SWITCH_FREE_VF to free the resources for any PCIe Virtual Functions (VFs) that it previously allocated on the NIC switch. The driver allocates resources for the VF by issuing OID method requests of OID_NIC_SWITCH_ALLOCATE_VF to the PF miniport driver.

   For more information, see Freeing Resources for a Virtual Function.

   Note When resources for the VF are freed, NDIS calls the *MiniportHaltEx* function of the VF miniport driver. For more information, see Halting a VF Miniport Driver.

After all receive filters, nondefault VPorts, and VFs have been deleted from the NIC switch, NDIS follows these steps:

- NDIS deletes all NIC switches by issuing OID set requests of OID_NIC_SWITCH_DELETE_SWITCH to the PF miniport driver. For more information on how a NIC switch is deleted, see Deleting a NIC Switch.

  Note Starting with Windows Server 2012, the SR-IOV interface only supports the default NIC switch on the network adapter.

- After all NIC switches have been successfully deleted, NDIS calls the *MiniportHaltEx* function of the PF miniport driver.

Actions Performed by the PF Miniport Driver When *MiniportHaltEx* is Called

When NDIS calls *MiniportHaltEx*, the PF miniport driver must follow these steps:

1. If the PF miniport driver supports the static creation of NIC switches and all the NIC switches have been deleted, the driver must disable the virtualization on the adapter by calling **NdisMEnableVirtualization** with *EnableVirtualization* parameter set to FALSE and the *NumVFs* parameter set to zero.

   NdisMEnableVirtualization clears the **NumVFs** member and the **VF Enable** bit in the SR-IOV Extended Capability structure in the PCIe configuration space of the network adapter's PF.

   Note If the PF miniport driver supports dynamic creation and configuration of NIC switches, it must call **NdisMEnableVirtualization** when the driver handles the OID set request of OID_NIC_SWITCH_DELETE_SWITCH. This OID request is issued before *MiniportHaltEx* is called.

2. The PF miniport driver performs the other tasks associated with a miniport halt operation. For more information, see Halting a Miniport Adapter.

INF Requirements for PF Miniport Drivers

Article • 12/15/2021

This section describes the INF requirements for the miniport driver of a network adapter that supports the single root I/O virtualization (SR-IOV) interface. These are only requirements for the miniport driver of the adapter's PCI Express (PCIe) Physical Function (PF).

This section includes the following topics:

Standardized INF Keywords for SR-IOV

INF *DDInstall*.HW Section for PF Miniport Drivers

Handling SR-IOV, VMQ, and RSS Standardized INF Keywords

Standardized INF Keywords for SR-IOV

Article • 12/15/2021

This topic describes the standardized INF keywords for the single root I/O virtualization (SR-IOV) interface. These keywords apply to the INF file for the miniport driver of the PCI Express (PCIe) Physical Function (PF) of an SR-IOV network adapter.

The SR-IOV INF keywords are described in the following sections:

Standardized INF Keywords for the Enabling or Disabling SR-IOV Support

Standardized INF Keywords for Configuration of the Default NIC Switch

Standardized INF Keywords for Enabling or Disabling SR-IOV Support

Standardized INF keywords are defined to enable or disable support for the SR-IOV features of a network adapter.

***SRIOV**
A value that describes whether the device has enabled or disabled the SR-IOV feature.

After the driver is installed, administrators can update the ***SRIOV** keyword value in the **Advanced** property page for the network adapter. For more information about advanced properties, see Specifying Configuration Parameters for the Advanced Properties Page.

Note The miniport driver is automatically restarted after a change is made in the **Advanced** property page for the adapter.

***SriovPreferred**
A value that defines whether SR-IOV capabilities should be enabled instead of virtual machine queue (VMQ) or receive side scaling (RSS) capabilities.

This is a hidden keyword value that must not be specified in the INF file and is not displayed in **Advanced** property page for the network adapter.

For more information about how to interpret SR-IOV, VMQ, and RSS keywords, see Handling SR-IOV, VMQ, and RSS Standardized INF Keywords.

The SR-IOV standardized INF keywords are enumeration keywords and are described in the following table. The columns in this table describe the following attributes for an enumeration keyword:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params\** key for the network adapter.

ParamDesc
The display text that is associated with the **SubkeyName** keyword.

Note The independent hardware vendor (IHV) can define any descriptive text for the SubkeyName.

Value
The enumeration integer value that is associated with each **SubkeyName** keyword in the list.

EnumDesc
The display text that is associated with each value that appears in the menu.

SubkeyName	ParamDesc	Value	EnumDesc
***SRIOV**	SR-IOV	0	Disabled
		1 (Default)	Enabled
***SriovPreferred**	The ParamDesc and EnumDesc entries for this subkey cannot be used in either INF files or a user interface.	0 (Default)	Report RSS or VMQ capabilities based on the ***VmqOrRssPreferrence** keyword. Do not report SR-IOV capabilities. For more information about the ***VmqOrRssPreferrence** keyword, see Standardized INF Keywords for VMQ.
		1	Report SR-IOV capabilities.

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

Standardized INF Keywords for Configuration of the Default NIC Switch

Starting with Windows Server 2012, the SR-IOV interface supports only one NIC switch on the network adapter. This switch is known as the *default NIC switch*, and is referenced by the NDIS_DEFAULT_SWITCH_ID identifier.

The INF file for the PF miniport driver must specify the configuration of the default NIC switch on the SR-IOV network adapter. This allows the network installer to copy the default switch configuration information from the INF to the miniport registry configuration under the subkey for the default switch (**NDI\params\NicSwitches\0**).

These keywords are not displayed in the **Advanced** property page for the network adapter and cannot be configured by the user. These keywords are specified by using the **AddReg** directive in the **DDInstall** section of the INF file. Each keyword is specified by a separate **AddReg** directive.

The following table describes the INF keywords for the default NIC switch configuration of the SR-IOV network adapter. The columns in this table describe the following attributes for these keywords:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params\NicSwitches\0** key for the network adapter.

Data value
The value that is associated with the **SubkeyName** keyword.

Data type
The type of the data value.

SubkeyName	Data value	Data type	Notes
***Flags**	0	REG_DWORD	The keyword must be assigned this value.
***SwitchType**	1	REG_DWORD	The keyword must be assigned this value.
***SwitchId**	0	REG_DWORD	The keyword must be assigned this value.
***SwitchName**	“Default Switch”	REG_SZ	The keyword must be assigned this value.
***NumVFs**	(0-*n*),	REG_DWORD	*n* is the maximum number of PCIe Virtual Functions (VFs) that are supported by the SR-IOV network adapter. **Note** This registry key defines the maximum number of VFs that the network adapter supports. When the miniport driver calls NdisMSetMiniportAttributes, it can advertise less than this value depending on the available hardware resources on the network adapter. For more information, see Determining NIC Switch Capabilities.

The following is an example of **AddReg** directives for the default NIC switch configuration of an SR-IOV network adapter:

syntax

```
HKR, NicSwitches\0, *SwitchId,   0x00010001, 0
HKR, NicSwitches\0, *SwitchName, 0x00000000, “Default Switch”
```

For more information about the syntax of the **AddReg** directive, see **INF AddReg Directive**.

For more information about the default NIC switch, see NIC Switches.

INF DDInstall.HW Section for PF Miniport Drivers

Article • 12/15/2021

INF *DDInstall*.**HW** sections are typically used for setting up any device-specific information in the registry, whether with explicit **AddReg** directives or with **Include** and **Needs** entries.

The INF file for the miniport driver of the PCI Express (PCIe) Physical Function (PF) network adapter must have a *DDInstall*.**HW** section that contains the following INF entries:

- An **Include** entry that specifies the pci.inf file that is included with the Windows operating system.

- A **Needs** entry that specifies the **PciSriovSupported.HW** section to include from the pci.inf file. This section defines standard INF settings that apply to all PF miniport drivers for network adapters that support the single root I/O virtualization (SR-IOV) interface.

The following is an example of a *DDInstall*.**HW** section for a PF miniport driver:

```syntax
[Device_Inst.NT.HW]

Include=pci.inf
Needs=PciSriovSupported.HW
```

For more information about the *DDInstall* section, see DDInstall Section in a Network INF File.

For more information about the *DDInstall*.HW section, see **INF DDInstall.HW Section**.

Handling SR-IOV, VMQ, and RSS Standardized INF Keywords

Article • 12/15/2021

Network adapters that support single root I/O virtualization (SR-IOV), virtual machine queue (VMQ), and receive side scaling (RSS) can enable the use of these interfaces in the following way:

- SR-IOV and VMQ can be enabled individually or at the same time.
- RSS cannot be enabled on the network adapter when SR-IOV or VMQ is enabled.

The operating system enables the use of the SR-IOV, VMQ, or RSS interfaces in the following way:

- When the network adapter is bound to the TCP/IP stack, the operating enables the use of the RSS feature.
- When the network adapter is bound to the Hyper-V extensible switch driver stack, the operating system enables the use of either the SR-IOV or VMQ feature.

  For more information about the Hyper-V extensible switch, see Hyper-V Extensible Switch.

When the network adapter is unbound from the TCP/IP stack and the Hyper-V extensible switch driver stack, the miniport driver is halted and then reinitialized. Because of this, it is not possible for such network adapters to switch between RSS, VMQ, and SR-IOV automatically.

When NDIS calls the *MiniportInitializeEx* function, the miniport driver follows these steps before it reports its currently enabled SR-IOV, VMQ, or RSS capabilities to NDIS:

1. The miniport driver reads the ***SriovPreferred** keyword before reporting its currently enabled capabilities to NDIS.

   If the value of the ***SriovPreferred** keyword is one, the miniport driver is configured for SR-IOV preference.

2. The miniport driver reads the ***RssOrVmqPreference** keyword before reporting its currently enabled capabilities to NDIS.

   If the value of the ***RssOrVmqPreference** keyword is one, the miniport driver is configured for VMQ preference.

If the value of the ***RssOrVmqPreference** keyword is zero or the keyword is not present, the miniport driver is configured for RSS preference.

3. If the miniport driver is configured for SR-IOV preference, it must read the ***SRIOV** keyword to determine whether SR-IOV is enabled on the network adapter. If the keyword is set to one, the driver reports the currently enabled SR-IOV settings.

   For more information on how the miniport driver reports SR-IOV settings, see Determining SR-IOV Capabilities.

   For more information about the SR-IOV keywords, see Standardized INF Keywords for SR-IOV.

   Note If the miniport driver is configured for SR-IOV preference, it must not read any of the RSS standardized keywords. However, the driver must read the VMQ ***VMQVlanFiltering** standardized keyword. This keyword specifies whether the miniport driver is enabled to filter network packets by using the virtual VLAN (VLAN) identifier in the media access control (MAC) header. The miniport driver reports this capability by setting the NDIS_RECEIVE_FILTER_MAC_HEADER_VLAN_ID_SUPPORTED flag in the **SupportedMacHeaderFields** member of the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure. For more information on the ***VMQVlanFiltering** standardized keyword, see Standardized INF Keywords for VMQ.

4. If the miniport driver is configured for VMQ preference, it must read the ***VMQ** keyword to determine whether VMQ is enabled on the network adapter. If the keyword is set to one, the driver reports the currently enabled VMQ settings. For more information on how the miniport driver reports VMQ settings, see Determining the VMQ Capabilities of a Network Adapter.

   For more information about VMQ keywords, see Standardized INF Keywords for VMQ.

   Note If the miniport driver is configured for VMQ preference, it must not read any of the RSS or SR-IOV standardized keywords.

5. If the miniport driver is configured for RSS preference, it must read the ***RSS** keyword to determine whether RSS is enabled on the network adapter. If the keyword is set to one, the driver reports the currently enabled RSS settings. For more information on how the miniport driver reports RSS settings, see RSS Configuration.

   For more information about the RSS keywords, see Standardized INF Keywords for RSS.

> **Note** If the miniport driver is configured for RSS preference, it must not read any of the VMQ or SR-IOV standardized keywords.

The following table describes how the miniport driver determines SR-IOV, VMQ, or RSS preference in order to enable the correct interface in the network adapter.

SriovPreferred	RssOrVmqPreference	*SRIOV*	VMQ	*RSS	Enabled interface
1	1	1	1	N/A	SR-IOV and VMQ
1	1	0	1	N/A	VMQ
1	1, 0, or not present in registry	0	0	N/A	None
0, or not present in registry	1	N/A	1	N/A	VMQ
0, or not present in registry	1	N/A	0	N/A	None
0, or not present in registry	0, or not present in registry	N/A	N/A	1	RSS
0, or not present in registry	0, or not present in registry	N/A	N/A	0	None

Note When the SR-IOV and VMQ interfaces are both enabled, SR-IOV nondefault virtual ports (VPorts) that are attached to the PCI Express (PCIe) Physical Function (PF) are used instead of VM queues for the VMQ interface. For more information, see Nondefault Virtual Ports and VMQ.

The miniport driver must advertise the capabilities of the currently enabled interface. For example, if SR-IOV is enabled, the miniport driver must advertise the SR-IOV capabilities but not the capabilities for VMQ or RSS. However, the miniport driver must always report the complete RSS, VMQ, and SR-IOV hardware capabilities regardless of which interface is enabled on the network adapter.

Note The VMQ and SR-IOV interfaces use receive filtering over VM queues or SR-IOV virtual ports (VPorts). As a result, some receive filtering capabilities require the same or different settings when either of these interfaces are enabled. For more information on how to report the receive filtering capabilities for the SR-IOV interface, see Determining Receive Filtering Capabilities. For more information on how to report the receive filtering capabilities for the VMQ interface, see Determining the VMQ Capabilities of a Network Adapter.

Writing SR-IOV VF Miniport Drivers Overview

Article • 12/15/2021

This section discusses the requirements and guidelines for writing an NDIS miniport driver for the PCI Express (PCIe) Virtual Function (VF) of a single root I/O virtualization (SR-IOV) network adapter.

This section includes the following topics:

Initializing a VF Miniport Driver

Halting a VF Miniport Driver

INF Requirements for VF Miniport Drivers

Note For information on how to write a miniport driver for a PCIe Physical Function (PF) of the SR-IOV network adapter, see Writing SR-IOV PF Miniport Drivers.

Initializing a VF Miniport Driver

Article • 12/15/2021

This topic describes the guidelines for writing a *MiniportInitializeEx* function for the miniport driver for a PCI Express (PCIe) Virtual Function (VF). The VF is exposed by a network adapter that supports single root I/O virtualization (SR-IOV).

> ⓘ Note
>
> These guidelines only apply to VF miniport drivers of the SR-IOV network adapter. For initialization guidelines for the miniport driver of a PCIe Physical Function (PF) of the adapter, see **Initializing a PF Miniport Driver**.

The VF miniport driver follows the same steps as any NDIS miniport driver when its *MiniportInitializeEx* function is called. For more information about these steps, see Initializing a Miniport Driver.

In addition to these steps, the VF miniport driver must follow these additional steps when NDIS calls the driver's *MiniportInitializeEx* function:

- The VF miniport driver calls the **NdisGetHypervisorInfo** function to verify that it is running in the Hyper-V child partition. This function returns an **NDIS_HYPERVISOR_INFO** structure which defines the partition type. If the partition type is reported as **NdisHypervisorPartitionMsHvChild**, the miniport driver is running in a Hyper-V child partition that is attached to the PF on the adapter.

  > ⓘ Note
  >
  > If the partition type is reported as **NdisHypervisorPartitionMsHvParent**, the miniport driver is running in the Hyper-V parent partition that is attached to the PF on the adapter. In this case, the miniport driver must not initialize as a VF driver. If possible, the driver must initialize as a PF driver as described in **Initialization Sequence for PF Miniport Drivers**.

- Unlike the PF miniport driver, the VF miniport driver must not be installed with the SR-IOV standardized keywords and must not attempt to read these keywords. For more information about these keywords, see Standardized INF Keywords for SR-IOV.

- The VF miniport driver reports the SR-IOV hardware capabilities of the underlying virtual network adapter through an NDIS_SRIOV_CAPABILITIES structure that is initialized in the following way:

  1. The miniport driver initializes the **Header** member. The driver sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT.

     Starting with NDIS 6.30, the miniport driver sets the **Revision** member of **Header** to NDIS_SRIOV_CAPABILITIES _REVISION_1 and the **Size** member to NDIS_SIZEOF_SRIOV_CAPABILITIES_REVISION_1.

  2. The miniport driver sets the NDIS_SRIOV_CAPS_PF_MINIPORT flag in the **SriovCapabilities** member to report SR-IOV capabilities.

     > ⓘ Note
     >
     > The VF miniport driver must set both the NDIS_SRIOV_CAPS_VF_MINIPORT flag and the NDIS_SRIOV_CAPS_SRIOV_SUPPORTED flag.

  The VF miniport driver registers the SR-IOV capabilities of the network adapter by following these steps:

  1. The miniport driver initializes an NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

     The miniport driver sets the **HardwareSriovCapabilities** and **CurrentSriovCapabilities** members to a pointer to the previouslyinitialized NDIS_SRIOV_CAPABILITIES structure.

  2. The driver calls NdisMSetMiniportAttributes and sets the *MiniportAttributes* parameter to a pointer to the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

- The VF miniport driver must not advertise virtual machine queue (VMQ) capabilities. However, the driver can advertise support for other NDIS technologies, such as power management and receive side scaling (RSS).

  For more information about RSS, see Receive Side Scaling.

Halting a VF Miniport Driver

Article • 12/15/2021

The VF miniport driver follows the same steps as any NDIS miniport driver when its *MiniportHaltEx* function is called. For more information about these steps, see Halting a Miniport Adapter.

INF Requirements for VF Miniport Drivers

Article • 12/15/2021

The INF file for the miniport driver of a PCI Express (PCIe) Virtual Function (VF) does not specify any standardized INF keywords for single root I/O virtualization (SR-IOV). Only the INF file of a PCIe Physical Function (PF) specifies standardized SR-IOV keywords. For more information about these keywords, see INF Requirements for PF Miniport Drivers.

The INF for a VF miniport driver follows (with one exception) the same requirements as other INF files for network adapters. For more information, see Standardized INF Keywords for Network Devices.

The only exception is that the INF file for the VF miniport driver must define the binding relationships to the services that manage the SR-IOV data paths. This is needed to ensure that network access can fail over to the synthetic data path if the VF data path is torn down for any reason. For more information about these data paths, see SR-IOV Data Paths.

To bind to the services that manage these data paths, the INF file for the VF miniport driver must specify the following settings for the **UpperRange** and **LowerRange** entries:

syntax

```
HKR, Ndi\Interfaces, UpperRange, 0, "ndisvf"
HKR, Ndi\Interfaces, LowerRange, 0, "iovvf"
```

SR-IOV PF/VF Backchannel Communication Overview

Article • 12/15/2021

The single root I/O virtualization (SR-IOV) interface provides a communication channel, or *backchannel*, between the miniport drivers of a PCI Express (PCIe) Virtual Function (VF) and the PCIe Physical Function (PF). Each VF miniport driver can issue requests over the backchannel to the PF miniport driver. The PF miniport driver can issue status notifications over the backchannel to individual VF miniport drivers.

Data exchanged between the PF and VF miniport drivers over the backchannel interface involves the use of a *VF configuration block*. Each VF configuration block is similar in concept to an interprocess communication (IPC) message, in which each block has a proprietary format, length, and block identifier. The independent hardware vendor (IHV) can define one or more VF configuration blocks for the PF and VF miniport drivers.

This section includes the following topics:

Backchannel Communication from a VF Miniport Driver

Backchannel Communication from the PF Miniport Driver

Backchannel Communication from a VF Miniport Driver

Article • 12/15/2021

A miniport driver of a PCI Express (PCIe) Virtual Function (VF) communicates with the miniport driver of the PCIe Physical Function (PF) to read or write data from a VF configuration block.

A VF configuration block is used for backchannel communication between the PF and VF miniport drivers. The independent hardware vendor (IHV) can define one or more VF configuration blocks for the device. Each VF configuration block has an IHV-defined format, length, and block ID. For example, the IHV can define a VF configuration block that can be used for the media access control (MAC) address of the VF miniport driver. Another VF configuration block can be used for the current VF and virtual port (VPort) configuration.

Note Data from each VF configuration block is used only by the PF and VF miniport drivers. The format and content of this data is opaque to components of the Windows operating system.

Each VF configuration block is assigned a unique identifier by the IHV. This allows the VF miniport driver to query or set information on specific VF configuration blocks.

VF miniport drivers initiate the read or write operation on a specified VF configuration block through the following functions:

- **NdisMReadConfigBlock**, which reads data from a specified VF configuration block. When the VF miniport driver calls this function, it specifies the block identifier and length of the data to be read. The driver also passes a pointer to the buffer that will contain the requested data.

- **NdisMWriteConfigBlock**, which writes data to a specified VF configuration block. When the VF miniport driver calls this function, it specifies the block identifier and length of the data to be written. The driver also passes a pointer to the buffer from which the data is to be written.

The PF miniport driver manages access to the specified VF configuration block in the following ways:

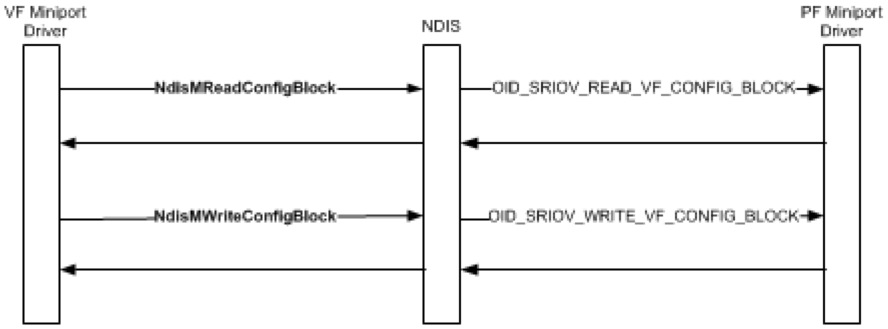
- When the VF miniport driver calls **NdisMReadConfigBlock**, NDIS issues an object identifier (OID) method request of OID_SRIOV_READ_VF_CONFIG_BLOCK to the PF miniport driver. This OID request contains the parameter data that was passed by the VF miniport driver in the function call.

The PF miniport driver performs the read operation and returns the requested data when the driver completes the OID request. After the OID request is completed, NDIS returns from the call to **NdisMReadConfigBlock**.

- When the VF miniport driver calls **NdisMWriteConfigBlock**, NDIS issues an OID method request of OID_SRIOV_WRITE_VF_CONFIG_BLOCK to the PF miniport driver. This OID request contains the parameter data that was passed by the VF miniport driver in the function call.

  The PF miniport driver performs the write operation and completes the OID request. After the OID request is completed, NDIS returns from the call to **NdisMWriteConfigBlock**.

The following figure shows the process involved in reading and writing VF configuration blocks over the SR-IOV backchannel interface.



Backchannel Communication from the PF Miniport Driver

Article • 12/15/2021

A miniport driver of the PCI Express (PCIe) Physical Function (PF) communicates with a miniport driver of the PCIe Virtual Function (VF) to issue notifications about changes in the data of a VF configuration block. The PF miniport driver issues these notifications to *invalidate* the data in the VF configuration block. In response to this notification, the VF miniport driver can issue a backchannel request to the PF miniport driver to read the data from an invalidated VF configuration block.

A VF configuration block is used for backchannel communication between the PF and VF miniport drivers. The IHV can define one or more VF configuration blocks for the device. Each VF configuration block has an IHV-defined format, length, and block ID.

Note Data from each VF configuration block is used only by the PF and VF miniport drivers. The format and content of this data is opaque to components of the Windows operating system.

The following steps occur when issuing and handling notifications of invalid VF configuration data:

1. In the guest operating system, NDIS issues an I/O control request of IOCTL_VPCI_INVALIDATE_BLOCK. When this IOCTL is completed, NDIS is notified that VF configuration data has changed.

2. In the management operating system that runs in the Hyper-V parent partition, the following steps occur:

   a. The PF miniport driver calls the NdisMInvalidateConfigBlock function to notify NDIS that VF configuration data has changed and is no longer valid. The driver sets the *BlockMask* parameter to a ULONGLONG bitmask that specifies which VF configuration blocks have changed. Each bit in the bitmask corresponds to a VF configuration block. If the bit is set to one, the data in the corresponding VF configuration block has changed.

   b. NDIS signals the virtualization stack, which runs in the management operating system, about the change to VF configuration block data. The virtualization stack caches the *BlockMask* parameter data.

      Note Each time that the PF miniport driver calls NdisMInvalidateConfigBlock, the virtualization stack ORs the *BlockMask* parameter data with the current value in its cache.

   c. The virtualization stack notifies the virtual PCI (VPCI) driver, which runs in the guest operating system, about the invalidation of VF configuration data. The virtualization stack sends the cached *BlockMask* parameter data to the VPCI driver.

3. In the guest operating system that runs in a Hyper-V child partition, the following steps occur:

   a. The VPCI driver saves the cached *BlockMask* parameter data in the **BlockMask** member of the **VPCI_INVALIDATE_BLOCK_OUTPUT** structure that is associated with the **IOCTL_VPCI_INVALIDATE_BLOCK** request.

   b. The VPCI driver successfully completes the **IOCTL_VPCI_INVALIDATE_BLOCK** request. When this happens, NDIS issues an object identifier (OID) method request of OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK to the VF miniport driver. An **NDIS_SRIOV_VF_INVALIDATE_CONFIG_BLOCK_INFO** is passed along in the OID request. This structure contains the cached *BlockMask* parameter data.

      NDIS also issues another **IOCTL_VPCI_INVALIDATE_BLOCK** request to handle successive notifications of changes to VF configuration data.

   c. When the VF driver handles the OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK request, it can read data from the specified VF configuration blocks by calling **NdisMReadConfigBlock**. For more information about this process, see Backchannel Communication from a VF Miniport Driver.

SR-IOV OIDs

Article • 12/15/2021

The single root I/O virtualization (SR-IOV) object identifiers (OIDs) apply to miniport and overlying drivers that support the SR-IOV interface. This interface is supported in NDIS version 6.30 and later versions.

The following table defines the characteristics of the SR-IOV OIDs. The following abbreviations are used to specify the OIDs' characteristics in the table.

- Q
  The OID is used only in query requests.
- S
  The OID is used only in set requests.
- M
  The OID is used only in method requests. These requests could be issued for set or query operations.
- N
  The OID request is handled directly by NDIS and not by the miniport driver. The driver will not be issued these OIDs.
- P
  The OID request is issued only to the miniport driver of the network adapter's physical function (PF).
  The PF driver must support these OIDs. The driver must also list these OIDs in the **SupportedOidList** member of the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure that the driver passes in the *MiniportAttributes* parameter of the call to NdisMSetMiniportAttributes.
- V
  The OID request is issued only to the miniport driver of one of the network's virtual functions (VFs).
  The VF driver must support these OIDs. The driver must also list these OIDs in the **SupportedOidList** member of the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure that the driver passes in the *MiniportAttributes* parameter of the call to NdisMSetMiniportAttributes.

Name	Q	S	M	N	P	V
OID_NIC_SWITCH_ALLOCATE_VF			X		X	
OID_NIC_SWITCH_CREATE_SWITCH			X		X	
OID_NIC_SWITCH_CREATE_VPORT			X		X	

Name	Q	S	M	N	P	V
OID_NIC_SWITCH_CURRENT_CAPABILITIES	X			X		
OID_NIC_SWITCH_DELETE_SWITCH		X			X	
OID_NIC_SWITCH_DELETE_VPORT		X			X	
OID_NIC_SWITCH_ENUM_SWITCHES	X			X		
OID_NIC_SWITCH_ENUM_VFS	X			X		
OID_NIC_SWITCH_ENUM_VPORTS	X			X		
OID_NIC_SWITCH_FREE_VF		X			X	
OID_NIC_SWITCH_HARDWARE_CAPABILITIES	X			X		
OID_NIC_SWITCH_PARAMETERS			X		X	
OID_NIC_SWITCH_VF_PARAMETERS			X		X	
OID_NIC_SWITCH_VPORT_PARAMETERS			X		X	
OID_SRIOV_BAR_RESOURCES		X			X	
OID_SRIOV_CURRENT_CAPABILITIES	X			X		
OID_SRIOV_HARDWARE_CAPABILITIES	X			X		
OID_SRIOV_PF_LUID	X			X		
OID_SRIOV_PROBED_BARS	X				X	
OID_SRIOV_READ_VF_CONFIG_BLOCK			X		X	
OID_SRIOV_READ_VF_CONFIG_SPACE			X		X	
OID_SRIOV_RESET_VF		X			X	
OID_SRIOV_SET_VF_POWER_STATE		X			X	
OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK			X			X
OID_SRIOV_VF_SERIAL_NUMBER	X			X		
OID_SRIOV_VF_VENDOR_DEVICE_ID			X		X	
OID_SRIOV_WRITE_VF_CONFIG_BLOCK		X			X	
OID_SRIOV_WRITE_VF_CONFIG_SPACE		X			X	

NDIS_STATUS_NIC_SWITCH_CURRENT_CAPABILITIES

Article • 03/14/2023

The **NDIS_STATUS_NIC_SWITCH_CURRENT_CAPABILITIES** status indicates to NDIS and overlying drivers that the currently enabled hardware capabilities of the NIC switch in a network adapter have changed.

The status indication is made by the miniport driver of the network adapter's PCI Express (PCIe) Physical Function (PF). The PF miniport driver runs in the management operating system of the Hyper-V parent partition.

Remarks

The PF miniport driver must issue an **NDIS_STATUS_NIC_SWITCH_CURRENT_CAPABILITIES** status indication whenever it detects a change to the currently enabled hardware capabilities of the NIC switch on the network adapter. These capabilities could change when one of the following conditions is true:

- The currently enabled NIC switch hardware capabilities are changed through a management application developed by the independent hardware vendor (IHV).
- The currently enabled NIC switch hardware capabilities change for one or more network adapters that belong to a load balancing failover (LBFO) team managed by a MUX intermediate driver. For more information, see NDIS MUX Intermediate Drivers.

When the PF miniport driver issues the **NDIS_STATUS_NIC_SWITCH_CURRENT_CAPABILITIES** status indication, it must follow these steps:

1. The miniport driver initializes an NDIS_NIC_SWITCH_CAPABILITIES structure with the currently enabled hardware capabilities of the network adapter's NIC switch.
2. The miniport driver initializes an NDIS_STATUS_INDICATION structure in the following way:
    - The **StatusCode** member must be set to **NDIS_STATUS_NIC_SWITCH_CURRENT_CAPABILITIES**.

- The **StatusBuffer** member must be set to the pointer to a NDIS_NIC_SWITCH_CAPABILITIES structure. This structure contains the currently enabled hardware capabilities of the NIC switch.

- The **StatusBufferSize** member must be set to sizeof(NDIS_NIC_SWITCH_CAPABILITIES).

3. The PF miniport driver issues the status notification by calling NdisMIndicateStatusEx. The driver must pass a pointer to the NDIS_STATUS_INDICATION structure to the *StatusIndication* parameter.

Overlying drivers can use the **NDIS_STATUS_NIC_SWITCH_CURRENT_CAPABILITIES** status indication to determine the currently enabled NIC switch capabilities on the network adapter. Alternatively, these drivers can also issue OID query requests of OID_NIC_SWITCH_CURRENT_CAPABILITIES to obtain these capabilities at any time.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h

See also

NDIS_NIC_SWITCH_CAPABILITIES

NDIS_STATUS_INDICATION

OID_NIC_SWITCH_CURRENT_CAPABILITIES

NDIS_STATUS_NIC_SWITCH_HARDWARE_CAPABILITIES

Article • 03/14/2023

The **NDIS_STATUS_NIC_SWITCH_HARDWARE_CAPABILITIES** status indicates to NDIS and overlying drivers that the hardware capabilities of the NIC switch in a network adapter have changed. These capabilities include the hardware capabilities that are currently disabled by INF file settings or through the **Advanced** properties page.

The status indication is made by the miniport driver of the network adapter's PCI Express (PCIe) Physical Function (PF). The PF miniport driver runs in the management operating system of the Hyper-V parent partition.

Remarks

The PF miniport driver must issue an **NDIS_STATUS_NIC_SWITCH_HARDWARE_CAPABILITIES** status indication whenever it detects a change to the hardware capabilities of the NIC switch on the network adapter. These capabilities could change when one of the following conditions is true:

- The NIC switch hardware capabilities are enabled or disabled through a management application developed by the independent hardware vendor (IHV).
- The NIC switch hardware capabilities change for one or more network adapters that belong to a load balancing failover (LBFO) team managed by a MUX intermediate driver. For more information, see NDIS MUX Intermediate Drivers.

When the PF miniport driver issues the **NDIS_STATUS_NIC_SWITCH_HARDWARE_CAPABILITIES** status indication, it must follow these steps:

1. The miniport driver initializes an **NDIS_NIC_SWITCH_CAPABILITIES** structure with the hardware capabilities of the network adapter's NIC switch.
2. The miniport driver initializes an **NDIS_STATUS_INDICATION** structure in the following way:
    - The **StatusCode** member must be set to **NDIS_STATUS_NIC_SWITCH_HARDWARE_CAPABILITIES**.

- The **StatusBuffer** member must be set to the pointer to a NDIS_NIC_SWITCH_CAPABILITIES structure. This structure contains the hardware capabilities of the NIC switch.

- The **StatusBufferSize** member must be set to sizeof(NDIS_NIC_SWITCH_CAPABILITIES).

3. The PF miniport driver issues the status notification by calling NdisMIndicateStatusEx. The driver must pass a pointer to the NDIS_STATUS_INDICATION structure to the *StatusIndication* parameter.

Overlying drivers can use the **NDIS_STATUS_NIC_SWITCH_HARDWARE_CAPABILITIES** status indication to determine the currently enabled NIC switch capabilities on the network adapter. Alternatively, these drivers can also issue OID query requests of OID_NIC_SWITCH_HARDWARE_CAPABILITIES to obtain these capabilities at any time.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h

See also

NDIS_NIC_SWITCH_CAPABILITIES

NDIS_STATUS_INDICATION

OID_NIC_SWITCH_HARDWARE_CAPABILITIES

Virtual Machine Queue (VMQ) Overview

Article • 09/12/2022

The NDIS Virtual Machine Queue (VMQ) interface supports Microsoft Hyper-V network performance improvements in NDIS 6.20 and later in Windows Server 2008 R2 and later versions of Windows Server.

The VMQ interface supports:

- Classification of received packets in network adapter hardware by using the destination media access control (MAC) address to route the packets to different receive queues.
- Shared memory; For more information see NDIS Memory Management Interface.
- Scaling to multiple processors by processing packets for different virtual machines on different processors.

The NDIS VMQ architecture provides advantages for virtualization such as:

- Virtualization impacts performance and VMQ helps overcome those effects.
- VMQ supports live migration.
- VMQ coexists with NDIS task offloads and other optimizations.

This section provides high-level information about the NDIS VMQ interface. You should read this section before writing an NDIS driver that supports VMQ.

For information about writing VMQ drivers, see Writing VMQ Drivers.

> **Note**
>
> Be sure to study the **NDIS Virtual Miniport Driver sample**, especially the vmq.c and vmq.h source files.

This section includes the following topics:

Introduction to NDIS Virtual Machine Queue (VMQ)

VMQ Components

VMQ Receive Queues

VMQ Receive Filters

Security Issues with NDIS Virtual Machine (VM) Shared Memory

NDIS VMQ Live Migration Support

NDIS VM Queue States

Introduction to NDIS Virtual Machine Queue (VMQ)

Article • 03/14/2023

Many network adapters can support more than one unicast media access control (MAC) address for a network server. Therefore, the network adapter can receive network data frames with a destination MAC address that matches any of the unicast MAC addresses that are set on the network adapter hardware without being in promiscuous mode. Such hardware can allocate a receive queue for each MAC address and route incoming frames with a matching MAC address to the queue. This feature, coupled with the ability to allocate receive buffers for each queue from the memory address space that is assigned to each virtual machine, are the primary capabilities that are required for VMQ support.

A VMQ-capable network adapter can use DMA to transfer all incoming frames that should be routed to a receive queue to the receive buffers that are allocated for that queue. The miniport driver can indicate all of the frames that are in a receive queue in one receive indication call.

VMQ provides the following features:

- Improves network throughput by distributing processing of network traffic for multiple virtual machines (VMs) among multiple processors.

  Note In Hyper-V, a child partition is also known as a VM.

- Reduces CPU utilization by offloading receive packet filtering to network adapter hardware.

- Prevents network data copy by using DMA to transfer data directly to VM memory.

- Splits network data to provide a secure environment. For more information about security issues, see Security Issues with NDIS Virtual Machine (VM) Shared Memory.

  Note Starting with NDIS 6.30 and Windows Server 2012, splitting network data into separate lookahead buffers is no longer supported.

- Supports live migration. For more information about live migration, see NDIS VMQ Live Migration Support.

To introduce high-level VMQ concepts, this section includes the following additional topics:

VMQ Components

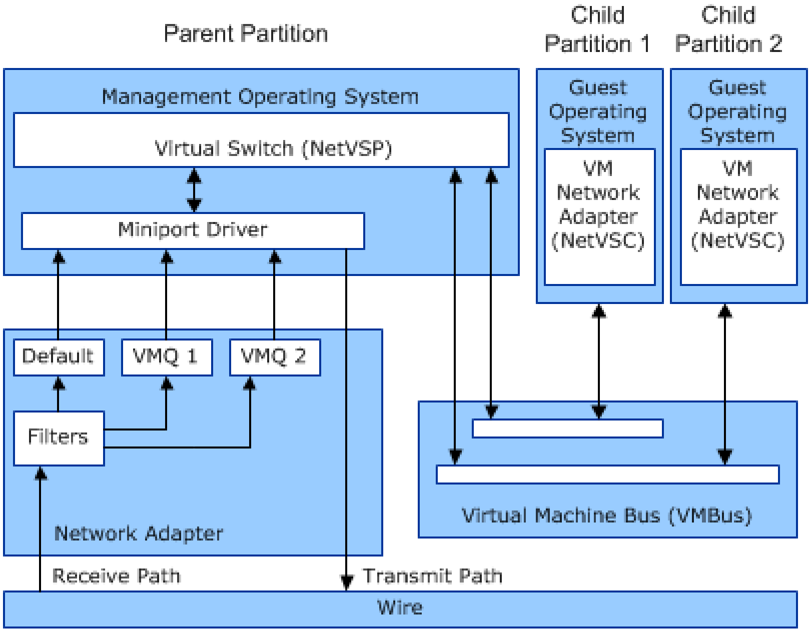
VMQ Receive Queues

VMQ Receive Filters

VMQ Components

Article • 12/15/2021

The following illustration shows the relationships among the various components in a virtual machine queue (VMQ) operating environment.



The preceding figure illustrates the following VMQ components:

Network Virtual Service Provider (NetVSP)
An NDIS driver that runs in the management operating system of the Hyper-V parent partition. This driver provides services to support networking access by the Hyper-V child partitions.

Note Starting with Windows Server 2008, the Hyper-V extensible switch component provides NetVSP support to the NetVSC components that run in the guest operating system. For more information about this component, see Hyper-V Extensible Switch.

Network Virtual Service Client (NetVSC)
An NDIS driver that runs in the guest operating system of a Hyper-V child partition. NetVSC exposes a virtualized view of the physical network adapter on the host computer. This virtualized device is known as the *VM network adapter*.

The NetVSC provides the following functionality:

- Supports networking device functionality in Hyper-V child partitions.
- Accesses the physical network adapter by passing messages over the virtual machine bus (VMBus) to the associated NetVSP driver. This driver runs in the management operating system of the Hyper-V parent partition.

Virtual Machine Bus (VMBus)
A virtual communications bus that passes control and data messages between the Hyper-V parent and child partitions.

Note In Hyper-V, a child partition is also known as a virtual machine (VM).

VM Bus Channel
A communications channel on the VMBus between a NetVSC in a Hyper-V child partition and the NetVSP in the Hyper-V parent partition.

VM Queue
A queue for received data. A network adapter that supports VMQ has hardware to route data to a VM queue.

VMQ Filter
A filter to test incoming data. A network adapter that supports VMQ uses filters to test packet data in order to assign the packet to a queue.

VMQ Receive Queues

Article • 12/15/2021

A virtual machine queue (VMQ) service provider allocates VMQ receive queues. The network adapter hardware assigns an incoming network data packet to a queue if the packet passes the filter tests that are set on the queue.

A VMQ receive queue has the following properties:

- A queue identifier that is unique to the associated network adapter.
- Processor affinity for interrupts.
- Filters that are set on the queue.
- Receive buffers that are assigned to the queue.

There is also a default queue that has the following properties:

- The default queue always exists. Other queues must be allocated.
- The default queue receives packets that do not pass the filter tests for the other queues.

Miniport drivers allocate shared memory for the receive buffers that are associated with a VMQ. Depending on the Windows Server version, miniport drivers must follow guidelines for buffer allocation that are described in the following sections:

- Allocating Shared Memory for VMQ Receive Buffers (Windows Server 2008 R2)
- Allocating Shared Memory for VMQ Receive Buffers (Windows Server 2012 and Later Versions)

VMQ shared memory requirements are designed to address potential security issues for virtual machines (VMs). For more information about VMQ security issues, see Security Issues with NDIS Virtual Machine (VM) Shared Memory.

Allocating Shared Memory for VMQ Receive Buffers (Windows Server 2008 R2)

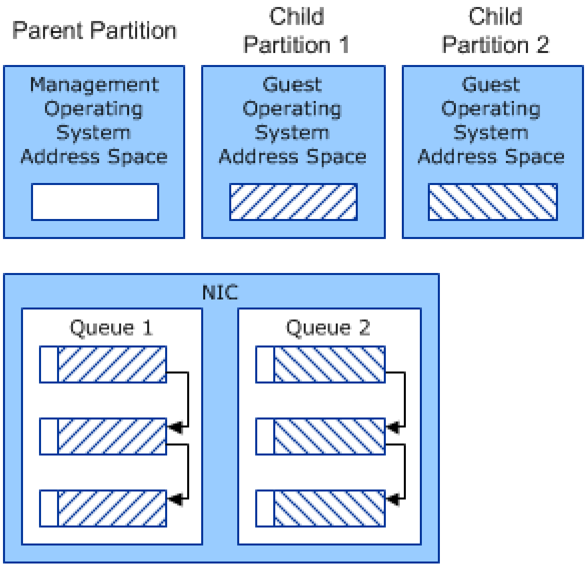
For NDIS 6.20 in Windows Server 2008 R2, if the miniport driver supports splitting packet data into separate lookahead buffers, it can allocate shared memory in the following way:

- The miniport driver allocates the shared memory for the pre-lookahead buffer from the address space of the management operating system that runs in a Hyper-V parent partition. The pre-lookahead buffer is the part of the packet that is inspected by the management operating system.

- The miniport driver allocates the shared memory for the post-lookahead buffer from the address space of the guest operating system that runs in a Hyper-V child partition. The post-lookahead buffer is the part of the packet that is inspected by the guest operating system.

  Note The Hyper-V child partition is also known as a VM.

The following figure shows the shared memory in the queues, the management operating system, and the guest operating systems.


Parent Partition
Child Partition 1
Child Partition 2
Management Operating System Address Space
Guest Operating System Address Space
Guest Operating System Address Space
NIC
Queue 1
Queue 2

In the figure, each packet in a queue is shown with header information that was allocated from the management operating system address space and data that was allocated from the guest operating system address space.

Allocating Shared Memory for VMQ Receive Buffers (Windows Server 2012 and Later Versions)

Starting with NDIS 6.30, splitting VMQ receive buffers into separate lookahead buffers is no longer supported. The miniport driver must allocate memory for each receive buffer from the address space of the management operating system.

VMQ Receive Filters

Article • 12/15/2021

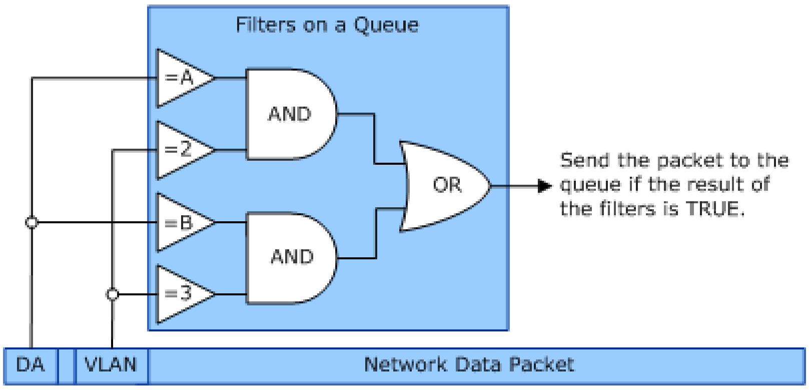
A network virtual service provider (VSP) sets VMQ receive filters on VMQ receive queues. Such a filter includes a set of network header field tests. The network adapter hardware performs these tests on header fields in incoming packets to determine the receive queue assignments for the packets. Each filter that is set on a queue has a unique filter identifier for a network adapter. That is, the filter identifiers are not duplicated on different queues that the network adapter manages.

The VMQ interface uses fields in the media access control (MAC) header in filter tests. Within the MAC header, VMQ filter tests use the virtual local area network (VLAN) identifier and the destination MAC address fields.

Multiple field tests can be specified in a receive filter. All of the tests must pass to match the criterion for the filter and assign a packet to a receive queue. VMQ filters test for fields that are equal to a specified value. For example, the destination MAC address is equal to a specified address.

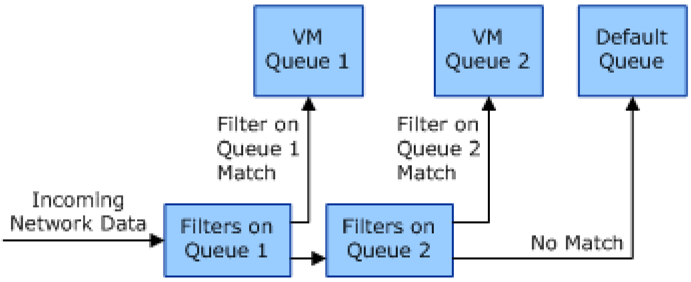
Multiple filters can be set on a receive queue. If any of the filters on a queue match (that is, all of the tests for that filter were passed), the network adapter assigns the packet to that receive queue.

This following figure shows how filter tests are performed and how filters determine a queue assignment.


Filters on a Queue
=A
=2
=B
=3
AND
AND
OR
Send the packet to the queue if the result of the filters is TRUE.
DA
VLAN
Network Data Packet

In the preceding figure, the destination address (DA) is tested (compared to A and B). Also, the VLAN identifier is tested (compared to 2 and 3). The AND operation illustrates that both the DA and VLAN identifier must be equal to the specified values to have a filter match. The OR operation illustrates that any filter on the queue that matches results in the assignment of the network data packet to that queue.

This following figure shows how filters and queues affect the receive data flow.



If an incoming packet matches a filter on a queue, it is assigned to that queue. Otherwise, the packet is tested against the filters on the next queue and so on. If there is no filter match on any of the queues, the network adapter assigns the packet to the default queue.

Security Issues with NDIS Virtual Machine (VM) Shared Memory

Article • 03/14/2023

This topic discusses the potential security issues involved with allocating shared memory from a virtual machine (VM) for virtual machine queue (VMQ) receive buffers. The topic includes the following sections:

- Overview of the Security Issues with VM Shared Memory
- How Windows Server 2008 R2 Addresses the Security Issue
- How Windows Server 2012 and Later Versions Address the Security Issue

Note In Hyper-V, a child partition is also known as a VM.

Overview of the Security Issues with VM Shared Memory

VMs are not trusted software entities. That is, a malicious VM must not be able to interfere with other VMs or the management operating system that runs in the Hyper-V parent partition. This section provides background information and requirements to ensure that driver writers understand VMQ security issues and requirements for shared memory. For more information about shared memory, see the Shared Memory Resource Allocation topic in the Writing VMQ Drivers section.

In the virtualized environment, VM shared memory can be viewed or modified by the VM. However, viewing or modifying data that is associated with other VMs is not allowed. VMs are also not allowed to access the management operating address space.

The header portion of the received packets must be protected. A VM is not allowed to affect the behavior of the Hyper-V extensible switch in a network virtual service provider (VSP). Therefore, VLAN (virtual LAN) filtering must happen before the network adapter uses DMA to transfer the data to VM shared memory. Also, the media access control (MAC) address learning of the switch cannot be affected.

If the Hyper-V extensible switch port that is connected to a VM has an associated VLAN identifier, the host computer must ensure that the destination MAC address and the VLAN identifier of the incoming frame match those respective attributes of the port before the host forwards the packet to the VM's virtual network adapter. If the VLAN identifier of the frame does not match the VLAN identifier of the port, the packet is dropped. When the receive buffers for a virtual network adapter are allocated from host memory, the host can check the VLAN identifier and drop the frame if necessary before making the contents of the frame visible to the target VM. If the frame is not copied to a VM's address space, it cannot be accessed by that VM.

However, when VMQ is configured to use shared memory, the network adapter uses DMA to transfer incoming frames directly to the VM address space. This transfer introduces a security issue in which a VM can examine the contents of the received frames without waiting for the extensible switch to apply the required VLAN filtering.

How Windows Server 2008 R2 Addresses the Security Issue

In Windows Server 2008 R2, before the VSP configures a VM queue to use shared memory that is allocated from the VM address space, it uses the following filtering test for the queue.

syntax

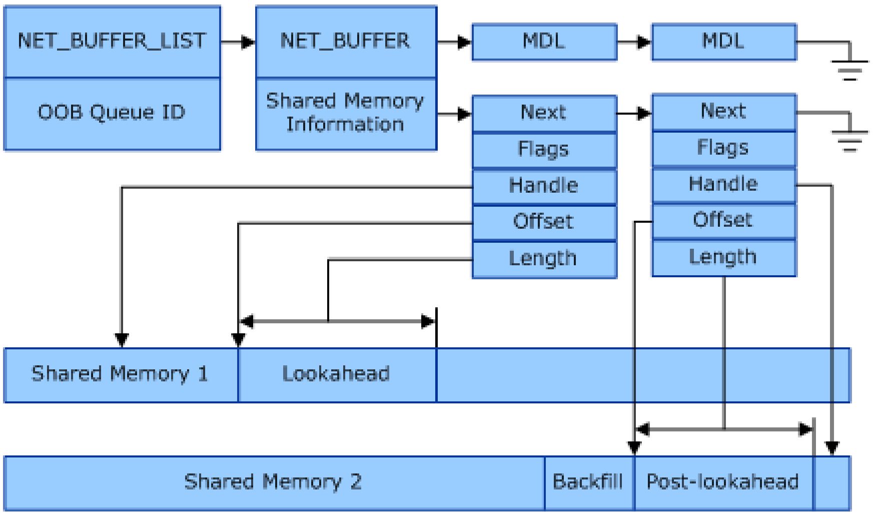
```
(MAC address == x) && (VLAN identifier == n)
```

If the network adapter hardware can support this test before the DMA transfer to the receive buffers, the network adapter can either drop frames with invalid VLAN identifiers or send them to the default queue so that they can be filtered out by the extensible switch. If the miniport driver succeeds in a request to set a filter with this test on a queue, the extensible switch can use VM shared memory for that queue. However, if the network adapter hardware is not capable of filtering the frames based on both destination MAC address and VLAN identifier, the extensible switch uses host shared memory for that queue.

The extensible switch inspects the source MAC address of received frames to configure the routing information for transmit frames—that is, it is similar to a physical learning switch. It is possible to install firewall filter drivers in the host stack; for example, above the miniport driver for the network adapter hardware and below the extensible switch driver. Firewall filter drivers can access data in a received frame before the extensible switch. If the entire receive buffer for each frame is allocated from VM address space, a malicious VM could access parts of the frame that would be examined by either a filter driver or the extensible switch that runs in the host.

To address this security issue, when using VM shared memory for a VM queue, the network adapter must split the packet at a byte offset that is at least the lookahead size, which is a predetermined fixed value. Any lookahead data— meaning data that is ahead of the byte offset for the lookahead size—must be transferred with DMA to shared memory that was allocated for lookahead data. The post-lookahead data—the rest of the frame payload—should be transferred with DMA to shared memory that was allocated for the post-lookahead data.

The following illustration shows the relationships for the network data structures when the incoming data is split into lookahead and post-lookahead shared memory buffers.



The summary requirements for VMQ shared memory are as follows:

- A network adapter can split a received frame at a network-header boundary that is larger than the lookahead size. However, when requested by NDIS, and without exception, all of the frames that are received and assigned to a VMQ must be split at or beyond the lookahead size boundary that NDIS requests.

- The lookahead data must be transferred with DMA to shared memory that is allocated by the miniport driver. The miniport driver must specify in the allocation call that the memory will be used for lookahead data.

- The post-lookahead data must be transferred with DMA to shared memory that is allocated by the miniport driver. The miniport driver must specify in the allocation call that the memory will be used for post-lookahead data.

- Miniport drivers must not be dependent upon which address space NDIS will use to complete the shared memory allocation request. That is, the shared memory address space for lookahead or post-lookahead data is implementation specific. In many cases, NDIS or the extensible switch might satisfy all the requests, including those for post-lookahead use, from host memory address space.

- The order in which frames are received on a VMQ receive queue must be preserved when the frames in that queue are indicated up the driver stack.

- The network adapter must allocate enough backfill memory space in each post-lookahead buffer. This allocation allows the lookahead data to be copied to the backfill portion of the post-lookahead buffer, and allows the frame to be delivered to the VM in a contiguous buffer.

If there is no mechanism in hardware to meet these requirements for VMQ shared memory, the hardware that supports scatter-gather DMA on the receive side might achieve the same results by allocating two receive buffers for each received frame. In this case, the size of the first buffer is limited to the requested lookahead size.

If the network adapter cannot meet these requirements for VMQ shared memory by any method, the VSP will allocate memory for the VMQ receive buffers from the host address space and will copy the received packets from the network adapter receive buffers to VM address space.

How Windows Server 2012 and Later Versions Address the Security Issue

Starting with Windows Server 2012, the VSP does not allocate shared memory from the VM for the VMQ receive buffers. Instead, the VSP allocates memory for the VMQ receive buffers from the host address space and then copies the received packets from the network adapter receive buffers to VM address space.

The following points apply to VMQ miniport drivers that run on Windows Server 2012 and later versions of Windows:

- For NDIS 6.20 VMQ miniport drivers, no change is required. However, when the VSP allocates a VM queue by issuing an OID (object identifier) method request of OID_RECEIVE_FILTER_ALLOCATE_QUEUE, it will set the **LookaheadSize** member of the **NDIS_RECEIVE_QUEUE_PARAMETERS** structure to zero. This will force a miniport driver to not split the packet into pre-lookahead and post-lookahead buffers.

- Starting with NDIS 6.30, VMQ miniport drivers must not advertise support for splitting packet data into pre-lookahead and post-lookahead buffers. When a miniport driver registers its VMQ capabilities, it must follow these rules when it initializes the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure:

  - The miniport driver must not set the **NDIS_RECEIVE_FILTER_LOOKAHEAD_SPLIT_SUPPORTED** flag in the **Flags** member.

  - The miniport driver must set the **MinLookaheadSplitSize** and **MaxLookaheadSplitSize** members to zero.

For more information about how to register VMQ capabilities, see Determining the VMQ Capabilities of a Network Adapter.

NDIS VMQ Live Migration Support

Article • 03/14/2023

To support live migration, a virtual machine (VM) can be paused at any instruction or pending I/O boundary. That is, the VM might not finish pending receive requests. So, the network virtual service provider (VSP) returns all of the received packets to the underlying network adapter that the VM did not return.

Note In Hyper-V, a child partition is also known as a VM.

When the VM is restarted on another host, the network VSP on the new host handles the receive packets that the resumed VM returns and does not pass them down to the new underlying in miniport driver. After the migration is complete, the receive queue that was associated with the VM is freed and it can be reused for another VM.

Note The new network adapter might not support VMQ.

When NDIS requests a miniport driver to free a VMQ receive queue, it follows these steps:

1. The network adapter stops the DMA transfer of data to receive buffers that are associated with the receive queue, after which the queue must enter the DMA Stopped state. The network adapter probably stopped the DMA activity when it received the OID_RECEIVE_FILTER_CLEAR_FILTER OID request to clear the last set filter on the receive queue.

2. The miniport driver generates an **NDIS_STATUS_RECEIVE_QUEUE_STATE** status indication with the **QueueState** member of the **NDIS_RECEIVE_QUEUE_STATE** structure set to **NdisReceiveQueueOperationalStateDmaStopped** to notify NDIS that the DMA transfer has been stopped.

3. The miniport driver waits for all the indicated receive packets for that queue to be returned to the miniport driver.

4. The miniport driver frees all the shared memory that it allocated for the network adapter's receive buffers that are associated with the queue by calling **NdisFreeSharedMemory**.

5. The miniport driver completes the OID_RECEIVE_FILTER_FREE_QUEUE OID request to free the receive queue.

For more information about queue states, see NDIS VM Queue States.

NDIS Virtual Machine Queue States

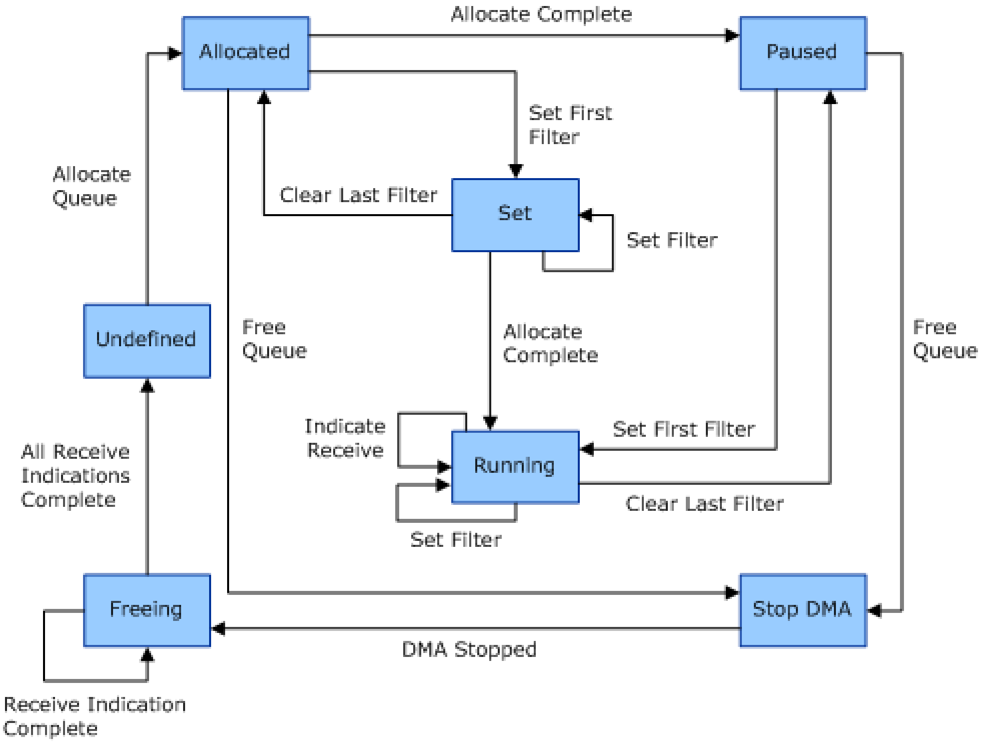
Article • 03/14/2023

This topic provides an overview of the operational states of NDIS virtual machine queues (VMQs). For more information about queue states, see the Queue States and Operations topic in the Writing VMQ Drivers section.

For each queue, a network adapter must support the following set of operational states:

- Undefined
- Allocated
- Set
- Paused
- Running
- Stop DMA
- Freeing

The following figure shows the relationships between these states.



The following defines the adapter states:

Undefined

Undefined is the initial state of a queue. In this state, the queue is not allocated. A queue (except the default queue which always exists) is undefined until the miniport driver receives a queue allocation request. Also, it is undefined after the free operation is complete and any receive indications that had been started are complete.

Allocated

A queue is in the *Allocated* state after an allocation request and before there are any filters set on the queue. The filter can also enter the Allocated state if the queue is in the Set state and the last filter is cleared on the queue. The queue enters the Paused state if the miniport driver receives an allocation complete request while the miniport driver is in the Allocated state. The queue enters the Stop DMA state if the miniport driver receives a free queue request.

Set

In the *Set* state, a queue is allocated and a least one filter is set on the queue but the miniport driver did not receive an allocation complete OID yet. The queue enters the Running state if it receives an allocation complete request. The queue enters the Allocated state if the last filter on the queue is cleared. Note that the queue cannot be freed while there are filters set on the queue.

Paused

In the *Paused* state, the queue is allocated but its miniport driver does not indicate received packets because there are no filters set on the queue. The miniport driver can enter the Paused state either from the Allocated state or from the Running state. The queue enters the Running state when it receives a filter set request. The queue enters the Stop DMA state when it receives a free queue request.

Running

In the *Running* state, a queue has filters set, the queue allocation is complete, and the network adapter indicates receive packets. The queue enters the Paused state if the last filter on the queue is cleared. Note that the queue cannot be freed while there are filters set on the queue. Also, the miniport driver can stop the DMA if the last filter is cleared. However, the miniport driver should not send the DMA stopped state indication in this case.

Stop DMA

In the *Stop DMA* state, the miniport driver received a free queue request and the DMA activity must be stopped. The miniport driver must send a DMA stopped state indication. After the miniport driver sends the status indication, the queue is in the Freeing state. Note that the DMA was probably already stopped when the last filter was cleared. However, the miniport driver should only send the status indication when it receives the free queue request.

Freeing
In the *Freeing* state, a miniport driver is waiting for all of the outstanding receive indications on a queue to complete and freeing the resources that are associated with the queue. After all of the resources are freed, the queue enters the Undefined state.

Getting Started Writing VMQ Drivers

Article • 12/15/2021

This section provides information about writing NDIS virtual machine queue (VMQ) drivers. You should already understand the Virtual Machine Queue Architecture before you read this section.

Note Be sure to study the NDIS Virtual Miniport Driver sample, especially the vmq.c and vmq.h source files.

A miniport driver that supports VMQ manages NICs that provide the VMQ hardware support. Such a NIC provides hardware services to filter incoming network data, and assign it to VM receive queues.

This section describes how overlying drivers obtain information about the underlying network adapter and set the VMQ configuration. Overlying drivers and user applications can obtain the current configuration and enable or disable VMQ features.

This section includes the following topics:

- VMQ Driver Configuration
- Queue States and Operations
- VMQ Interrupt Requirements
- Allocating and Freeing VM Queues
- Setting and Clearing VMQ Filters
- Obtaining and Updating VM Queue Parameters
- VMQ Send and Receive Operations
- Obtaining VMQ Information

VMQ Driver Configuration

Article • 12/15/2021

This section describes the features that NDIS provides to determine the virtual machine queue (VMQ) capabilities of network adapters.

A miniport driver reports its VMQ capabilities to NDIS and overlying drivers during network adapter initialization.

Standardized INF file entries can enable or disable VMQ features when a driver is installed.

This section includes the following topics:

Determining the VMQ Capabilities of a Network Adapter

Standardized INF Keywords for VMQ

Determining the VMQ Capabilities of a Network Adapter

Article • 12/15/2021

NDIS provides the interface to determine the VMQ capabilities of a network adapter, such as:

- The generic filtering capabilities of a network adapter.
- Supported VM queue capabilities.
- Lookahead support to allow splitting of the networking data memory into two separate buffers.

  Note Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported.

Miniport drivers provide the following information to NDIS during network adapter initialization:

- The VMQ hardware capabilities that the network adapter can support.
- The VMQ capabilities that are currently enabled.
- The global receive filtering features that are enabled or disabled on a network adapter.

Overlying drivers and applications can use the following OID query requests to obtain the network adapter capabilities.

OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES

OID_RECEIVE_FILTER_CURRENT_CAPABILITIES

OID_RECEIVE_FILTER_GLOBAL_PARAMETERS

NDIS handles these OID query requests for miniport drivers. Therefore, the query is not requested for miniport drivers. NDIS reports the currently enabled receive VMQ capabilities of a network adapter during initialization. Therefore, overlying drivers do not have to query these OIDs.

The **NDIS_RECEIVE_FILTER_CAPABILITIES** structure specifies the filtering capabilities of a network adapter. This structure is used in the following ways:

- When NDIS calls the *MiniportInitializeEx* function, the miniport driver registers its filtering capabilities by initializing an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure. The driver then sets the **HardwareReceiveFilterCapabilities** member of the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure to point to the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure. The driver next calls the **NdisMSetMiniportAttributes** function and then sets the *MiniportAttributes* parameter to a pointer to the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

- An overlying protocol driver receives the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure in the **NDIS_BIND_PARAMETERS** structure when NDIS calls the driver's *ProtocolBindAdapterEx* function.

- An overlying filter driver receives the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure in the **NDIS_FILTER_ATTACH_PARAMETERS** structure when NDIS calls the driver's *FilterAttach* function.

- Overlying drivers receive the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure by issuing an OID query request of OID_RECEIVE_FILTER_CURRENT_CAPABILITIES or OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES. The **HardwareReceiveFilterCapabilities** and **CurrentReceiveFilterCapabilities** members point to an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure.

The **NDIS_RECEIVE_FILTER_CAPABILITIES** structure includes the following information:

EnabledFilterTypes
The types of the supported receive filters. The NDIS_RECEIVE_FILTER_VMQ_FILTERS_ENABLED flag specifies that virtual machine queue (VMQ) filters are enabled.

EnabledQueueTypes
The types of supported receive queues. The NDIS_RECEIVE_FILTER_VM_QUEUES_ENABLED flag specifies that virtual machine (VM) queues are enabled.

NumQueues
The number of receive queues that the network adapter supports. To support VMQ, this number must be equal to or less than the number of unicast MAC addresses that the NIC supports. This number must not include the default queue.

Note The number of unicast MAC addresses or VM queues that a network adapter supports does not include the MAC address of the associated NIC.

SupportedQueueProperties

The queue properties that the network adapter supports. The NDIS_RECEIVE_FILTER_VM_QUEUE_SUPPORTED flag specifies that the network adapter provides the minimum requirements to support VMQ filtering. A VMQ-capable NIC must provide an MSI-X table entry for each receive queue. Therefore, a VMQ miniport driver must set the NDIS_RECEIVE_FILTER_MSI_X_SUPPORTED flag.

SupportedFilterTests

The filter test operations that a miniport driver supports. For example, the network adapter supports testing the selected header field to determine whether it is equal to a given value. A VMQ miniport driver must set the NDIS_RECEIVE_FILTER_TEST_HEADER_FIELD_EQUAL_SUPPORTED flag.

SupportedHeaders

The types of network packet headers that a miniport driver can inspect. For example, the network adapter can inspect the MAC header of a network packet. The MAC header includes the packet type, destination and source MAC addresses, the VLAN identifier, and the priority tag fields. A VMQ miniport driver must set the NDIS_RECEIVE_FILTER_MAC_HEADER_SUPPORTED flag.

SupportedMacHeaderFields

The types of MAC header fields that a miniport driver can inspect. A VMQ miniport driver must set the NDIS_RECEIVE_FILTER_MAC_HEADER_DEST_ADDR_SUPPORTED flag.

MaxMacHeaderFilters

The maximum number of MAC header filters that the miniport driver supports. There should be at least as many header filters as there are VM queues.

MaxQueueGroups

This member is reserved for NDIS.

MaxQueuesPerQueueGroup

This member is reserved for NDIS.

MinLookaheadSplitSize

The minimum size, in bytes, that the network adapter supports for lookahead packet segments.

Note Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported. Miniport drivers that support NDIS 6.30 or later versions must set this member to zero.

MaxLookaheadSplitSize

The maximum size, in bytes, that the network adapter supports for lookahead packet segments.

Note Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported. Miniport drivers that support NDIS 6.30 or later versions must set this member to zero.

After a successful return from the OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES OID query, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an NDIS_RECEIVE_FILTER_CAPABILITIES structure. These capabilities can include VMQ hardware capabilities that are currently disabled by INF file settings or through the **Advanced** properties page. For more information about VMQ INF files settings, see VMQ Standard INF Entries.

NDIS miniport drivers supply the receive-filtering hardware capabilities during initialization in the **HardwareReceiveFilterCapabilities** member of the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

After a successful return from the OID_RECEIVE_FILTER_CURRENT_CAPABILITIES OID query, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure. These capabilities include the currently enabled VMQ capabilities.

NDIS miniport drivers supply the currently enabled receive filtering capabilities during initialization in the **CurrentReceiveFilterCapabilities** member of the **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

NDIS reports the currently enabled receive filtering capabilities of an underlying network adapter to overlying protocol drivers in the **ReceiveFilterCapabilities** member of the **NDIS_BIND_PARAMETERS** structure during the bind operation.

The **NDIS_RECEIVE_FILTER_GLOBAL_PARAMETERS** structure is used in the OID_RECEIVE_FILTER_GLOBAL_PARAMETERS query OID to obtain the current global receive filter settings.

NDIS_RECEIVE_FILTER_GLOBAL_PARAMETERS includes the following information:

EnabledFilterTypes
The types of enabled receive filters. The NDIS_RECEIVE_FILTER_VMQ_FILTERS_ENABLED flag specifies that virtual machine queue (VMQ) filters are enabled.

EnabledQueueTypes
The types of enabled receive queues. The NDIS_RECEIVE_FILTER_VM_QUEUES_ENABLED flag specifies that virtual machine (VM) queues are enabled.

After a successful return from the OID_RECEIVE_FILTER_GLOBAL_PARAMETERS OID query, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_GLOBAL_PARAMETERS** structure. The NDIS_RECEIVE_FILTER_GLOBAL_PARAMETERS structure specifies the receive-filtering features that are enabled or disabled on a network adapter.

NDIS protocol drivers use OID_RECEIVE_FILTER_GLOBAL_PARAMETERS to query the current global configuration parameters for receive filtering on a network adapter. For example, protocol drivers can use this OID to determine whether types of receive filters or receive queues are enabled or disabled.

Standardized INF Keywords for VMQ

Article • 12/15/2021

The following standardized INF keywords are defined to enable or disable support for the virtual machine queue (VMQ) features of network adapters.

***VMQ**
A value that describes whether the device has enabled or disabled the VMQ feature.

***VMQLookaheadSplit**
A value that describes whether the device has enabled or disabled the ability to split receive buffers into lookahead and post-lookahead buffers. The miniport driver reports this capability with the NDIS_RECEIVE_FILTER_LOOKAHEAD_SPLIT_SUPPORTED flag in the **SupportedQueueProperties** member of the NDIS_RECEIVE_FILTER_CAPABILITIES structure. For more information about this feature, see Shared Memory in Receive Buffers.

Note Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported. Starting with Windows Server 2012, this INF keyword is obsolete.

***VMQVlanFiltering**
A value that describes whether the device has enabled or disabled the ability to filter network packets by using the VLAN identifier in the media access control (MAC) header. The miniport driver reports this capability with the NDIS_RECEIVE_FILTER_MAC_HEADER_VLAN_ID_SUPPORTED flag in **SupportedMacHeaderFields** member of the NDIS_RECEIVE_FILTER_CAPABILITIES structure.

***RssOrVmqPreference**
A value that defines whether VMQ capabilities should be enabled instead of receive side scaling (RSS) capabilities.

This is a hidden keyword value that must not be specified in the INF file and is not displayed in **Advanced** property page for the network adapter. For more information, see Handling VMQ and RSS INF Keywords.

VMQ standardized INF keywords are enumeration keywords. The following table describes the possible INF entries for the VMQ standardized INF keywords.

SubkeyName	ParamDesc	Value	EnumDesc
***VMQ**	Virtual Machine Queues	0	Disabled

SubkeyName	ParamDesc	Value	EnumDesc
		1 (Default)	Enabled
***VMQLookaheadSplit**	VMQ Lookahead Split	0	Disabled **Note** Starting with NDIS 6.30, this keyword is no longer supported.
		1 (Default)	Enabled
***VMQVlanFiltering**	VMQ VLAN Filtering	0	Disabled
		1 (Default)	Enabled
***RssOrVmqPreference**	Note: The ParamDesc and EnumDesc entries for this subkey cannot be used in either INF files or a user interface. For more information, see Handling VMQ and RSS INF Keywords.	0 (Default)	**Note** Report RSS capabilities
		1	**Note** Report VMQ capabilities

The columns in this table describe the following attributes for an enumeration keyword:

SubkeyName
The name of the keyword that you must specify in the INF file. This name also appears in the registry under the **NDI\params** key for the network adapter.

ParamDesc
The display text that is associated with the SubkeyName INF entry.

Note The independent hardware vendor (IHV) can define any descriptive text for the SubkeyName.

Value
The enumeration integer value that is associated with each SubkeyName in the list.

EnumDesc
The display text that is associated with each value that appears in the **Advanced**

property page.

For more information about standardized INF keywords, see Standardized INF Keywords for Network Devices.

Handling VMQ and RSS INF Keywords

Network adapters that support VMQ and receive side scaling (RSS) cannot use these features simultaneously. The operating system enables the use of the RSS or VMQ features in the following way:

- When the network adapter is bound to the TCP/IP stack, the operating enables the use of the RSS feature.
- When the network adapter is bound to the Hyper-V extensible switch driver stack, the operating system enables the use of the VMQ feature.

  For more information, see Hyper-V Extensible Switch.

Because the network adapter is not disabled and then re-enabled when it is unbound from the TCP/IP stack and bound to the Hyper-V driver stack (or the reverse), it is not possible for such network adapters to switch between VMQ and RSS automatically.

When NDIS calls the *MiniportInitializeEx* function, the miniport driver follows these steps before it reports its currently-enabled VMQ or RSS capabilities to NDIS:

1. The miniport driver reads the ***RssOrVmqPreference** keyword before reporting its currently-enabled capabilities to NDIS.

   If the value of the ***RssOrVmqPreference** keyword is 1, the miniport driver is configured for VMQ preference.

   If the value of the ***RssOrVmqPreference** keyword is zero or the keyword is not present, the miniport driver is configured for RSS preference.

2. If the miniport driver is configured for VMQ preference, it must read the ***VMQ** keyword to determine if VMQ is enabled on the network adapter. If the keyword is set to 1, the driver reports the currently-enabled VMQ settings. For more information on how the miniport driver reports VMQ settings, see Determining the VMQ Capabilities of a Network Adapter.

   For more information about the VMQ keywords, see Standardized INF Keywords for VMQ.

Note If the miniport driver is configured for VMQ preference, it must not read any of the RSS standardized keywords.

3. If the miniport driver is configured for RSS preference, it must read the ***RSS** keyword to determine if RSS is enabled on the network adapter. If the keyword is set to 1, the driver reports the currently-enabled RSS settings. For more information on how the miniport driver reports RSS settings, see RSS Configuration.

   For more information about the RSS keywords, see Standardized INF Keywords for RSS.

   Note If the miniport driver is configured for RSS preference, it must not read any of the VMQ standardized keywords.

The following table describes how the miniport driver determines RSS or VMQ preference and advertises capabilities based on registry keywords:

*RssOrVmqPreference	*VMQ	*RSS	VMQ or RSS capabilities advertised
1	1	N/A	VMQ
1	0	N/A	None
0, or not present in registry	N/A	1	RSS
0, or not present in registry	N/A	0	None

Note The miniport driver must always report the complete RSS and VMQ hardware capabilities regardless of the values of these keywords. These keyword settings only affect how the driver reports the currently-enabled RSS and VMQ capabilities.

Reserved Registry Keywords

If the miniport driver supports VMQ and the VMQ interface is enabled on the network adapter, the driver must not read the following RSS INF entries:

SubkeyName	ParamDesc	Value
*RssMaxProcNumber	The maximum processor number of the RSS interface.	0 through (MAXIMUM_PROC_PER_GROUP-1),
*MaxRssProcessors	The maximum number of RSS processors.	1 through MAXIMUM_PROC_PER_SYSTEM.

The miniport driver that supports VMQ must not read the following subkeys under the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\VMSMP\Parameters** registry key.

SubkeyName	ParamDesc	Value
***TenGigVmqEnabled**	Enable or disable VMQ on all 10 gigabits per second (Gbps) network adapters.	0=System default (disabled for Windows Server 2008 R2).
		1=Enabled.
		2=Explicitly disabled.
***BelowTenGigVmqEnabled**	Enable or disable VMQ on all network adapters that support less than 10 Gbps.	0=System default (disabled for Windows Server 2008 R2).
		1=Enabled.
		2=Explicitly disabled.
***RssMaxProcNumber**	The maximum processor number of the RSS interface.	0 through (MAXIMUM_PROC_PER_GROUP-1),
***MaxRssProcessors**	The maximum number of RSS processors.	1 through MAXIMUM_PROC_PER_SYSTEM.

Queue States and Operations

Article • 12/15/2021

For each queue, a network adapter must support the following set of operational states:

Undefined
The queue is not allocated. To allocate a queue, an overlying driver sends an OID_RECEIVE_FILTER_ALLOCATE_QUEUE OID request.

Allocated
The *Allocated* state is the initial state for a queue. When a queue is in the Allocated state, the overlying driver usually sets filters on the queue with the OID_RECEIVE_FILTER_SET_FILTER OID or completes the queue allocation with the OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE OID request.

Set
In the *Set* state, a queue has at least one filter allocated but the overlying driver has not sent the OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE OID.

Running
In the *Running* state, the queue has filters set, the queue allocation is complete and the miniport adapter is indicating receive packets for the queue.

Paused
In the *Paused* state, the network adapter does not indicate received network data for the queue. Either there were no filters set on the queue before the OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE OID request or all of the filters that were set on the queue were cleared with the OID_RECEIVE_FILTER_CLEAR_FILTER OID request.

DMA Stopped
In the *DMA Stopped* state, a miniport driver received an OID_RECEIVE_FILTER_FREE_QUEUE OID request. When the DMA is stopped and the driver has issued the DMA-stopped status indication (with **NDIS_STATUS_RECEIVE_QUEUE_STATE**), the driver enters the Freeing state.

Freeing
In the *Freeing* state, a miniport driver completes the operations that are required to stop send and receive operations for the queue, and releases the associated resources. After all of the outstanding receive indications are complete, the queue is deleted and the queue is Undefined.

In the following table, the headings are the queue states. Major events are listed in the first column. The rest of the entries in the table specify the next state that the queue enters after an event occurs within a state. The blank entries represent invalid event/state combinations.

Event \ State	Undefined	Allocated	Set	Running	Paused	Stop DMA	Freeing
OID_RECEIVE_FILTER_ALLOCATE_QUEUE - method (set)	Allocated						
OID_RECEIVE_FILTER_QUEUE_PARAMETERS - method (query) request		Allocated	Set	Running	Paused		
OID_RECEIVE_FILTER_QUEUE_PARAMETERS - set request		Allocated	Set	Running	Paused		
OID_RECEIVE_FILTER_SET_FILTER - method (set) request		Set	Set	Running	Running		
OID_RECEIVE_FILTER_CLEAR_FILTER - set request (last filter)			Allocated	Paused			

Event \ State	**Undefined**	**Allocated**	**Set**	**Running**	**Paused**	**Stop DMA**	**Freeing**
OID_RECEIVE_FILTER_CLEAR_FILTER - set request (not last filter)			Set	Running			
OID_RECEIVE_FILTER_ENUM_FILTERS - method (query request)		Allocated	Set	Running	Paused		
OID_RECEIVE_FILTER_PARAMETERS - method (query) request			Set	Running			
OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE - method (set) request		Paused	Running				
Receive packet				Running			
OID_RECEIVE_FILTER_FREE_QUEUE set request		Stop DMA			Stop DMA		
DMA is stopped and NDIS_STATUS_RECEIVE_QUEUE_STATE status indication was sent (Note: DMA was probably already stopped in Allocated or Paused state)						Freeing	
All receive indications are complete and the queue resources are freed							Undefined

Note The events listed in the preceding table include some secondary events that do not result in a state change. These secondary events are included in the table to specify the states where these events are valid.

The primary queue events are defined as follows:

OID_RECEIVE_FILTER_ALLOCATE_QUEUE - method (set) request
An overlying driver allocated a queue. For more information about allocating queues, see Allocating and Freeing VM Queues.

OID_RECEIVE_FILTER_SET_FILTER - method (set) request
An overlying driver set a filter on a queue. If the overlying driver has not sent the OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE OID, the queue is in the Set state. Otherwise, the queue is in the Running state. For more information about setting filters on queues, see Setting and Clearing VMQ Filters.

OID_RECEIVE_FILTER_CLEAR_FILTER - set request
An overlying driver cleared a filter on a queue. If the last filter was cleared on a running queue, the DMA can be stopped; receive indications are stopped and the queue should be cleared of received data, if any. For more information about clearing filters on queues, see Setting and Clearing VMQ Filters.

OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE - method (set) request
An overlying driver completed the queue allocation. If there are filters set on the queue, it is in the Running state and receive indications can start. For more information about completing queue allocation, see Allocating and Freeing VM Queues.

Receive packet
A miniport driver can only indicate receive packets for a queue that is in the Running state. For more information about indicating received data for queues, see VMQ Send and Receive Operations.

OID_RECEIVE_FILTER_FREE_QUEUE set request.
An overlying driver freed a queue. The driver issues the DMA-stopped status indication (with **NDIS_STATUS_RECEIVE_QUEUE_STATE**), the driver enters the Freeing state. When all of the outstanding receive indications are complete and the queue resources are freed, the queue is undefined.

VMQ Interrupt Requirements

Article • 12/15/2021

A miniport driver that supports the virtual machine queue (VMQ) functionality must also support the following interrupt allocation requirements:

- The miniport driver must support MSI-X. The driver must set the **NDIS_RECEIVE_FILTER_MSI_X_SUPPORTED** flag in the **SupportedQueueProperties** member of the NDIS_RECEIVE_FILTER_CAPABILITIES structure.

  The driver returns this structure in the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure that the driver uses in its call to the NdisMSetMiniportAttributes function.

- The miniport driver must call the NdisGetRssProcessorInformation function to obtain processor information for allocating interrupt vectors. It must not rely on registry keys or information obtained from other sources for interrupt allocation.

  NdisGetRssProcessorInformation returns information about the set of processors that a miniport driver can use for RSS and VMQ. This information is contained in an NDIS_RSS_PROCESSOR_INFO structure.

- The miniport driver should allocate only one interrupt vector for each processor that is specified in the NDIS_RSS_PROCESSOR_INFO structure.

  The miniport driver should allocate no more than two interrupt vectors for other events that are not related to send or receive packet operations. For example, the driver could allocate an IDT for link status events.

- The miniport driver must support the minimum number of MSI-X interrupt vectors as defined in the following table:

Number of queues	Minimum number of required MSI-X interrupt vectors
1–16	1–16
17–64	16–32
65 or more	32 or more

Allocating and Freeing VM Queues

Article • 12/15/2021

This section includes the following topics:

Allocating a VM Queue

Article • 12/15/2021

To allocate a queue with an initial set of configuration parameters, an overlying driver issues an OID_RECEIVE_FILTER_ALLOCATE_QUEUE method OID request. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure. After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure that has a new queue identifier and an MSI-X table entry.

The **NDIS_RECEIVE_QUEUE_PARAMETERS** structure is used in the OID_RECEIVE_FILTER_ALLOCATE_QUEUE OID and the OID_RECEIVE_FILTER_QUEUE_PARAMETERS OID. For more information about VM queue parameters, see Obtaining and Updating VM Queue Parameters.

The overlying driver initializes the **NDIS_RECEIVE_QUEUE_PARAMETERS** structure with the following queue configuration parameters:

- The queue type (**NdisReceiveQueueTypeVMQueue** from the **NDIS_RECEIVE_QUEUE_TYPE** enumeration.)

- The processor affinity for the queue.

- The queue name and the virtual machine name.

- The lookahead-split parameters.

  Note Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported.

Note The overlying driver can set the NDIS_RECEIVE_QUEUE_PARAMETERS_PER_QUEUE_RECEIVE_INDICATION and NDIS_RECEIVE_QUEUE_PARAMETERS_LOOKAHEAD_SPLIT_REQUIRED flags in the **Flags** member of the **NDIS_RECEIVE_QUEUE_PARAMETERS** structure. The other flags are not used for queue allocation.

When NDIS receives an OID request to allocate a receive queue, it verifies the queue parameters. After NDIS allocates the necessary resources and the queue identifier, it submits the OID request to the underlying miniport driver. The queue identifier is unique to the associated network adapter.

If the miniport driver can successfully allocate the necessary software and hardware resources for the receive queue, it completes the OID request with a success status.

Before NDIS sends the OID request to the miniport driver, NDIS assigns a queue identifier in the **QueueId** member of the **NDIS_RECEIVE_QUEUE_PARAMETERS** structure and passes the method request to the miniport driver. The miniport driver provides the MSI-X table entry in the **MSIXTableEntry** member.

The miniport driver must retain the queue identifiers for the allocated receive queues. NDIS uses the queue identifier of a receive queue for subsequent calls to the miniport driver to set a receive filter on the receive queue, change the receive queue parameters, or free the receive queue.

Note The default queue (queue identifier zero) is always allocated and cannot be freed.

The overlying driver must use the queue identifier that NDIS provides in subsequent OID requests, for example, to modify the queue parameters or free the queue. The queue identifier is also included in the OOB data on all **NET_BUFFER_LIST** structures that are associated with the queue. Drivers use the **NET_BUFFER_LIST_RECEIVE_QUEUE_ID** macro to retrieve the queue identifier in a NET_BUFFER_LIST structure.

Note A protocol driver can set VMQ filters at any time after it successfully allocates a queue and before the queue is deleted.

The protocol driver issues an OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE method OID request to complete the queue allocation. The miniport driver can allocate shared memory and other resources when the allocation is complete. For more information about allocating shared memory resources, see Shared Memory Resource Allocation.

After a miniport driver receives an OID_RECEIVE_FILTER_QUEUE_ALLOCATION OID request and handles it successfully, the queue is in the *Allocated* state. For more information about queue states, see Queue States and Operations.

After an overlying driver allocates one or more receive queues (and optionally sets the initial filters), it must issue OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE set OID requests to notify the miniport driver that the allocation is complete for the current batch of receive queues.

The miniport driver must not retain any packets in a receive queue if there are no filters set on that queue. If a queue never had any filters set or all the filters were cleared, the queue should be empty and any packets should be discarded. That is, they are not indicated up the driver stack or retained in the queue.

Overlying drivers use the OID_RECEIVE_FILTER_FREE_QUEUE OID to free queues that they allocate. For more information about freeing queues, see Freeing a VM Queue.

Freeing a VM Queue

Article • 12/15/2021

To free a receive queue, an overlying driver issues an OID_RECEIVE_FILTER_FREE_QUEUE set OID request. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_QUEUE_FREE_PARAMETERS** structure with a queue identifier of type **NDIS_RECEIVE_QUEUE_ID**.

OID_RECEIVE_FILTER_FREE_QUEUE frees a receive queue that an overlying driver allocated by using the OID_RECEIVE_FILTER_ALLOCATE_QUEUE OID. For more information about allocating a receive queue, see Allocating a VM Queue.

Note The default queue, which has a queue identifier of **NDIS_DEFAULT_RECEIVE_QUEUE_ID**, is always allocated and cannot be freed.

An overlying driver must free all the filters that it sets on a queue before it frees the queue. Also, an overlying driver must free all the receive queues that it allocated on a network adapter before it calls the **NdisCloseAdapterEx** function to close a binding to the network adapter. NDIS frees all the queues that are allocated on a network adapter before it calls the miniport driver's *MiniportHaltEx* function.

When a miniport driver receives a request to free a queue, it does the following:

- Must immediately stop DMA to shared memory resources that are associated with the queue.
- Generates a status indication to indicate that the DMA is stopped.
- Waits for all outstanding **NET_BUFFER_LIST** structures, which are associated with the queue, to be returned.
- Frees the associated shared memory and hardware resources.

When a miniport driver receives an OID_RECEIVE_FILTER_FREE_QUEUE set request, the queue must enter the Stop DMA state, it stops the DMA on a queue and the miniport driver must indicate the status change by using the **NDIS_STATUS_RECEIVE_QUEUE_STATE** status indication. For more information about queue states, see Queue States and Operations.

After the miniport driver issues the **NDIS_STATUS_RECEIVE_QUEUE_STATE** status indication, it must wait for all the pending receive indications to complete before it can free the associated shared memory. For more information about freeing shared memory, see Shared Memory Resource Allocation.

Shared Memory Resource Allocation

Article • 12/15/2021

To allocate shared memory resources for a VM queue, a miniport driver calls the **NdisAllocateSharedMemory** function. For example, the miniport driver allocates shared memory when it receives the OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE OID. Also, a miniport driver can allocate shared memory for the default queue during network adapter initialization. For more information about allocating queues, see Allocating a VM Queue.

The miniport driver can allocate more memory for the queue until the queue is freed. For more information about freeing a queue, see Freeing a VM Queue.

The **NDIS_SHARED_MEMORY_PARAMETERS** structure specifies the shared memory parameters for a shared memory allocation request. Miniport drivers pass this structure to the **NdisAllocateSharedMemory** function. NDIS passes this structure to the **NetAllocateSharedMemory** function (that is, the ALLOCATE_SHARED_MEMORY_HANDLER entry point).

When a miniport driver allocates shared memory, it specifies the following:

- Queue identifier.
- Shared memory length.
- How the shared memory is used. For example, the miniport driver specifies **NdisSharedMemoryUsageReceive** if the shared memory is to be used for receive buffers.

If the NDIS_SHARED_MEM_PARAMETERS_CONTIGOUS flag is not set in the **Flags** member, shared memory can be specified in a scatter-gather list that is contained in non-contiguous memory.

The **NDIS_SHARED_MEMORY_USAGE** enumeration specifies how shared memory is used. The NDIS_SHARED_MEMORY_USAGE enumeration is used in the **NDIS_SHARED_MEMORY_PARAMETERS** and **NDIS_SCATTER_GATHER_LIST_PARAMETERS** structures. For information about shared memory parameters in VMQ receive data buffers, see Shared Memory in Receive Buffers.

The **NdisAllocateSharedMemory** function provides the following to the caller:

- Virtual address of the allocated memory.
- Scatter-gather list.

- Shared memory handle - for receive indications.

- Allocation handle - used to identify the memory later.

Miniport drivers call the **NdisFreeSharedMemory** function to free shared memory for a queue. If the miniport driver allocated the shared memory for a nondefault queue, it frees the shared memory in the context of the OID_RECEIVE_FILTER_FREE_QUEUE OID while it is freeing the queue. Miniport drivers free shared memory that they allocated for the default queue in the context of the *MiniportHaltEx* function.

Enumerating the Allocated Queues

Article • 12/15/2021

To get a list of all the receive queues that are allocated on a network adapter, an overlying driver issues an OID_RECEIVE_FILTER_ENUM_QUEUES query OID request. After a successful return from the OID query request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_QUEUE_INFO_ARRAY** structure that is followed by an **NDIS_RECEIVE_QUEUE_INFO** structure for each queue.

NDIS handles the OID_RECEIVE_FILTER_ENUM_QUEUES query OID request for miniport drivers. NDIS obtained the information from an internal cache of the data that it received from the OID_RECEIVE_FILTER_ALLOCATE_QUEUE and OID_RECEIVE_FILTER_QUEUE_PARAMETERS OID requests.

Overlying drivers and user-mode applications can use the OID_RECEIVE_FILTER_ENUM_QUEUES OID query request to enumerate the receive queues on a network adapter.

If a protocol driver issues the request, the request type in the **NDIS_OID_REQUEST** structure is set to **NdisRequestQueryInformation** and this OID returns an array of all the receive queues that the protocol driver allocated on the network adapter. If a user-mode application issued the request, the request type in the NDIS_OID_REQUEST is set to **NdisRequestQueryStatistics**, and this OID returns an array of information for all the receive queues on the miniport adapter.

Setting and Clearing VMQ Filters

Article • 12/15/2021

Filters can be set and cleared any time after the queue has been allocated. For information about allocating and freeing queues, see Allocating and Freeing VM Queues.

This section includes the following topics:

Setting a VMQ Filter

Clearing a VMQ Filter

Enumerating Filters on a VMQ

Setting a VMQ Filter

Article • 12/15/2021

After a receive queue is allocated, overlying drivers can set filters on the receive queue. Only the driver that allocated a receive queue can set a filter on that queue.

Note Because the default receive queue (**NDIS_DEFAULT_RECEIVE_QUEUE_ID**) always exists, overlying drivers can always set a receive filter on the default queue. Overlying drivers do not own the default queue. Therefore, all protocol drivers that are bound to a network adapter can use the default queue.

Setting a Filter on a Receive Queue

To set a filter on a receive queue with an initial set of configuration parameters, an overlying driver issues an OID_RECEIVE_FILTER_SET_FILTER method object identifier (OID) request. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure with a new filter identifier.

The overlying driver initializes the **NDIS_RECEIVE_FILTER_PARAMETERS** structure with the following filter configuration parameters for the receive queue:

- The filter type that is specified through an **NDIS_RECEIVE_FILTER_TYPE** enumeration value.

  Note Starting with NDIS 6.20, only **NdisReceiveFilterTypeVMQueue** filter types are supported for the virtual machine queue (VMQ) interface.

- The queue identifier.

- One or more field test parameters that are formatted as **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures. For VMQ, the following field test parameters are defined.

  - The destination media access control (MAC) address in the packet equals the specified MAC address.

  - The virtual LAN (VLAN) identifier in the packet equals the specified VLAN identifier.

The **NDIS_RECEIVE_FILTER_PARAMETERS** structure is used with the OID_RECEIVE_FILTER_PARAMETERS OID and the OID_RECEIVE_FILTER_SET_FILTER OID to specify the configuration parameters of a filter.

The **FieldParametersArrayOffset**, **FieldParametersArrayNumElements**, and **FieldParametersArrayElementSize** members of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure define an array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures. Each **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure in the array sets the filter test criterion for one field in a network header.

The **Flags** member of the **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure specifies actions to be performed for the receive filter. The following points apply to the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag:

- If the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is set in the **Flags** member of the **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structure, the network adapter must indicate only received packets that match all of the following test criteria:

    - A packet with a matching MAC address.

    - A packet that has no VLAN tag or has a VLAN identifier of zero.

  If the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is set, the network adapter must not indicate packets that have a matching MAC address and a nonzero VLAN identifier.

  Note If the Hyper-V extensible switch sets the MAC address filter and no VLAN identifier filter is configured in OID_RECEIVE_FILTER_SET_FILTER, the switch also sets the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag.

- If the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag is not set and there is no VLAN identifier filter configured by an OID set request of OID_RECEIVE_FILTER_SET_FILTER, the miniport driver must do one of the following:

    - If the miniport driver supports NDIS 6.20, it must return a failed status for the OID request of OID_RECEIVE_FILTER_SET_FILTER.

    - If the miniport driver supports NDIS 6.30 or later versions of NDIS, it must configure the network adapter to inspect and filter the specified MAC address fields. If a VLAN tag is present in the received packet, the network adapter must remove it from the packet data. The miniport driver must put the VLAN tag in an **NDIS_NET_BUFFER_LIST_8021Q_INFO** that is associated with the packet's **NET_BUFFER_LIST** structure.

- If a protocol driver sets a MAC address filter and a VLAN identifier filter with the OID_RECEIVE_FILTER_SET_FILTER OID, it does not set the **NDIS_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO** flag in either of the filter fields. In this case, the miniport driver should indicate packets that match both the specified MAC address and the VLAN identifier. That is, the miniport driver should not indicate packets with a matching MAC address that have a zero VLAN identifier or are untagged packets.

Using the Filter Identifier

NDIS assigns a filter identifier in the **FilterId** member of the **NDIS_RECEIVE_FILTER_PARAMETERS** structure and passes the OID method request of OID_RECEIVE_FILTER_SET_FILTER to the underlying miniport driver. Each filter that is set on a receive queue has a unique filter identifier for a network adapter. That is, the filter identifiers are not duplicated on different queues that the network adapter manages.

The overlying driver must use the filter identifier that NDIS provides in later OID requests; for example, to modify the filter parameters or to free a filter.

When NDIS receives an OID request to set a filter on a receive queue, it verifies the filter parameters. After NDIS allocates the necessary resources and the filter identifier, it submits the OID request to the underlying network adapter. If the network adapter can successfully allocate the necessary software and hardware resources for the filter, it completes the OID request with **NDIS_STATUS_SUCCESS**.

The miniport driver must retain the filter identifiers for the allocated receive filters. NDIS uses the filter identifier of a filter with later OID requests in order to change the receive filter parameters or clear the receive filter. For more information about how to change parameters and clear filters, see Obtaining and Updating VM Queue Parameters and Clearing a VMQ Filter.

Handling the Filter on a Receive Queue

The miniport driver programs the network adapter based on the filters in the following way:

- All field test parameters for a particular filter must match in order to assign a packet to the queue.

- Multiple filters can be set on a queue.
- Packets must be assigned to the receive queue if any of the filters pass.

The network adapter combines the results from all the field tests with a logical **AND** operation. That is, if any field test that is included in the array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures fails, the network packet does not meet the specified filter criterion.

When a network adapter tests a received packet against these filter criteria, it must ignore all fields in the packet that have no test criteria specified.

Receiving Packets from a Receive Queue

After a miniport driver receives an OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE request and has filters that are set on the queue, the queue is in the *Running* state. While the queue is in this state, the miniport driver can indicate packets on the queue. For more information about queue states, see Queue States and Operations.

If the miniport driver has received an OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE OID request for a queue but there are no filters set on the queue, the miniport driver must not indicate any receive packets on that queue. In this case, when the miniport driver receives an OID_RECEIVE_FILTER_SET_FILTER OID request for the queue, and possibly before it completes the OID request, it can indicate packets on that queue. If the miniport driver indicates packets on a queue while it is processing an OID_RECEIVE_FILTER_SET_FILTER OID request, the miniport driver must complete the OID_RECEIVE_FILTER_SET_FILTER request that has an **NDIS_STATUS_SUCCESS** return code.

Clearing a VMQ Filter

Article • 12/15/2021

To free a filter on a receive queue, an overlying driver issues an OID_RECEIVE_FILTER_CLEAR_FILTER set OID request. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS** structure.

The protocol driver obtained the filter identifier from an earlier OID_RECEIVE_FILTER_SET_FILTER method OID request. For more information about setting filters, see Setting a VMQ Filter.

A protocol driver must clear all the filters that it set on a queue before it frees the queue. A protocol driver must also clear all the filters that it set on the default queue before it closes its binding to the network adapter.

A miniport driver must not indicate packets on a nondefault queue if it has completed the OID_RECEIVE_FILTER_CLEAR_FILTER OID request to clear the last filter on the queue or if it has completed an OID_RECEIVE_FILTER_FREE_QUEUE OID request to free the queue.

Enumerating Filters on a VMQ

Article • 12/15/2021

To obtain a list of all the filters that are set on a receive queue, overlying drivers and applications can use the OID_RECEIVE_FILTER_ENUM_FILTERS method object identifier (OID) request.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure. When it formats the **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure, the overlying driver or application must set the **QueueId** member to the identifier (ID) of the receive queue. The receive queue ID is obtained in the following ways:

- The overlying driver obtained the receive queue ID value from earlier OID method requests of OID_RECEIVE_FILTER_ALLOCATE_QUEUE or OID_RECEIVE_FILTER_ENUM_QUEUES. The driver can also specify **NDIS_DEFAULT_RECEIVE_QUEUE_ID** for the default receive queue.

- An application obtained the receive queue ID value from an earlier OID method request of OID_RECEIVE_FILTER_ENUM_QUEUES. The application can also specify **NDIS_DEFAULT_RECEIVE_QUEUE_ID** for the default receive queue.

After a successful return from the OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an updated **NDIS_RECEIVE_FILTER_INFO_ARRAY** structure that is followed by one or more **NDIS_RECEIVE_FILTER_INFO** structures. Each **NDIS_RECEIVE_FILTER_INFO** structure specifies the ID for a filter that is set on the specified receive queue.

Overlying drivers or applications can use the OID_RECEIVE_FILTER_PARAMETERS OID method request to obtain the parameters of a specific filter on a receive queue.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. The overlying driver or application formats the **NDIS_RECEIVE_FILTER_PARAMETERS** structure by setting the **FilterId** member to the nonzero ID value of the filter whose parameters are to be returned.

Note The overlying driver obtained the filter ID from an earlier OID method request of OID_RECEIVE_FILTER_SET_FILTER or OID_RECEIVE_FILTER_ENUM_FILTERS. The application can obtain the filter ID only from an earlier OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS.

NDIS handles the OID_RECEIVE_FILTER_ENUM_FILTERS and OID_RECEIVE_FILTER_PARAMETERS method OID requests for miniport drivers. NDIS obtained the information from an internal cache of the data that it received from the OID_RECEIVE_FILTER_SET_FILTER OID request.

Obtaining and Updating VM Queue Parameters

Article • 12/15/2021

An overlying driver can set the configuration parameters of a VM queue after it is allocated. Also, an overlying driver or application can obtain the current parameters for a queue and parameters for the filters that are set on a queue.

To change the current configuration parameters of a queue, overlying drivers can use the OID_RECEIVE_FILTER_QUEUE_PARAMETERS set OID request. The overlying driver provides a pointer to an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure in the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure.

The NDIS_RECEIVE_QUEUE_PARAMETERS structure is used in the OID_RECEIVE_FILTER_ALLOCATE_QUEUE OID and the OID_RECEIVE_FILTER_QUEUE_PARAMETERS OID. For more information about allocating queues, see Allocating a VM Queue.

To get the current configuration parameters of a queue, overlying drivers can use the OID_RECEIVE_FILTER_QUEUE_PARAMETERS method OID request. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure with a queue identifier of type NDIS_RECEIVE_QUEUE_ID. After a successful return from the OID method request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure.

NDIS handles the method request for miniport drivers. Therefore, the OID_RECEIVE_FILTER_QUEUE_PARAMETERS method OID request is not requested for miniport drivers. NDIS obtained the information from an internal cache of the data that it received from the OID_RECEIVE_FILTER_ALLOCATE_QUEUE and OID_RECEIVE_FILTER_QUEUE_PARAMETERS OID requests.

To get the current configuration parameters of a filter on a receive queue, overlying drivers can use the OID_RECEIVE_FILTER_PARAMETERS method OID request. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure initially contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. NDIS uses the **FilterId** member in the input structure to identify the filter. After a successful return from the method request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an updated NDIS_RECEIVE_FILTER_PARAMETERS structure.

NDIS handles the OID_RECEIVE_FILTER_PARAMETERS method OID request for miniport drivers. NDIS obtained the information from an internal cache of the data that it received from the OID_RECEIVE_FILTER_SET_FILTER OID request.

Overlying drivers can use the OID_RECEIVE_FILTER_PARAMETERS method OID request to get the configuration parameters for a filter on a receive queue.

The overlying driver obtained the filter identifier from an earlier OID_RECEIVE_FILTER_SET_FILTER method OID request or from the OID_RECEIVE_FILTER_ENUM_FILTERS OID request. Only drivers can use the OID_RECEIVE_FILTER_SET_FILTER request.

An application obtained the filter identifier from the OID_RECEIVE_FILTER_ENUM_FILTERS OID request.

VMQ Send and Receive Operations

Article • 12/15/2021

This section provides information about implementing send and receive operations in NDIS drivers that support VMQ.

To support VMQ send and receive operations, the VMQ interface requires network adapter hardware that supports the VMQ filter operations. These filters determine the assignment of packets to receive queues.

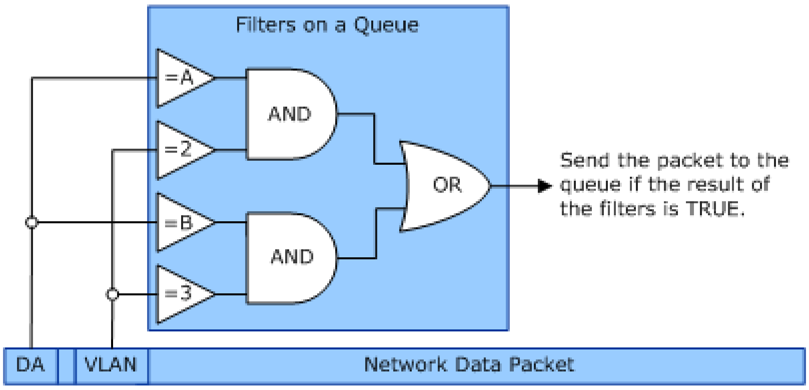
This section includes the following topics:

VMQ Filter Operations

Article • 12/15/2021

Multiple receive filters and can be set on a receive queue. Also, the current VMQ implementation supports filters on the destination media access control (MAC) address of the incoming packets and optional filters that inspect the virtual local area network (VLAN) identifier.

The following illustration shows the relationship between the VLAN identifier and MAC address tests, filters, and queues.


Filters on a Queue
=A
=2
=B
=3
AND
AND
OR
Send the packet to the queue if the result of the filters is TRUE.
DA
VLAN
Network Data Packet

In the preceding illustration, the network data packet includes a destination address (DA) and VLAN identifier field. The network adapter hardware implements the filters on the queue based on the settings that the miniport driver received and set in the network adapter hardware. For more information about setting filters on a receive queue, see Setting and Clearing VMQ Filters.

In this illustration, there are two filters; each filter compares a destination address and a VLAN identifier to the fields in the incoming packet. If both the VLAN and DA tests match, then the criterion for that filter is met and the incoming packet is assigned to the queue. If there is more than one filter on the queue and then a match for any filter, then the network adapter assigns the packet to the queue.

Shared Memory in Receive Buffers

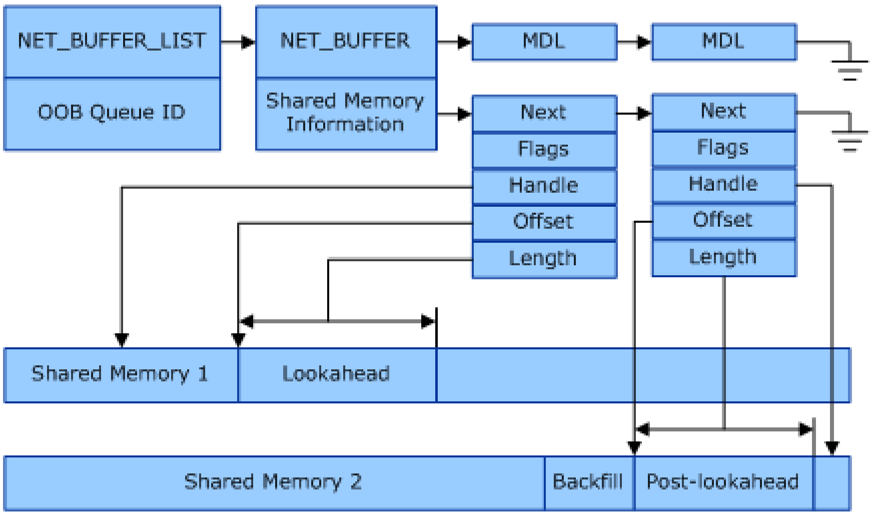
Article • 12/15/2021

This section describes the layout of the shared memory in VMQ receive buffers.For more information about using the buffers in receive indications, see VMQ Receive Path.

If the overlying protocol driver set the NDIS_RECEIVE_QUEUE_PARAMETERS_LOOKAHEAD_SPLIT_REQUIRED flag in the **Flags** member of the NDIS_RECEIVE_QUEUE_PARAMETERS structure, the network adapter should split a received packet at an offset equal to or greater than the requested lookahead size and use DMA to transfer the lookahead data and the post-lookahead data to separate shared memory segments.

Miniport drivers specify the settings for the lookahead type (**NdisSharedMemoryUsageReceiveLookahead**) or other shared memory types when the shared memory is allocated. For example, the miniport driver calls the NdisAllocateSharedMemory function and sets the **Usage** member in the NDIS_SHARED_MEMORY_PARAMETERS structure to **NdisSharedMemoryUsageReceiveLookahead**. Miniport drivers should allocated shared memory for a queue when the queue allocation is complete. For information about allocating and freeing shared memory resources for queues, see Shared Memory Resource Allocation.

The following illustration shows the relationships for the network data when the incoming data is split into two shared memory buffers.



The NET_BUFFER_SHARED_MEMORY structure specifies shared memory information. There can be a linked list of such shared memory buffers that are associated with a NET_BUFFER structure.

Use the NET_BUFFER_SHARED_MEM_NEXT_SEGMENT, NET_BUFFER_SHARED_MEM_FLAGS, NET_BUFFER_SHARED_MEM_HANDLE, NET_BUFFER_SHARED_MEM_OFFSET, and NET_BUFFER_SHARED_MEM_LENGTH macros to access the NET_BUFFER_SHARED_MEMORY in a NET_BUFFER structure. The **SharedMemoryInfo** member of the NET_BUFFER structure contains the first NET_BUFFER_SHARED_MEMORY structure in the linked list.

Note Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported. Starting with Windows Server 2012, the overlying protocol driver will not set the **NDIS_RECEIVE_QUEUE_PARAMETERS_LOOKAHEAD_SPLIT_REQUIRED** flag in the **Flags** member of the NDIS_RECEIVE_QUEUE_PARAMETERS structure.

VMQ Receive Path

Article • 12/15/2021

A network adapter indicates a received packet on a queue only if it passes all the filter field tests for a filter that is set on that queue. For more information about filter tests, see VMQ Filter Operations.

If the overlying protocol driver set the **NDIS_RECEIVE_QUEUE_PARAMETERS_PER_QUEUE_RECEIVE_INDICATION** flag in the **Flags** member of the NDIS_RECEIVE_QUEUE_PARAMETERS structure, the miniport driver must not mix NET_BUFFER_LIST structures for other receive queues with the **NET_BUFFER_LIST** structures for this queue in a single call to the NdisMIndicateReceiveNetBufferLists function. Also, the driver must set the **NDIS_RECEIVE_FLAGS_SINGLE_QUEUE** flag in the *ReceiveFlags* parameter of the **NdisMIndicateReceiveNetBufferLists** function.

If **NDIS_RECEIVE_QUEUE_PARAMETERS_PER_QUEUE_RECEIVE_INDICATION** was not set, miniport drivers can link NET_BUFFER_LIST structures for frames from different VM queues and indicate them in a single call to NdisMIndicateReceiveNetBufferLists. In this case, the indicated linked list of **NET_BUFFER_LIST** structures is not required to be sorted by queue number. **NET_BUFFER_LIST** structures for different queues are not required to be grouped together.

When a protocol driver sets **NDIS_RETURN_FLAGS_SINGLE_QUEUE** and it returns receive buffers, all of the NET_BUFFER_LIST structures in the *NetBufferLists* parameter of the NdisReturnNetBufferLists function must belong to the same VM queue. However, protocol drivers are not required to return all the **NET_BUFFER_LIST** structures that were indicated in a single call to the ProtocolReceiveNetBufferLists function in a single call to **NdisReturnNetBufferLists**. Also, the returned list can include **NET_BUFFER_LIST** structures from multiple receive indications if they belong to the same VM queue.

Protocol drivers set the **NDIS_RETURN_FLAGS_SINGLE_QUEUE** bit on the *ReturnFlags* parameter of NdisReturnNetBufferLists to indicate that all of the returned NET_BUFFER_LIST structures belong to the same VM queue.

VMQ receive indications must include the following out of band (OOB) information in the **NetBufferListInfo** member of the NET_BUFFER_LIST structures.

- Specify the queue identifier in the **NetBufferListFilteringInfo** information.
- Set the filter identifier in the **NetBufferListFilteringInfo** information to zero.

The **NetBufferListFilteringInfo** information is specified in an NDIS_NET_BUFFER_LIST_FILTERING_INFO structure. To access the NDIS_NET_BUFFER_LIST_FILTERING_INFO structure in the NET_BUFFER_LIST OOB data, an NDIS driver calls the NET_BUFFER_LIST_INFO macro and specifies the **NetBufferListFilteringInfo** information type.

To access the filter identifier and queue identifier directly, use the NET_BUFFER_LIST_RECEIVE_FILTER_ID and NET_BUFFER_LIST_RECEIVE_QUEUE_ID macros.

VMQ receive indications must define shared memory information at the **SharedMemoryInfo** member of the NET_BUFFER structure.

Note When a VMQ is deleted (for example, during VM live migration), it is possible for the miniport driver to receive an NBL that contains an invalid **QueueId** value. If this happens, the miniport should ignore the invalid queue ID and use 0 (the default queue) instead. The **QueueId** is found in the **NetBufferListFilteringInfo** portion of the NBL's OOB data, and is retrieved by using the NET_BUFFER_LIST_RECEIVE_QUEUE_ID macro.

To indicate that the NET_BUFFER_SHARED_MEMORY pointer at **SharedMemoryInfo** is valid, the miniport driver must set the NDIS_RECEIVE_FLAGS_SHARED_MEMORY_INFO_VALID flag in the *ReceiveFlags* parameter of the NdisMIndicateReceiveNetBufferLists function. For more information about the layout of shared memory buffers in VMQ receive buffers, see Shared Memory in Receive Buffers.

The receive indication must include the following information in the NET_BUFFER_SHARED_MEMORY structure.

NextSharedMemorySegment
A pointer to the next NET_BUFFER_SHARED_MEMORY structure in a **NULL**-terminated linked list of such structures.

SharedMemoryHandle
An NDIS shared memory handle that NdisAllocateSharedMemory returned.

SharedMemoryOffset
An offset, in bytes, to the start of the data from the beginning of the shared memory buffer.

SharedMemoryLength
The length, in bytes, of the shared memory segment.

If the overlying protocol driver set the **NDIS_RECEIVE_QUEUE_PARAMETERS_LOOKAHEAD_SPLIT_REQUIRED** flag in the **Flags** member of the NDIS_RECEIVE_QUEUE_PARAMETERS structure, each NET_BUFFER includes:

- Two MDLs and corresponding **SharedMemoryInfo** structures.
- A post-lookahead buffer with backfill space.

If necessary, the protocol driver copies the contents of the lookahead buffer to the backfill area. However, backfill space must exist even if the packet is entirely in the lookahead buffer.

If the overlying driver does not set the **NDIS_RECEIVE_QUEUE_PARAMETERS_LOOKAHEAD_SPLIT_REQUIRED** flag, each NET_BUFFER structure includes a single MDL and a single **SharedMemoryInfo** structure.

The byte count and byte offset in the MDL and the **DataLength** and **DataOffset** members in the NET_BUFFER_DATA structure are set in the same way as they are set for drivers that do not use VMQ. The **SharedMemoryLength** and **SharedMemoryOffset** members in the **SharedMemoryInfo** structure can be set once during initialization. The miniport driver is not required to update the **SharedMemoryLength** and **SharedMemoryOffset** members for every packet that is received because the overlying drivers and NDIS can use the NET_BUFFER **DataLength** member and the MDL byte count to determine the packet start and size.

Note Starting with NDIS 6.30 and Windows Server 2012, splitting packet data into separate lookahead buffers is no longer supported. The overlying protocol driver will not set the **NDIS_RECEIVE_QUEUE_PARAMETERS_LOOKAHEAD_SPLIT_REQUIRED** flag.

VMQ Transmit Path

Article • 12/15/2021

For transmit requests, the overlying driver uses the NET_BUFFER_LIST_RECEIVE_QUEUE_ID macro to set the queue identifier of the outgoing queue in the outgoing data with the **NetBufferListFilteringInfo** OOB information. The **NetBufferListFilteringInfo** information is specified in an NDIS_NET_BUFFER_LIST_FILTERING_INFO structure.

NDIS drivers can use the NET_BUFFER_LIST_RECEIVE_QUEUE_ID macro to set or get the queue identifier of a NET_BUFFER_LIST structure. If a queue group contains more than one VM queue, the queue identifier of the transmit packet might be set to the queue identifier of any of the VM queues in the group.

Protocol drivers set the NDIS_SEND_FLAGS_SINGLE_QUEUE bit on the *SendFlags* parameter of the NdisSendNetBufferLists function to indicate that all of the transmit NET_BUFFER_LIST structures in the call are for the same transmit queue.

Miniport drivers set the NDIS_SEND_COMPLETE_FLAGS_SINGLE_QUEUE bit on the *SendCompleteFlags* parameter of the NdisMSendNetBufferListsComplete function to indicate that all NET_BUFFER_LISTs in the call were sent on the same transmit queue.

For more information about filter tests, see VMQ Filter Operations.

Note When a VMQ is deleted (for example, during VM live migration), it is possible for the miniport driver to receive an NBL that contains an invalid **QueueId** value. If this happens, the miniport should ignore the invalid queue ID and use 0 (the default queue) instead. The **QueueId** is found in the **NetBufferListFilteringInfo** portion of the NBL's OOB data, and is retrieved by using the NET_BUFFER_LIST_RECEIVE_QUEUE_ID macro.

Obtaining VMQ Information

Article • 12/15/2021

The VMQ interface includes OID requests and WMI GUIDs that allow overlying drivers and applications to obtain information about the underlying VMQ configuration.

OID_RECEIVE_FILTER_ENUM_QUEUES enumerates the queues allocated on a network adapter. For more information about enumerating queues, see Enumerating the Allocated Queues.

As a method OID request, overlying drivers can use the OID_RECEIVE_FILTER_QUEUE_PARAMETERS OID to obtain the parameter settings of a particular queue. For more information about obtaining queue parameter settings, see Obtaining and Updating VM Queue Parameters.

OID_RECEIVE_FILTER_ENUM_FILTERS enumerates the filters that are allocated on a particular queue. For more information about enumerating the filters that are set on a queue, see Enumerating Filters on a VMQ.

As a method OID request, overlying drivers can use the OID_RECEIVE_FILTER_PARAMETERS OID to obtain the parameter settings of a filter. For more information about obtaining filter parameter settings, see Obtaining and Updating VM Queue Parameters.

Overlying drivers and applications can issue the following OID query requests to obtain the VMQ capabilities.

OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES

OID_RECEIVE_FILTER_CURRENT_CAPABILITIES

OID_RECEIVE_FILTER_GLOBAL_PARAMETERS

For more information about obtaining VMQ capabilities, see Determining the VMQ Capabilities of a Network Adapter.

Introduction to Hyper-V Extensible Switch

Article • 12/15/2021

The Hyper-V extensible switch supports an interface that allows instances of NDIS filter drivers (known as *extensible switch extensions*) to bind within the extensible switch driver stack. After they are bound and enabled, extensions can monitor, modify, and forward packets to extensible switch ports. This also allows extensions to reject, redirect, or originate packets to ports that are used by the Hyper-V partitions.

The Hyper-V extensible switch is supported starting with NDIS 6.30 in Windows Server 2012.

This section includes the following topics that describe the Hyper-V extensible switch and its interface:

Getting Started Writing a Hyper-V Extensible Switch Extension

Article • 07/08/2024

A Hyper-V Extensible Switch extension is an NDIS filter or Windows Filtering Platform (WFP) filter that runs inside the Hyper-V Extensible Switch (also called the "Hyper-V virtual switch").

There are 3 classes of extensions: capture, filtering, and forwarding. All of them can be implemented as NDIS filter drivers. Filtering extensions can also be implemented as WFP filter drivers.

For an architectural overview for driver developers, see Overview of the Hyper-V Extensible Switch.

To create a Hyper-V Extensible Switch extension, follow these steps:

1. Learn the extension architecture and programming model.

   - Read the online documentation for NDIS-based extensions, beginning with Hyper-V Extensible Switch. Capture, filtering, and forwarding extensions use the standard NDIS filtering API. The NDIS interfaces have been enhanced to provide configuration, notifications, and identification of virtual switches and virtual machines.
     - Hyper-V Extensible Switch Functions
     - Hyper-V Extensible Switch Enumerations
     - Hyper-V Extensible Switch Structures and Unions
     - Hyper-V Extensible Switch OIDs
     - Hyper-V Extensible Switch Status Indications
     - Hyper-V Extensible Switch Macros
   - Read the online documentation for WFP-based extensions, beginning with Using Virtual Switch Filtering.
   - There are several PowerShell commands that can be used to manage extensions. These are listed in Managing Installed Hyper-V Extensible Switch Extensions.

2. Set up your development environment.

   - Install Microsoft Visual Studio Professional.
   - Download and install Windows Driver Kit.

3. Study the sample extensions.

- Download the NDIS forwarding extension sample.
- Download the WFP sample. This is a functioning prototype that includes vSwitch capability.

4. Write your extension.

    - You can use one of the samples as a starting point, port existing filter code, or write your extension from scratch.
    - If you're developing an NDIS extension, you can use the standard NDIS INF with a few changes as outlined in INF Requirements for Hyper-V Extensible Switch Extensions.

5. Build your extension and unit-test it.

    - You must use Visual Studio to build your extension.
    - You can familiarize yourself with the extension build process by using Visual Studio to compile and run the sample extensions.

6. Learn about the Windows certification (logo) process for getting an extension signed.

    - An extension must pass the tests in the Windows Hardware Lab Kit (HLK).
    - The requirements for an extension are listed under the Filter.Driver.vSwitchExtension.ExtensionRequirements in the Windows Hardware Certification Requirements - Filter Driver .

7. Set up your Windows Hardware Lab Kit environment.

    - Download and install the Windows Hardware Lab Kit (HLK).

8. Run the WHCK tests for extensions:

    - Filter.Driver.Fundamentals
    - Filter.Driver.Security
    - Filter.Driver.vSwitchExtension

9. After your final extension passes WHCK certification, submit it to Microsoft.

    - Your extension must be submitted as an MSI install package with a specific format to ensure that it can be tracked and deployed by management packages, such as System Center Virtual Machine Manager (SCVMM) 2012. The MSI format is defined in Extension Driver MSI Packaging Requirements.

10. List your extension on WindowsServerCatalog.com.

    - List a brief description of your extension on WindowsServerCatalog.com.

- Information on listing a certified extension on WindowsServerCatalog.com will be available soon.

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

Overview of the Hyper-V Extensible Switch

Article • 09/27/2024

Windows Server 2012 introduces the Hyper-V Extensible Switch (also called the Hyper-V Virtual Switch), which is a virtual Ethernet switch that runs in the management operating system of the Hyper-V parent partition. This page covers the following subjects:

- Background Reading
- Types of Hyper-V extensible switches and network adapters
- Types of extensible switch extensions
- Hyper-V extensible switch architectural diagrams

Background Reading

For high-level technical overviews of this technology and its underpinnings, see the following TechNet documentation:

- Hyper-V Virtual Switch Overview
- Hyper-V Network Virtualization Overview
- Hyper-V Overview

Types of Hyper-V extensible switches and network adapters

The Hyper-V Virtual Network Manager can be used to create, configure, or delete one or more extensible switches of the following types:

- An external extensible switch that supports ports that connect to a single external network adapter as well as one or more virtual machine (VM) network adapters. This type of switch allows packets to be sent or received between all Hyper-V partitions and the physical network interface on the host.

  Also, applications and drivers that run in the management operating system can send or receive packets through this type of switch.

- An internal extensible switch that supports ports that connect to one or more internal network adapters as well as one or more VM network adapters. This type of switch allows packets to be sent or received between the Hyper-V parent partition and one or more Hyper-V child partitions on the host.

Also, applications and drivers that run in the management operating system can send or receive packets through this type of switch.

- A private extensible switch that supports ports that connect to one or more VM network adapters. This type of switch allows packets to be sent or received only between Hyper-V child partitions.

  Note Applications and drivers that run in the management operating system cannot send or receive packets through this type of switch.

Each extensible switch module routes incoming and outgoing packets over the network adapters that are used by the Hyper-V child and parent partitions. These network adapters include the following:

- External network adapters that provide the connection to the physical network interface that is available on the host.

  For more information about this type of network adapter, see External Network Adapters.

  Note Only external extensible switches provide access to an external network adapter.

- Internal network adapters that provide access to an extensible switch for processes that run in the management operating system of the Hyper-V parent partition. This allows these processes to send or receive packets over the extensible switch.

  For more information about this type of network adapter, see Internal Network Adapters.

  Note Only external and internal extensible switches provide access to an internal network adapter.

- VM network adapters that are exposed within the guest operating system that runs in a Hyper-V child partition. VM network adapters provide a connection to the extensible switch for packets to be sent or received by processes that run in the guest operating system of the child partition.

  For more information about this type of network adapter, see Virtual Machine Network Adapters.

Each Hyper-V child partition can be configured to have one or more VM network adapters. Each VM network adapter is configured to be associated with an instance of an extensible switch. This allows a child partition to be configured in the following way:

- The child partition can be configured to have a single VM network adapter that is associated with one instance of an extensible switch.
- The child partition can be configured to have multiple VM network adapters, with each VM network adapter associated with an instance of an extensible switch.
- The child partition can be configured to have multiple VM network adapters, with one or more VM network adapters associated with the same instance of an extensible switch.

Types of extensible switch extensions

The Hyper-V extensible switch supports an interface in which independent software vendors (ISVs) can extend the switch functionality in the following ways:

- The Hyper-V extensible switch supports an interface that allows NDIS filter drivers, known as *extensions*, to bind within the extensible switch driver stack. This allows extensions to capture, filter, and forward packets to extensible switch ports. This also allows extensions to inject, drop, or redirect packets to ports that are connected to the network adapters exposed in the Hyper-V partitions.

  After extensions are installed, they can be enabled or disabled on separate instances of a Hyper-V extensible switch. For more information, see Installing Hyper-V Extensible Switch Extensions.

- The Windows Filtering Platform (WFP) provides an in-box filtering extension (Wfplwfs.sys) that allows WFP filters or callout drivers to intercept packets along the Hyper-V extensible switch data path. This allows the WFP filters or callout drivers to perform packet inspection or modification by using the WFP management and system functions.

  For an overview of WFP, see Windows Filtering Platform.

  For an overview of WFP callout drivers, see Windows Filtering Platform Callout Drivers.

  Note To perform WFP-based filtering of extensible switch packet traffic, ISVs only need to extend their WFP filters and callout drivers to use extended WFP calls and data types. ISVs do not need to develop their own extensions.

The extensible switch interface supports the following types of extensions:

Capturing Extensions
Extensions that capture and monitor packet traffic. This type of extension cannot modify

or drop packets, or exclude packets from being delivered to extensible switch ports. However, capturing extensions can originate packet traffic, such as packets that contain traffic statistics that the extension sends to a host application.

Multiple capturing extensions can be bound and enabled in each instance of an extensible switch.

For more information on this type of extension, see Capturing Extensions.

Filtering Extensions
These extensions have the same capabilities as capturing extensions. However, based on port or switch policy settings, this type of extension can inspect and drop packets, or exclude packet delivery to extensible switch ports. Filtering extensions can also originate, duplicate, or clone packets and inject them into the extensible switch data path.

Multiple filtering extensions can be bound and enabled in each instance of an extensible switch.

For more information on this type of extension, see Filtering Extensions.

Forwarding Extensions
These extensions have the same capabilities as filtering extensions, but are responsible for performing the core packet forwarding and filtering tasks of extensible switches. These tasks include the following:

- Determining the destination ports for a packet, unless the packet is an NVGRE packet. For more information, see Hybrid Forwarding.
- Filtering packets by enforcing standard port policies, such as security, profile, or virtual LAN (VLAN) policies.

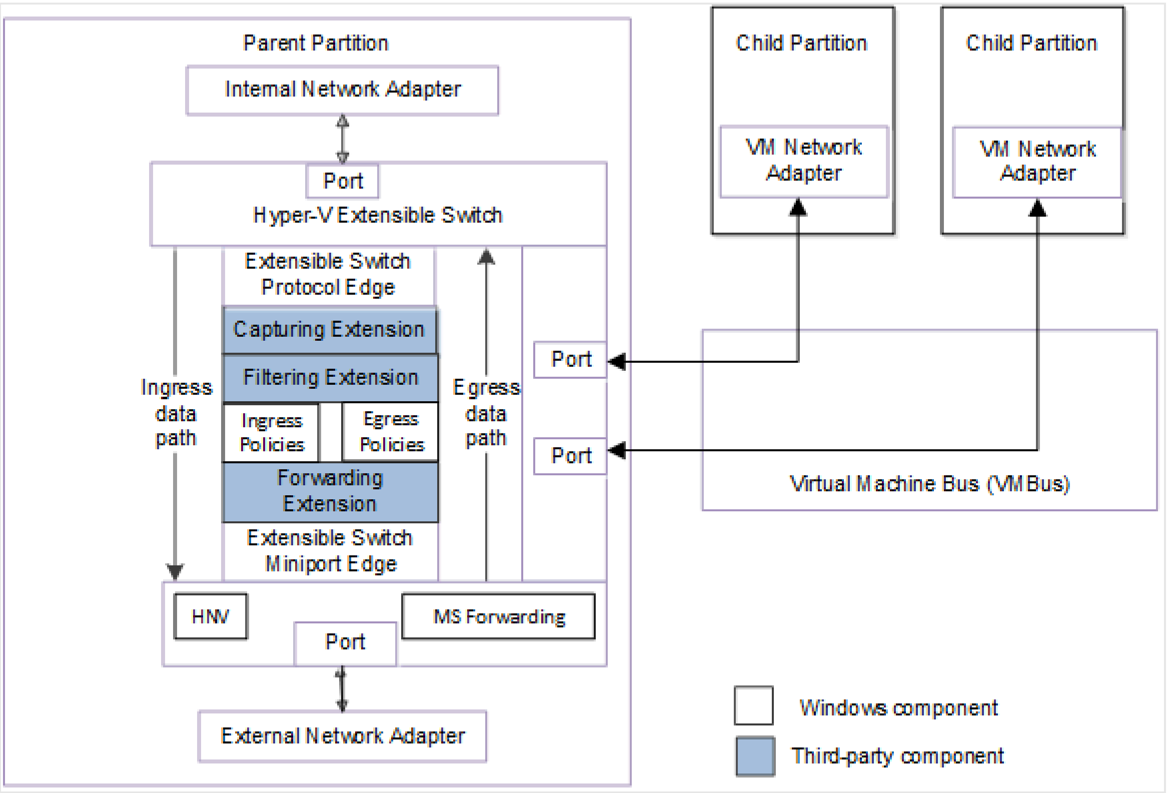
Note If a forwarding extension is not installed and enabled in the extensible switch, the switch determines a packet's destination ports as well as filters packets based on standard port settings.

Only one forwarding extension can be bound and enabled in each instance of an extensible switch.

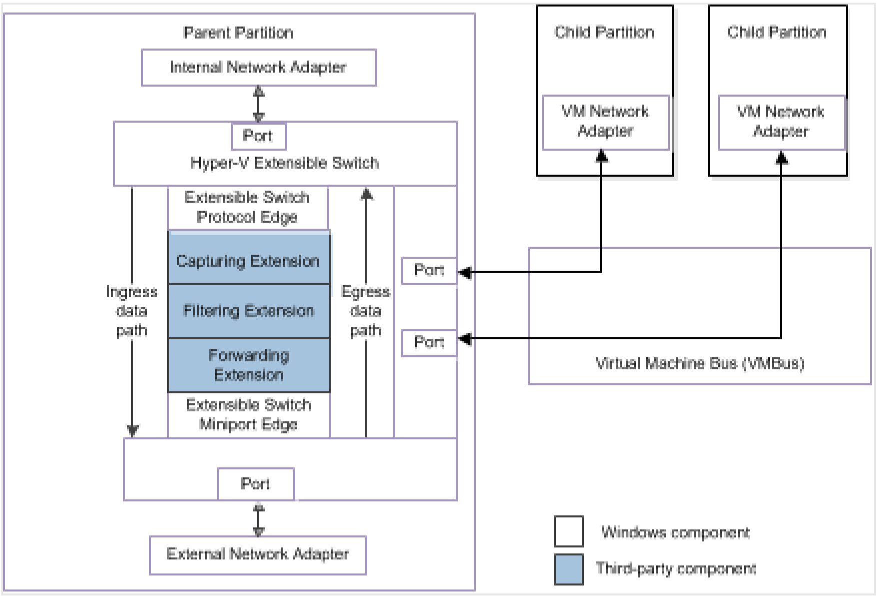
For more information on this type of extension, see Forwarding Extensions.

Hyper-V extensible switch architectural diagrams

The following figure shows the components of the extensible switch interface for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the components of the extensible switch interface for NDIS 6.30 (Windows Server 2012).



For more information about the components for the extensible switch interface, see Hyper-V Extensible Switch Architecture.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Hyper-V Extensible Switch Architecture Topics

Article • 12/15/2021

This section describes the architecture of the Microsoft Hyper-V extensible switch interface. This section includes the following topics:

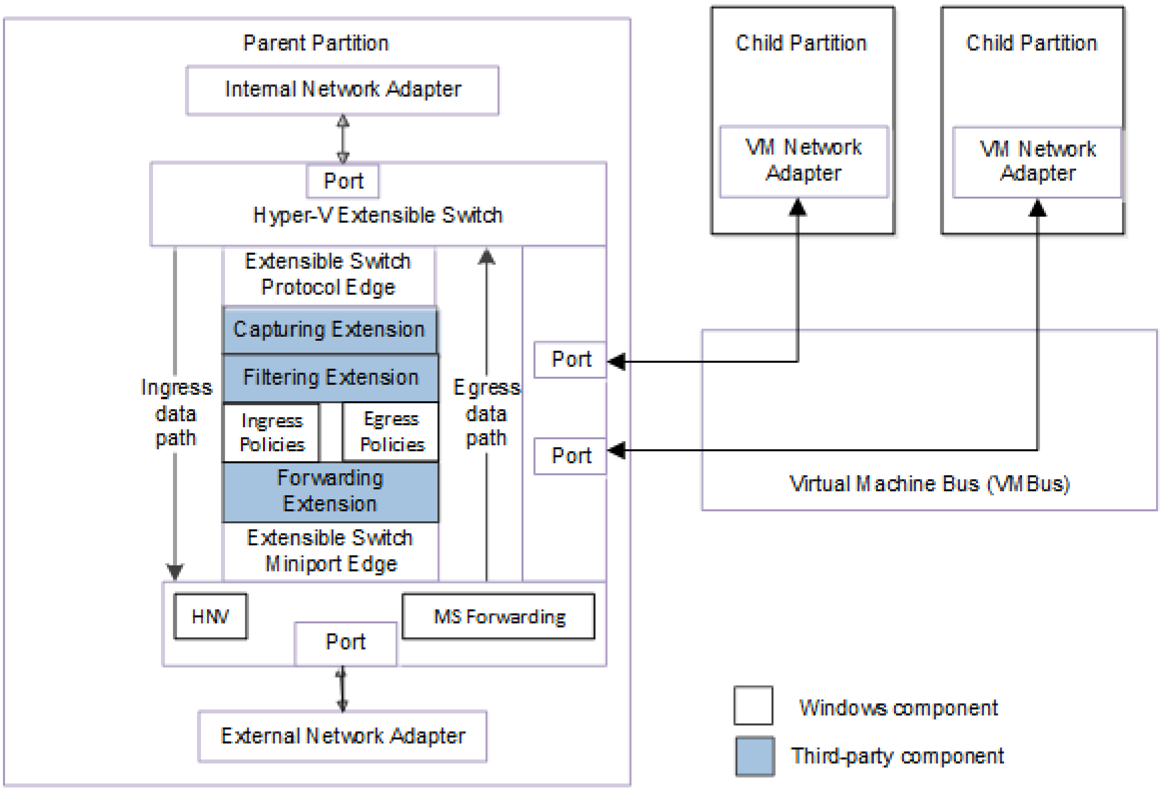
Hyper-V Extensible Switch Components Overview

Article • 12/15/2021

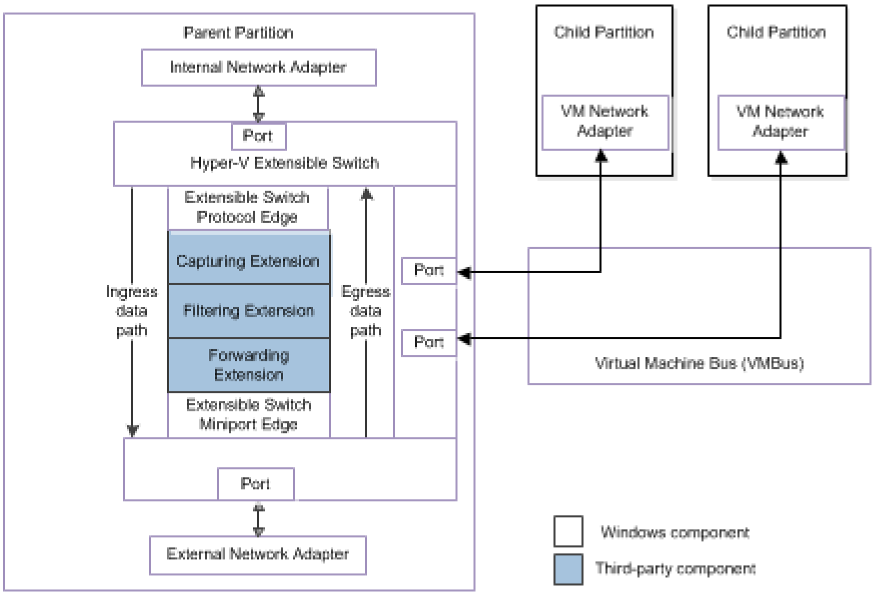
Starting with Windows Server 2012, the Hyper-V extensible switch supports an interface that allows NDIS filter drivers (known as *Hyper-V extensible switch extensions*) to bind within the extensible switch driver stack. This allows extensions to monitor, modify, and forward packets to extensible switch ports. This also allows extensions to drop, redirect, or originate packets to ports that are used by the Hyper-V partitions.

Extensions can be provisioned with policies that apply to packet traffic over an individual extensible switch port or the switch itself. This allows the extension to allow a packet to be sent or deny a packet from being sent.

The following figure shows the components of the extensible switch interface for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the components of the extensible switch interface for NDIS 6.30 (Windows Server 2012).



This section includes the following topics that describe the extensible switch components:

Hyper-V Extensible Switch Extensions

Hyper-V Extensible Switch Ports

Hyper-V Extensible Switch Network Adapters

Hyper-V Extensible Switch Port and Network Adapter States

Hyper-V Extensible Switch Extensions

Article • 12/15/2021

This section describes the type of NDIS filter drivers that are supported in the extensible switch interface. These filter drivers are known as *extensible switch extensions*.

This section includes the following topics:

Capturing Extensions

Filtering Extensions

Forwarding Extensions

Note If multiple extensions of the same type are bound to an extensible switch instance, the order in which they are layered within the extensible switch driver stack can be modified. For more information, see Reordering Hyper-V Extensible Switch Extensions.

Capturing Extensions

Article • 12/15/2021

A Hyper-V extensible switch capturing extension inspects packet traffic, object identifier (OID) requests, and NDIS status indications. This type of extension cannot modify or drop packets, or exclude packets from being delivered to extensible switch ports. However, capturing extensions can originate packet traffic, such as packets that contain traffic statistics that the extension sends to a host application.

Capturing extensions are invoked at the start of the ingress data path and at the end of the egress data path. For more information about these data paths, see Hyper-V Extensible Switch Data Path.

A capturing extension has the following requirements and restrictions:

- A capturing extension must be developed as an NDIS filter driver that supports the extensible switch interface.

  For more information about filter drivers, see NDIS Filter Drivers.

  For more information on how to write a capturing extension, see Writing Hyper-V Extensible Switch Extensions.

- A capturing extension provides the same functionality as a standard NDIS monitoring filter driver. However, the INF file for a capturing extension must install it as a modifying filter driver.

  For more information about modifying filter drivers, see Types of Filter Drivers.

  For more information about the INF requirements for modifying filter drivers, see Configuring an INF File for a Modifying Filter Driver.

- A capturing extension can monitor packets over the ingress and egress extensible switch data path. However, this type of extension must always call **NdisFSendNetBufferLists** to forward the packets to underlying drivers in the extensible switch driver stack and not complete them.

- A capturing extension must not modify the data within the packets nor add port destinations to the out-of-band (OOB) data of the packet. The extension must not exempt the delivery of the packet to any extensible switch port.

- A capturing extension can originate packets. For example, the extension may originate packets in order to report traffic conditions to a remote monitoring application.

For more information on originating packets by an extension, see Originating Packet Traffic.

Note As with other extensions, the capturing extension can only originate packet traffic in the extensible switch ingress data path.

- A capturing extension can monitor packets, OID requests, and NDIS status indications that are issued over the extensible switch driver stack. However, this type of extension must forward packets, OID requests, and NDIS status indications through the extensible switch driver stack. The extension must not modify the data within the packets, OID requests, or NDIS status indications that it monitors.

- The **FilterClass** value in the INF file for the extension must be set to **ms_switch_capture**. For more information, see INF Requirements for Hyper-V Extensible Switch Extensions.

- Any number of capturing extensions can be bound to an extensible switch instance. By default, multiple capturing extensions are ordered based on when they were installed. For example, multiple capturing extensions are layered in the extensible switch driver stack with the most recently installed extension layered above other capturing extensions in the stack.

  Once bound to an extensible switch instance, the layering of capturing extensions in the extensible switch driver stack can be reordered. For more information, see Reordering Hyper-V Extensible Switch Extensions.

Filtering Extensions

Article • 12/15/2021

A Hyper-V extensible switch filtering extension can inspect, modify, and insert packets into the extensible switch data path. Based on extensible switch port and switch policy settings, the extension can drop a packet or exclude its delivery to one or more destination ports.

Filtering extensions are invoked after capturing extensions in the ingress data path and after the forwarding extensions in the egress data path. For more information about these data paths, see Hyper-V Extensible Switch Data Path.

A filtering extension can do the following with packets that were obtained on the ingress data path:

- Filter packet traffic and enforce custom port or switch policies for packet delivery through the extensible switch. When the filtering extension filters packets in the ingress data path, it can apply filtering rules based only on the source port and network adapter connection from which the packet originated. This information is stored in the out-of-band (OOB) data of a packet's **NET_BUFFER_LIST** structure and can be obtained by using the **NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL** macro.

  Note Packets obtained on the ingress data path do not contain destination ports. Filtering packets based on destination ports can be done only on packets obtained on the egress data path.

  Custom policies are defined by the independent software vendor (ISV). Property settings for this policy type are managed through the Hyper-V WMI management layer. The filtering extension is configured with these property settings through an object identifier (OID) request of OID_SWITCH_PORT_PROPERTY_UPDATE and OID_SWITCH_PROPERTY_UPDATE.

  For more information on custom extensible port or switch policies, see Managing Hyper-V Extensible Switch Policies.

  Note Only forwarding extensions can enforce standard port policies for packet delivery through the extensible switch.

- Inject new, modified, or cloned packets into the ingress data path.

  For more information, see Hyper-V Extensible Switch Send and Receive Operations.

A filtering extension can do the following with packets that were obtained on the egress data path:

- Filter packet traffic and enforce custom port or switch policies for packet delivery through the extensible switch. When the filtering extension filters packets in the egress data path, it can apply filtering rules based on the source or destination ports for a packet. Destination port data is stored in the OOB data of a packet's **NET_BUFFER_LIST** structure. Extensions obtain this information by calling the *GetNetBufferListDestinations* function.

- Exclude the delivery of the packet to one or more extensible switch destination ports. This allows the filtering extension to exclude the delivery of a packet to extensible switch ports.

  For more information on how to exclude packet delivery to extensible switch ports, see Excluding Packet Delivery to Extensible Switch Destination Ports.

- Manage the traffic flow to one or more destination ports by postponing the forwarding of packets up the egress data path.

  For example, a filtering extension that supports quality of service (QoS) functionality may want to immediately call **NdisFSendNetBufferLists** to forward packets that are specified with a higher priority value. Depending on the traffic flow, the extension may want to forward packets with a lower priority value at a later time.

- Modify the packet data. If the filtering extension needs to modify the data in a packet, it must first clone the packet without preserving port destinations. Then, the extension must inject the modified packet into the ingress data path. This allows the underlying extensions to enforce policies on the modified packet and the forwarding extension can add port destinations.

  For more information, see Cloning Packet Traffic.

Besides inspecting OID requests and NDIS status indications, a filtering extension can do the following:

- Veto the creation of an extensible switch port or network adapter connection by returning STATUS_DATA_NOT_ACCEPTED for the applicable extensible switch OIDs. For example, the filtering extension can veto a port creation request by returning STATUS_DATA_NOT_ACCEPTED when the driver receives an OID set request of OID_SWITCH_PORT_CREATE.

Note Filtering extensions do not create or delete ports or network adapter connections. The protocol edge of the extensible switch issues OIDs to notify the underlying filter drivers about the creation or deletion of ports or network adapter connections. For more information, see Hyper-V Extensible Switch Port and Network Adapter States.

- Veto the addition or update of an extensible switch or port policy by returning STATUS_DATA_NOT_ACCEPTED for the applicable extensible switch OIDs. For example, the filtering extension can veto the addition of a port policy by returning STATUS_DATA_NOT_ACCEPTED when the extension receives an OID set request of OID_SWITCH_PORT_PROPERTY_ADD.

  For more information about extensible switch policies, see Managing Hyper-V Extensible Switch Policies.

A filtering extension has the following requirements:

- A filtering extension must be developed as an NDIS filter driver that supports the extensible switch interface.

  For more information about filter drivers, see NDIS Filter Drivers.

  For more information on how to write a filtering extension, see Writing Hyper-V Extensible Switch Extensions.

  Note The Windows Filtering Platform (WFP) provides an in-box extensible switch filtering extension (Wfplwfs.sys). This extension allows WFP filters or callout drivers to intercept packets along the Hyper-V extensible switch data path. This allows the filters or callout drivers to perform packet inspection or modification by using the WFP management and system functions. For an overview of WFP, see Windows Filtering Platform.

- The INF file for a filtering extension must install the driver as a modifying filter driver. NDIS-monitoring filter drivers cannot be installed in the extensible switch driver stack.

  For more information about modifying filter drivers, see Types of Filter Drivers.

  For more information about the INF requirements for modifying filter drivers, see Configuring an INF File for a Modifying Filter Driver.

- The **FilterClass** value in the INF file for the filter driver must be set to **ms_switch_filter**. For more information, see INF Requirements for Hyper-V Extensible Switch Extensions.

- Any number of filtering extensions can be bound and enabled in the driver stack for each instance of an extensible switch. By default, multiple filtering extensions are ordered based on when they were installed. For example, multiple filtering extensions are layered in the extensible switch driver stack with the most recently installed extension layered above other filtering extensions in the stack.

  After they are bound and enabled in an extensible switch instance, the layering of filtering extensions in the extensible switch driver stack can be reordered. For more information, see Reordering Hyper-V Extensible Switch Extensions.

Forwarding Extensions

Article • 12/15/2021

A forwarding extension has the same capabilities as a filtering extension, but is responsible for performing the core packet forwarding and filtering tasks of the extensible switch. These tasks include the following:

- Determining the destination ports for a packet.

  Note If the packet is an NVGRE packet, the Hyper-V Network Virtualization (HNV) component of the extensible switch determines the destination ports and forwards the packet. For more information, see Hybrid Forwarding.

- Filtering packets by enforcing standard port policies, such as security, profile, or virtual LAN (VLAN) policies.

  Note The extensible switch still performs filtering based on built-in policies. These policies include access control lists (ACLs) and quality of service (QoS).

Note If a forwarding extension is not installed and enabled in the extensible switch, the switch determines a packet's destination ports as well as filters packets based on standard port settings.

Forwarding extensions are layered immediately above the extensible switch extension miniport driver in the egress and ingress data path. For more information about these data paths, see Hyper-V Extensible Switch Data Path.

A forwarding extension can do the following with packets that were obtained on the ingress data path:

- It can filter packet traffic and enforce custom and standard port or switch policies for packet delivery through the extensible switch. When the forwarding extension filters packets in the ingress data path, it applies filtering rules based on the source port as well as the destination ports that the extension assigns to the packet.

  Custom policies are defined by the independent software vendor (ISV). Standard policies are defined by the extensible switch interface. Property settings for these types of policies are managed through the Hyper-V WMI management layer. The forwarding extension is configured with these property settings through an object identifier (OID) request of OID_SWITCH_PORT_PROPERTY_UPDATE and OID_SWITCH_PROPERTY_UPDATE.

For more information on extensible switch policies, see Managing Hyper-V Extensible Switch Policies.

- It can inject new, modified, or cloned packets into the ingress data path.

  For more information, see Hyper-V Extensible Switch Send and Receive Operations.

- It can determine the delivery of the packet to one or more extensible switch destination ports. This allows the forwarding extension to add destination ports for the delivery of a packet to extensible switch ports.

  For more information on how to add destination ports, see Adding Extensible Switch Destination Port Data to a Packet.

A forwarding extension can do the following with packets that were obtained on the egress data path:

- It can filter packet traffic and enforce custom and standard port or switch policies for packet delivery through the extensible switch. When the forwarding extension filters packets in the egress data path, it can apply filtering rules based on the source or destination ports for a packet.

- It can exclude the delivery of the packet to one or more extensible switch destination ports. This allows the forwarding extension to exclude the delivery of a packet to extensible switch ports.

  For more information on how to exclude packet delivery to extensible switch ports, see Excluding Packet Delivery to Extensible Switch Destination Ports.

  Note The forwarding extension can only exclude packet delivery when it handles the packet on the egress data path. The extension can only add or modify destination ports for the packet on the ingress data path.

- It can modify the packet data. If the forwarding extension needs to modify the data in a packet, it must first clone the packet before it assigns port destinations. After the packet has been modified and port destinations assigned, the extension must inject the modified packet into the egress data path.

  For more information, see Cloning Packet Traffic.

Besides inspecting OID requests and NDIS status indications, a forwarding extension can do the following:

- It can inject OIDs or NDIS status indications into the extensible switch control path. This allows the forwarding extension to create or modify OIDs and status indications and forward them to or from underlying physical network adapters.

For example, the extensible switch external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX intermediate driver itself can be bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*.

In this configuration, the extensible switch extensions are exposed to every network adapter in the extensible switch team. This allows the forwarding extension in the extensible switch driver stack to manage the configuration and use of individual network adapters in the team. For example, the extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. Such an extension is known as a *teaming provider*.

By acting as a teaming provider, the forwarding extension can create or modify OID requests to enable or disable hardware capabilities on an adapter in the team. The teaming provider can also create or modify NDIS status indications based on changes to one or more adapters in the team.

For more information about teaming providers, see Teaming Provider Extensions.

- It can veto the creation of an extensible switch port or network adapter connection by returning STATUS_DATA_NOT_ACCEPTED for the applicable extensible switch OIDs. For example, the forwarding extension can veto a port creation request by returning STATUS_DATA_NOT_ACCEPTED when the driver receives an OID set request of OID_SWITCH_PORT_CREATE.

  Note Forwarding extensions do not create or delete ports or network adapter connections. The protocol edge of the extensible switch issues OIDs to notify the underlying extensions about the creation or deletion of ports or network adapter connections. For more information, see Hyper-V Extensible Switch Port and Network Adapter States.

- It can veto the addition or update of an extensible switch or port policy by returning STATUS_DATA_NOT_ACCEPTED for the applicable extensible switch OIDs. For example, the forwarding extension can veto the addition of a port policy by returning STATUS_DATA_NOT_ACCEPTED when the driver receives an OID set request of OID_SWITCH_PORT_PROPERTY_ADD.

  For more information about extensible switch policies, see Managing Hyper-V Extensible Switch Policies.

A forwarding extension has the following requirements:

- A forwarding extension must be developed as an NDIS filter driver that supports the extensible switch interface.

  For more information about filter drivers, see NDIS Filter Drivers.

  For more information on how to write a forwarding extension, see Writing Hyper-V Extensible Switch Extensions.

- The INF file for a forwarding extension must install the extension as a modifying filter driver. NDIS-monitoring filter drivers cannot be installed in the extensible switch driver stack.

  For more information about modifying filter drivers, see Types of Filter Drivers.

  For more information about the INF requirements for modifying filter drivers, see Configuring an INF File for a Modifying Filter Driver.

- The **FilterClass** value in the INF file for the extension must be set to **ms_switch_forward**. For more information, see INF Requirements for Hyper-V Extensible Switch Extensions.

- Only one forwarding extension can be enabled in the driver stack for each instance of an extensible switch.

For more information about extensible switch teams, see Types of Physical Network Adapter Configurations.

For more information about forwarding extensions, see the following pages:

- Teaming Provider Extensions
- Hybrid Forwarding

Related topics

Forwarding Packets to Hyper-V Extensible Switch Ports

Forwarding Packets to Physical Network Adapters

Overview of the Hyper-V Extensible Switch

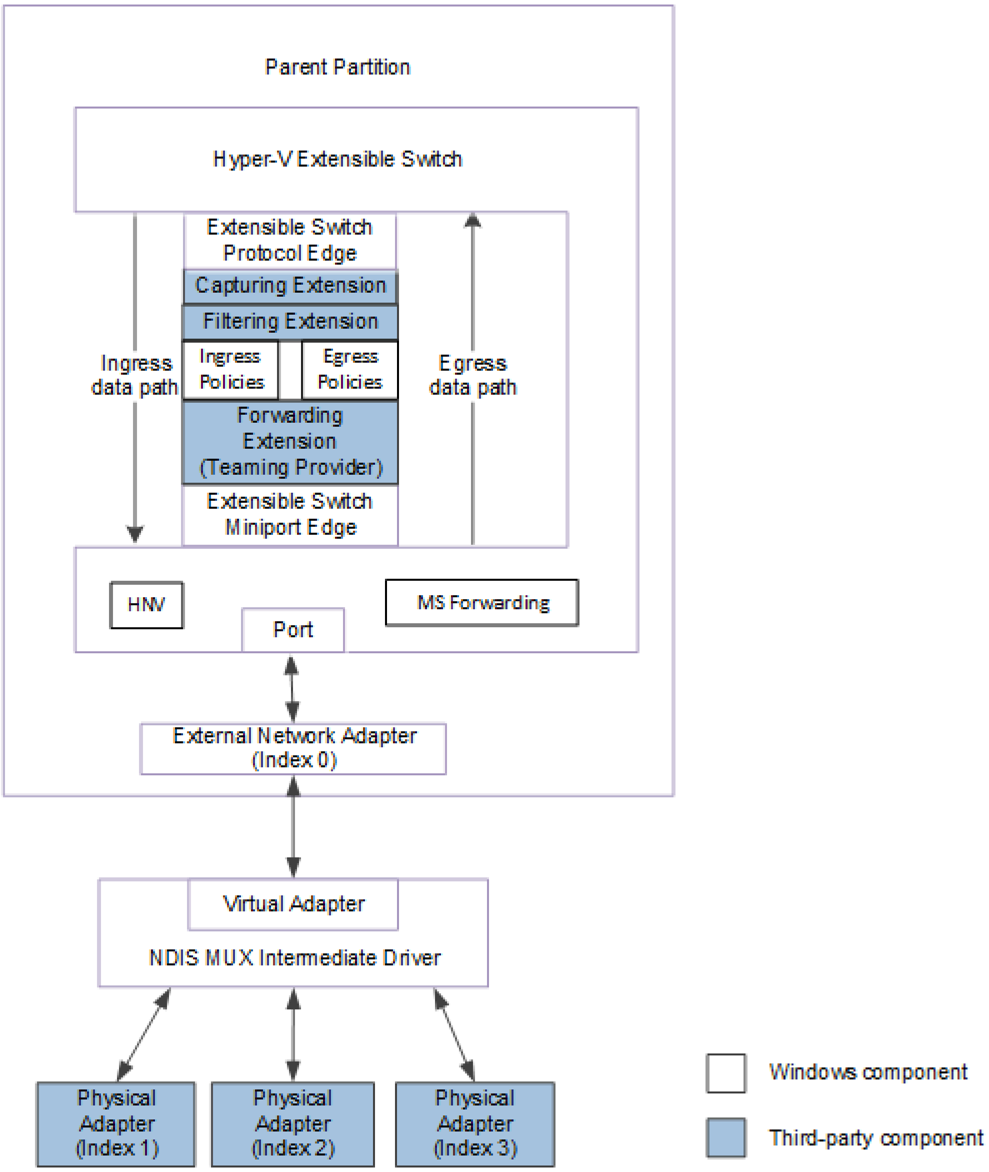
Teaming Provider Extensions

Article • 12/15/2021

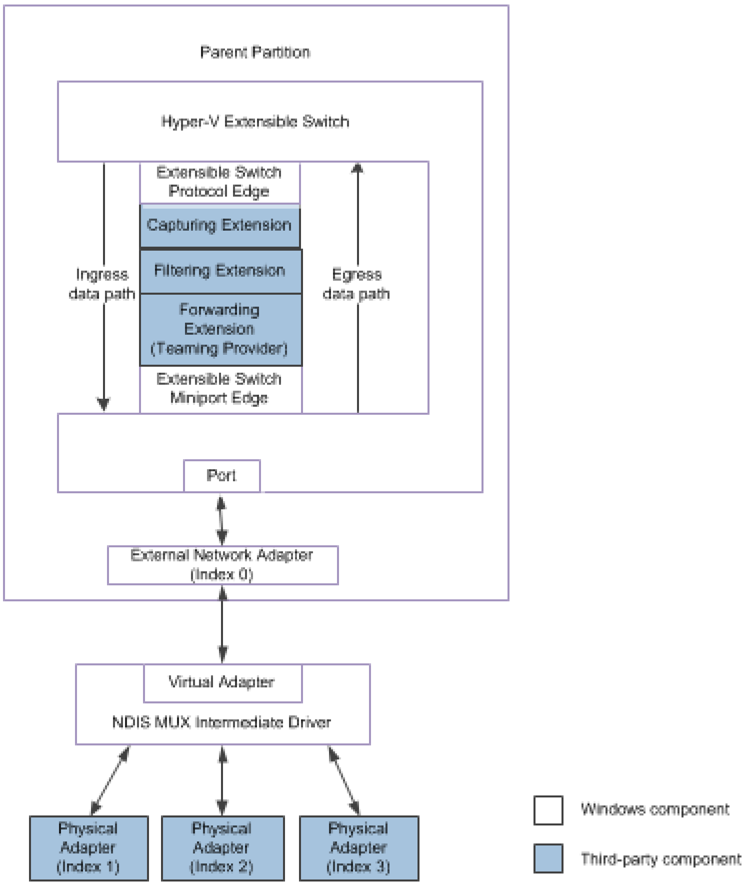
The extensible switch external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX intermediate driver itself can be bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*. For more information about extensible switch teams, see Types of Physical Network Adapter Configurations.

In this configuration, the extensible switch extensions are exposed to every network adapter in the extensible switch team. This allows the forwarding extension in the extensible switch driver stack to manage the configuration and use of individual network adapters in the team. For example, the extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. Such an extension is known as a *teaming provider*.

The following figure shows the data path for packet traffic to or from the underlying extensible switch team that is bound to the external network adapter for NDIS 6.40 (Windows Server 2012 R2) and later.


Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Ingress Policies
Egress Policies
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
HNV
Port
MS Forwarding
External Network Adapter (Index 0)
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index 3)
Windows component
Third-party component

The following figure shows the data path for packet traffic to or from the underlying extensible switch team that is bound to the external network adapter for NDIS 6.30 (Windows Server 2012).


Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
Port
External Network Adapter (Index 0)
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index 3)
□ Windows component
▨ Third-party component

Teaming providers can do everything that a forwarding extension can. In addition, teaming providers can do the following.

- Forward outgoing packets to an individual physical adapter in the extensible switch team. This ability is especially useful for LBFO functionality.

- Forward standard NDIS object identifier (OID) requests to an individual physical adapter in the extensible switch team. This ability is especially useful for configuring the adapters in the team for hardware offloads.

  For example, the MUX driver advertises the common capabilities of the entire extensible switch team. However, the teaming provider can issue OID requests to query the individual capabilities of adapters within the team. Then, the teaming provider can issue OID requests to the extensible switch external network adapter to set the capabilities that apply to the entire team.

- Forward private OID requests to an individual physical adapter in the extensible switch team. These private OID requests are defined by the independent hardware vendor (IHV) for the physical network adapters. This allows a teaming provider that was also developed by the IHV to enable or disable proprietary attributes on individual physical adapters in the team.

- Modify NDIS status indications from the extensible switch team. This ability is especially useful for managing the extensible switch team for hardware offloads.

For example, the MUX driver issues NDIS status indications with settings that are common for the entire extensible switch team. If the status indication was for a hardware offload that the teaming provider enabled for a network adapter in the extensible switch team, the teaming provider can first issue an OID request to query the current capabilities on that adapter. Then, the teaming provider can modify the indication data to set those attributes that may have changed on the adapter.

Teaming providers must follow these guidelines when managing an extensible switch team:

- The teaming provider must maintain state for every physical network adapter for which an extensible switch network connection had been established.

  For every physical network adapter that is bound to the external network adapter, the protocol edge of the extensible switch issues a separate OID set request of OID_SWITCH_NIC_CREATE. This OID request notifies the extension about the creation of a network connection to an underlying physical adapter.

- When the network connection to the physical network adapter is created, it is assigned a nonzero index value that is unique for the port to which the external network adapter is connected.

  The teaming provider must specify the network adapter index value when it issues or forwards packets or OID requests to an underlying physical network adapter.

  For more information, see Network Adapter Index Values.

- If the teaming provider issues or forwards packets to a physical adapter, it must specify the nonzero network adapter index value of the physical adapter connection.

  When the provider receives packets, it can determine the source network adapter index value from the packet's out-of-band forwarding context in the **NET_BUFFER_LIST** structure. For more information about the forwarding context, see Hyper-V Extensible Switch Forwarding Context.

  For more information, see Hyper-V Extensible Switch Data Path.

- To issue forward OID requests to a physical adapter, the teaming provider must encapsulate the OID request within an **NDIS_SWITCH_NIC_OID_REQUEST** structure. The provider must set the **DestinationNicIndex** member to the nonzero network adapter index value of the physical adapter connection. The provider then issues an OID set request of OID_SWITCH_NIC_REQUEST to deliver the encapsulated OID request to the target physical adapter.

For more information, see Managing OID Requests to Physical Network Adapters.

- The teaming provider can issue NDIS status indications on behalf of an underlying physical adapter. To do this, the provider must encapsulate the indication within an **NDIS_SWITCH_NIC_STATUS_INDICATION** structure. The provider must set the **SourceNicIndex** member to the nonzero network adapter index value of the physical adapter connection. The provider then issues an NDIS status indication of **NDIS_STATUS_SWITCH_NIC_STATUS** to deliver the encapsulated status indication to overlying drivers in the extensible switch driver stack.

  For more information, see Managing NDIS Status Indications from Physical Network Adapters.

For more information about forwarding extensions, see Forwarding Extensions.

For more information on MUX drivers, see NDIS MUX Intermediate Drivers.

Hybrid Forwarding

Article • 12/15/2021

Starting with NDIS 6.40 (Windows Server 2012 R2, the Hyper-V extensible switch architecture supports hybrid forwarding by the Hyper-V Network Virtualization (HNV) component of the extensible switch and by forwarding extensions.

Note This page assumes that you are familiar with Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload and Overview of the Hyper-V Extensible Switch.

NVGRE and non-NVGRE packets

In a hybrid forwarding environment, there are two types of packets that enter and leave the Hyper-V extensible switch: NVGRE packets and non-NVGRE packets:

- NVGRE packets have the encapsulated format that is specified in the NVGRE: Network Virtualization using Generic Routing Encapsulation Internet Draft. NVGRE packets are forwarded by the HNV component of the Hyper-V extensible switch.
- Non-NVGRE packets are just normal network packets. Non-NVGRE packets are forwarded by the forwarding extension (or, in the absence of a forwarding extension, the extensible switch itself).

Flow of NVGRE and non-NVGRE packets through the switch

In the ingress data path, after the capturing and filtering extensions but before the forwarding extension, if a packet is an NVGRE packet, the extensible switch sets the **NativeForwardingRequired** flag in the NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO structure for the packet. This structure is contained in the **NetBufferListInfo** member of the packet's NET_BUFFER_LIST structure.

Note The **NetBufferListInfo** member of the NET_BUFFER_LIST is often referred to as the packet's "out-of-band (OOB) data."

If the **NativeForwardingRequired** flag is set in the packet's OOB data, the packet is an NVGRE packet. If it is not set, the packet is a non-NVGRE packet.

Extensions should use the NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL macro to check the value of the **NativeForwardingRequired** flag.

NVGRE and non-NVGRE packets are treated as follows:

- The HNV component of the Hyper-V extensible switch forwards (i.e., determines the destination table for) all NVGRE packets
- The HNV component performs NVGRE encapsulation and decapsulation as needed.
- The forwarding extension forwards all non-NVGRE packets.
- The forwarding extension cannot forward NVGRE packets, but it can perform the same filtering actions as a filtering extension, including adding or excluding destination ports or even dropping packets.
- If there is no forwarding extension, the Hyper-V extensible switch forwards all packets.

For more information, see Packet Flow through the Extensible Switch Data Path.

Support for third-party network virtualization

A **VirtualSubnetId** can be configured on a VM network adapter port as an external virtual subnet. This feature was added to enable forwarding extensions to provide third-party network virtualization solutions. On ingress, the Hyper-V extensible switch will not set the **NativeForwardingRequired** flag in the NET_BUFFER_LIST structures for these packets. A forwarding extension may then modify the packet headers, as required, during forwarding. Packets that are being modified must be cloned and their **ParentNetBufferList** pointers set to the original **NET_BUFFER_LIST**. (See Cloning Packet Traffic.)

Related topics

Adding Extensible Switch Destination Port Data to a Packet

Cloning Packet Traffic

Forwarding Extensions

Packet Flow through the Extensible Switch Data Path

NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL

NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO

Hyper-V Extensible Switch Ports Topics

Article • 12/15/2021

This section includes the following topics that describe the characteristics of Hyper-V extensible switch ports:

Overview of Hyper-V Extensible Switch Ports

Validation Ports

Operational Ports

Note NDIS ports and extensible switch ports are different objects. Packets that move through the extensible switch data path are always assigned to the NDIS port number of **NDIS_DEFAULT_PORT_NUMBER**. However, the packet's source and destination extensible switch port number can be a value of **NDIS_SWITCH_DEFAULT_PORT_ID** or greater. For more information, see Hyper-V Extensible Switch Data Path.

Overview of Hyper-V Extensible Switch Ports

Article • 12/15/2021

Each network connection to the Hyper-V extensible switch is represented by a port. The extensible switch interface creates and configures a port before a network connection is made. After the network connection is torn down, the interface may delete the port or reuse it for another network connection.

Every Hyper-V child partition that is configured with a network interface is assigned a port on the extensible switch. When a Hyper-V child partition is started, the extensible switch interface creates a port before the virtual machine (VM) network adapter is exposed within the guest operating system. After the VM network adapter is exposed and initialized, the extensible switch interface creates a network connection between the VM network adapter and the extensible switch port. If the child partition is stopped, the extensible switch interface first deletes the network connection and then deletes the extensible switch port.

When an extensible switch port is created, it is configured with a unique identifier and name. After it is created, the extensible switch port can be provisioned with policies that define various attributes for the management of packet traffic over the port. For example, standard port policies can be defined for virtual LAN (VLAN) attributes and access restrictions for port traffic. In addition, independent software vendors (ISVs) can define custom policies that individual ports can be provisioned with. For more information, see Port Policies.

Extensible switch ports consist of the following types:

Validation ports
Validation ports are used to validate and verify port settings. These ports are temporary and are created under certain conditions.

For example, when a Hyper-V child partition is created or reconfigured for network access, the extensible switch interface creates a validation port. The interface uses this port to verify the settings for the network connection to the virtual machine (VM) network adapter of the partition. After the verification is completed, the validation port is deleted and an operational port is created.

For more information, see Validation Ports.

Operational ports
Operational ports are created to host an extensible switch network adapter connection. When an operational port is created, it is assigned a port type. This port type is in effect after the port is created and before it is torn down. For ports assigned to Hyper-V child partitions, the operational port type stays in effect while the partition is running and operational.

For more information, see Operational Ports.

Extensible switch extensions are notified of port creation, update and deletion through the following extensible switch object identifier (OID) requests:

OID_SWITCH_PORT_CREATE
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_CREATE to notify extensible switch extensions about the creation of an extensible switch port.

The extension can veto the creation notification by returning STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot allocate resources to enforce its configured policies on the port, the extension vetoes the creation notification.

If the extension accepts the creation notification, it must forward the OID request down the extensible switch driver stack. The extension monitors the completion status of this OID request to determine whether underlying extensions have vetoed the port creation notification.

Extensions cannot forward packets to the newly created port until a network connection is created. For more information on this process, see Hyper-V Extensible Switch Network Adapters.

OID_SWITCH_PORT_UPDATED
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_UPDATED to notify extensible switch extensions that an extensible switch port's parameters are being updated. The OID will only be issued for ports that have already been created, and have not yet begun the teardown/delete process. Currently only the *PortFriendlyName* field is subject to update after creation.

The protocol edge of the extensible switch issues this OID request when the previous network connection to the port has been torn down and all OID requests to the port have been completed.

Note This OID request could be issued if a network adapter connection was not previously made to the port.

The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

OID_SWITCH_PORT_TEARDOWN
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_TEARDOWN to notify extensible switch extensions that an extensible switch port is being deleted. The protocol edge of the extensible switch issues this OID request when the previous network connection to the port has been torn down and all OID requests to the port have been completed.

Note This OID request could be issued if a network adapter connection was not previously made to the port.

The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

After the extension forwards this OID request, it can no longer issue OID requests for the port being deleted.

OID_SWITCH_PORT_DELETE
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_DELETE to notify extensible switch extensions that an extensible switch port has been deleted. The protocol edge of the extensible switch issues this OID request after it issues the OID_SWITCH_PORT_TEARDOWN request and OID requests that target the port have been completed.

The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

All extensible switch ports that are created for network connections are assigned an identifier greater than **NDIS_SWITCH_DEFAULT_PORT_ID**. The **NDIS_SWITCH_DEFAULT_PORT_ID** identifier is reserved and used in the following ways:

- The source port identifier for a packet is stored in the packet's out-of-band (OOB) forwarding context that is associated with its **NET_BUFFER_LIST** structure. A source port identifier of **NDIS_SWITCH_DEFAULT_PORT_ID** specifies that the packet originated from the extensible switch extension and not from an extensible switch port. A packet with a source port identifier of **NDIS_SWITCH_DEFAULT_PORT_ID** is trusted and bypasses the extensible switch port policies, such as access control lists (ACLs) and quality of service (QoS).

  The extension may want the packet to be treated as if it originated from a particular port. This allows the policies for that port to be applied to the packet. The extension calls *SetNetBufferListSource* to change the source port for the packet.

However, there may be situations where the extension may want to assign the packet's source port identifier to **NDIS_SWITCH_DEFAULT_PORT_ID**. For example, the extension may want to set the source port identifier to **NDIS_SWITCH_DEFAULT_PORT_ID** for proprietary control packets that are sent to a device on the external network.

For more information about the forwarding context, see Hyper-V Extensible Switch Forwarding Context.

- Object identifier (OID) requests of OID_SWITCH_NIC_REQUEST are issued by the extensible switch interface to encapsulate OID requests that are issued to the extensible switch external network adapter. For example, hardware offload OID requests are encapsulated by the interface before they are issued down the extensible switch driver stack.

  An extension can also issue encapsulated OID requests in order to forward requests down the extensible switch control path. This allows extensions to query or configure the capabilities of an underlying physical network adapter.

  The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for this OID request contains a pointer to an **NDIS_SWITCH_NIC_OID_REQUEST** structure. If the **SourcePortId** member is set to **NDIS_SWITCH_DEFAULT_PORT_ID**, this specifies that the OID request was originated by the extensible switch interface. If the **DestinationPortId** is set to **NDIS_SWITCH_DEFAULT_PORT_ID**, this specifies that the OID request is targeted for processing by an extension in the extensible switch driver stack.

  For more information about the control path for OID requests, see Hyper-V Extensible Switch Control Path for OID Requests.

- NDIS status indications of **NDIS_STATUS_SWITCH_NIC_STATUS** are issued by the miniport edge of the extensible switch to encapsulate a status indication from the extensible switch external network adapter.

  An extension can also issue encapsulated NDIS status indications in order to forward indications up the extensible switch control path. This allows extensions to change the reported capabilities of an underlying physical network adapter.

  The **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure for this indication contains a pointer to an **NDIS_SWITCH_NIC_STATUS_INDICATION** structure. If the **SourcePortId** member is set to **NDIS_SWITCH_DEFAULT_PORT_ID**, this specifies that the status indication was originated by the extensible switch interface. If the **DestinationPortId** is set to **NDIS_SWITCH_DEFAULT_PORT_ID**, this specifies that the OID request is targeted for processing by an extension in the extensible switch driver stack.

For more information about the control path for NDIS status indications, see Hyper-V Extensible Switch Control Path for NDIS Status Indications.

The extensible switch interface maintains a reference counter for each port that has been created. A port will not be deleted if its reference counter has a nonzero value. The interface provides the following handler functions for incrementing or decrementing an extensible switch port's reference counters:

ReferenceSwitchPort
The extensible switch extension calls this function to increment a port's reference counter. While the reference counter has a nonzero value, the protocol edge of the extensible switch will not issue an object identifier (OID) set request of OID_SWITCH_PORT_DELETE to delete the extensible switch port.

The extension must call *ReferenceSwitchPort* before it performs any operation that requires the port to be in an active state. For example, the extension must call *ReferenceSwitchPort* before it issues an OID method request of OID_SWITCH_PORT_PROPERTY_ENUM.

Note The extension must not call *ReferenceSwitchPort* for a port after it receives an OID set request of OID_SWITCH_PORT_TEARDOWN for that port.

DereferenceSwitchPort
The extensible switch extension calls this function to decrement a port's reference counter.

The extension must call *DereferenceSwitchPort* after the operation being performed on the port has completed. For example, if the extension called *ReferenceSwitchPort* before if issued an OID_SWITCH_PORT_PROPERTY_ENUM request, the extension must call *DereferenceSwitchPort* after the OID request has completed.

Note NDIS ports and extensible switch ports are different objects. Packets that move through the extensible switch data path are always assigned to the NDIS port number of **NDIS_DEFAULT_PORT_NUMBER**. However, the packet's source and destination extensible switch port number can be a value of **NDIS_SWITCH_DEFAULT_PORT_ID** or greater. For more information, see Hyper-V Extensible Switch Data Path.

Validation Ports

Article • 12/15/2021

Starting with NDIS 6.30 in Windows Server 2012, the extensible switch interface creates an operational port to host an extensible switch network adapter connection. Under certain conditions, the extensible switch interface creates a validation port before it creates the operational port for a Hyper-V child partition. The validation port is used to validate and verify settings for the operational port that will be connected to the extensible switch virtual machine (VM) network adapter of the child partition.

Note In Hyper-V, a child partition is also known as a VM.

This validation port is created under the following conditions:

- The VM is first created. Once the VM is powered on, the validation port is deleted and the operational port is created in its place.

- The VM enters a saved state. When the VM is restored and powered on, the validation port is deleted and the operational port is created in its place.

  For more information, see Hyper-V Extensible Switch Save and Restore Operations.

- The VM is stopped and powered down. Once the VM is powered on, the validation port is deleted and the operational port is created in its place.

- The VM is being live migrated to another host computer. Once the VM is created and powered on in the new host computer, the validation port is deleted and the operational port is created in its place.

After the validation port is created, the extensible switch interface issues OID requests to download port policies for the port. Because these ports are created for policy validation and verification, such as when a VM is first configured, it is important that the validation that occurs is appropriate for configuration time rather than run time. Extensions should perform the following types of policy validation for these ports:

- Syntax validation. This validation fails if the values are not properly formatted.

- Range validation. This validation fails if the settings do not conform to the expected range of minimum and maximum values.

- Applicability validation. This validation fails if the settings do not apply to the extensible switch. For example, a policy profile that defines an external network service-level agreement (SLA) would not apply to an extensible switch that does not have access to the external networking interface.

- Conflict detection. The validation fails if the settings conflict with other settings that are already set on the same port.

When the extensible switch extension validates port and policy settings for a validation port, it must follow these guidelines:

- Because the validation port is temporal, the extension must not validate and fail policy and configuration settings that cannot be currently satisfied by the extensible switch.

  For example, an extensible switch, which supports a maximum of 10 gigabits of bandwidth, may currently only have 1 gigabit of bandwidth available for reservation. The extension does not fail the validation of a port property that is reserving more than 1 gigabit of bandwidth. This kind of validation should instead occur when the operational port is created. This is because the settings that are being validated may still be applied to an operational port in which the bandwidth is available. This allows system administrators to initially configure VMs without being restricted by run-time constraints.

- The extension must not allocate or reserve resources for the validation port. For example, bandwidth reservation settings on a validation port should not deduct from the available bandwidth of the extensible switch. Reservation should occur only when the operational port is created.

For more information on extensible switch operational ports, see Operational Ports.

Operational Ports

Article • 12/15/2021

Starting with NDIS 6.30 in Windows Server 2012, the extensible switch interface creates an operational port to host an extensible switch network adapter connection. When an extensible switch port is created, it is assigned a port type. This port type is in effect after the port is created and before it is torn down. For ports assigned to Hyper-V child partitions, the operational port type stays in effect while the partition is running and operational.

Operational port types define the type of extensible switch network adapter that can connect to it. The extensible switch interface defines the following operational port types:

NdisSwitchPortTypeExternal
This is a port that is configured to be connected to the external network adapter of the extensible switch. This adapter is exposed in the management operating system that runs in the Hyper-V parent partition.

The external network adapter provides a connection to the physical network interface that is available on the host. The external network adapter can be accessed by the Hyper-V parent partition and all child partitions.

Note An extensible switch supports only one external network adapter connection.

NdisSwitchPortTypeInternal
This is a port that is configured to be connected to the internal network adapter of the extensible switch. This adapter is exposed in the management operating system that runs in the Hyper-V parent partition.

The internal network adapter provides access to an extensible switch for processes that run in the management operating system. This allows these processes to send or receive packets over the extensible switch.

Note An extensible switch supports only one internal network adapter.

NdisSwitchPortTypeSynthetic
This is a port that is configured to be connected to a synthetic network adapter. This adapter is exposed in a guest operating system that runs in a Hyper-V child partition.

Note A synthetic network adapter is a type of virtual machine (VM) network adapter. This adapter is exposed in a guest operating system that is running Windows Vista or a later version of Windows.

NdisSwitchPortTypeEmulated
This value specifies a port that is configured to be connected to an emulated network adapter. This adapter is exposed in a guest operating system.

Note An emulated network adapter is a type of VM network adapter. This adapter can be exposed in a guest operating system that is running Windows XP or a non-Windows operating system.

Hyper-V Extensible Switch Network Adapters Topics

Article • 12/15/2021

This section includes the following topics that describe the characteristics of extensible switch network adapters and their connections to extensible switch ports:

Overview of Hyper-V Extensible Switch Network Adapters

External Network Adapters

Internal Network Adapters

Virtual Machine Network Adapters

Network Adapter Index Values

Overview of Hyper-V Extensible Switch Network Adapters

Article • 12/15/2021

The Hyper-V extensible switch supports connections from various types of virtual or physical network adapters. The connection to these types of network adapters is made through an extensible switch port. Ports are created before a virtual network adapter connection is made, and are deleted after the network adapter connection is torn down.

For example, when a Hyper-V child partition is started, the extensible switch interface creates a port before the virtual machine (VM) network adapter is exposed within the guest operating system. After the VM network adapter is exposed and enumerated, the extensible switch interface creates a network connection between the VM network adapter and the extensible switch port. If the child partition is stopped, the extensible switch interface first deletes the network connection and then deletes the extensible switch port.

The Hyper-V extensible switch supports connections from the following types of virtual network adapters:

External network adapters
This is an extensible switch network adapter that is exposed in the management operating system that runs in the Hyper-V parent partition. Each extensible switch supports only one external network adapter connection.

The external network adapter provides a connection to the physical network interface that is available on the host. The external network adapter can be accessed by the Hyper-V parent partition and all child partitions.

For more information about this type of network adapter, see External Network Adapters.

Internal network adapters
This is an extensible switch network adapter that is exposed in the management operating system that runs in the Hyper-V parent partition. Each extensible switch supports only one internal network adapter connection.

The internal network adapter provides access to an extensible switch for processes that run in the management operating system. This allows these processes to send or receive packets over the extensible switch.

For more information about this type of network adapter, see Internal Network Adapters.

Virtual machine (VM) network adapters
This is an extensible switch network adapter that is exposed in the guest operating system that runs in the Hyper-V child partition.

Note In Hyper-V, a child partition is also known as a VM.

The VM network adapter supports the following virtualization types:

- The VM network adapter could be a synthetic virtualization of a network adapter (*synthetic network adapter*). In this case, the network virtual service client (NetVSC) that runs in the VM exposes this virtual network adapter. NetVSC forwards packets to and from the extensible switch port over the VM bus (VMBus).

- The VM network adapter could be an emulated virtualization of a physical network adapter (*emulated network adapter*). In this case, the VM network adapter mimics an Intel network adapter and uses hardware emulation to forward packets to and from the extensible switch port.

For more information about this type of network adapter, see Virtual Machine Network Adapters.

Extensible switch network adapter connections are created, updated, and deleted through the following extensible switch OID requests:

OID_SWITCH_NIC_CREATE
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CREATE to notify extensible switch extensions about the creation of a network adapter connection to an extensible switch port. The port must have been previously created through an OID set request of OID_SWITCH_PORT_CREATE.

The OID_SWITCH_NIC_CREATE request only notifies the extension that a new extensible switch network adapter connection is being brought up and that packet traffic may soon begin to occur over the specified port.

The extension can veto the creation notification by returning STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot satisfy its configured policies on the port that is used for the network adapter connection, the extension should veto the creation notification.

If the extension accepts the creation notification, it must forward the OID request down the extensible switch driver stack. The extension monitors the completion status of this

OID request to determine whether underlying extensions have vetoed the creation notification.

When the network adapter connection is created, it is assigned an NDIS_SWITCH_NIC_INDEX value. This index value identifies the network adapter connection on an extensible switch port. Connections to the external, internal, and VM network adapters are assigned an NDIS_SWITCH_NIC_INDEX value of **NDIS_SWITCH_DEFAULT_NIC_INDEX**. Each physical or virtual network adapter that is bound to the external network adapter is assigned an NDIS_SWITCH_NIC_INDEX value in the following way:

- If the physical or virtual network adapter is directly bound to the external network adapter, it is assigned an NDIS_SWITCH_NIC_INDEX value of one.

- If the physical network adapter is part of an extensible switch team, it is assigned an NDIS_SWITCH_NIC_INDEX value that is greater than or equal to one. An extensible switch team is a configuration in which a team of one or more physical network adapters are bound to the extensible switch external network adapter.

For more information about the different configurations in which physical network adapters can be bound to the external network adapter, see Types of Physical Network Adapter Configurations.

For more information on NDIS_SWITCH_NIC_INDEX values, see Network Adapter Index Values.

Note The extension cannot generate or forward packets over the network adapter connection until the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT.

OID_SWITCH_NIC_CONNECT
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT to notify extensible switch extensions that an extensible switch network adapter connection is fully operational.

The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

After the OID request has completed with NDIS_STATUS_SUCCESS, the network adapter connection and extensible switch port are fully operational. When the network adapter connection is in this state, the extension can do the following:

- Generate or forward packet traffic to the port's network adapter connection.

- Issue extensible switch OIDs or status indications that use the port as the source port.
- Call *ReferenceSwitchNic* to increment a reference counter for the network adapter connection. The extensible switch interface will not tear down a network adapter connection while the reference counter has a nonzero value.

OID_SWITCH_NIC_UPDATED
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_UPDATED to notify extensible switch extensions that the parameters for an extensible switch network adapter have been updated. The OID will only be issued for NICs that have already been connected, and have not yet begun the disconnect process. These run-time configuration changes can include *NicFriendlyName*, *MTU*, *NetCfgInstanceId*, *PermanentMacAddress*, *VMMacAddress*, *CurrentMacAddress*, and *VFAssigned*.

The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

OID_SWITCH_NIC_DISCONNECT
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DISCONNECT to notify extensible switch extensions that an extensible switch network adapter connection is being torn down. After the connection has been completely torn down, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DELETE.

The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

After the extension forwards this OID request, it can no longer generate or forward packets to the port on which the network adapter connection is being torn down. Also, the extension can no longer call *ReferenceSwitchNic* for the network adapter connection.

OID_SWITCH_NIC_DELETE
The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DELETE to notify extensible switch extensions that an extensible switch network adapter connection has been torn down and deleted. This OID request is only issued for network connections for which an OID set request of OID_SWITCH_NIC_DISCONNECT was previously issued.

Note The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

After this OID request is completed, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_TEARDOWN to start the deletion process for the port that was used for the network adapter connection.

The extension must always forward this OID set request down the extensible switch driver stack. The extension must not fail the request.

The extensible switch interface maintains a reference counter for each network adapter connection that has been created. A network adapter connection will not be deleted if its reference counter has a nonzero value. The interface provides the following handler functions for incrementing or decrementing the reference counter of an extensible switch network adapter connection:

ReferenceSwitchNic
The extensible switch extension calls this function to increment a network adapter connection's reference counter. Although the reference counter has a nonzero value, the extensible switch interface does not delete the network adapter connection.

The extension should call *ReferenceSwitchNic* before it performs the following operations:

- Forwards an OID_SWITCH_NIC_REQUEST request down the extensible switch driver stack to an underlying external adapter.

- Forwards an **NDIS_STATUS_SWITCH_NIC_STATUS** status indication up the extensible switch driver stack from an underlying external adapter.

Note The extension must not call *ReferenceSwitchNic* for a network adapter connection after it receives an OID set request of OID_SWITCH_NIC_DISCONNECT for that connection.

DereferenceSwitchNic
The extensible switch extension calls this function to decrement a port's reference counter.

If the extension calls *ReferenceSwitchNic*, it must call *DereferenceSwitchNic* after the OID_SWITCH_NIC_REQUEST or **NDIS_STATUS_SWITCH_NIC_STATUS** indication have completed.

External Network Adapters

Article • 12/15/2021

The external network adapter is exposed in the management operating system that runs in the Hyper-V parent partition. The external network adapter provides the connection to a Hyper-V external network. This network forwards packet traffic over the physical network interface of the host.

The external network is accessed by the Hyper-V parent partition and all child partitions that are connected to the extensible switch. Each instance of the extensible switch supports no more than one external network adapter connection.

The external network adapter is a virtual representation of the underlying physical network adapter on the host. The external network adapter forwards packets, object identifier (OIDs) requests, and NDIS status indications to and from one or more underlying physical network adapters.

Internally, the external network adapter binds to various configurations of underlying physical network adapters. Each of these configurations provides access to the external network interface through one or more physical network adapters. For more information about these physical adapter configurations, see Types of Physical Network Adapter Configurations.

If the extensible switch is configured to provide an external network adapter connection, the following steps occur when the switch is started:

1. The protocol edge of the extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_CREATE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a port is being created for the external network adapter.

2. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CREATE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a network connection for the external network adapter is being created for the port that was previously created.

3. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a network connection for the external network adapter is connected and operational. At this

point, the extension can inspect, inject, and forward packets to the port that is connected to the external network adapter.

The following steps occur when the extensible switch with an external network adapter connection is stopped:

1. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DISCONNECT down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the connection to the external network adapter is being torn down.

2. After all packet traffic and OID requests that target the network connection are completed, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DELETE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the connection to the external network adapter has been gracefully torn down and deleted.

3. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_TEARDOWN down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the port that was used for the external network adapter connection is being torn down.

4. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_DELETE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the port that was used for the external network adapter connection has been torn down and deleted.

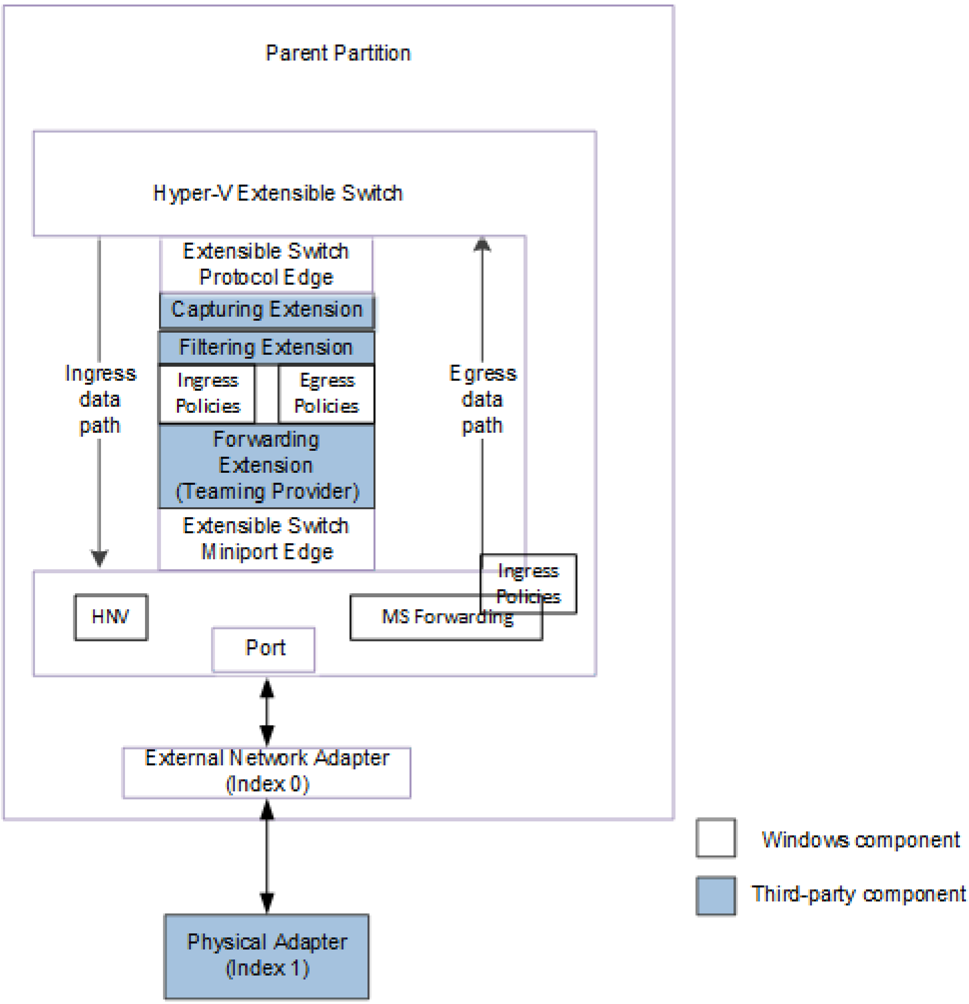
Types of Physical Network Adapter Configurations

Article • 12/15/2021

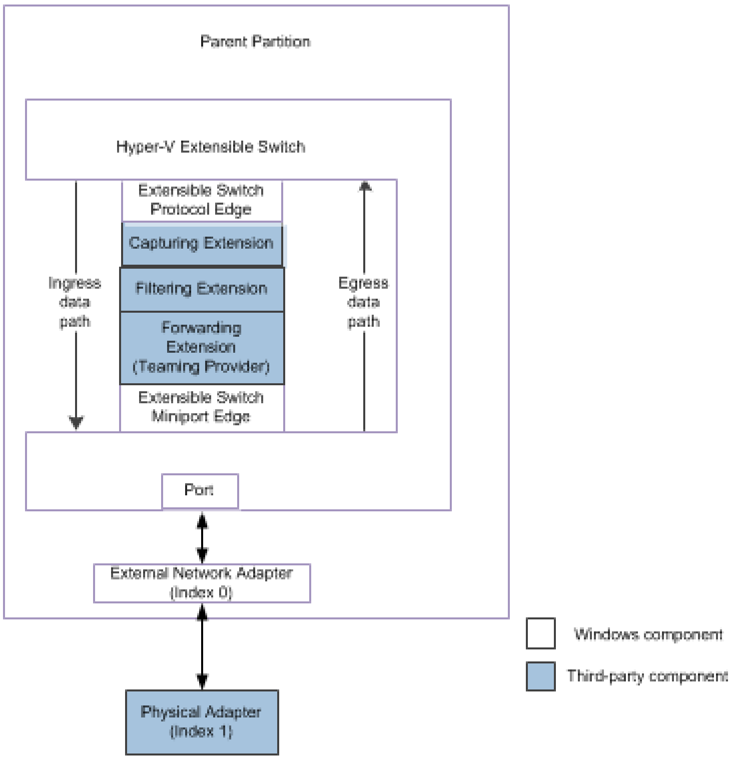
The Hyper-V extensible switch architecture supports the connection to a single external network adapter for access to the underlying physical medium. The external network adapter can be bound to one of the following physical network adapter configurations:

- The external network adapter can be bound to a single underlying physical network adapter. In this configuration, an extensible switch extension is exposed to and manages only one underlying network adapter on the host.

  The following figure shows the extensible switch configuration in which the external network adapter is bound to a single physical network adapter for NDIS 6.40 (Windows Server 2012 R2) and later.



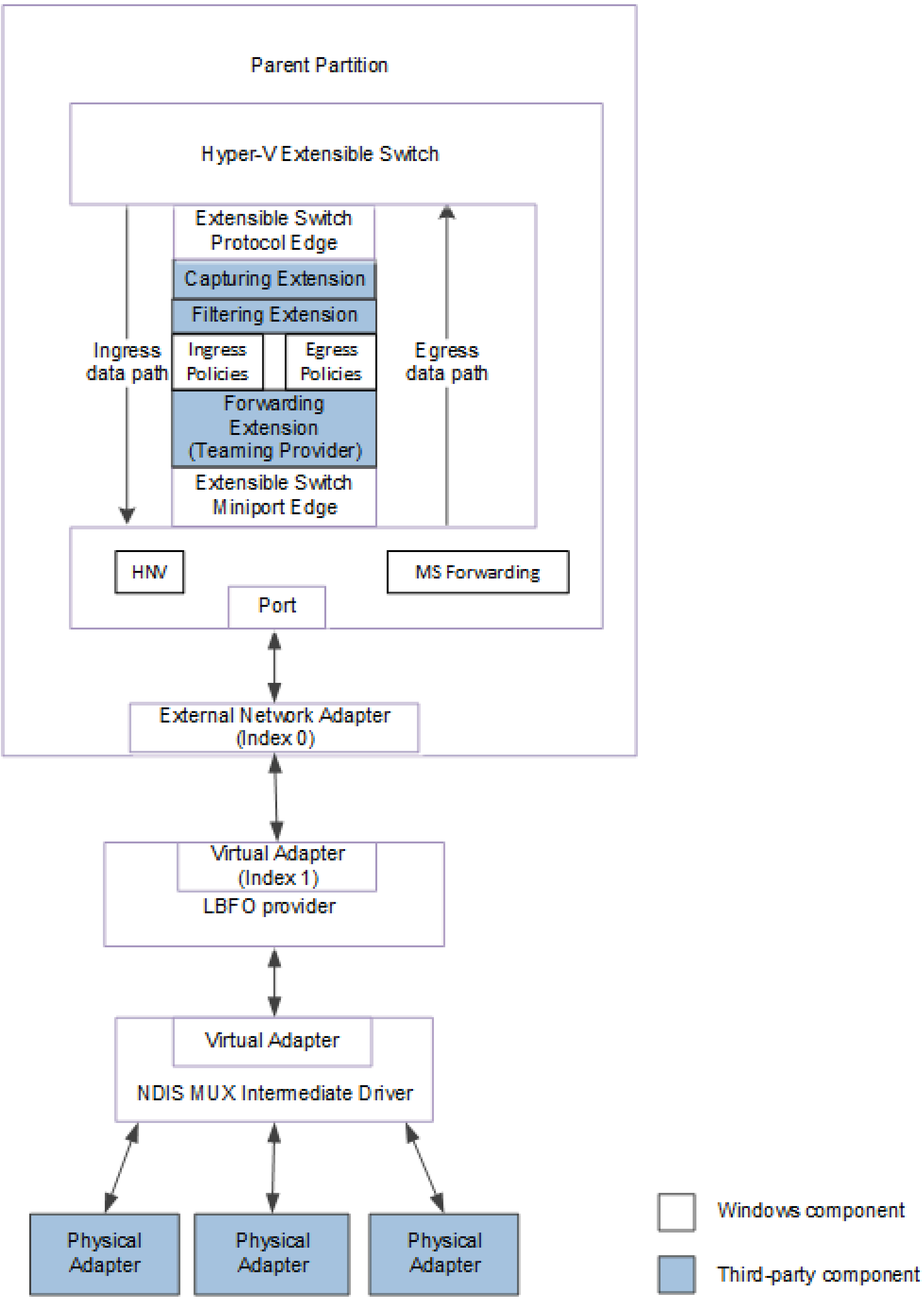
  The following figure shows the extensible switch configuration in which the external network adapter is bound to a single physical network adapter for NDIS 6.30 (Windows Server 2012).


Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
Port
External Network Adapter (Index 0)
Physical Adapter (Index 1)
Windows component
Third-party component

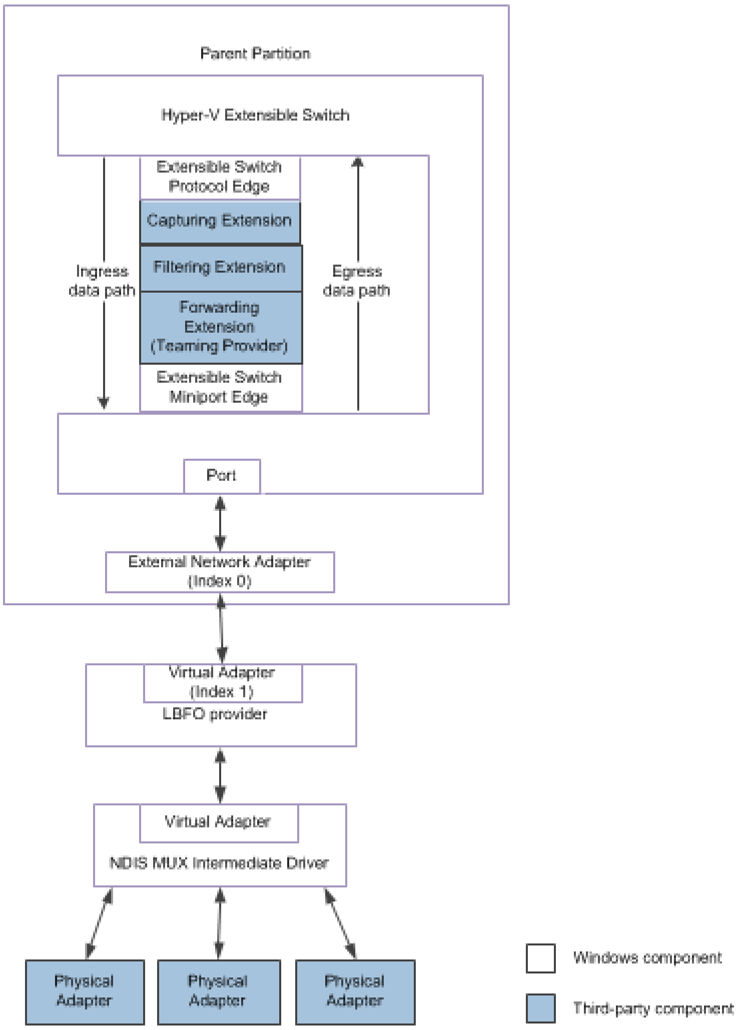
- The external network adapter can be bound to the virtual miniport edge of a load balancing failover (LBFO) provider. This is an NDIS filter driver that is layered above an NDIS multiplexer (MUX) driver, which may be bound to a team of one or more physical networks on the host. This configuration is known as an *LBFO team*.

  In this configuration, the extensible switch extensions are exposed to only the underlying virtual miniport edge as a network adapter. This allows the provider to support an LBFO solution by binding to multiple physical network adapters. These physical network adapters are not managed by a forwarding extension that runs in the extensible switch driver stack.

  The following figure shows an example of an LBFO team configuration for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows an example of an LBFO team configuration for NDIS 6.30 (Windows Server 2012).

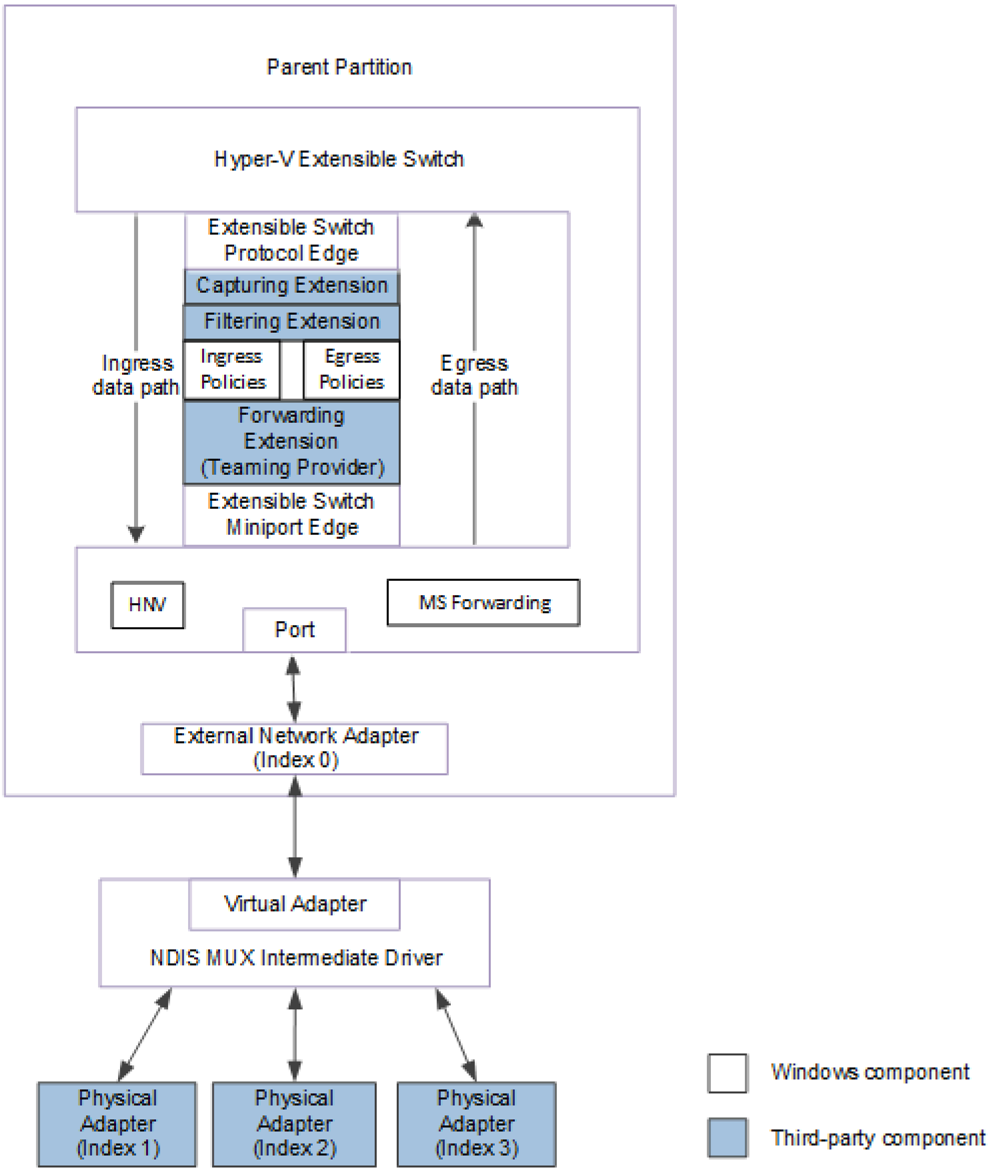

Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
Port
External Network Adapter (Index 0)
Virtual Adapter (Index 1)
LBFO provider
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter
Physical Adapter
Physical Adapter
Windows component
Third-party component

Note To extensible switch extensions, an underlying LBFO team appears as a single virtual network adapter that is bound to the external network adapter.

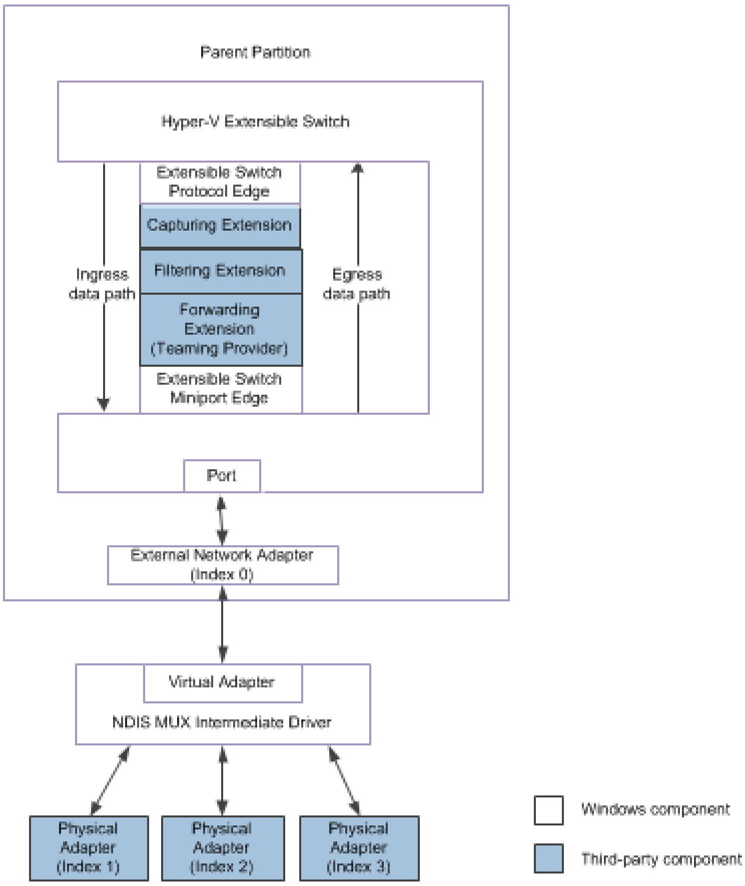
- The external network adapter can be bound to the virtual miniport edge of an NDIS MUX intermediate driver. The MUX driver is bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*.

  In this configuration, an extensible switch extension is exposed to every network adapter in the team. This allows the extension to manage the configuration and use of individual network adapters in the team. For example, the extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters.

  The following figure shows an example of an extensible switch team for NDIS 6.40 (Windows Server 2012 R2) and later.


Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Ingress Policies
Egress Policies
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
HNV
Port
MS Forwarding
External Network Adapter (Index 0)
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index 3)
Windows component
Third-party component

The following figure shows an example of an extensible switch team for NDIS 6.30 (Windows Server 2012).


Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
Port
External Network Adapter (Index 0)
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index 3)
Windows component
Third-party component

For more information on MUX drivers, see NDIS MUX Intermediate Drivers.

Internal Network Adapters

Article • 12/15/2021

The internal network adapter is exposed in the management operating system that runs in the Hyper-V parent partition. This type of network adapter provides access to an extensible switch for processes that run in the management operating system. This allows these processes to send or receive packets over the extensible switch.

If the extensible switch is configured to provide an internal network adapter connection, the following steps occur when the switch is started:

1. The protocol edge of the extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_CREATE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a port is being created for the internal network adapter

2. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CREATE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a network connection for the internal network adapter is being created for the port that was previously created.

3. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a network connection for the internal network adapter is connected and operational. At this point, the extension can inspect, inject, and forward packets to the port that is connected to the internal network adapter.

The following steps occur when the extensible switch with an internal network adapter connection is stopped:

1. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DISCONNECT down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the connection to the internal network adapter is being torn down.

2. After all packet traffic and OID requests that target the network connection are completed, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DELETE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the connection to the internal network adapter has been gracefully torn down and deleted.

3. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_TEARDOWN down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the port that was used for the internal network adapter connection is being torn down.

4. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_DELETE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the port that was used for the internal network adapter connection has been torn down and deleted.

Virtual Machine Network Adapters

Article • 12/15/2021

The virtual machine (VM) network adapter is exposed in the guest operating system that runs in the Hyper-V child partition.

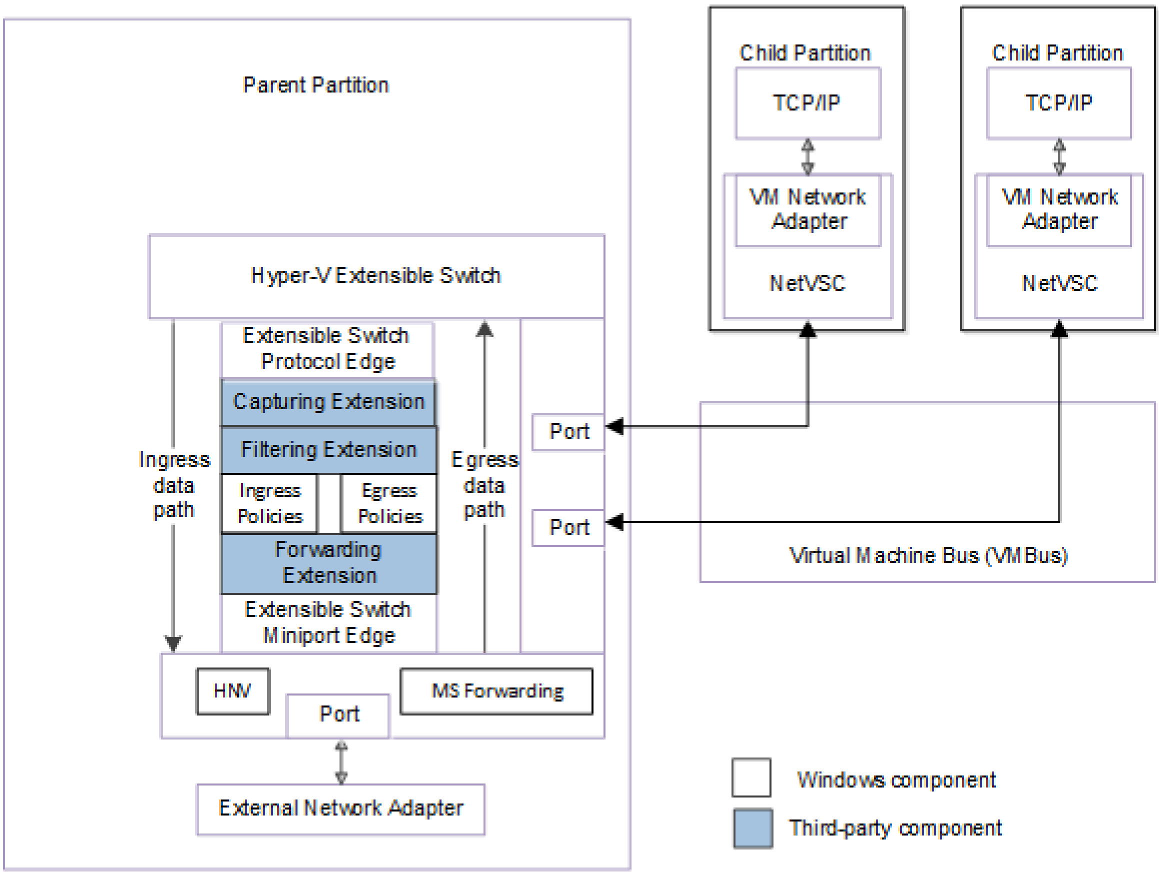
Note In Hyper-V, a child partition is also known as a VM.

The VM network adapter supports the following virtualization types:

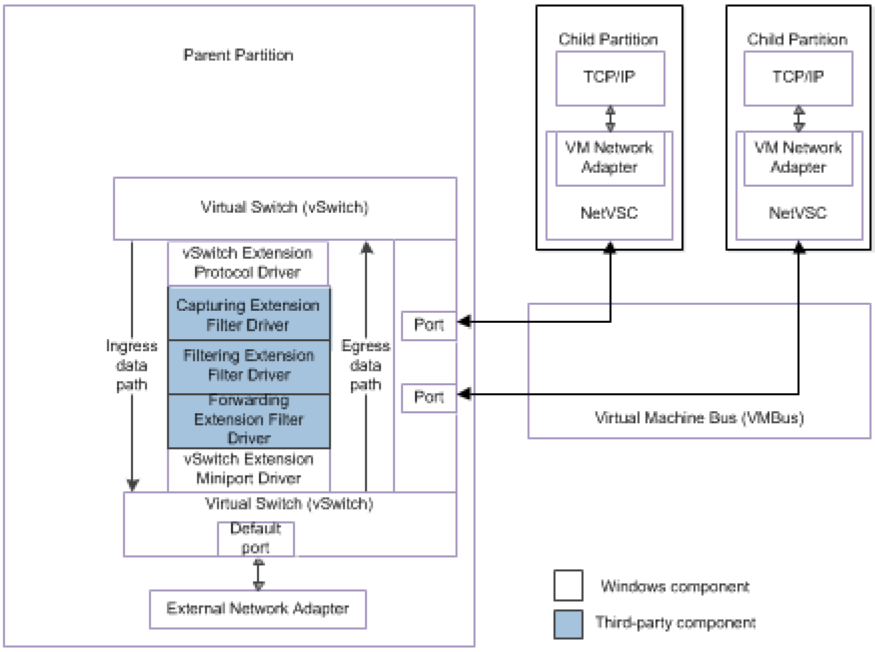
- The VM network adapter could be a synthetic virtualization of a network adapter (*synthetic network adapter*). In this case, the network virtual service client (NetVSC) that runs in the VM exposes this virtual network adapter. NetVSC forwards packets to and from the extensible switch port over the VM bus (VMBus).

- The VM network adapter could be an emulated virtualization of a physical network adapter (*emulated network adapter*). In this case, the VM network adapter mimics an Intel network adapter and uses hardware emulation to forward packets to and from the extensible switch port.

The following figure shows the interface between VM network adapters and the extensible switch NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the interface between VM network adapters and the extensible switch for NDIS 6.30 (Windows Server 2012).



The following steps occur when the user starts a Hyper-V VM:

1. The protocol edge of the extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_CREATE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a port is being created for the VM.

2. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CREATE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a network connection for the VM network adapter is being created for the VM port that was previously created.

3. When the networking stacks are operational and have bound to the VM network adapter, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that a network connection for the VM network adapter is connected and operational. At this point, the extension can inspect, inject, and forward packets to the port that is connected to the VM network adapter.

The following steps occur when the user stops a Hyper-V VM:

1. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DISCONNECT down the extensible switch driver stack. This OID

request notifies the underlying extensible switch extensions that the connection to the VM network adapter is being torn down.

2. After all packet traffic and OID requests that target the network connection are completed, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_DELETE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the connection to the VM network adapter has been gracefully torn down and deleted.

3. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_TEARDOWN down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the port that was used for the VM network adapter connection is being torn down.

4. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_DELETE down the extensible switch driver stack. This OID request notifies the underlying extensible switch extensions that the VM port has been torn down and deleted.

Network Adapter Index Values

Article • 12/15/2021

In the Hyper-V extensible switch interface, each network adapter that is connected to a port is assigned an NDIS_SWITCH_NIC_INDEX value. This index value identifies the network connection on an extensible switch port.

The index value is unique for each network adapter connection to a port. Although most network adapters require only one index value, the port connection to the external network adapter may be assigned multiple index values. For example, if the external network adapter is bound to a team of physical network adapters, the external network adapter and each physical network adapter is assigned a unique index value.

The connections to network adapters in the extensible switch are identified through the following NDIS_SWITCH_NIC_INDEX values:

NDIS_SWITCH_DEFAULT_NIC_INDEX
This index value specifies the index for the network adapter that is connected to an extensible switch port. This index value applies to any network adapter that is directly connected to an extensible switch port, such as an external network adapter, internal network adapter, or virtual machine (VM) network adapter.

Note NDIS_SWITCH_DEFAULT_NIC_INDEX is defined to be zero.

1-32
This index value specifies the index for an underlying physical network adapter that is bound to the extensible switch external network adapter. Index values are assigned based on the following configurations:

- If the external network adapter is bound to a single physical network adapter, it is assigned an index value of one.

- If the external network adapter is bound to a load balancing fail over (LBFO) team of physical network adapters, the entire team is assigned an index value of one. An LBFO team is a configuration in which the external network adapter is bound to the virtual miniport edge of an LBFO provider. The LBFO provider itself can bind to a team of one or more physical network adapters.

  Note To extensible switch extensions, an underlying LBFO team appears as a single network adapter that is bound to the external network adapter.

- If the external network adapter is bound to an extensible switch team of physical network adapters, each adapter in the team is assigned a unique index value that is greater than or equal to one. An extensible switch team is a configuration in which a team of one or more physical network adapters is bound to the external network adapter.

For more information about the different configurations in which physical network adapters can be bound to the external network adapter, see Types of Physical Network Adapter Configurations.

Hyper-V Extensible Switch Port and Network Adapter States

Article • 12/15/2021

The Hyper-V extensible switch interface manages the lifetime of the following components:

Hyper-V Extensible Switch Ports
Each network adapter connection to the extensible switch is represented by a port. Ports are created when a Hyper-V child partition is configured to connect to an instance of an extensible switch. Depending on the switch type, ports are also created for the external and internal network adapter connections. For more information about switch types, see Overview of the Hyper-V Extensible Switch.

Each port is used to hold the configuration for the network interface connection. If the configuration for the network interface connection is removed or the child partition is stopped, the port is torn down and deleted.

For more information about this component, see Hyper-V Extensible Switch Ports.

Hyper-V Extensible Switch Network Adapters
These are virtual network adapters that connect to the extensible switch port. These virtual network adapters are exposed within the Hyper-V child and parent partitions. This includes the virtual machine (VM) network adapter exposed in a child partition and the external network adapter that is teamed with the underlying physical network adapter.

Each network adapter connection requires a corresponding extensible switch port. The port must have been created before the network adapter connection is brought up. Similarly, the network adapter connection must be deleted before the port can be torn down and deleted.

Note In some situations, extensible switch ports could be created and deleted without ever having a network adapter connection.

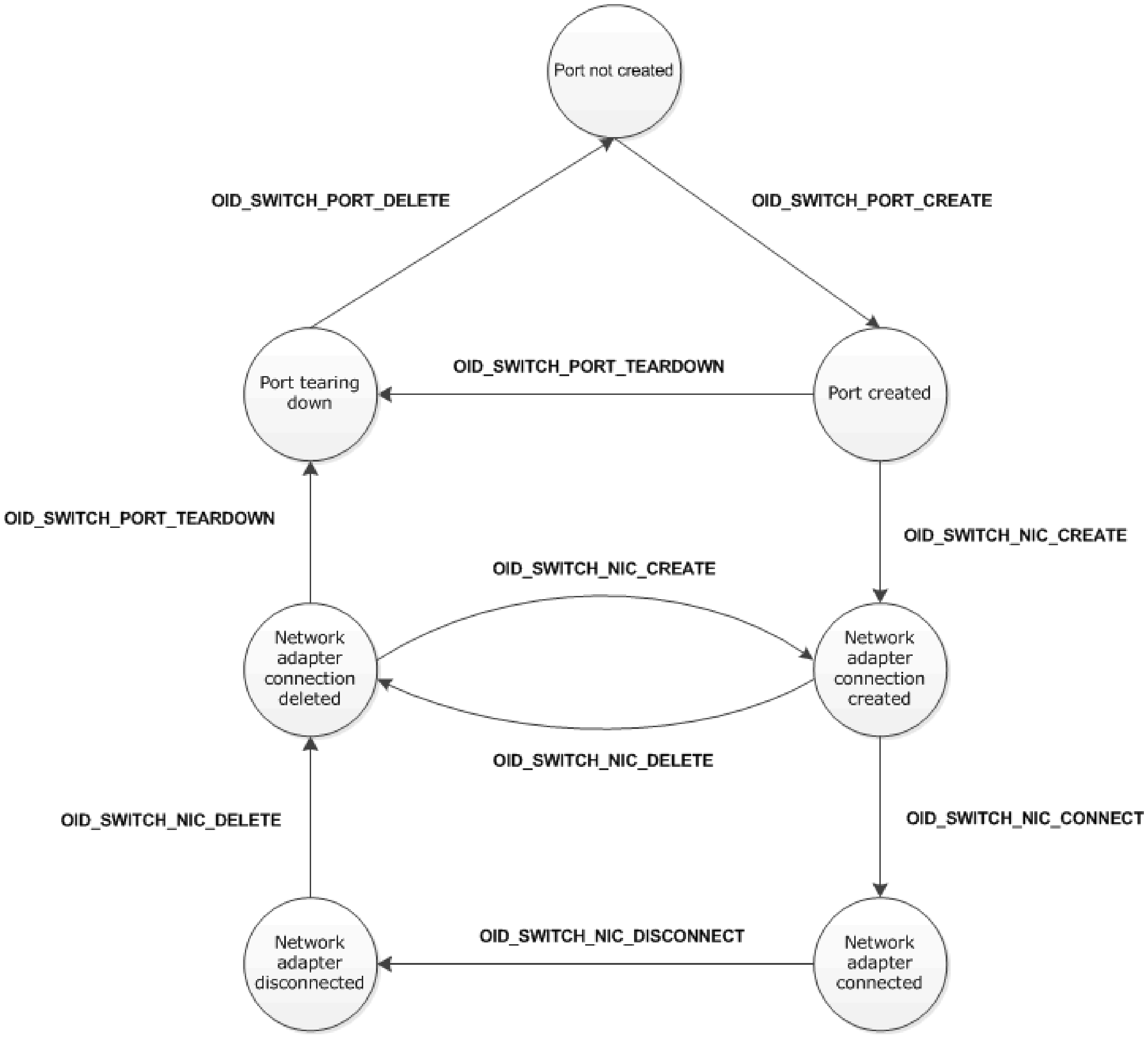
For example, when a Hyper-V child partition is started, the extensible switch interface creates a port before the VM network adapter is exposed within the guest operating system. After the VM network adapter is exposed and enumerated, the extensible switch interface creates a network connection between the VM network adapter and the extensible switch port. If the child partition is stopped, the extensible switch interface first deletes the network connection and then deletes the extensible switch port.

For more information about this component, see Hyper-V Extensible Switch Network Adapters.

When the extensible switch interface creates, deletes, or changes the configuration of these components, it issues object identifier (OID) set requests down the extensible switch driver stack. This operation is performed so that underlying extensible switch extensions can be notified about the state of the component and its configuration. Each OID set request results in a state transition for these components.

When an extension is bound and enabled on an extensible switch instance, it can issue OIDs to discover the existing port and network adapter connection configuration of the switch.

The following diagram shows the various states for the extensible switch port and network adapter connection components. The diagram also shows the OID set requests that cause the state transition for the component.



The following list describes the various states of the extensible switch port and network adapter connection components:

Port not created
In this state, an extensible switch port does not exist on the extensible switch. OID requests that target a previously created port cannot be issued after the port has entered this state.

Port created
When the extensible switch interface issues an OID set request of OID_SWITCH_PORT_CREATE, the port is created on the extensible switch. In this state, the extensible switch interface and extension can issue OID requests that target the port.

For more information about OID traffic through the extensible switch driver stack, see Hyper-V Extensible Switch Control Path.

Note An underlying extension can fail the OID set request and veto the port creation. The extension does this by completing the OID request with STATUS_DATA_NOT_ACCEPTED. If this is done, the port is not created on the extensible switch. For more information on this procedure, see Hyper-V Extensible Switch Ports.

Network adapter connection created
When the extensible switch interface issues an OID set request of OID_SWITCH_NIC_CREATE, the network adapter connection to the port is created on the extensible switch. In this state, the extensible switch interface can do the following:

- Issue OID requests that target the network adapter connection.
- Forward packet traffic to or from the network adapter connection.

It is also possible for a new adapter to connect to an existing port without going through a port teardown and create sequence.

In this state, the extension must forward these packets and OID requests through the extensible switch extension stack. However, the extension cannot originate or redirect packets or OID requests to other network adapter connections on the extensible switch.

Note In this state, the extension must not issue OID requests or originate packet traffic to the network adapter connection.

For more information about OID traffic through the extensible switch driver stack, see Hyper-V Extensible Switch Control Path.

For more information about packet traffic through the extensible switch driver stack, see Hyper-V Extensible Switch Data Path.

Note An underlying extension can fail the OID set request and veto the creation of the network adapter connection. If so, the connection is not created on the extensible switch port. For more information on this procedure, see Hyper-V Extensible Switch Network Adapters.

Network adapter connected
When the extensible switch interface issues an OID set request of OID_SWITCH_NIC_CONNECT, the network adapter is fully connected to the extensible switch port. In this state, the extension can now do the following:

- Issue OID requests that target the network adapter connection.
- Originate packet traffic to the network adapter connection.
- Redirect packet traffic to the network adapter connection. For example, the extension can redirect packets from one network adapter connection to another connection on the extensible switch.

  Note Only forwarding extensions can perform this operation. For more information, see Forwarding Extensions.

Network adapter disconnected
When the extensible switch interface issues an OID set request of OID_SWITCH_NIC_DISCONNECT, the network adapter is being disconnected from the extensible switch port. For example, this OID request is issued when the child partition, which exposed a VM network adapter, is stopped or the external network adapter is disabled.

In this state, the extensible switch extension can no longer originate packets or OID requests that target the connection. Also, forwarding extensions can no longer redirect packets to the connection.

Note Pending packets and OID requests that were issued by the extensible switch interface before the connection became disconnected may still be delivered to the extension. However, the extension must forward the packets and OID requests without making any modifications.

Network adapter connection deleted
After all packet traffic and OID requests that target the network adapter connection are completed, the extensible switch interface issues an OID set request of OID_SWITCH_NIC_DELETE to delete the connection from the extensible switch.

In this state, the extensible switch interface will no longer issue packets or OID requests that target the connection.

Port tearing down
When the extensible switch interface issues an OID set request of

OID_SWITCH_PORT_TEARDOWN, the extensible switch port is being torn down in preparation to being deleted.

In this state, the extensible switch extension can no longer originate OID requests that target the port.

Note Pending OID requests that were issued by the extensible switch interface before the port started its tear down process may still be delivered to the extension. However, the extension must forward the OID requests without making any modifications.

After all pending OID requests that target the port are completed, the extensible switch interface issues an OID set request of OID_SWITCH_PORT_DELETE. This causes the port to transition to a *Port not created* state.

The extension can call an extensible switch handler function to increment or decrement a reference counter on a port or network adapter connection component. While a component's reference counter is nonzero, the extensible switch interface cannot delete the component.

The extension can call *ReferenceSwitchPort* or *DereferenceSwitchPort* to increment or decrement a reference counter for an extensible switch port. These calls can be made after the port has reached the *Port created* state. These calls must not be made after the port has reached the *Port tearing down* or *Port not created* states.

The extension can call *ReferenceSwitchNic* or *DereferenceSwitchNic* to increment or decrement a reference counter for an extensible switch network adapter connection. These calls can be made after the connection has reached the *Network adapter connected* state. These calls must not be made after the connection has reached the *Network adapter disconnected* or *Network adapter deleted* states.

The following table describes the operations that are allowed based on the state of the extensible switch port or network adapter connection components.

Component state	Calls to *ReferenceSwitchPort* or *DereferenceSwitchPort* allowed?	Calls to *ReferenceSwitchNic* or *DereferenceSwitchNic* allowed?
Port not created	No	No
Port created	Yes	No
Network adapter connection created	Yes	No
Network adapter connected	Yes	Yes

Component state	Calls to *ReferenceSwitchPort* or *DereferenceSwitchPort* allowed?	Calls to *ReferenceSwitchNic* or *DereferenceSwitchNic* allowed?
Network adapter disconnected	Yes	No
Network adapter connection deleted	Yes	No
Port tearing down	No	No

Component state	OID requests from extensible switch allowed for port?	OID requests from extensions allowed for port?	OID requests from extensible switch allowed for network adapter connection?	OID requests from extensions allowed for network adapter connection?	Packet traffic from extensible switch allowed over network adapter connection?	Packet traffic from extensions allowed over network adapter connection?
Port not created	No	No	No	No	No	No
Port created	Yes	Yes	No	No	No	No
Network adapter connection created	Yes	Yes	Yes	No	Yes	No
Network adapter connected	Yes	Yes	Yes	Yes	Yes	Yes
Network adapter disconnected	Yes	Yes	Yes	No	Yes	No
Network adapter connection deleted	Yes	Yes	No	No	No	No
Port tearing down	Yes	No	No	No	No	No

Hyper-V Extensible Switch Data Path Topics

Article • 12/15/2021

This section discusses the Hyper-V extensible switch data path that packets move across. This section includes the following topics:

Overview of the Hyper-V Extensible Switch Data Path

Hyper-V Extensible Switch Forwarding Context

Hyper-V Extensible Switch Send and Receive Flags

Overview of the Hyper-V Extensible Switch Data Path

Article • 12/15/2021

This section describes the Hyper-V extensible switch data path over which packets move to or from extensible switch ports. It includes the following topics:

Packet Flow through the Extensible Switch Data Path

Packet Management Guidelines for the Extensible Switch Data Path

Packet Flow through the Extensible Switch Data Path

Article • 12/15/2021

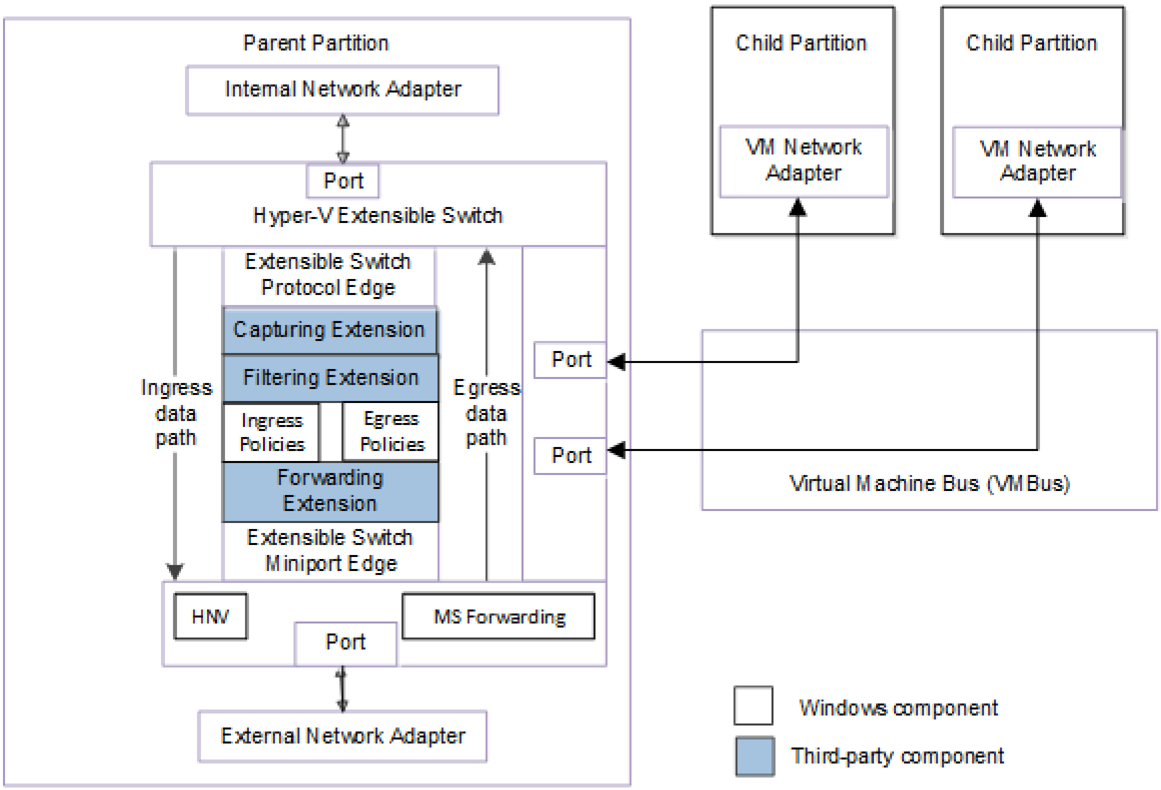
This topic describes how packets move to or from extensible switch ports through the Hyper-V extensible switch data path.

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*. For more information about the extensions, see Hyper-V Extensible Switch Extensions.

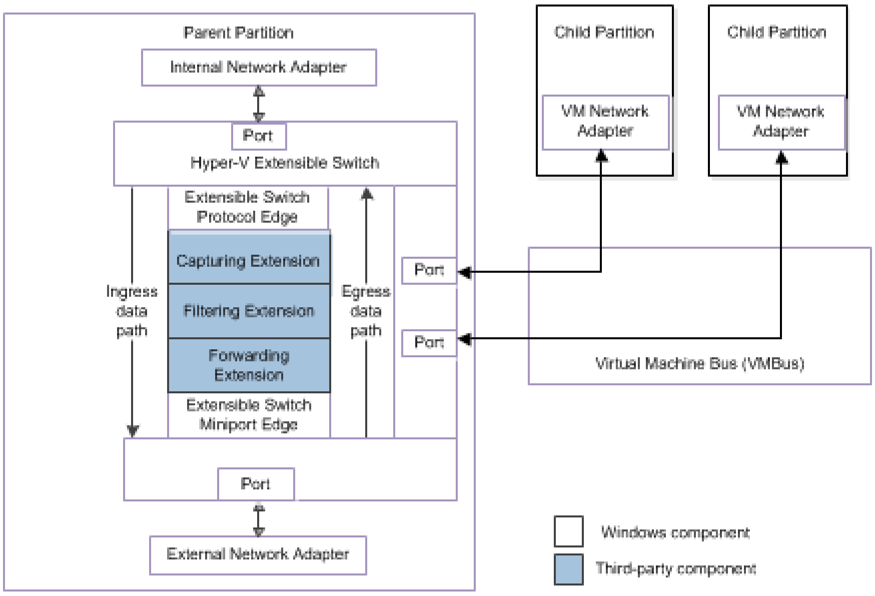
Note This page assumes that you are familiar with the information in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

All packet traffic that arrives at the extensible switch from its ports follows the same path through the extensible switch driver stack. For example, packet traffic received from the external network adapter connection or sent from a virtual machine (VM) network adapter connection moves through the same data path.

The following figure shows the extensible switch data path for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the extensible switch data path for NDIS 6.30 (Windows Server 2012).



For more information about the components for the extensible switch interface, see Hyper-V Extensible Switch Architecture.

The extensible switch data path has the following parts, listed in the order that packets flow through them:

- Overlying protocol edge
- Ingress data path
- Underlying miniport edge
- Egress data path

Overlying protocol edge

1. Packets arrive at the extensible switch from network adapters that are connected to the switch ports. These packets are first issued as send requests from the protocol edge of the extensible switch down the extensible switch ingress data path.

   The protocol edge of the extensible switch prepares the packets for the ingress data path. The protocol edge allocates a context area for these packets that contains the out-of-band (OOB) extensible switch forwarding context. It populates the OOB data with information about the source port and network adapter connection from which the packet was delivered to the extensible switch.

   For more information about the forwarding context, see Hyper-V Extensible Switch Forwarding Context.

2. In NDIS 6.40 (Windows Server 2012 R2) and later, if the packet is an NVGRE packet from an external network adapter, the extensible switch sets the

NativeForwardingRequired flag in the packet's out-of-band (OOB) information. For more information, see Hybrid Forwarding.

3. If the packet arrived on a port where the traffic has a virtual subnet, the extensible switch sets the **VirtualSubnetId** member of the **NDIS_NET_BUFFER_LIST_VIRTUAL_SUBNET_INFO** structure for the packet.

   Note The virtual subnet could be an HNV subnet or a third-party virtual subnet.

Ingress data path

1. An extension obtains a packet from the ingress data path when its *FilterSendNetBufferLists* function is called. The extension forwards the packet to underlying extensions on the ingress data path by calling **NdisFSendNetBufferLists**. Filtering and forwarding extensions can also drop the packet from the ingress data path by calling **NdisFSendNetBufferListsComplete**.

2. When capturing extensions obtain packets on the ingress data path, they can inspect the packet data. However, capturing extensions must not complete the send requests for packets on the ingress data path. These extensions must always forward the packets to underlying extensions in the extensible switch driver stack.

   A capturing extension can also originate packets on the ingress data path. For example, the extension may originate packets in order to report traffic conditions to a remote monitoring application.

   For more information on originating packets by an extension, see Originating Packet Traffic.

3. When filtering extensions obtain packets on the ingress data path, they can do the following:

   - Drop packets based on custom extensible switch or port policies.

     For more information about these policies, see Hyper-V Extensible Switch Policies.

     Note Packets obtained on the ingress data path do not have destination ports defined in the packet's OOB data. As a result, filtering extensions must only enforce custom policies based on the packet data or the packet's source port or network adapter connection.

- Clone or modify packets obtained from the ingress data path.

- Inject new packets into the ingress data path.

4. In NDIS 6.40 and later, after the capturing and filtering extensions but before the forwarding extension on the ingress data path, the extensible switch does the following:

    - If the packet is an NVGRE packet from an external network adapter, the address in the packet header is a provider address (PA) space address. The extensible switch indicates this by setting the **NativeForwardingRequired** flag in the packet's out-of-band (OOB) information. For more information, see Hybrid Forwarding.

    - The extensible switch applies the built-in ingress policies to the packet. These policies may include ingress access control lists (ACLs), DHCP Guard, and Router Guard.

5. If a forwarding extension is not enabled in the extensible switch driver stack, the destination port array for a packet is determined by the extensible switch.

6. If a forwarding extension is enabled, it must do the following when it obtains packets on the ingress data path:

    - In NDIS 6.40 and later, if the packet is an NVGRE packet (see Hybrid Forwarding), the forwarding extension cannot modify the destination port array in the OOB data of the packet in the ingress data path. However, it can drop the packet.

    - If the packet is not an NVGRE packet, the forwarding extension must add destination ports to the destination port array in the OOB data of the packet.

    - The forwarding extension must drop packets based on standard or custom extensible switch or port policies. Standard switch or port policies include security and virtual LAN (VLAN) properties. If a forwarding extension is not enabled in the extensible switch driver stack, these policies are enforced by the extensible switch.

      Note When the forwarding extension filters packets in the ingress data path, it applies filtering rules based on the source port as well as the destination ports that the extension assigns to the packet.

In addition, the forwarding extension can do the following:

- Clone or modify packets obtained from the ingress data path.

- Inject new packets into the ingress data path.

Underlying miniport edge

1. When the packet arrives at the underlying miniport edge of the extensible switch, the extensible switch applies its built-in policies to the packet. These policies include access control lists (ACLs) and quality of service (QoS) properties. If the packet is not dropped because of these policies, the extensible switch originates a receive indication for the packet and forwards the packet up the egress data path.

   Note If port mirroring is enabled on a port that the packet is to be delivered to, the miniport edge adds a destination port to the packet's OOB data for the mirror port. The miniport edge does this regardless of whether a forwarding extension is installed and enabled in the extensible switch driver stack. The miniport edge only adds the mirror port if it is not already specified in the array of destination ports for the packet.

2. If a forwarding extension is not enabled, the extensible switch determines the destination ports for the packet and add these destination ports to the packet's OOB data before it forwards the packet up the egress data path.

3. In NDIS 6.40 and later, the HNV component performs any needed NVGRE encapsulation or decapsulation after ingress and before egress, so that the forwarding extension can see the packet in encapsulated and decapsulated form. For example, if the packet arrived from an external network adapter and is destined for an internal VM, the forwarding extension obtains the encapsulated packet on ingress and the decapsulated packet on egress.

   Note In the encapsulated packet, the address in the packet header is a provider address (PA) space address. In the decapsulated packet, it is a customer address (CA) space address.

   a. If the packet is an NVGRE packet that arrived from an external network adapter, the Hyper-V Network Virtualization (HNV) component of the extensible switch performs NVGRE decapsulation on the packet. The HNV component determines the destinations for the packet according to HNV policies, and then the extensible switch forwards the packet up the egress data path.

   b. If the packet arrived from an internal VM, the HNV component will perform NVGRE encapsulation on the packet if HNV policies are set for the packet. The HNV component determines the destinations for the packet according to HNV policies, and then the extensible switch forwards the packet up the egress data path.

   c. Otherwise, the forwarding extension forwards the packet up the egress data path.

4. In NDIS 6.30, if a forwarding extension is enabled, it must forward the packet up the egress data path.

Egress data path

1. An extension obtains a packet from the egress data path when its *FilterReceiveNetBufferLists* function is called. The extension forwards the packet to overlying extensions on the egress data path by calling **NdisFIndicateReceiveNetBufferLists**. Filtering and forwarding extensions can also drop the packet from the egress data path by calling **NdisFReturnNetBufferLists**.

2. When the forwarding extension obtains a packet on the egress data path, it can inspect the packet's destination port information in the OOB data.

   Note The extension obtains this information from the OOB data by calling *GetNetBufferListDestinations*.

Based on standard or custom switch or port policies, the extension can exclude the delivery of the packet to one or more destination ports that are contained within the OOB data.

3. In NDIS 6.40 (Windows Server 2012 R2) and later, after the forwarding extension but before the filtering and capturing extensions on the egress data path, the extensible switch applies the built-in egress policies to the packet. These policies may include trunk mode, monitoring mode, egress ACLs, and quality of service (QoS) properties.

4. When filtering extensions obtain a packet on the egress data path, they can inspect the packet's destination port information in the OOB data. Based on custom switch or port policies, the extension can exclude the delivery of the packet to one or more destination ports that are contained within the OOB data.

   If the filtering extension needs to modify the data in a packet, it must first clone the packet without preserving port destinations. Then, the extension must inject the modified packet into the ingress data path. This allows the underlying extensions to enforce policies on the modified packet and the forwarding extension can add port destinations.

   For more information, see Cloning or Packet Traffic.

5. When capturing extensions obtain packets on the egress data path, they can inspect the packet data. However, if the capturing extension needs to originate packets in order to report traffic conditions to a remote monitoring application, it must originate this packet traffic by calling **NdisFSendNetBufferLists** to initiate a send operation on the ingress data path.

6. When the packet arrives at the overlying protocol edge of the extensible switch, the extensible switch interface forwards the packet to all specified destination ports.

7. Once the packet has been forwarded, the interface completes the packet through the same path in reverse. First, the interface calls the extension's *FilterReturnNetBufferLists* function to complete packets forwarded on the egress data path. Then, the interface calls the extension's *FilterSendNetBufferListsComplete* function to complete packets forwarded on the ingress data path.

   When the packet is completed on both the egress and ingress data path, the extension performs any necessary packet cleanup and post-processing that may be required.

Packet Management Guidelines for the Extensible Switch Data Path

Article • 12/15/2021

This topic describes the guidelines that Hyper-V extensible switch extensions must follow for managing packets obtained on the extensible switch data path.

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*. For more information about the extensions, see Hyper-V Extensible Switch Extensions.

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

Extensions must follow these guidelines for packet management in the extensible switch data path:

- Extensions that originate packets must call **NdisFSendNetBufferLists** to initiate a send request on the ingress data path. This must be done in this way to allow for proper forwarding of the packet through the extensible switch.

- A capturing extension can monitor packets on the extensible switch ingress and egress data path. However, this type of extension must always forward packets and must not drop the packets. Also, the capturing extension must not modify the packet data before it forwards the packet.

- On the extensible switch ingress data path, filtering and forwarding extensions can do the following:

  - Filtering extensions can filter packet traffic and enforce only custom port or switch policies for packet delivery through the extensible switch. When the extension filters packets in the ingress data path, it can only apply filtering rules based only on the source port and network adapter connection from which the packet originated. This information is stored in the OOB data of a packet's NET_BUFFER_LIST structure and can be obtained by using the NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL macro.

    Note Packets obtained on the ingress data path do not contain destination ports. Filtering packets based on destination ports can only be done on packets obtained on the egress data path.

  - Forwarding extensions can filter packet traffic and enforce custom and standard port or switch policies for packet delivery through the extensible switch. When the forwarding extension filters packets in the ingress data path, it applies filtering rules based on the source port as well as the destination ports that the forwarding extension assigns to the packet.

- On the extensible switch egress data path, filtering and forwarding extensions can do the following:

  - Filtering extensions can filter packet traffic and enforce only custom port or switch policies for packet delivery through the extensible switch. When the filtering extension filters packets in the egress data path, it can apply filtering rules based only on the destination ports for a packet.

    Destination port data is stored in the OOB data of a packet's **NET_BUFFER_LIST** structure. Extensions obtain this information by calling the *GetNetBufferListDestinations* function.

  - Forwarding extensions can filter packet traffic and enforce custom and standard port or switch policies for packet delivery through the extensible switch. When the forwarding extension filters packets in the egress data path, it can apply filtering rules based on the source or destination ports for a packet.

  - Based on the policies enforced on a packet, the filtering or forwarding extension can exclude the delivery of the packet to one or more destinations. For more information on this procedure, see Excluding Packet Delivery to Extensible Switch Destination Ports.

    Based on the policies enforced on a packet, the forwarding extension can exclude the delivery of the packet to one or more destinations. For more information, see Hybrid Forwarding.

- On the extensible switch egress data path, filtering and forwarding extensions must not do the following:

  - Modify the packet data before forwarding the packet on the egress data path.

    If a filtering extension needs to modify the data in a packet, it must first clone the packet without preserving port destinations. Then, the extension must inject the modified packet into the ingress data path. This allows the underlying extensions to enforce policies on the modified packet and the forwarding extension can add port destinations.

If the forwarding extension needs to modify the data in a packet, it must first clone the packet before it assigns port destinations. After the packet has been modified and port destinations assigned, the extension must inject the modified packet into the ingress data path.

For more information, see Cloning Packet Traffic.

Note If the extension clones a packet that was obtained on the egress data path, it can inject the new packet into the egress data path only if it has not changed the packet data and has preserved the original destination port data.

  - Add destination ports to the packet before forwarding the packet.

    Note Forwarding extensions are allowed to add destination ports to packets obtained on the ingress data path.

  - Inject new or cloned data packets into the egress data path.

- In the standard NDIS data path, non-extensible switch OOB data often has different formats depending on whether the packet is being indicated as a send or a receive. For example, the **NDIS_IPSEC_OFFLOAD_V2_HEADER_NET_BUFFER_LIST_INFO** OOB data is a union of send-and receive–specific structures.

  In the extensible switch data path, all packets move through the extension driver stack as both sends and receives. Because of this, the non-extensible switch OOB data within the packet's **NET_BUFFER_LIST** structure will be in either a send or receive format through the duration of the flow through the driver stack.

  The format of this OOB data depends upon the source extensible switch port from which the packet arrived at the extensible switch. If the source port is connected to the external network adapter, the non-extensible switch OOB data will be in a receive format. For other ports, this OOB data will be in a send format.

  Note If the extension clones a packet's **NET_BUFFER_LIST** structure, it must take the non-extensible switch OOB data into consideration if it adds or modifies the OOB data. The extension must call *CopyNetBufferListInfo* to copy the OOB data associated with the extensible switch data path from a source packet to a cloned packet. This function will maintain the OOB send or receive format when the data is copied to the cloned packet.

- If an extension drops a packet from either the ingress of egress data path, it must call *ReportFilteredNetBufferLists*. When this function is called, the extensible switch interface increments counters and logs events for the dropped or excluded packets.

Hyper-V Extensible Switch Forwarding Context Overview

Article • 12/15/2021

The **NET_BUFFER_LIST** structure for each packet that traverses the Hyper-V extensible switch data path contains out-of-band (OOB) data. This data specifies the source port from where the packet originated, as well as one or more destination ports for packet delivery. This OOB data is known as the *extensible switch forwarding context*.

This section includes the following topics about the extensible switch forwarding context:

Hyper-V Extensible Switch Forwarding Context Data Types

Managing the Hyper-V Extensible Switch Forwarding Context

Hyper-V Extensible Switch Forwarding Context Data Types

Article • 12/15/2021

The NET_BUFFER_LIST structure for each packet that traverses the Hyper-V extensible switch data path contains out-of-band (OOB) data. This data specifies the source port from where the packet originated, as well as one or more destination ports for packet delivery. This OOB data is known as the *extensible switch forwarding context*.

The following data types have been declared to access the extensible switch forwarding context within a packet's NET_BUFFER_LIST structure:

NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO
This is a 64-bit union that contains the forwarding characteristics of a packet. This data includes the identifiers for the source port and network adapter connection from which the packet originated. This data also includes the number of unused elements that are available in the destination port array.

The extensible switch extension can access this data by using the NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL macro.

NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY
This structure defines the destination port array for the packet. Each element in this array is formatted as an NDIS_SWITCH_PORT_DESTINATION structure.

The NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure contains members that specify the current number of the total number of elements as well as the number of used elements in the array.

The extensible switch extension can obtain this array by calling the *GetNetBufferListDestinations* function. If the driver adds or modifies elements in the array for a packet with multiple destination ports, it must call the *UpdateNetBufferListDestinations* function. This function commits those changes to the destination port array in the packet's forwarding context.

Note To commit changes to a packet with only one destination port, it is more efficient for the driver to call the *AddNetBufferListDestination* function.

NDIS_SWITCH_PORT_DESTINATION
This structure defines a destination port for the packet. For packets with a single destination port, there is only one NDIS_SWITCH_PORT_DESTINATION element in the

destination port array. For packets with multiple destination ports, there are one or more of these elements in the array.

After the extensible switch extension has called *GetNetBufferListDestinations* to obtain the packet's destination port array, it can access individual elements in the array by using the **NDIS_SWITCH_PORT_DESTINATION_AT_ARRAY_INDEX** macro.

Managing the Hyper-V Extensible Switch Forwarding Context

Article • 12/15/2021

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

The **NET_BUFFER_LIST** structure for each packet that traverses the Hyper-V extensible switch data path contains out-of-band (OOB) data. This data specifies the source port from where the packet originated, as well as one or more destination ports for packet delivery. This OOB data is known as the *extensible switch forwarding context*.

Note The extensible switch forwarding context is different from the **NET_BUFFER_LIST_CONTEXT** structure. This allows extensions to allocate their own context structures without affecting the forwarding context.

The extensible switch forwarding context is allocated and freed in the following way:

- When a packet arrives at the extensible switch from a network adapter, the extensible switch interface allocates the forwarding context and associates it with the packet's **NET_BUFFER_LIST** structure.

  When the packet is delivered to its destination ports, the interface frees the forwarding context from the packet's **NET_BUFFER_LIST** structure.

- If an extensible switch extension injects a new or cloned packet into the extensible switch data path, it must allocate the forwarding context before it calls **NdisFSendNetBufferLists**.

  After the extension allocates a **NET_BUFFER_LIST** structure for a new or cloned packet, it must call the *AllocateNetBufferListForwardingContext* function to allocate the forwarding context for the packet. When the send packet request is completed, the extension must call *FreeNetBufferListForwardingContext* before it frees or reuses the **NET_BUFFER_LIST** structure.

  Note When the extension calls *AllocateNetBufferListForwardingContext*, the source port for the packet is set to **NDIS_SWITCH_DEFAULT_PORT_ID**. This specifies that the packet originated from an extension instead of arriving at an extensible switch port. Under certain conditions, the extension may want to change the source port for the packet. For more information, see Modifying a Packet's Extensible Switch Source Port Data.

For more information, see Hyper-V Extensible Switch Send and Receive Operations.

All extensible switch extensions can call the following extensible switch handler functions to access the data within the packet's forwarding context:

AllocateNetBufferListForwardingContext
Allocates the extensible switch forwarding context and prepares a **NET_BUFFER_LIST** structure for send or receive operations within the extensible switch.

CopyNetBufferListInfo
Copies the forwarding context from a source packet's **NET_BUFFER_LIST** structure to a destination packet's **NET_BUFFER_LIST** structure. This data includes the extensible switch source port and network adapter information. The extensible switch destination port information can also be copied to the destination packet.

FreeNetBufferListForwardingContext
Frees the resources in the extensible switch forwarding context of a **NET_BUFFER_LIST** structure. This data was used for send or receive operations in a Hyper-V extensible switch, and was previously allocated by calling the *AllocateNetBufferListForwardingContext* function.

GetNetBufferListDestinations
Returns the destination ports from the forwarding context of a packet's **NET_BUFFER_LIST** structure.

A forwarding extension is responsible for adding destination ports for a packet, unless the packet is an NVGRE packet. (For more information, see Hybrid Forwarding.) The extension calls the following extensible switch handler functions to add or update the destination ports within the packet's forwarding context:

AddNetBufferListDestination
Adds a single destination to the extensible switch forwarding context area for a packet that is specified by a **NET_BUFFER_LIST** structure.

Note This call commits the change to the forwarding context area. In this case, the forwarding extension does not need to call *UpdateNetBufferListDestinations*.

GrowNetBufferListDestinations
Increases the size of the destination port array in the forwarding context area of a packet's **NET_BUFFER_LIST** structure.

UpdateNetBufferListDestinations
Commits modifications that the extension made to one or more extensible switch

destination ports of a packet. This function updates the forwarding context of a packet's **NET_BUFFER_LIST** structure with these changes.

Note After the forwarding extension commits the changes for destination ports to the forwarding context, destination ports cannot be removed and only the **IsExcluded** member of a destination port's **NDIS_SWITCH_PORT_DESTINATION** structure can be changed. For more information, see Excluding Packet Delivery to Extensible Switch Destination Ports.

Related topics

Hyper-V Extensible Switch Forwarding Context

Hyper-V Extensible Switch Forwarding Context Data Types

Hyper-V Extensible Switch Send and Receive Flags

Article • 12/15/2021

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

Packet traffic that moves over the Hyper-V extensible switch data path is obtained by extensions in the following way:

- An extension obtains a packet from the ingress data path when its *FilterSendNetBufferLists* function is called. The extension forwards the packet to underlying extensions on the ingress data path by calling **NdisFSendNetBufferLists**. Filtering and forwarding extensions can also drop the packet from the ingress data path by calling **NdisFSendNetBufferListsComplete**.

- An extension obtains a packet from the egress data path when its *FilterReceiveNetBufferLists* function is called. The extension forwards the packet to overlying extensions on the egress data path by calling **NdisFIndicateReceiveNetBufferLists**. Filtering and forwarding extensions can also drop the packet from the egress data path by calling **NdisFReturnNetBufferLists**.

The following flags may be set in the *SendFlags* parameter of *FilterSendNetBufferLists* or **NdisFSendNetBufferLists**:

NDIS_SEND_FLAGS_SWITCH_SINGLE_SOURCE
If this flag is set, all packets in a linked list of **NET_BUFFER_LIST** structures originated from the same Hyper-V extensible switch source port.

When NDIS calls *FilterSendNetBufferLists*, it will set this flag if the extensible switch extensible interface has grouped multiple packets from the same source port. For the best performance, the extensions should keep this grouping in place and set this flag when it calls **NdisFSendNetBufferLists**. The extension can also add any originated or cloned packets to the linked list of **NET_BUFFER_LIST** structures if the extension uses the same source port as the other packets in the list.

Note If each packet in the linked list of **NET_BUFFER_LIST** structures uses the same source port, the extension should set the **NDIS_SEND_COMPLETE_FLAGS_SWITCH_SINGLE_SOURCE** flag in the *SendCompleteFlags* parameter of **NdisFSendNetBufferListsComplete** when it completes the send request.

NDIS_SEND_FLAGS_SWITCH_DESTINATION_GROUP
If this flag is set, all packets in a linked list of **NET_BUFFER_LIST** structures are to be forwarded to the same extensible switch destination port.

A forwarding extension can use this flag for a linked list of **NET_BUFFER_LIST** structures that it forwards on the ingress data path after it determines each packet's destination ports. This flag is consumed and removed by the underlying miniport edge of the extensible switch before it forwards the packets up the egress data path.

Capturing and filtering extensions cannot use this flag.

Note The forwarding extension only determines the packet's destination ports for non-NVGRE packets. If the packet is an NVGRE packet, the Hyper-V Network Virtualization (HNV) component determines the packet's destination ports and forwards the packet. For more information, see Hybrid Forwarding.

For the best performance, forwarding extensions should set this flag if all packets in the linked list are to be forwarded to the same destination port. By setting this flag, the extension is acknowledging that all packets in the linked list have the same destination port elements in the extensible switch forwarding context.

Note The forwarding extension must not set this flag for a linked list of packets that have multiple destination ports.

The following flags may be set in the *ReceiveFlags* parameter of *FilterReceiveNetBufferLists* or **NdisFIndicateReceiveNetBufferLists**:

NDIS_RECEIVE_FLAGS_SWITCH_SINGLE_SOURCE
If this flag is set, all packets in a linked list of **NET_BUFFER_LIST** structures originated from the same Hyper-V extensible switch source port.

When NDIS calls *FilterReceiveNetBufferLists*, it will set this flag if the extensible switch has grouped multiple packets from the same source port. For the best performance, the extensions should keep this grouping in place and set this flag when it calls **NdisMIndicateReceiveNetBufferLists**. The extensions should also add any originated or cloned packets to the linked list of **NET_BUFFER_LIST** structures if the packet has the same source port as the other packets in the list.

Note If each packet in the linked list of **NET_BUFFER_LIST** structures use the same source port, the extension should set the **NDIS_RETURN_FLAGS_SWITCH_SINGLE_SOURCE** flag in the *ReturnFlags* parameter of *FilterReturnNetBufferLists* when the receive request completes. The extension must set this flag in the *ReturnFlags* parameter if it calls **NdisFReturnNetBufferLists** to return packets that it did not originate or clone.

NDIS_RECEIVE_FLAGS_SWITCH_DESTINATION_GROUP
If this flag is set, all packets in a linked list of **NET_BUFFER_LIST** structures are to be forwarded to the same extensible switch destination port.

When NDIS calls *FilterReceiveNetBufferLists*, it will set this flag if the extensible switch has grouped multiple packets that have the same destination ports. For the best performance, the extensions should keep this grouping in place and set this flag when it calls **NdisMIndicateReceiveNetBufferLists**. The extensions should also add any originated or cloned packets to the linked list of **NET_BUFFER_LIST** structures if the packet has the same destination ports as the other packets in the list.

Note When an extension calls **NdisFIndicateReceiveNetBufferLists**, it must not set the **NDIS_RECEIVE_FLAGS_RESOURCES** flag in the *ReceiveFlags* parameter. The extensible switch interface ignores this flag and will complete the receive indication by calling *FilterReturnNetBufferLists*.

Hyper-V Extensible Switch Control Path Topics

Article • 12/15/2021

This section discusses the control path that Hyper-V extensible switch object identifier (OID) requests and NDIS status indications move across.

This section includes the following topics:

Hyper-V Extensible Switch Control Path for OID Requests

Hyper-V Extensible Switch Control Path for NDIS Status Indications

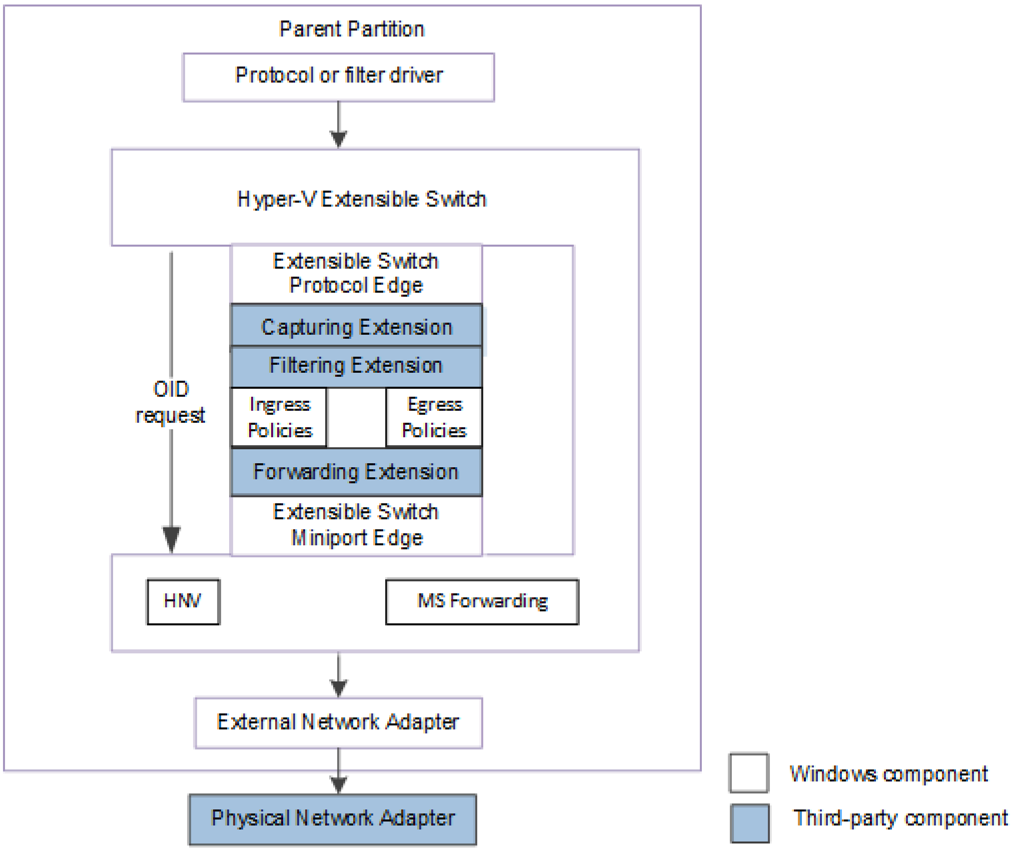
Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

Hyper-V Extensible Switch Control Path for OID Requests

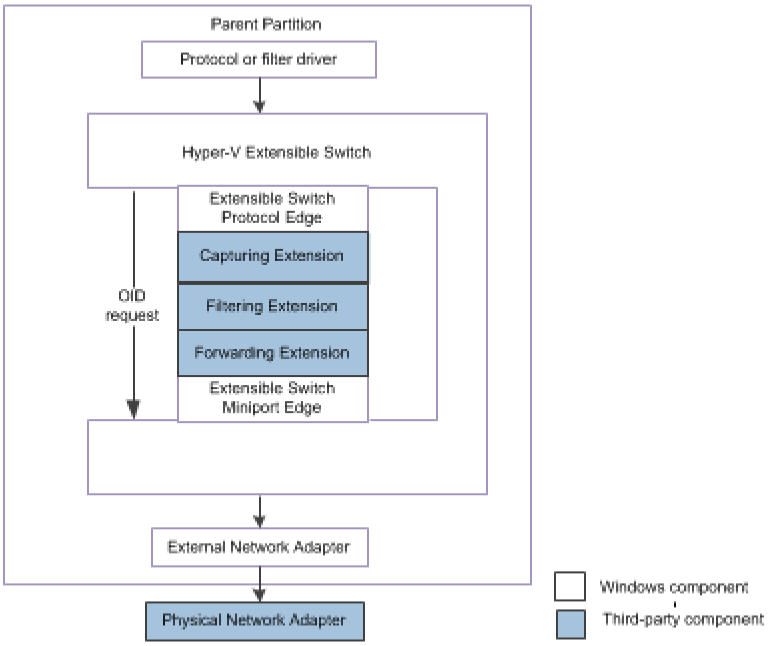
Article • 12/15/2021

This topic discusses the control path that Hyper-V extensible switch object identifier (OID) requests move across.

The following figure shows the extensible switch control path for OID requests for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the extensible switch control path for OID requests for NDIS 6.30 (Windows Server 2012).



Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

Extensible switch extensions, such as filtering and forwarding extensions, are responsible for allowing or rejecting packet traffic based on port or switch policies. In order for these extensions to apply policy decisions, these extensions must be able to do the following:

- Receive the necessary information from the extensible switch interface about the new or updated configuration and state of the extensible switch, its ports, and its network adapter connections.
- Receive the necessary information from the extensible switch interface about the new or updated properties for a switch or port policy.
- Issue OID requests to the extensible switch interface to obtain the current configuration of the extensible switch, its ports, and its network adapter connections.

The extensible switch interface notifies underlying extensions about changes to its component configuration and policy parameters by issuing extensible switch OID set requests. These requests are issued by the protocol edge of the extensible switch to notify underlying extension about these changes. These OID requests move through the extensible switch driver stack to the underlying miniport edge of the extensible switch.

The miniport edge of the extensible switch is responsible for completing the OID requests. However, with some extensible switch OID requests, an underlying extension can fail an OID request in order to veto a notification. For example, when the protocol edge of the extensible switch notifies the extensions about a new port that will be created, it issues an OID set request of OID_SWITCH_PORT_CREATE. An underlying filtering or forwarding extension can veto the port creation by completing the OID request with STATUS_DATA_NOT_ACCEPTED. For more information on this procedure, see Receiving OID Requests about Hyper-V Extensible Switch Configuration Changes.

Note If the extension does not veto an extensible switch OID request, it should monitor the status when the request is completed. The extension should do this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

Note Stack restart requests using **NdisFRestartFilter** will not complete while an extensible switch OID request is pending. For this reason, an extension that is waiting for a stack restart must complete any ongoing OID requests.

Most of the extensible switch OID requests can only be issued by the extensible switch interface. However, some extensible switch OID requests can be issued by an extension to obtain information about the configuration of the extensible switch, its ports, and its network adapter connections. For more information, see Querying the Hyper-V Extensible Switch Configuration.

Hyper-V Extensible Switch Control Path for NDIS Status Indications

Article • 03/14/2023

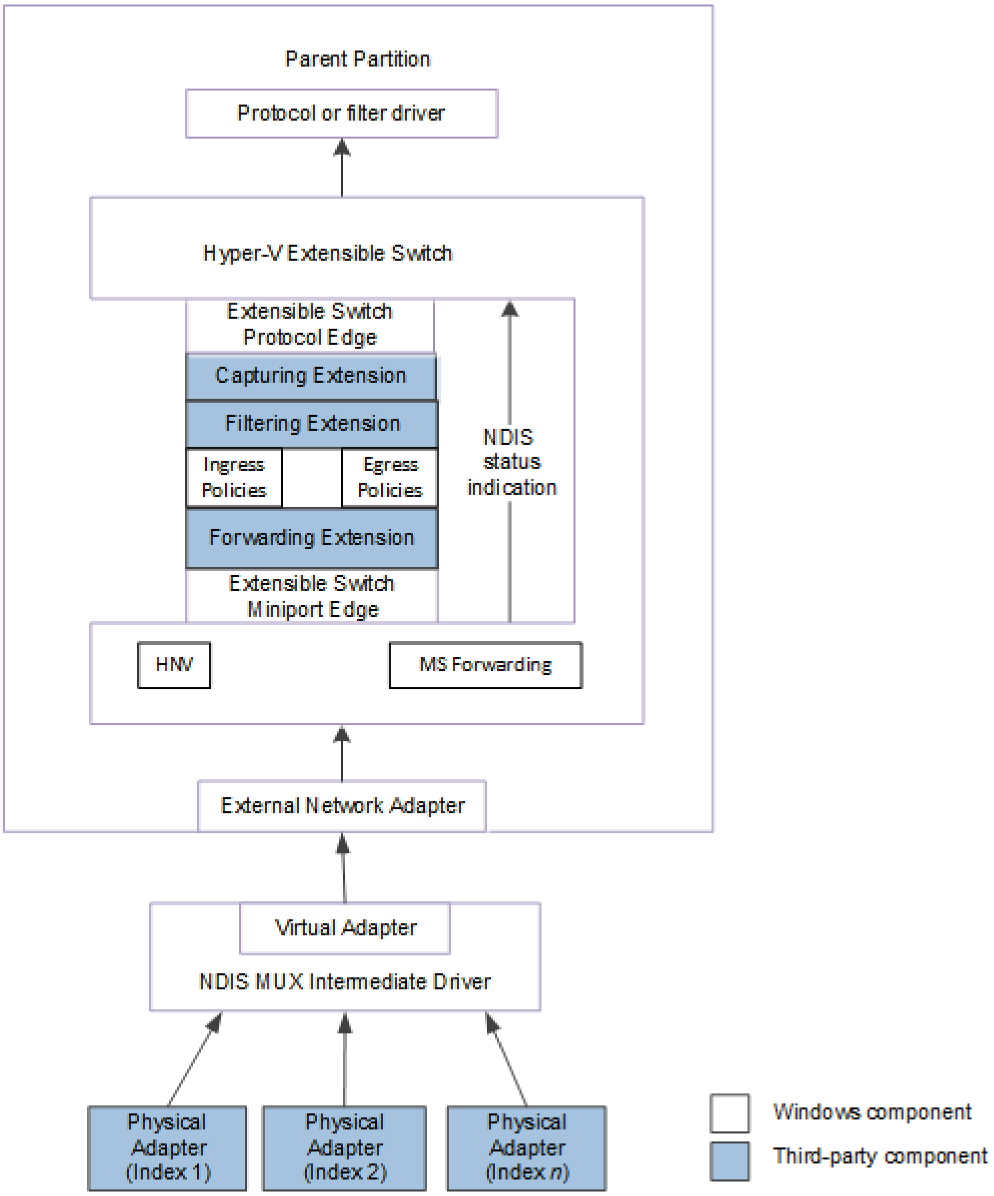
This topic discusses the control path that NDIS status indications from an underlying physical adapter move across. One or more underlying physical adapters can be teamed with the Hyper-V extensible switch external network adapter.

For example, the extensible switch external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX intermediate driver itself can be bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*. For more information about extensible switch teams, see Types of Physical Network Adapter Configurations.

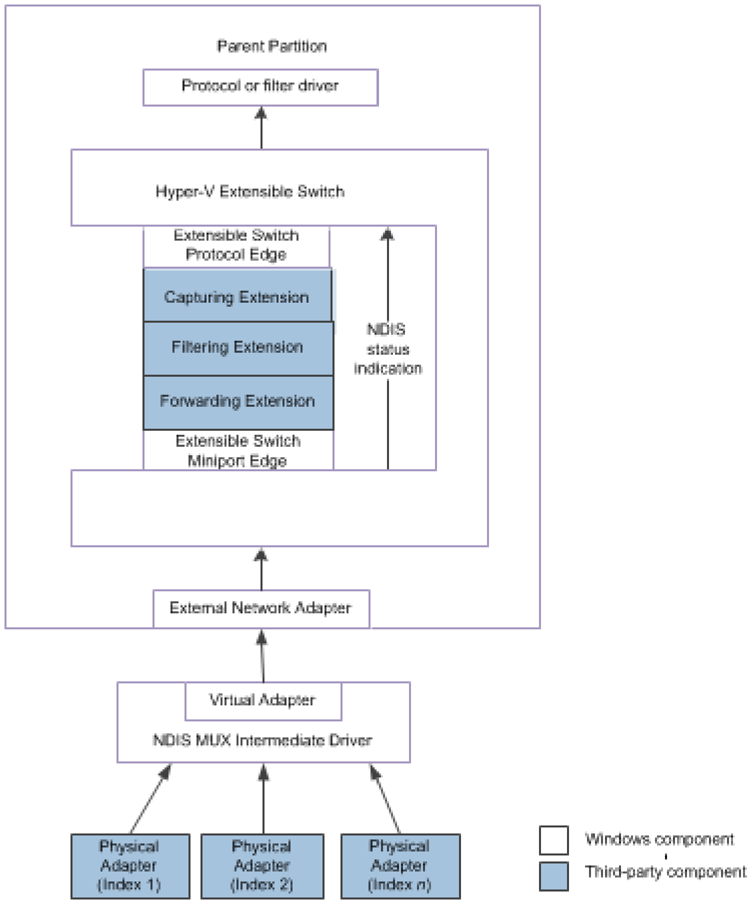
In this configuration, the extensible switch extensions are exposed to every network adapter in the extensible switch team. This allows the forwarding extension in the extensible switch driver stack to manage the configuration and use of individual network adapters in the team. For example, the extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. Such an extension is known as a *teaming provider*. For more information about teaming providers, see Teaming Provider Extensions.

Note Operations of this sort can only be performed by a forwarding extension. For more information about this type of driver, see Forwarding Extensions.

The following figure shows the extensible switch control path for NDIS status indications issued by an underlying extensible switch team for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the extensible switch control path for NDIS status indications issued by an underlying extensible switch team for NDIS 6.30 (Windows Server 2012).


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension
Extensible Switch Miniport Edge
NDIS status indication
External Network Adapter
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index n)
Windows component
Third-party component

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

The extensible switch supports NDIS status indications from the underlying physical adapter or extensible switch team in the following ways:

- When an NDIS status indication arrives at the extensible switch interface, it encapsulates the indication inside an NDIS_SWITCH_NIC_STATUS_INDICATION structure. Then, the miniport edge of the extensible switch issues an NDIS_STATUS_SWITCH_NIC_STATUS indication that contains this structure.

  When a forwarding extension receives this indication, it can duplicate the indication to change the encapsulated data. This allows the forwarding extension to change the indicated status or capabilities of the underlying extensible switch team.

- A forwarding extension that operates as a teaming provider can participate in the configuration of the adapter team for hardware offloads by initiating NDIS_STATUS_SWITCH_NIC_STATUS indications that are related to the offload technology.

  For example, the provider can initiate an NDIS_STATUS_SWITCH_NIC_STATUS indication with an encapsulated

NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES indication to modify the offload capabilities for the virtual machine queue (VMQ) on the adapter team.

- Teaming providers can also initiate an NDIS_STATUS_SWITCH_NIC_STATUS indication to modify other network adapter configurations other than an extensible switch team.

  For example, the extension can initiate an NDIS_STATUS_SWITCH_NIC_STATUS with an encapsulated NDIS_STATUS_SWITCH_PORT_REMOVE_VF indication. This indication removes the binding between a virtual machine (VM) network adapter and a PCI Express (PCIe) virtual function (VF). The VF is exposed by an underlying physical network adapter that supports the single root I/O virtualization (SR-IOV) interface.

  After this binding is removed, packets are delivered through an extensible switch port instead of directly between the VM network adapter and the VF of the underlying SR-IOV physical adapter. This allows the extensible switch port policies to be applied to packets that are received or sent over the extensible switch port.

Note The extensible switch extension must follow the same guidelines for filtering NDIS status indications that applies to all NDIS filter drivers. For more information, see Filter Module Status Indications.

For more information on how forwarding extensions can initiate NDIS_STATUS_SWITCH_NIC_STATUS indications, see Managing NDIS Status Indications from Physical Network Adapters.

Hyper-V Extensible Switch Policies Topics

Article • 12/15/2021

The Hyper-V platform and the extensible switch interface provide an infrastructure to manage switch and port policies for an extensible switch through PowerShell cmdlets and WMI. Hyper-V also provides the infrastructure to store and migrate policies.

This section includes the following topics that describe the various extensible switch policies:

Overview of Hyper-V Extensible Switch Policies

Port Policies

Switch Policies

Overview of Hyper-V Extensible Switch Policies

Article • 12/15/2021

The Hyper-V platform and the extensible switch interface provide an infrastructure to manage switch and port policies for an extensible switch. These policies are managed through PowerShell cmdlets and WMI-based application programs. This infrastructure also provides support for the storage and migration of policies.

Independent software vendors (ISVs) can use this infrastructure to register their own custom policies. After they are registered, these policies can be discovered and managed through the built-in Hyper-V policy interfaces. Properties of policies can be configured either on a per-port level or a per-switch level.

In addition to custom policy properties, the Hyper-V extensible switch interface provides the infrastructure to obtain status information for custom policy properties on a per-port or a per-switch basis. This status information is known as *feature status* information.

Extensible switch custom policy data is registered with the WMI management layer by using managed object format (MOF) class definitions. The following shows an example of a MOF class for a custom port policy property.

C++

```
#pragma namespace("\\\\.\\root\\virtualization\\v2")

[ Dynamic,
 UUID("F2F73F23-2B8E-457a-96C4-F541201C9150"),
 ExtensionId("5CBF81BE-5055-47CD-9055-A76B2B4E369E"),
 Provider("VmmsWmiInstanceAndMethodProvider"),
 Locale(0x409),
 InterfaceVersion("1"),
 InterfaceRevision("0"),
DisplayName("VendorName Port Settings Friendly Name") : Amended,
Description("VendorName Port Settings detailed description.") : Amended]
class Vendor_SampleFeatureSettingData:
Msvm_EthernetSwitchPortFeatureSettingDataMsvm
{
  [WmiDataId(1),
   InterfaceVersion("1"),
   InterfaceRevision("0")]
  uint8  IntValue8 = 0;

  [WmiDataId(2),
   InterfaceVersion("1"),
   InterfaceRevision("0")]
  uint16 IntValue16 = 0;

  [WmiDataId(3),
   InterfaceVersion("1"),
   InterfaceRevision("0")]
  uint32 IntValue32 = 0;

  [WmiDataId(4),
   InterfaceVersion("1"),
   InterfaceRevision("0")]
  uint64 IntValue64 = 0;

  [WmiDataId(5),
   InterfaceVersion("1"),
   InterfaceRevision("0"),
   MaxLen(255)]
  string FixedLengthString = "";

  [WmiDataId(6),
   InterfaceVersion("1"),
   InterfaceRevision("0")]
  string VariableLengthString = "";

  [WmiDataId(7),
   InterfaceVersion("1"),
   InterfaceRevision("0"),
   Max(8)]
  uint32 FixedLengthArray[] = {};

  [WmiDataId(8),
   InterfaceVersion("1"),
   InterfaceRevision("0")]
  uint32 VariableLengthArray[] = {};

};
```

The WMI management layer serializes the MOF data when it is transferred to an underlying extensible switch extension. The MOF class is serialized to a corresponding C structure that can be processed by the Hyper-V extensible switch extension. The following shows an example of the C structure that was serialized for the MOF class from the previous example.

C++

```
#pragma pack(8)

typedef struct _VARIABLE_LENGTH_ARRAY
{
    UINT32 Buffer[1];
} VARIABLE_LENGTH_ARRAY;

typedef struct _SAMPLE_FEATURE_SETTINGS
```

```
{
    UINT8  IntValue8;
    UINT32 IntValue16;
    UINT32 IntValue32;
    UINT64 IntValue64;
    UINT16 FixedLengthStringByteCount;
    WCHAR  FixedLengthString[256];
    UINT32 VariableLengthStringOffset;    // offset to
VARIABLE_LENGTH_STRING structure
    UINT32 FixedLengthArrayElementCount;
    UINT32 FixedLengthArray[8];
    UINT32 VariableLengthArrayElementCount;
    UINT32 VariableLengthArrayOffset;   // offset to VARIABLE_LENGTH_ARRAY
} SAMPLE_FEATURE_SETTINGS;

typedef struct _VARIABLE_LENGTH_STRING
{
    USHORT StringLength;
    WCHAR  StringBuffer[1];
} VARIABLE_LENGTH_STRING;
```

This example highlights the following points that occur when a MOF class is serialized to a corresponding C structure for an extensible switch policy property:

- The version definition in MOF files is converted into a USHORT value, where the high-order bits contain the major version and the low-order bits contain the minor version. The version is serialized by using the following code:

  `(((MajorVersion) << 8) + (MinorVersion))`

  For example, Version("1") above would be serialized to a value of 0x0100 through `(((1) << 8) + (0))`. Version ("1.1") would be serialized to a value of 0x0101 through `(((1) << 8) + (1))`.

  When a custom policy property is issued to an underlying extension, the **PropertyVersion** member of the structures that define policy properties contains the serialized version value.

  For example, when the extensible switch interface issues an object identifier (OID) request of OID_SWITCH_PORT_PROPERTY_ADD, the OID is associated with an **NDIS_SWITCH_PORT_PROPERTY_PARAMETERS** structure. The **PropertyVersion** member of that structure contains the serialized version value.

- All variable-length strings are serialized into offsets within the buffer that contains the serialized C structure. Each variable-length string is formatted as a **VARIABLE_LENGTH_STRING** structure within this buffer offset.

Port Policies

Article • 12/15/2021

This section includes the following topics that describe custom properties for Hyper-V extensible switch port policies:

Overview of Port Policies

Custom Port Property Definition and Registration

Overview of Port Policies

Article • 12/15/2021

Starting with NDIS 6.30, the following types of policies are supported for Hyper-V extensible switch ports:

Built-in Port Policies
Built-in port policies specify properties that are enforced by the extensible switch interface. Extensions in the extensible switch driver stack are not provisioned with the properties of these policies.

Built-in port policies include access control lists (ACLs) and quality of service (QoS) properties. When a packet arrives at the miniport edge of the extensible switch on the ingress data path, the switch filters the packet and enforces these policies. If the packet passes the filtering, the switch forwards the packet up the egress data path for additional handling and filtering by overlying extensions.

For more information about the extensible switch data path, see Hyper-V Extensible Switch Data Path.

Standard Port Policies
Standard port policies specify security, profile, or virtual LAN (VLAN) properties. These properties are provisioned by object identifier (OID) requests issued by the protocol edge of the extensible switch. If a forwarding extension is not installed and enabled in the extensible switch data path, these policies are enforced by the underlying extensible switch's miniport edge. Otherwise, the forwarding extension enforces these policies if it allows the policy to be provisioned.

Standard port properties are specified by an **NDIS_SWITCH_PORT_PROPERTY_TYPE** enumeration value of **NdisSwitchPortPropertyTypeSecurity**, **NdisSwitchPortPropertyTypeVlan**, and **NdisSwitchPortPropertyTypeProfile**.

Note If a forwarding extension does not manage or enforce VLAN port properties, it must return STATUS_DATA_NOT_ACCEPTED for the OID_SWITCH_PORT_PROPERTY_ADD and OID_SWITCH_PORT_PROPERTY_UPDATE requests that add or update the property. VLAN port properties have a property type of **NdisSwitchPortPropertyTypeVlan**.

Custom Port Policies
Custom port policies specify proprietary properties that are defined by an independent software vendor (ISV). These properties are provisioned by OID requests issued by the extensible switch's protocol edge of the extensible switch and enforced by the underlying extension that manages the custom port policy.

Custom port properties are defined through managed object format (MOF) class definitions. The ISV defines the format of the custom port properties through the MOF class definition. After the MOF file is registered with the WMI management layer, the underlying extensions are provisioned with the custom port policy.

A custom port property is specified by **NDIS_SWITCH_PORT_PROPERTY_TYPE** enumeration value of **NdisSwitchPortPropertyTypeCustom**. Each custom port property is uniquely defined through a GUID value. The extension manages those custom port properties for which it has been configured with the property's GUID value.

Note The method by which the extension is configured with the property's GUID value is proprietary to the ISV.

Standard and custom port policies are provisioned through the following OID requests:

- The protocol edge issues OID set requests of OID_SWITCH_PORT_PROPERTY_ADD to notify underlying extensions of the addition of a standard or custom port property.
- The protocol edge issues OID set requests of OID_SWITCH_PORT_PROPERTY_UPDATE to notify underlying extensions of the update to a standard or custom port property.
- The protocol edge issues OID set requests of OID_SWITCH_PORT_PROPERTY_DELETE to notify underlying extensions of the deletion of a standard or custom port property.

A forwarding extension can block the provisioning of the new or updated port policy by vetoing the OID request. The extension does this by completing the OID request with STATUS_DATA_NOT_ACCEPTED. If the extension does not veto the OID request, it must call **NdisFOidRequest** to forward the OID request down the extensible switch control path.

Note If the extension does not veto the OID request, it monitors the status when the request is completed. The extension does this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

For more information on how to manage port policies and properties, see Managing Port Policies.

Custom Port Property Definition and Registration

Article • 12/15/2021

Custom property definitions for a Hyper-V extensible switch port policy are registered with the WMI management layer by using managed object format (MOF) class definitions. In addition to the structure members that define the attributes of the custom port property, the MOF class must also contain the following:

- A UUID that uniquely identifies the custom port property.
- A GUID that uniquely identifies the extensible switch extension. This GUID is declared as the **ExtensionId** qualifier of the MOF class and must match the value of the **NetCfgInstanceId** entry that is declared in the extension's INF file.
- A descriptive class name string. The name of the vendor must be included in the string.

The following shows an example of a MOF class for a custom property of an extensible switch port policy.

```C++
#pragma namespace("\\\\.\\root\\virtualization\\v2")

[ Dynamic,
 UUID("EB29F0F2-F5DC-45C6-81BB-3CD9F219BBBB"),
 ExtensionId("5CBF81BE-5055-47CD-9055-A76B2B4E369E"),
 Provider("VmmsWmiInstanceAndMethodProvider"),
 Locale(0x409),
 InterfaceVersion("1"),
 InterfaceRevison("0"),
DisplayName("Fabrikam, Inc. Port Settings Friendly Name") : Amended,
Description("Fabrikam, Inc. Port Settings detailed description.") : Amended]
class Fabrikam_PortCustomSettingData :
Msvm_EthernetSwitchPortFeatureSettingData {

    [ Read,
      Write,
      WmiDataId(1),
      InterfaceVersion("1"),
      InterfaceRevision("0"),
      Description (
         "int32 setting.") : Amended]
    uint32 SettingIntA = 0;

    [ Read,
      Write,
      WmiDataId(2),
      InterfaceVersion("1"),
      InterfaceRevision("0"),
      Description (
         "int64 setting.") : Amended]
    uint64 SettingIntB = 0;
};
```

The MOF classes for custom properties of a port policy are registered in the common information model (CIM) repository by using the MOF compiler (Mofcomp.exe). Once registered, the MOF class can be configured through PowerShell cmdlets and WMI-based application programs.

The following example shows the commands that must be entered to register a file (Fabrikam_PortCustomSettingData.mof) that contains the MOF class for a custom port property.

PowerShell

```
net stop vmms
mofcomp -N:root\virtualization\v2 Fabrikam_PortCustomSettingData.mof
net start vmms
```

For more information about how to use the MOF compiler, see Compiling a Driver's MOF File.

The following example shows how you can configure the sample feature. In this example, the Fabrikam_PortCustomSettingData MOF class is used to configure a port from a Hyper-V partition named "TestVm".

PowerShell

```
# Retrieve the template object for the custom configuration. We know the ID
already so
# we can retrieve it directly, otherwise Get-
VmSystemSwitchExtensionPortFeature can list all available
# properties.
PS C:\> $feature = Get-VMSystemSwitchExtensionPortFeature -FeatureId
EB29F0F2-F5DC-45C6-81BB-3CD9F219BBBB

# Output the values
PS C:\> $feature

Id            : eb29f0f2-f5dc-45c6-81bb-3cd9f219bbbb
ExtensionId   : 5cbf81bd-5055-47cd-9055-a76b2b4e369d
ExtensionName : Fabrican Extension
Name          : Fabrikam, Inc. Port Settings Friendly Name
```

```
ComputerName  : TEST_SERVER
SettingData   :
\\TEST_SERVER\root\virtualization\v2:VendorName_SwitchPortCustomSettingData.
InstanceID="Microsoft:Defini
                tion\\EB29F0F2-F5DC-45C6-81BB-3CD9F219BBBB\\Default"

# Cast the SettingsData to a WMI object to see the actual configurable
values.
PS C:\> $wmiObj = [wmi]$feature.SettingData
PS C:\> $wmiObj

__GENUS            : 2
__CLASS            : Fabrikam_PortCustomSettingData
__SUPERCLASS       : Msvm_EthernetSwitchFeatureSettingData
__DYNASTY          : CIM_ManagedElement
__RELPATH          :
Fabrikam_PortCustomSettingData.InstanceID="Microsoft:Definition\\EB29F0F2-
F5DC-45C6-81BB-3CD
                   9F219BBBB\\Default"
__PROPERTY_COUNT : 6
__DERIVATION       : {Msvm_EthernetSwitchFeatureSettingData, CIM_SettingData,
CIM_ManagedElement}
__SERVER           : TEST_SERVER
__NAMESPACE        : root\virtualization\v2
__PATH             :
\\TEST_SERVER\root\virtualization\v2:Fabrikam_PortCustomSettingData.Instance
ID="Microsoft:Def
                   inition\\EB29F0F2-F5DC-45C6-81BB-3CD9F219BBBB\\Default"
Caption            : Fabrikam, Inc. Port Settings Friendly Name
Description        : Fabrikam, Inc. Port Settings detailed description.
ElementName        : Fabrikam, Inc. Port Settings Friendly Name
InstanceID         : Microsoft:Definition\EB29F0F2-F5DC-45C6-81BB-
3CD9F219BBBB\Default
SettingIntA        : 0
SettingIntB        : 0

# Update the property settings and add to the NIC attached to TestVm
PS C:\> $wmiObj.SettingIntA = 100
PS C:\> $wmiObj.SettingIntB = 9999
PS C:\> Add-VMSwitchExtensionPortFeature -VMSwitchExtensionFeature $feature
-VmName TestVm

# Validate that the properties are now set on the VM’s NIC
PS C:\> $feature = Get-VmSwitchExtensionPortFeature -FeatureId EB29F0F2-
F5DC-45C6-81BB-3CD9F219BBBB -VmName TestVm

PS C:\> [wmi]$feature.SettingData


__GENUS            : 2
__CLASS            : Fabrikam_PortCustomSettingData
__SUPERCLASS       : Msvm_EthernetSwitchFeatureSettingData
__DYNASTY          : CIM_ManagedElement
__RELPATH          :
Fabrikam_PortCustomSettingData.InstanceID="Microsoft:6208FB20-2490-4DC1-
```

```
B121-877B68B4CE11\\4
                   DDC57F5-6DAE-4A36-9D62-7A838D5601F2\\C\\EB29F0F2-F5DC-
45C6-81BB-3CD9F219BBBB\\CB323B56-FA54-4506-B58
                   B-78C70C0B3229"
__PROPERTY_COUNT : 6
__DERIVATION     : {Msvm_EthernetSwitchFeatureSettingData, CIM_SettingData,
CIM_ManagedElement}
__SERVER         : TEST_SERVER
__NAMESPACE      : root\virtualization\v2
__PATH           :
\\TEST_SERVER\root\virtualization\v2:Fabrikam_PortCustomSettingData.Instance
ID="Microsoft:620
                   8FB20-2490-4DC1-B121-877B68B4CE11\\4DDC57F5-6DAE-4A36-
9D62-7A838D5601F2\\C\\EB29F0F2-F5DC-45C6-81BB-
                   3CD9F219BBBB\\CB323B56-FA54-4506-B58B-78C70C0B3229"
Caption          : Fabrikam, Inc. Port Settings Friendly Name
Description      : Fabrikam, Inc. Port Settings detailed description.
ElementName      : Fabrikam, Inc. Port Settings Friendly Name
InstanceID       : Microsoft:6208FB20-2490-4DC1-B121-877B68B4CE11\4DDC57F5-
6DAE-4A36-9D62-7A838D5601F2\C\EB29F0F2-F5DC-
                   45C6-81BB-3CD9F219BBBB\CB323B56-FA54-4506-B58B-
78C70C0B3229
SettingIntA      : 100
SettingIntB      : 9999
```

For more information on how extensible switch extensions manage port policies, see Managing Port Policies.

Switch Policies

Article • 12/15/2021

This section includes the following topics that describe custom properties for Hyper-V extensible switch policies:

Overview of Switch Policies

Custom Switch Property Definition and Registration

Overview of Switch Policies

Article • 12/15/2021

Starting with NDIS 6.30, the following types of policies are supported for Hyper-V extensible switches:

Built-in Switch Policies
Built-in switch policies specify properties that are enforced by the extensible switch interface. Extensions in the extensible switch driver stack are not provisioned with the properties of these policies.

Built-in switch policies include properties that affect the switch configuration in general but do not affect traffic flow over individual switch ports. For example, one such built-in policy configures the switch to allow hardware offloads to physical adapters that support the single root I/O virtualization (SR-IOV) interface. For more information about this interface, see Single Root I/O Virtualization (SR-IOV).

Custom Switch Policies
Custom switch policies specify proprietary properties that are defined by an independent software vendor (ISV). These properties are provisioned by the protocol edge of the extensible switch and enforced by the underlying extension that manages the custom switch policy.

The ISV defines the format for the custom switch properties. The format of the custom switch property is proprietary to the ISV.

Custom switch properties are defined through managed object format (MOF) class definitions. After the MOF file is registered with the WMI management layer, the underlying extensions are provisioned with the custom switch policy.

A custom switch property is specified by the **NDIS_SWITCH_PROPERTY_TYPE** enumeration value of **NdisSwitchPropertyTypeCustom**. Each custom switch property is uniquely defined through a GUID value. The extension manages those custom switch properties for which it has been configured with the property's GUID value.

Note The method by which the extension is configured with the property's GUID value is proprietary to the ISV.

Custom switch policies are provisioned through the following OID requests:

- The protocol edge issues OID set requests of OID_SWITCH_PROPERTY_ADD to notify underlying extensions of the addition of a custom switch property.

- The protocol edge issues OID set requests of OID_SWITCH_PROPERTY_UPDATE to notify underlying extensions of the update to a custom switch property.
- The protocol edge issues OID set requests of OID_SWITCH_PROPERTY_DELETE to notify underlying extensions of the deletion of a custom switch property.

A forwarding extension can block the provisioning of the new or updated switch policy by vetoing the OID request. The extension does this by completing the OID request with STATUS_DATA_NOT_ACCEPTED. If the extension does not veto the OID request, it must call **NdisFOidRequest** to forward the OID request down the extensible switch control path.

Note If the extension does not veto the OID request, it monitors the status when the request is completed. The extension does this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

For more information on how to manage switch policies and properties, see Managing Switch Policies.

Custom Switch Property Definition and Registration

Article • 12/15/2021

Custom property definitions for a Hyper-V extensible switch policy are registered with the WMI management layer by using managed object format (MOF) class definitions. In addition to the structure members that define the attributes of the custom switch property, the MOF class must also contain the following:

- A UUID that uniquely identifies the custom switch property.
- A GUID that uniquely identifies the extensible switch extension. This GUID is declared as the **ExtensionId** qualifier of the MOF class and must match the value of the **NetCfgInstanceId** entry that is declared in the extension's INF file.
- A descriptive class name string. The name of the vendor must be included in the string.

The following shows an example of a MOF class for a custom property of an extensible switch policy.

```C++
#pragma namespace("\\\\.\\root\\virtualization\\v2")

[ Dynamic,
 UUID("FF36C3A6-D2F1-46ed-A376-32B43D6B8390"),
 ExtensionId("5CBF81BE-5055-47CD-9055-A76B2B4E369E"),
 Provider("VmmsWmiInstanceAndMethodProvider"),
  Locale(0x409),
 InterfaceVersion("1"),
 InterfaceRevison("0"),
DisplayName("Fabrikam, Inc.  Switch Settings Friendly Name") : Amended,
Description("Fabrikam, Inc.  Switch Settings detailed description.") :
Amended]
class Fabrikam_SwitchCustomSettingData :
Msvm_EthernetSwitchFeatureSettingData {

    [ Read,
      Write,
      WmiDataId(1),
      InterfaceVersion("1"),
      InterfaceRevision("0"),
      Description (
         "int32 setting.") : Amended]
    uint32 SwitchSettingIntA = 0;
    [ Read,
      Write,
      WmiDataId(2),
      InterfaceVersion("1"),
      InterfaceRevision("0"),
      Description (
         "int64 setting.") : Amended]
    uint64 SwitchSettingIntB = 0;
};
```

The MOF classes for custom properties of a switch policy are registered in the common information model (CIM) repository by using the MOF compiler (Mofcomp.exe). Once registered, the MOF class can be configured through PowerShell cmdlets and WMI-based application programs.

The following example shows the commands that must be entered to register a file (Fabrikam_SwitchCustomSettingData.mof) that contains the MOF class for a custom port property.

PowerShell

```
net stop vmms
mofcomp -N:root\virtualization\v2 Fabrikam_SwitchCustomSettingData.mof
net start vmms
```

For more information about how to use the MOF compiler, see Compiling a Driver's MOF File.

The following example shows how you can configure the sample feature. In this example, the Fabrikam_SwitchCustomSettingData MOF class is used to configure a switch named “TestSwitch”.

PowerShell

```
# Retrieve the template object for the custom configuration. We know the ID
already so
# we can retrieve it directly, otherwise Get-
VMSystemSwitchExtensionSwitchFeature can list all available
# properties.
PS C:\> $feature = Get-VMSystemSwitchExtensionSwitchFeature -FeatureId
FF36C3A6-D2F1-46ed-A376-32B43D6B8390

# Output the values
PS C:\temp> $feature


Id             : ff36c3a6-d2f1-46ed-a376-32b43d6b8390
ExtensionId    : 5CBF81BE-5055-47CD-9055-A76B2B4E369E
```

```
ExtensionName : Fabrican Extension
Name          : Fabrikam, Inc. Switch Settings Friendly Name
ComputerName  : TEST_SERVER
SettingData   :
\\TEST_SERVER\root\virtualization\v2:VendorName_SwitchCustomSettingData.Inst
anceID="Microsoft:Definition\\
                FF36C3A6-D2F1-46ED-A376-32B43D6B8390\\Default"

# Cast the SettingsData to a WMI object to see the actual configurable
values.
PS C:\> $wmiObj = [wmi]$feature.SettingData
PS C:\> $wmiObj


__GENUS            : 2
__CLASS            : Fabrikam_SwitchCustomSettingData
__SUPERCLASS       : Msvm_EthernetSwitchFeatureSettingData
__DYNASTY          : CIM_ManagedElement
__RELPATH          : Fabrikam_SwitchCustomSettingData
.InstanceID="Microsoft:Definition\\FF36C3A6-D2F1-46ED-A376-32B43D
                     6B8390\\Default"
__PROPERTY_COUNT   : 6
__DERIVATION       : {Msvm_EthernetSwitchFeatureSettingData,
Msvm_FeatureSettingData, CIM_SettingData,
                     CIM_ManagedElement}
__SERVER           : TEST_SERVER
__NAMESPACE        : root\virtualization\v2
__PATH             :
\\TEST_SERVER\root\virtualization\v2:Fabrikam_SwitchCustomSettingData
.InstanceID="Microsoft:Definiti
                     on\\FF36C3A6-D2F1-46ED-A376-32B43D6B8390\\Default"
Caption            : Fabrikam, Inc. Switch Settings Friendly Name
Description        : Fabrikam, Inc. Switch Settings detailed description.
ElementName        : Fabrikam, Inc. Switch Settings Friendly Name
InstanceID         : Microsoft:Definition\FF36C3A6-D2F1-46ED-A376-
32B43D6B8390\Default
SwitchSettingIntA : 0
SwitchSettingIntB : 0
PSComputerName     : TEST_SERVER

# Update the property settings and add to the NIC attached to TestVm
PS C:\> $wmiObj.SwitchSettingIntA = 100
PS C:\> $wmiObj.SwitchSettingIntB = 9999
PS C:\> Add-VMSwitchExtensionSwitchFeature -VMSwitchExtensionFeature
$feature -SwitchName TestSwitch

# Validate that the properties are now set on the VM’s NIC
PS C:\> $feature = Get-VmSwitchExtensionSwitchFeature -FeatureId FF36C3A6-
D2F1-46ed-A376-32B43D6B8390 -SwitchName TestSwitch

PS C:\> [wmi]$feature.SettingData


__GENUS            : 2
__CLASS            : Fabrikam_SwitchCustomSettingData
```

```
__SUPERCLASS      : Msvm_EthernetSwitchFeatureSettingData
__DYNASTY         : CIM_ManagedElement
__RELPATH         : Fabrikam_SwitchCustomSettingData
.InstanceID="Microsoft:88835394-FDE1-437C-B249-D840575154E2\\FF36
                    C3A6-D2F1-46ED-A376-32B43D6B8390\\F9EA07E7-7B73-431A-
8705-26EC2B592306"
__PROPERTY_COUNT  : 6
__DERIVATION      : {Msvm_EthernetSwitchFeatureSettingData,
Msvm_FeatureSettingData, CIM_SettingData,
                    CIM_ManagedElement}
__SERVER          : TEST_SERVER
__NAMESPACE       : root\virtualization\v2
__PATH            :
\\TEST_SERVER\root\virtualization\v2:Fabrikam_SwitchCustomSettingData
.InstanceID="Microsoft:88835394
                    -FDE1-437C-B249-D840575154E2\\FF36C3A6-D2F1-46ED-A376-
32B43D6B8390\\F9EA07E7-7B73-431A-8705-26EC2B5
                    92306"
Caption           : Fabrikam, Inc. Switch Settings Friendly Name
Description       : Fabrikam, Inc. Switch Settings detailed description.
ElementName       : Fabrikam, Inc. Switch Settings Friendly Name
InstanceID        : Microsoft:88835394-FDE1-437C-B249-D840575154E2\FF36C3A6-
D2F1-46ED-A376-32B43D6B8390\F9EA07E7-7B73-4
                    31A-8705-26EC2B592306
SwitchSettingIntA : 100
SwitchSettingIntB : 9999
PSComputerName    : TEST_SERVER
```

For more information on how extensible switch extensions manage switch policies, see Managing Switch Policies.

Hyper-V Extensible Switch Feature Status Information Overview

Article • 12/15/2021

The Hyper-V extensible switch interface supports the ability to obtain custom status information for an extensible switch or one of its ports. This status information is known as *feature status* information.

This section includes the following topics that describe the support for feature status information:

Custom Port Feature Status

Custom Switch Feature Status

Custom Port Feature Status

Article • 12/15/2021

The Hyper-V platform and Hyper-V extensible switch interface provide the infrastructure to obtain custom status information for an extensible switch port. This information is known as *port feature status* information.

Custom feature status definitions for a Hyper-V extensible switch port property are registered with the WMI management layer by using managed object format (MOF) class definitions. In addition to the structure members that define the attributes of the custom port feature status definition, the MOF class must also contain the following:

- A UUID that uniquely identifies the custom port feature status definition.
- A GUID that uniquely identifies the extensible switch extension. This GUID is declared as the **ExtensionId** qualifier of the MOF class and must match the value of the **NetCfgInstanceId** entry that is declared in the extension's INF file.
- A descriptive class name string. The name of the vendor must be included in the string.

The following shows an example of a MOF class for a custom feature status definition of an extensible switch port.

C++

```
#pragma namespace("\\\\.\\root\\virtualization\\v2")

[ Dynamic,
  UUID("DAA0B7CC-74DB-41ef-8354-7002F9FA463E"),
  ExtensionId("5CBF81BE-5055-47CD-9055-A76B2B4E369E"),
  Provider("VmmsWmiInstanceAndMethodProvider"),
  InterfaceVersion("1"),
  InterfaceRevison("0"),
  Locale(0x409),
  Description("Fabricam, Inc. port custom feature status description.") :
Amended,
  DisplayName("Fabricam, Inc.port custom feature status friendly name.") :
Amended]
class Fabrikam_CustomPortData  : Msvm_EthernetPortData {
    [ Read,
       Write,
       WmiDataId(1),
      InterfaceVersion("1"),
      InterfaceRevision("0"),
       Description(
         "The current status of custom feature on this port.") : Amended]
    uint32 CurrentStatus = 0 ;
};
```

The MOF classes for custom feature status definition of a port are registered in the common information model (CIM) repository by using the MOF compiler (Mofcomp.exe). After it is registered, the MOF class can be configured through PowerShell cmdlets and WMI-based application programs.

The following example shows the commands that must be entered to register a file (Fabrikam_CustomPortData.mof) that contains the MOF class for a custom port feature status definition.

PowerShell

```
net stop vmms
mofcomp -N:root\virtualization\v2 Fabrikam_CustomPortData.mof
net start vmms
```

For more information about how to use the MOF compiler, see Compiling a Driver's MOF File.

The following example shows how you can use the custom port feature status definition to obtain port data. In this example, the Fabrikam_CustomPortData MOF class is used to obtain port status from a Hyper-V partition named "TestVm". The Fabrikam, Inc. extension is enabled on the vSwitch “TestSwitch”, and is returning 123 for the status.

PowerShell

```
PS C:\> $portData = Get-VMSwitchExtensionPortData -VmName TestVm -FeatureId
DAA0B7CC-74DB-41ef-8354-7002F9FA463E
# Output the current value
PS C:\> $portData.Data.CurrentStatus
123
```

For more information on how extensible switch extensions manage port feature status information, see Managing Custom Port Feature Status Information.

Custom Switch Feature Status

Article • 12/15/2021

The Hyper-V platform and Hyper-V extensible switch interface provide the infrastructure to obtain custom status information for an extensible switch. This information is known as *switch feature status* information.

Custom switch feature status definitions are registered with the WMI management layer by using managed object format (MOF) class definitions. In addition to the structure members that define the attributes of the custom switch feature status definition, the MOF class must also contain the following:

- A UUID that uniquely identifies the custom switch feature status definition.
- A GUID that uniquely identifies the extensible switch extension. This GUID is declared as the **ExtensionId** qualifier of the MOF class and must match the value of the **NetCfgInstanceId** entry that is declared in the extension's INF file.
- A descriptive class name string. The name of the vendor must be included in the string.

The following shows an example of a MOF class for a custom feature status definition of an extensible switch.

C++

```
#pragma namespace("\\\\.\\root\\virtualization\\v2")

[ Dynamic,
  UUID("B3E57D77-8E95-4977-97DE-524F8DAF03E4"),
  ExtensionId("5CBF81BE-5055-47CD-9055-A76B2B4E369E"),
  Provider("VmmsWmiInstanceAndMethodProvider"),
  InterfaceVersion("1"),
  InterfaceRevison("0"),
  Locale(0x409),
  Description(
   "Fabricam, Inc. Switch custom feature status description.") : Amended,
  DisplayName("Fabricam, Inc. Switch custom feature status friendly name.")
: Amended]
class Fabrikam_CustomSwitchData  : Msvm_EthernetSwitchFeatureSettingData{
    [ Read,
       Write,
       WmiDataId(1),
       InterfaceVersion("1"),
       InterfaceRevision("0"),
       Description(
         "The current status of custom feature on this switch.") : Amended]
```

```
      uint32 CurrentStatus = 0 ;
  };
```

The MOF classes for custom feature status definition of an extensible switch are registered in the common information model (CIM) repository by using the MOF compiler (Mofcomp.exe). After it is registered, the MOF class can be configured through PowerShell cmdlets and WMI-based application programs.

The following example shows the commands that must be entered to register a file (Fabrikam_CustomSwitchData.mof) that contains the MOF class for a custom switch feature status definition.

PowerShell

```
net stop vmms
mofcomp -N:root\virtualization\v2 Fabrikam_CustomSwitchData.mof
net start vmms
```

For more information about how to use the MOF compiler, see Compiling a Driver's MOF File.

The following example shows how you can use the custom switch feature status definition to obtain switch data. In this example, the Fabrikam_CustomSwitchData MOF class is used to obtain switch status from a switch named “TestSwitch”. The Fabrikam, Inc. extension is enabled on the vSwitch “TestSwitch”, and is returning 123 for the status.

PowerShell

```
PS C:\> $switchData = Get-VMSwitchExtensionSwitchData -SwitchName TestSwitch
-FeatureId B3E57D77-8E95-4977-97DE-524F8DAF03E4
# Output the current value
PS C:\> $switchData$customSwitchData.Data.CurrentStatus
123
```

For more information on how extensible switch extensions manage switch feature status information, see Managing Custom Switch Feature Status Information.

Hyper-V Extensible Switch Save and Restore Operations Overview

Article • 12/15/2021

When a Hyper-V child partition is stopped, saved, or live migrated, the run-time state of the partition is saved. When the partition is restarted or has completed the live migration to another host computer, the run-time state is restored. During the transition between saved and restored states, the settings of the network interfaces for the child partition are unchanged and network connections to the Hyper-V extensible switch are not torn down.

The extensible switch interface notifies underlying extensions of save and restore operations for the child partition. During the save operation, the extension can return run-time data for each extensible switch network adapter (NIC). During the restore operation, the interface returns the run-time data to the extension so that it can restore the state of the NIC.

This section includes the following topics:

Hyper-V Extensible Switch Save Operations

Hyper-V Extensible Switch Restore Operations

Hyper-V Extensible Switch Live Migration Support

Hyper-V Extensible Switch Save Operations

Article • 12/15/2021

When a Hyper-V child partition is stopped, saved, or live migrated, the run-time state of the partition is saved. During the save operation, a Hyper-V extensible switch extension can save run-time data about an extensible switch network adapter (NIC).

When a save operation is being performed on a Hyper-V child partition, the extensible switch interface notifies the extension about the operation. The extension is notified through the following object identifier (OID) requests:

OID_SWITCH_NIC_SAVE
The extensible switch interface signals the protocol edge of the extensible switch to issue this OID during the save operation for an extensible switch NIC. When it handles this OID request, the extension returns run-time data for the NIC. After the run-time data is saved, it is restored through OID set requests of OID_SWITCH_NIC_RESTORE.

When it receives the OID_SWITCH_NIC_SAVE method request, the extension can do one of the following:

- If the extension has run-time data to save, it initializes an **NDIS_SWITCH_NIC_SAVE_STATE** structure and sets the various members, such as the **ExtensionId** member, to identify itself and the data that it is saving. The extension also saves the data within the **NDIS_SWITCH_NIC_SAVE_STATE** structure starting SaveDataOffset bytes from the start of the structure, and then completes the OID method request with NDIS_STATUS_SUCCESS.

- If the **NDIS_SWITCH_NIC_SAVE_STATE** structure does not provide a sufficient buffer size, enumerated in the NDIS_OBJECT_HEADER **Size** member to hold the run-time state, the extension sets the method structure's *BytesNeeded* field to NDIS_SIZEOF_NDIS_SWITCH_NIC_SAVE_STATE_REVISION_1 plus the amount of buffer necessary to hold the save data, and completes the OID with NDIS_STATUS_BUFFER_TOO_SHORT. The OID will be reissued with the required size.

- If the extension does not have run-time data to save, it must call **NdisFOidRequest**. This forwards the OID method request to underlying extensions in the extensible switch driver stack. For more information about this procedure, see Filtering OID Requests in an NDIS Filter Driver.

For more information about this OID request, see Handling the OID_SWITCH_NIC_SAVE Request.

OID_SWITCH_NIC_SAVE_COMPLETE
The extensible switch interface signals the protocol edge of the extensible switch to issue this OID at the completion of the save operation of run-time data for an extensible switch NIC.

This OID request notifies the extension that the save operation has completed only for a specified extensible switch NIC.

For more information about this OID request, see Handling the OID_SWITCH_NIC_SAVE_COMPLETE Request.

During the save operation for run-time data, the protocol edge of the extensible switch issues OID requests of OID_SWITCH_NIC_SAVE and OID_SWITCH_NIC_SAVE_COMPLETE for the network interface of a Hyper-V child partition is connected. If multiple Hyper-V child partitions are stopped or live migrated, the protocol edge issues separate sets of OID_SWITCH_NIC_SAVE and OID_SWITCH_NIC_SAVE_COMPLETE requests for each network interface connection.

Note The protocol edge of the extensible switch will not interleave save operations for run-time data for the same NIC. The protocol edge will start a run-time data save operation for a NIC only after a previous save operation has completed on the same NIC. However, the protocol edge may start a save operation for a NIC while another save operation is in progress for another NIC. Because of this, we highly recommend that extensions perform save operations in a non-interleaved fashion. For example, extensions should not assume that a new save operation cannot start on another NIC before an ongoing save operation has completed for a different NIC.

Hyper-V Extensible Switch Restore Operations

Article • 12/15/2021

When a Hyper-V child partition is restarted after it was stopped or live migrated, the run-time state of the partition is restored. During the restore operation, a Hyper-V extensible switch extension driver can restore run-time data about an extensible switch network adapter (NIC).

When a restore operation is being performed on a Hyper-V child partition, the extensible switch interface signals the protocol edge of the extensible switch to issue an OID set request of OID_SWITCH_NIC_RESTORE. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for the OID_SWITCH_NIC_RESTORE request contains a pointer to an **NDIS_SWITCH_NIC_SAVE_STATE** structure.

When it handles this OID request, the extension restores the run-time data for the network adapter. This run-time data was previously saved through OID requests of OID_SWITCH_NIC_SAVE and OID_SWITCH_NIC_SAVE_COMPLETE.

When it receives the OID_SWITCH_NIC_RESTORE request, the extensible switch extension must first determine whether it owns the run-time data. The driver does this by comparing the value of the **ExtensionId** member of the **NDIS_SWITCH_NIC_SAVE_STATE** structure to the GUID value that the driver uses to identify itself.

If the extension owns the run-time data, it restores this data in the following way:

1. The extension copies the run-time data in the **SaveData** member to driver-allocated storage.

   Note The value of the **PortId** member of the **NDIS_SWITCH_NIC_SAVE_STATE** structure may be different from the **PortId** value at the time that the run-time data was saved. This can occur if run-time data was saved during a Live Migration from one host to another. However, the configuration of the extensible switch NIC is retained during the Live Migration. This enables the extension to restore the run-time data to the extensible switch NIC by using the new **PortId** value.

2. The extension completes the OID set request with NDIS_STATUS_SUCCESS.

If the extension does not own the run-time data, it must call **NdisFOidRequest**. This forwards the OID method request to underlying extensions in the extensible switch driver stack. For more information about this procedure, see Filtering OID Requests in an NDIS Filter Driver.

OID_SWITCH_NIC_RESTORE_COMPLETE
The extensible switch interface signals the protocol edge of the extensible switch to issue this OID at the completion of the restore operation of run-time data for an extensible switch NIC.

This OID request notifies the extension that the restore operation has completed only for a specified extensible switch NIC.

For more information about this OID request, see OID_SWITCH_NIC_RESTORE_COMPLETE.

During the restore operation for run-time data, the protocol edge of the extensible switch issues OID requests of OID_SWITCH_NIC_RESTORE and OID_SWITCH_NIC_RESTORE_COMPLETE for the network interface of a Hyper-V child partition is connected. If multiple Hyper-V child partitions are restored, the protocol edge issues separate sets of OID_SWITCH_NIC_RESTORE and OID_SWITCH_NIC_RESTORE_COMPLETE requests for each network interface connection.

Note The protocol edge of the extensible switch will not interleave restore operations for run-time data for the same NIC. The protocol edge will start a run-time data restore operation for a NIC only after a previous restore operation has completed on the same NIC. However, the protocol edge may start a restore operation for a NIC while another restore operation is in progress for another NIC. Because of this, we highly recommend that extensions perform restore operations in a non-interleaved fashion. For example, extensions should not assume that a new restore operation cannot start on another NIC before an ongoing restore operation has completed for a different NIC.

For more information about this OID request, see Restoring Hyper-V Extensible Switch Run-Time Data.

Hyper-V Extensible Switch Live Migration Support

Article • 12/15/2021

During a Hyper-V live migration, a child partition, or *virtual machine (VM)*, is stopped on one host computer (*source host*) and migrated to another host computer (*destination host*). During live migration, the following operations occur:

- When the live migration starts on the source host, the extensible switch interface requests underlying extensions to save run-time data for each port and its associated network adapter connection.

  For more information about this operation, see Hyper-V Extensible Switch Save Operations.

- Before the live migration completes on the destination host, the extensible switch interface requests underlying extensions to restore run-time data for each port and its associated network adapter connection.

  For more information about this operation, see Hyper-V Extensible Switch Restore Operations.

During the live migration setup stage, the source host creates a TCP connection with the destination physical host. Hyper-V transfers the source VM's configuration data over this connection to the destination physical host. A skeleton VM is set up on the destination host and memory is allocated to the destination VM. At this point, Hyper-V transfers the source VM's state, including its memory pages, to the destination VM.

The extensible switch interface also uses the TCP connection to synchronize steps and results during the live migration. For example, the interface that runs on the destination host requests the transfer of run-time data from the source host for the port and network adapter connection associated with the migrated VM.

Before the destination VM is brought online on the destination host, the extensible switch interface performs these steps:

1. A validation port is created on the destination host through an object identifier (OID) set request of OID_SWITCH_PORT_CREATE. If the port is created successfully, the extensible switch interface issues other OID requests to verify the properties of port policies by underlying extensions.

If the extension fails the port creation or invalidates any of the policy properties, the live migration does not continue for that destination node and switch.

For more information about the validation port and its usages, see Validation Ports.

2. After the verification of policy properties is completed successfully, the validation port is deleted on the destination host through an OID set request of OID_SWITCH_PORT_DELETE. After this port is deleted, an operational port is created on the destination host and an operational port is created in its place. The **NDIS_SWITCH_PORT_PARAMETERS** structure that is associated with the OID_SWITCH_PORT_CREATE request for the operational port contains the same data that was used to create the port on the source host.

   If the operational port is created successfully, port policies are added to the operational port.

3. If the settings are successfully applied to the operational port on the destination host, a save operation is issued for the operational port on the source host.

4. If the save operation is completed successfully, the operational port and its network adapter connection are deleted on the source host in the following way:

   a. The network connection is first disconnected through an OID set request of OID_SWITCH_NIC_DISCONNECT. After this OID request is completed, the network adapter connection on the source host is deleted through an OID set request of OID_SWITCH_NIC_DELETE.

   b. After the network adapter connection is deleted, the operational port is torn down through an OID set request of OID_SWITCH_PORT_TEARDOWN. After this OID request is completed, the operational port is deleted through an OID set request of OID_SWITCH_PORT_DELETE.

5. A network adapter connection is created for the operational port on the destination host through an OID set request of OID_SWITCH_NIC_CREATE. If this OID request completes successfully, the network adapter connection is established on the associated operation port through an OID set request of OID_SWITCH_NIC_CONNECT.

   If the network adapter connection is established successfully, the run-time data for the operational port and network adapter connection is restored on the target host.

   At this point, the underlying extensions can perform resource reservation and validation on the network adapter connection.

Writing Hyper-V Extensible Switch Extensions Topics

Article • 12/15/2021

The following topics provide information about how to write Hyper-V extensible switch extensions:

Hyper-V Extensible Switch Send and Receive Operations

Hyper-V Extensible Switch OID Requests

Managing Hyper-V Extensible Switch Source and Destination Port Data

Managing Hyper-V Extensible Switch Policies

Managing Hyper-V Extensible Switch Feature Status Information

Managing Hyper-V Extensible Switch Run-Time Data

Managing Physical Network Adapters

Note Extensible switch extensions are based on the NDIS filter driver architecture. For more information, see NDIS Filter Drivers.

Hyper-V Extensible Switch Send and Receive Operations

Article • 12/15/2021

This section describes the send and receive operations for Hyper-V extensible switch extensions. The following table describes the operations that an extension can perform on packets that are sent or received over the extensible switch data path.

Operation	Capturing Extensions	Filtering Extensions	Forwarding Extensions
Originating packets	X	X	X
Cloning packets		X	X
Forwarding packets			X

This section includes the following topics:

Originating Packet Traffic

Cloning Packet Traffic

Forwarding Packets to Hyper-V Extensible Switch Ports

For more information about the extensible switch data path, see Hyper-V Extensible Switch Data Path.

Originating Packet Traffic

Article • 12/15/2021

This topic describes how Hyper-V extensions originate new packets and inject them into the extensible switch data path.

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*. For more information about the extensions, see Hyper-V Extensible Switch Extensions.

Extensible switch extensions can only inject new packets into the extensible switch ingress data path. This ensures that the extensible switch interface can filter and forward these packets correctly. Extensions must follow these guidelines for injecting new packets into the ingress data path:

- The extension must first allocate a **NET_BUFFER_LIST** structure for a new packet.

- After the extension allocates a **NET_BUFFER_LIST** structure for a new packet, it must call the *AllocateNetBufferListForwardingContext* handler function to allocate the extensible switch forwarding context for the packet.

  The forwarding context resides in the out-of-band (OOB) data of the packet. It contains forwarding information for the packet, such as its source port and an array of one or more destination ports.

  For more information about the forwarding context, see Hyper-V Extensible Switch Forwarding Context.

- After the extension calls *AllocateNetBufferListForwardingContext*, the source port for the packet will be set to **NDIS_SWITCH_DEFAULT_PORT_ID**. A packet with a source port identifier of **NDIS_SWITCH_DEFAULT_PORT_ID** is trusted and bypasses the extensible switch port policies, such as access control lists (ACLs) and quality of service (QoS).

  The extension may want the packet to be treated as if it originated from a particular port. This allows the policies for that port to be applied to the packet. The extension calls *SetNetBufferListSource* to change the source port for the packet.

  However, there may be situations where the extension may want to assign the packet's source port identifier to **NDIS_SWITCH_DEFAULT_PORT_ID**. For example, the extension may want to set the source port identifier to **NDIS_SWITCH_DEFAULT_PORT_ID** for proprietary control packets that are sent to a device on the external network.

- If the forwarding extension is sending a new packet on the ingress data path, it must determine the destination ports for the packet. For more information on this procedure, see Adding Extensible Switch Destination Port Data to a Packet.

  Note A capturing or filtering extension cannot add new destination ports to the new packet.

- When the extension creates a new packet, the packet data is located in local, or *trusted*, memory in the parent operating system of the Hyper-V parent partition. This memory is not accessible by the child partition. Therefore, it is considered "safe" from unsynchronized updates by the guest operating system that runs in that partition.

  The extension must obtain the **NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO** union for the new packet by using the **NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL** macro. The extension must set the **IsPacketDataSafe** member to TRUE. This specifies that all of the packet data is located in trusted memory.

- When the extension calls **NdisFSendNetBufferLists** to inject the packet into the ingress data path, it must set the *Flags* parameter with the appropriate extensible switch flag settings. For more information about these flag settings, see Hyper-V Extensible Switch Send and Receive Flags.

- When NDIS calls the extension's *FilterSendNetBufferListsComplete* function to complete the send request of the new packet, the extension must call *FreeNetBufferListForwardingContext* to free the allocated forwarding context. The extension must do this before it frees or reuses the **NET_BUFFER_LIST** structure for the packet.

For more information about the extensible switch ingress and egress data paths, see Hyper-V Extensible Switch Data Path.

Cloning Packet Traffic

Article • 12/15/2021

This topic describes how Hyper-V extensible switch extensions clone, or duplicate, packets and inject them into the extensible switch data path. For more information on cloning packets, see Cloned NET_BUFFER_LIST Structures.

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*. For more information about the extensions, see Hyper-V Extensible Switch Extensions.

Extensible switch filtering and forwarding extensions can inject cloned packets into the extensible switch ingress or egress data path by following these guidelines:

- The extension must first allocate a **NET_BUFFER_LIST** structure for the cloned packet. The extension must then copy the packet data from the original packet to the cloned packet. For more information on how to clone packets, see Derived NET_BUFFER_LIST Structures.

- After the extension allocates a **NET_BUFFER_LIST** structure, it must call the *AllocateNetBufferListForwardingContext* handler function to allocate the extensible switch forwarding context for the packet.

  The forwarding context resides in the out-of-band (OOB) data of the packet. It contains forwarding information for the packet, such as its source port and an array of one or more destination ports.

  For more information about the forwarding context, see Hyper-V Extensible Switch Forwarding Context.

- The extension must copy the OOB data, including the existing source port, from the original packet to the cloned packet by calling *CopyNetBufferListInfo*. If the extension plans to inject the packet into the ingress data path, it must also copy the destination ports from the OOB data of the original packet.

  When it copies the OOB data, the extension must follow these guidelines:

  - If the filtering extension plans to inject the packet into the ingress data path, it must call *CopyNetBufferListInfo* with the NDIS_SWITCH_COPY_NBL_INFO_FLAGS_PRESERVE_DESTINATIONS flag unspecified. This causes the original packet's destination ports to not be copied to the cloned packets. When the filtering extension injects this packet into the ingress data path, destination ports will be added to the packet by either an underlying forwarding extension (if enabled in the driver stack) or the miniport edge of the extensible switch.

  - If the filtering extension plans to inject the packet into the egress data path, it must call *CopyNetBufferListInfo* with the NDIS_SWITCH_COPY_NBL_INFO_FLAGS_PRESERVE_DESTINATIONS flag specified. This causes the original packet's destination ports to be copied to the cloned packets.

- If the filtering extension is cloning or duplicating a packet that was obtained from the egress data path, it can change the destination ports for the packet after it calls *CopyNetBufferListInfo* with the NDIS_SWITCH_COPY_NBL_INFO_FLAGS_PRESERVE_DESTINATIONS flag specified. For more information on this procedure, see Modifying a Packet's Extensible Switch Source Port Data.

- If the forwarding extension is cloning or duplicating a packet that was obtained from the ingress data path, it must add new destination ports for the packet before it injects the packet into the ingress data path. For more information on this procedure, see Adding Extensible Switch Destination Port Data to a Packet.

- After the extension calls *CopyNetBufferListInfo*, the packet will be assigned the same source port information that was contained in the original packet.

  The extension can call *SetNetBufferListSource* to change the source port information in the packet's out-of-band (OOB) data.

  The extension may want the packet to be treated as if it originated from a particular port. This allows the policies for that port to be applied to the packet. The extension calls *SetNetBufferListSource* to change the source port for the packet.

  However, there may be situations where the extension may want to assign the packet's source port identifier to **NDIS_SWITCH_DEFAULT_PORT_ID**. For example, the extension may want to set the source port identifier to **NDIS_SWITCH_DEFAULT_PORT_ID** for proprietary control packets that are sent to a device on the external network.

- In the standard NDIS data path, non-extensible switch OOB data often has different values depending on whether the packet is being indicated as a send or a receive. For example, the

NDIS_IPSEC_OFFLOAD_V2_HEADER_NET_BUFFER_LIST_INFO OOB data is a union of send-and-receive–specific structures

In the extensible switch data path, all packets move through the extension driver stack as both sends and receives. Because of this, the non-extensible switch OOB data within the packet's NET_BUFFER_LIST structure will be in either a send or receive format through the duration of the flow through the driver stack.

The format of this OOB data depends on the source extensible switch port from which the packet arrived at the extensible switch. If the source port is connected to the external network adapter, the non-extensible switch OOB data will be in a receive format. For other ports, this OOB data will be in a send format.

The source port information is stored in the NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO union in the OOB data of the packet's NET_BUFFER_LIST structure. The extension obtains the data by using the NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL macro.

Note If the extension clones a packet's NET_BUFFER_LIST structure, it must take the non-extensible switch OOB data into consideration if it adds or modifies the OOB data. The extension can call *CopyNetBufferListInfo* to copy all OOB data from a source packet to a cloned packet. This function will maintain the OOB send or receive format when the data is copied to the packet.

- When the extension clones a packet, the cloned packet data is located in local, or *trusted*, memory in the parent operating system of the Hyper-V parent partition. This memory cannot be accessed by the child partition. Therefore, it is considered "safe" from unsynchronized updates by the guest operating system that runs in that partition.

  After the original packet has been cloned, the extension must obtain the NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO union in the cloned packet by using the NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL macro. The extension must set the **IsPacketDataSafe** member to TRUE. This specifies that all of the packet data is located in trusted memory.

Filtering and forwarding extensions must follow these guidelines for injecting cloned packets into the ingress or egress data path:

- The extension must call NdisFSendNetBufferLists to inject the cloned packet into the ingress data path. The extension must set the *SendFlags* parameter with the appropriate extensible switch flag settings. For more information about these flag settings, see Hyper-V Extensible Switch Send and Receive Flags.

When NDIS calls the extension's *FilterSendNetBufferListsComplete* function to complete the send request of the cloned packet, the extension must call *FreeNetBufferListForwardingContext* to free the allocated forwarding context. The extension must do this before it frees or reuses the **NET_BUFFER_LIST** structure for the packet.

Note The extension must inject the cloned packet into the ingress data path if it modifies the packet data or source port for a packet that it obtained from the egress data path. It must also inject the cloned packet into the ingress data path if the packet's destination ports are not preserved.

- The extension must call **NdisFIndicateReceiveNetBufferLists** to inject the cloned packet into the egress data path. The extension must set the *ReceiveFlags* parameter with the appropriate extensible switch flag settings.

  When NDIS calls the extension's *FilterReturnNetBufferLists* function to complete the receive request of the cloned packet, the extension must call *FreeNetBufferListForwardingContext* before it frees or reuses the **NET_BUFFER_LIST** structure for the packet.

  Note Before the forwarding extension calls **NdisFIndicateReceiveNetBufferLists**, it must have determined the cloned packet's destination ports and added this data to the packet's OOB data.

- If the extension clones a packet's **NET_BUFFER_LIST** structure, it must retain ownership of the original packet's **NET_BUFFER_LIST** structure until the cloned packet's send or receive request has completed. The extension must use the **ParentNetBufferList** member of the cloned packet's **NET_BUFFER_LIST** structure to link to the original packet's **NET_BUFFER_LIST** structure.

  Note In NDIS 6.30 (Windows Server 2012), the extension can use the **ParentNetBufferList** member to link to the original packet, but it is not required to do so. In NDIS 6.40 (Windows Server 2012 R2) and later, the extension is required to use the **ParentNetBufferList** member to link to the original packet.

  Once the cloned packet's send or receive request has completed, the extension must complete the send or receive request of the original packet.

  Note If the extension has cloned a packet's **NET_BUFFER_LIST** structure, it can complete the send or receive request of the original packet after it has been cloned.

- If the extension clones a packet, it can complete the send or receive request of the original packet as soon as it is cloned.

If the forwarding or filtering extension obtains a packet in the egress data path, it cannot inject a cloned version of the packet in this data path if the extension modified the packet data or changed the source port. However, the extension can inject these packets into the ingress data path. This allows the packet to be forwarded and filtered properly through the extensible switch data path.

Note Filtering extensions can only inject cloned packets into the ingress data path if the packet's destination ports are not preserved.

For example, assume that a packet with multiple destination ports was obtained in the extensible switch egress data path. If one destination port requires special handling, such as data encapsulation, the forwarding or filtering extension handles this by following these steps:

1. Exclude packet delivery to the port that requires special handling. The extension does this by setting the **IsExcluded** member of the destination port's **NDIS_SWITCH_PORT_DESTINATION** structure to a value of one. For more information on this procedure, see Excluding Packet Delivery to Extensible Switch Destination Ports.

2. Clone the original packet and perform the required handling of the packet data.

   Note The filtering extension must not add a destination port for the cloned packet. This data will be added later by the forwarding extension or the miniport edge of the extensible switch.

3. Forward the original packet on the egress data path by calling **NdisMIndicateReceiveNetBufferLists**.

4. Inject the cloned packet on the ingress data path by calling **NdisFSendNetBufferLists**.

For more information about the extensible switch ingress and egress data paths, see Hyper-V Extensible Switch Data Path.

Note Capturing extensions cannot clone packet traffic. However, they can originate packet traffic. For more information, see Originating Packet Traffic.

Forwarding Packets to Hyper-V Extensible Switch Ports

Article • 12/15/2021

This page describes how a Hyper-V extensible switch forwarding extension can forward packets to one or more ports. This type of extension can also forward packets to individual network adapters that are connected to the extensible switch external network adapter.

Note Only the extensible switch forwarding extension or the extensible switch itself can forward packets to extensible switch ports.

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*. For more information about extensions, see Hyper-V Extensible Switch Extensions.

If a forwarding extension is installed and enabled in the extensible switch driver stack, it is responsible for making forwarding decisions for each packet that it obtains on the extensible switch ingress data path. Based on these forwarding decisions, the extension adds destination ports into the destination port array in the out-of-band (OOB) data of the packet's **NET_BUFFER_LIST** structure. After the packet has completed its traversal of the extensible switch data path, the extensible switch interface delivers the packet to the specified destination ports.

Note If a forwarding extension is not installed or enabled, the extensible switch makes the forwarding decisions for packets it obtains from the ingress data path. The switch adds the destination ports to the OOB data of the packet's **NET_BUFFER_LIST** structure before it forwards the packet up the extensible switch egress data path.

Note If the packet is an NVGRE packet, the forwarding extension can add destination ports to the destination port array. However, the Hyper-V Network Virtualization (HNV) component of the extensible switch is responsible for determining the destination ports and forwarding the packet. For more information, see Hybrid Forwarding.

The forwarding extension can add port destinations only to packets obtained from the ingress data path. After the packet is forwarded up the egress data path, filtering and forwarding extensions can exclude packet delivery to extensible switch ports. For more information, see Excluding Packet Delivery to Extensible Switch Destination Ports.

Within the OOB data of a packet's NET_BUFFER_LIST structure, the data for destination ports are contained in an NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure. Each element in the array defines a destination port and is formatted as an NDIS_SWITCH_PORT_DESTINATION structure.

The forwarding extension can call the following Hyper-V Extensible Switch handler functions to manage the NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure and its NDIS_SWITCH_PORT_DESTINATION elements:

AddNetBufferListDestination
This function adds a single destination port to the NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure in the packet's OOB data.

GetNetBufferListDestinations
This function returns a pointer to the NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure in a packet's OOB data.

GrowNetBufferListDestinations
This function adds more destination port elements to the NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure of a packet's OOB data.

UpdateNetBufferListDestinations
This function commits the modifications that the extension made to add or exclude one or more destination ports for a packet. These changes are committed to the NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure in the packet's OOB data.

When the forwarding extension's *FilterSendNetBufferLists* function is called, the *NetBufferList* parameter contains a pointer to a linked list of NET_BUFFER_LIST structures. Each of these structures specifies a packet obtained from the ingress data path.

For each NET_BUFFER_LIST structure in this list, the forwarding extension adds destination ports for the packet by following these steps:

1. The extension makes forwarding decisions for the packet based on various types of criteria. These criteria include the following:

    - Policy criteria based on the packet's source port and network adapter connection. The forwarding extension obtains this information by using the NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL macro.

- Policy criteria for an extensible switch port based on the packet's payload data. For example, a policy for an extensible switch port may include a filter to allow delivery of only IP version 4 (IPv4) packets.

Note If the packet is an NVGRE packet, the HNV component of the extensible switch is responsible for forwarding the packet. However, the forwarding extension can apply its own policy criteria in the ingress and egress paths. For more information, see Hybrid Forwarding.

2. If the extension determines that the packet can be forwarded to one or more extensible switch ports, it must add destination ports to the packet's OOB data. For more information about this procedure, see Adding Extensible Switch Destination Port Data to a Packet.

   Note If the packet is an NVGRE packet, the forwarding extension is not required to add destination ports. For more information, see Hybrid Forwarding.

3. If the extension determines that the packet cannot be forwarded to any extensible switch port, it must drop the packet.

   Note This is not true if the packet is an NVGRE packet. For more information, see Hybrid Forwarding.

4. If the extension has added one or more destination ports for the packet, it must call **NdisFSendNetBufferLists** to forward the packet on the egress data path.

   Note If the packet is an NVGRE packet, the HNV component of the extensible switch is responsible for forwarding the packet. For more information, see Hybrid Forwarding.

For more information about the extensible switch ingress and egress data paths, see Hyper-V Extensible Switch Data Path.

Hyper-V Extensible Switch OID Requests

Article • 12/15/2021

The Hyper-V extensible switch interface includes object identifier (OID) requests that are used in the following ways:

- OID requests that are issued by an extensible switch extension to query the current configuration of the extensible switch. For example, the filter driver (also known as a *Hyper-V extensible switch extension*) can issue an OID query request of OID_SWITCH_NIC_ARRAY to obtain an array. Each element in the array specifies the configuration parameters of a network adapter that is associated with an extensible switch port.

  For more information, see Querying the Hyper-V Extensible Switch Configuration.

- OID requests that are issued by the extensible switch interface to notify underlying extensions about changes to the extensible switch configuration. For example, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_CREATE to notify extensions about the creation of an extensible switch port.

  For more information, see Receiving OID Requests about Hyper-V Extensible Switch Configuration Changes.

- OID requests from a Hyper-V child partition that are forwarded by the extensible switch interface to extensions over the extensible switch control path. This allows the extensions to obtain configuration information about the network interface that is used in the partition.

  For example, the protocol edge of the extensible switch in the extensibility interface forwards an OID set request of OID_802_3_ADD_MULTICAST_ADDRESS from a child partition down the extensible switch control path. This allows extensions to obtain the multicast address configuration that is used by the networking interface in that partition.

  For more information, see Forwarding OID Requests from a Hyper-V Child Partition.

For more information on how extensions and NDIS filter drivers handle OID requests, see Filter Module OID Requests.

Querying the Hyper-V Extensible Switch Configuration

Article • 12/15/2021

The Hyper-V extensible switch interface includes object identifier (OID) requests that are issued by an extensible switch extension to query the current configuration of the extensible switch, its ports, and its network adapter connections. These requests include the following OIDs:

OID_SWITCH_NIC_ARRAY
This OID query request returns an array. Each element in the array specifies the configuration parameters of a network adapter that is associated with an extensible switch port.

OID_SWITCH_PARAMETERS
This OID query request returns the current configuration of the extensible switch.

OID_SWITCH_PORT_ARRAY
This OID query request returns an array. Each element in the array specifies the configuration parameters for an extensible switch port.

OID_SWITCH_PORT_PROPERTY_ENUM
This OID method request returns an array. Each element in the array specifies the properties of a policy for a specified extensible switch port.

OID_SWITCH_PROPERTY_ENUM
This OID method request returns an array. Each element in the array specifies the properties of an extensible switch policy.

Note When a switch extension binds for a Hyper-v Extensible Switch, it must first issue the OID_SWITCH_PARAMETERS OID to obtain the basic switch information. If the **IsActive** member of the **NDIS_SWITCH_PARAMETERS** structure is FALSE, the extension must not issue the other query OIDs until the switch has finished activation. In this case, the **NetEventSwitchActivate** **NET_PNP_EVENT** notification specifies the switch activation event. If the **IsActive** member is TRUE at bind, the extension can safely issue the other query OIDs. Querying for the configuration while the Hyper-v Extensible Switch has not completed activation will result in the extension having an incomplete initial view of the switch configuration.

Note When an extension generates its own OID requests, it does this in the same way as any NDIS filter driver. For more information on how this is done, see Generating OID

Requests from an NDIS Filter Driver.

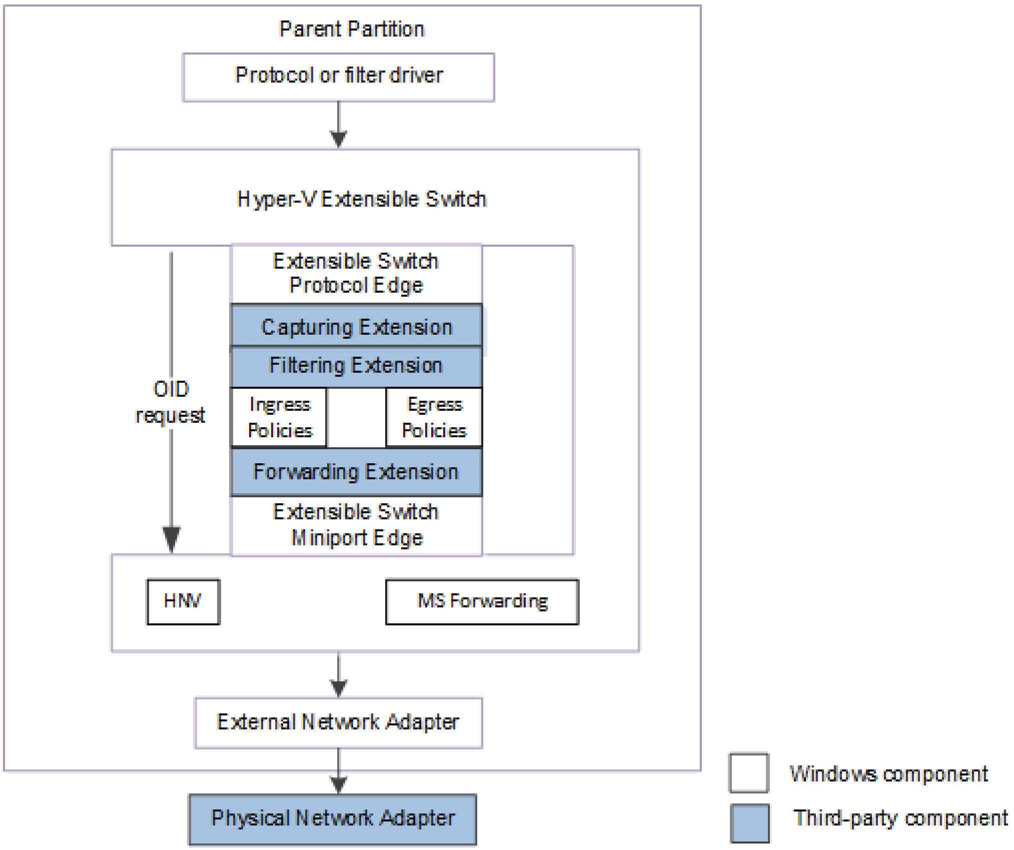
For more information on the control path for extensible switch OID requests, see Hyper-V Extensible Switch Control Path for OID Requests.

Receiving OID Requests about Hyper-V Extensible Switch Configuration Changes

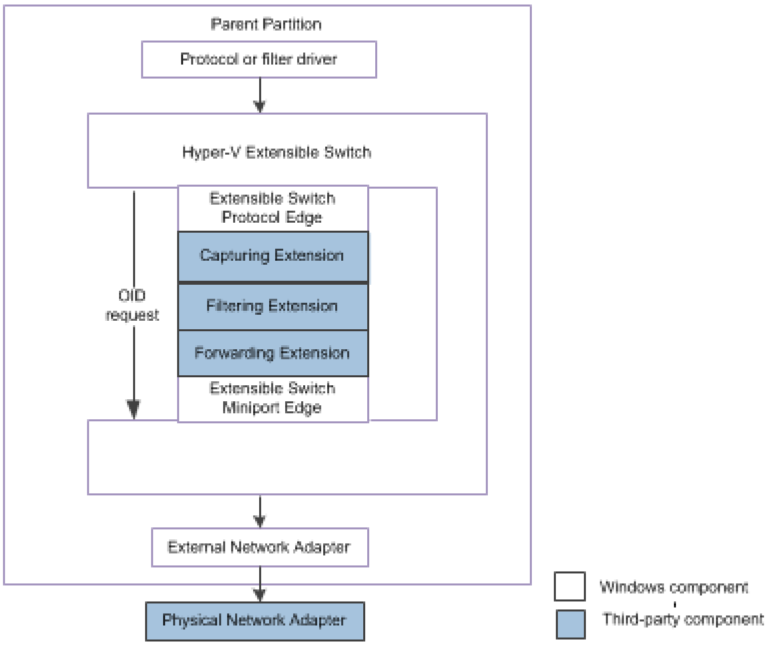
Article • 12/15/2021

The extensible switch interface notifies underlying extensions about changes to the extensible switch component configuration and policy parameters by issuing extensible switch object identifier (OID) set requests. These requests are issued by the protocol edge of the extensible switch to notify underlying extensions about changes to the extensible switch component configuration and policy parameters. These OID requests move through the extensible switch driver stack to the underlying miniport edge of the extensible switch.

The following figure shows the extensible switch control path for OID requests for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the extensible switch control path for OID requests for NDIS 6.30 (Windows Server 2012).



Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

The protocol edge of the extensible switch issues OID set requests for the following types of notifications:

- Changes to the port configuration on the extensible switch.

  For example, the protocol driver issues OID_SWITCH_PORT_CREATE to notify underlying extensions about the creation of a port on the extensible switch. Similarly, the protocol driver issues OID_SWITCH_PORT_DELETE to notify extensions about the deletion of a port.

  For more information about this type of OID notification, see Hyper-V Extensible Switch Ports.

- Changes to the network adapter connection to a port on the extensible switch.

  For example, the protocol driver issues OID_SWITCH_NIC_CONNECT to notify underlying extensions about the connection of a network adapter to a port on the extensible switch. Similarly, the protocol driver issues OID_SWITCH_NIC_DISCONNECT to notify extensions that the network adapter has been disconnected from the port.

  For more information about this type of OID notification, see Hyper-V Extensible Switch Network Adapters.

- Changes to the extensible switch port or switch policies.

  For example, the protocol driver issues OID_SWITCH_PROPERTY_ADD to notify underlying extensions about the addition of an extensible switch property.

Similarly, the protocol driver issues OID_SWITCH_PORT_PROPERTY_DELETE to notify extensions about the deletion of a port property.

For more information about this type of OID notification, see Managing Hyper-V Extensible Switch Policies.

Note The extension is not notified of changes to the default port or switch policies that are managed by the underlying miniport edge of the extensible switch.

- Save or restore run-time port data.

  For example, the protocol driver issues OID_SWITCH_NIC_SAVE to notify underlying extensions to save run-time data for a specified port on the extensible switch. These OIDs are issued when the Hyper-V state is being saved or migrated to another host. Similarly, the protocol driver issues OID_SWITCH_NIC_RESTORE to notify extensions that run-time port data is being restored on the extensible switch.

  For more information about this type of OID notification, see Managing Hyper-V Extensible Switch Run-Time Data.

The extensible switch extension miniport driver is responsible for completing these OID requests. However, with some extensible switch OID requests, an underlying extension can fail an OID request to veto a notification. For example, when the extensible switch protocol driver notifies the filter drivers about a new port that will be created on the extensible switch, it issues an OID set request of OID_SWITCH_PORT_CREATE. An underlying filtering or forwarding extension can veto the port creation by completing the OID request with STATUS_DATA_NOT_ACCEPTED.

The extensible switch extension must follow these guidelines when its *FilterOidRequest* function is called for an extensible switch OID request:

- The extension must not modify any data that is pointed to by the *OidRequest* parameter.

- For some extensible switch OID requests, the extension can complete the OID request with STATUS_DATA_NOT_ACCEPTED. This vetoes the operation on an extensible switch component for which the OID request was issued.

  For example, the extension can complete the OID_SWITCH_NIC_CREATE request with STATUS_DATA_NOT_ACCEPTED. The driver may need to do this if it cannot satisfy its configured policies on the specified port to which the network connection is being created.

  The extension can complete requests in this manner for the following OIDs:

  - OID_SWITCH_NIC_CREATE
  - OID_SWITCH_PORT_CREATE
  - OID_SWITCH_PORT_PROPERTY_ADD
  - OID_SWITCH_PORT_PROPERTY_DELETE
  - OID_SWITCH_PORT_PROPERTY_UPDATE
  - OID_SWITCH_PROPERTY_ADD
  - OID_SWITCH_PROPERTY_DELETE
  - OID_SWITCH_PROPERTY_UPDATE

- If the extension does not complete the OID request, it must call **NdisFOidRequest** to forward the request down the extensible switch driver stack.

  Note Before the driver calls **NdisFOidRequest**, the driver must call **NdisAllocateCloneOidRequest** to allocate an **NDIS_OID_REQUEST** structure and transfer the request information to the new structure.

  The extension should monitor the completion result of the OID request when its *FilterOidRequestComplete* function is called. This allows the extension to determine whether the operation on an extensible switch component completed successfully or was vetoed by an underlying extension.

  For more information on how to filter and forward an OID request, see Filtering OID Requests in an NDIS Filter Driver.

- NDIS and overlying protocol and filter drivers can issue OID requests for hardware offload technologies to the underlying physical network adapter. This includes OID requests for offload technologies, such as virtual machine queue (VMQ), Internet Protocol security (IPsec), and single root I/O virtualization (SR-IOV).

  When these OID requests arrive at the extensible switch interface, it encapsulates the OID request inside an **NDIS_SWITCH_NIC_OID_REQUEST**. Then, the protocol edge of the extensible switch issues an OID request of OID_SWITCH_NIC_REQUEST that contains this structure.

- An extensible switch forwarding extension can provide support for an NDIS hardware offload technology on one or more physical adapters that are bound to the external network adapter. In this configuration, the extensible switch external network adapter is bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX intermediate driver is bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*. For more information about extensible switch teams, see Types of Physical Network Adapter Configurations.

In this configuration, the extensible switch extensions are exposed to every network adapter in the team. This allows the forwarding extension in the extensible switch driver stack to manage the configuration and use of individual network adapters in the team. For example, the extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. Such an extension is known as a *teaming provider*. For more information about teaming providers, see Teaming Provider Extensions.

By handling the OID request of OID_SWITCH_NIC_REQUEST, teaming providers can participate in the configuration of the adapter team for hardware offloads. For example, the extension can generate its own OID request of OID_SWITCH_NIC_REQUEST to configure a physical adapter with parameters for the hardware offload.

For more information on how to handle the OID_SWITCH_NIC_REQUEST OID request, see Forwarding OID Requests to Physical Network Adapters.

Note Extension filter drivers can generate OID requests of OID_SWITCH_NIC_REQUEST to issue private OIDs to any physical adapter that is bound to the extensible switch external network adapter.

Note Stack restart requests using **NdisFRestartFilter** will not complete while an extensible switch OID request is pending. For this reason, an extension that is waiting for a stack restart must complete any ongoing OID requests.

For more information on the control path for extensible switch OID requests, see Hyper-V Extensible Switch Control Path for OID Requests.

Forwarding OID Requests from a Hyper-V Child Partition

Article • 12/15/2021

Multicast object identifier (OID) requests, including OID_802_3_ADD_MULTICAST_ADDRESS and OID_802_3_DELETE_MULTICAST_ADDRESS, are issued by overlying protocol and filter drivers that run in the following:

- The management operating system that runs in the Hyper-V parent partition.
- The guest operating system that runs Windows Vista or a later version of the Windows operating system in the Hyper-V child partition.

The extensible switch interface forwards these OID requests down the extensible switch control path. This allows the extensions to obtain configuration information about the network interface that is used in the partition.

For example, the protocol edge of the extensible switch forwards an OID set request of OID_802_3_ADD_MULTICAST_ADDRESS from a child partition down the extensible switch control path. This allows extensions to obtain the multicast address configuration that is used by the networking interface in that partition.

When these multicast OID requests arrive at the extensible switch interface, the protocol edge of the extensible switch encapsulates the OID request within an **NDIS_SWITCH_NIC_OID_REQUEST** structure. The protocol edge also sets the members of this structure in the following way:

- The **SourcePortId** and **SourceNicIndex** members are set to the corresponding values for the port and network adapter used by the partition from which the OID request originated.

  Note If the multicast OID request was originated from the management operating system, the protocol edge sets these members to the values for the extensible switch internal network adapter.

- The **DestinationPortId** and **DestinationNicIndex** members are set to zero. This specifies that the encapsulated OID request is to be delivered to extensions in the control path.
- The **OidRequest** member is set to the address of an **NDIS_OID_REQUEST** structure for the encapsulated OID request.

The protocol edge then issues the OID_SWITCH_NIC_REQUEST request to forward the encapsulated OID request down the extensible switch control path. Underlying forwarding extensions can inspect these encapsulated OID requests and retain the multicast address information that they specify. For example, the extension may need this information if it originates multicast packets that it forwards to an extensible switch port.

For more information about the extensible switch control path, see Hyper-V Extensible Switch Control Path.

Managing Hyper-V Extensible Switch Source and Destination Port Data

Article • 12/15/2021

This section describes how a Hyper-V extensible switch extension manages the source and destination extensible switch port data within a packet's out-of-band (OOB) data. This section includes the following topics:

Managing Hyper-V Extensible Switch Source Port Data

Managing Hyper-V Extensible Switch Destination Port Data

Managing Hyper-V Extensible Switch Source Port Data

Article • 12/15/2021

This section describes how a Hyper-V extensible switch extension manages the source extensible switch port data within a packet's out-of-band (OOB) data. This section includes the following topics:

Querying a Packet's Extensible Switch Source Port Data

Modifying a Packet's Extensible Switch Source Port Data

Querying a Packet's Extensible Switch Source Port Data

Article • 12/15/2021

The Hyper-V extensible switch source port is specified by the **SourcePortId** member in the NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO structure. This structure is contained in the out-of-band (OOB) forwarding context of the packet's NET_BUFFER_LIST structure. For more information on this context, see Hyper-V Extensible Switch Forwarding Context.

The extensible switch extension accesses the NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO structure by using the NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL macro. The following example shows how the driver can obtain the source port identifier from the packet's **NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO** structure.

```C++
PNDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO fwdDetail;
NDIS_SWITCH_PORT_ID sourcePortId;

fwdDetail = NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL(NetBufferList);
sourcePortId = fwdDetail->SourcePortId;
```

Modifying a Packet's Extensible Switch Source Port Data

Article • 12/15/2021

The Hyper-V extensible switch source port is specified by the **SourcePortId** member in the **NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO** structure. This structure is contained in the out-of-band (OOB) forwarding context of the packet's **NET_BUFFER_LIST** structure. For more information about this context, see Hyper-V Extensible Switch Forwarding Context.

The extensible switch extension must follow these guidelines for modifying a packet's source port identifier:

- The extensible switch extension must call *SetNetBufferListSource* to modify the source port for a packet. The extension must not directly modify the **SourcePortId** member of the **NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO** structure.

- If the extension is creating or cloning a packet, it must call the *AllocateNetBufferListForwardingContext* function after it calls **NdisAllocateNetBufferList**. This function allocates an extensible switch context area for the OOB data that is used for forwarding information for the packet.

  When the extension calls *AllocateNetBufferListForwardingContext*, the **SourcePortId** member is set to **NDIS_SWITCH_DEFAULT_PORT_ID**. This specifies that the packet originated from an extension instead of arriving at an extensible switch port.

  Packets with a source port of **NDIS_SWITCH_DEFAULT_PORT_ID** are treated by the extensible switch extension data path as privileged and trusted. Such traffic should not be subjected to the policies that are applied to packets from other source ports. For example, packets with a source port identifier of **NDIS_SWITCH_DEFAULT_PORT_ID** bypass the built-in extensible switch policies that are applied by the underlying miniport edge of the extensible switch. These policies include access control lists (ACLs) and quality of service (QoS).

  When the extension is originating packet traffic, it should use the source port of **NDIS_SWITCH_DEFAULT_PORT_ID** sparingly and carefully. In most cases, the extension should modify the source port identifier to an active port on the extensible switch. This allows the policies of that port to be applied to the packet.

However, there may be situations where the extension has to use the source port of **NDIS_SWITCH_DEFAULT_PORT_ID** for packets that it originates. For example, if the extension originates a control packet that has to be sent to its destination on the physical or virtual network, it should use **NDIS_SWITCH_DEFAULT_PORT_ID** for the source port identifier. This ensures that the packet will not be filtered and rejected by underlying extensions in the extensible switch driver stack.

Managing Hyper-V Extensible Switch Destination Port Data

Article • 12/15/2021

This section describes how a Hyper-V extensible switch extension manages the destination extensible switch port data within a packet's out-of-band (OOB) data. This section includes the following topics:

Querying a Packet's Extensible Switch Destination Port Data

Adding Extensible Switch Destination Port Data to a Packet

Excluding Packet Delivery to Extensible Switch Destination Ports

Querying a Packet's Extensible Switch Destination Port Data

Article • 12/15/2021

Each Hyper-V extensible switch destination port is specified by an **NDIS_SWITCH_PORT_DESTINATION** element within the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. This array is contained in the out-of-band (OOB) forwarding context of the packet's **NET_BUFFER_LIST** structure. For more information on this context, see Hyper-V Extensible Switch Forwarding Context.

The extensible switch extension calls the *GetNetBufferListDestinations* function to obtain a pointer to the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure within a packet's **NET_BUFFER_LIST** structure. Individual **NDIS_SWITCH_PORT_DESTINATION** elements within this structure can be accessed by using the **NDIS_SWITCH_PORT_DESTINATION_AT_ARRAY_INDEX** macro.

To improve performance, a forwarding extension can call the *GrowNetBufferListDestinations* function instead of *GetNetBufferListDestinations* to obtain a pointer to the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. The extension does this if it determines that it needs additional array elements in the packet's OOB data for destination ports. For more information, see Adding Extensible Switch Destination Port Data to a Packet.

Note Only packets obtained from the extensible switch egress data path will contain destination port information. For more information, see Hyper-V Extensible Switch Data Path.

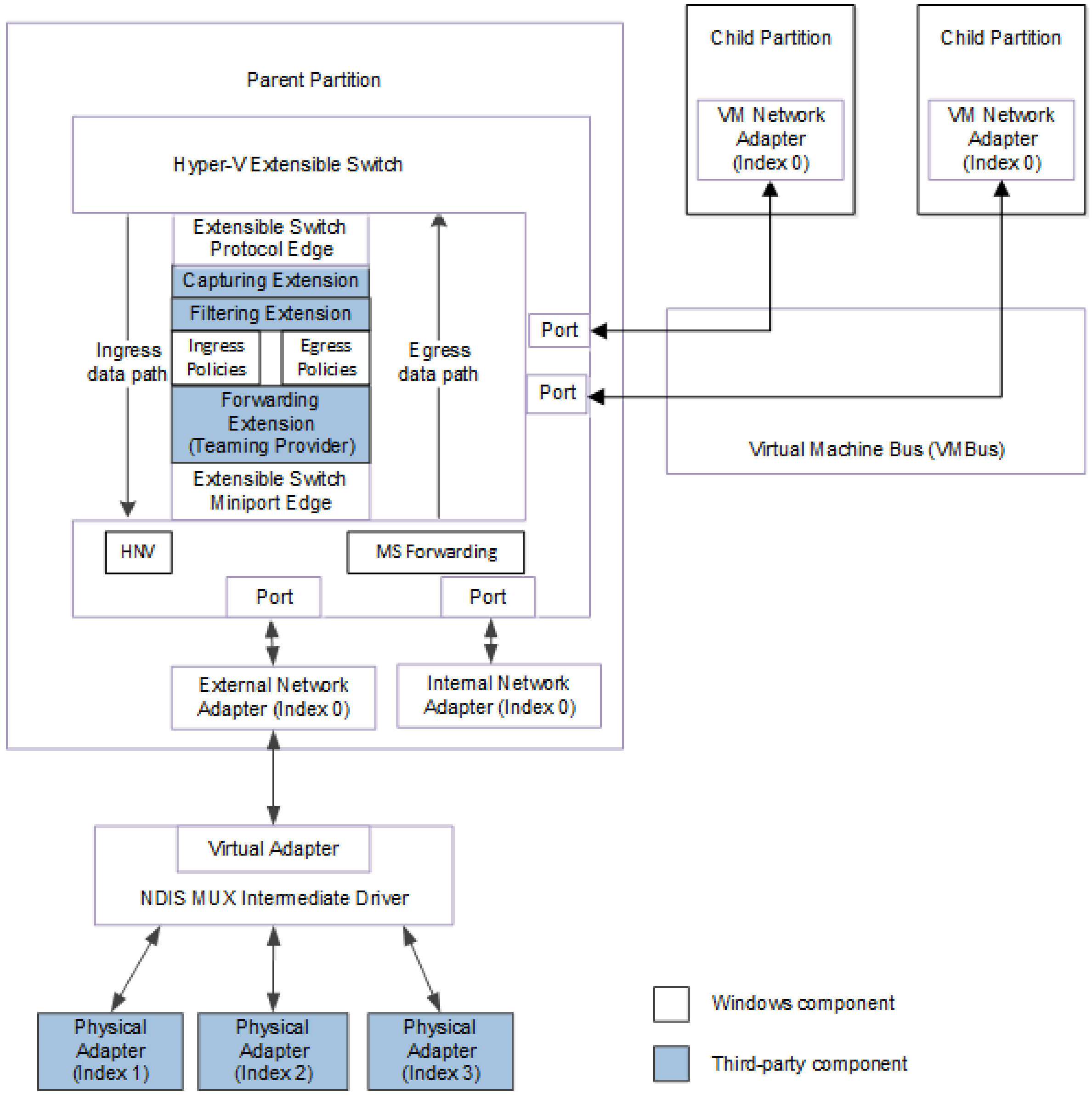
Adding Extensible Switch Destination Port Data to a Packet

Article • 12/15/2021

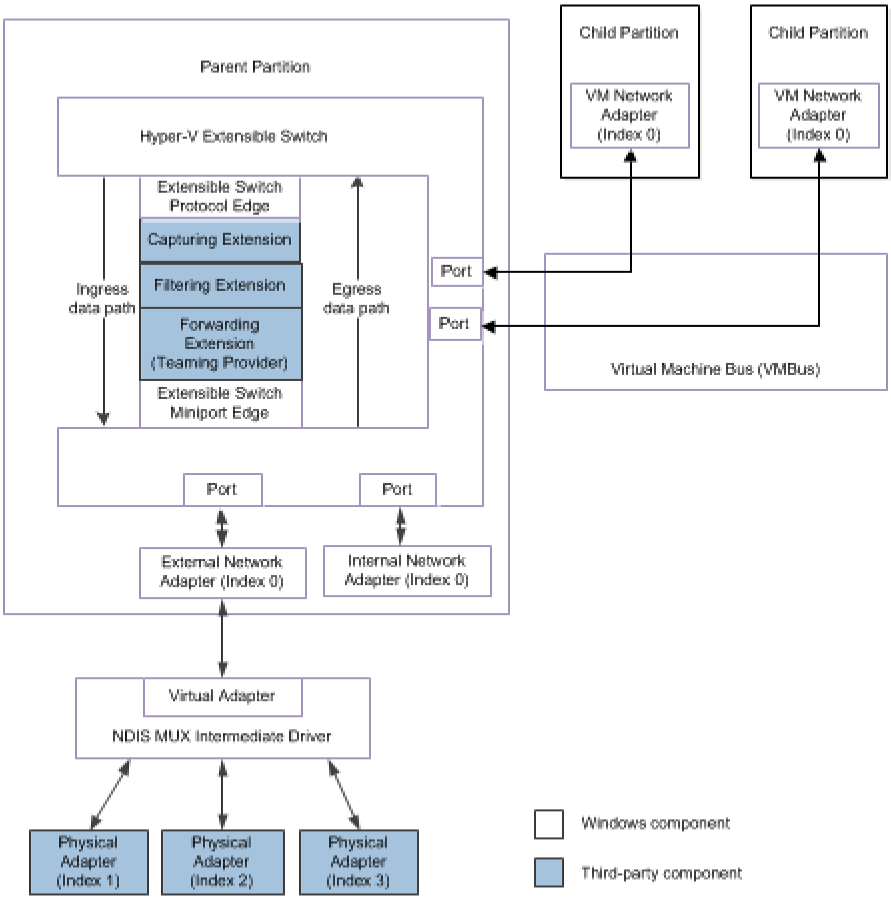
This topic describes how Hyper-V extensible switch forwarding extensions can specify the delivery of packets to one or more destination ports. These extensions can also forward packets to individual physical network adapters that are bound to the extensible switch external network adapter.

Note Only a forwarding extension or the switch itself can forward packets to extensible switch ports or individual network adapters.

The following figure shows the data path for packet traffic through the extensible switch driver stack for NDIS 6.40 (Windows Server 2012 R2) and later. Both figures also show the data path for packet traffic to or from the network adapters that are connected to extensible switch ports.


Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Ingress Policies
Egress Policies
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
Port
Port
HNV
MS Forwarding
Port
Port
External Network Adapter (Index 0)
Internal Network Adapter (Index 0)
Child Partition
VM Network Adapter (Index 0)
Child Partition
VM Network Adapter (Index 0)
Virtual Machine Bus (VMBus)
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index 3)
Windows component
Third-party component

The following figure shows the data path for packet traffic through the extensible switch driver stack for NDIS 6.30 (Windows Server 2012).


Parent Partition
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension (Teaming Provider)
Extensible Switch Miniport Edge
Ingress data path
Egress data path
Port
Port
Port
Port
External Network Adapter (Index 0)
Internal Network Adapter (Index 0)
Child Partition
VM Network Adapter (Index 0)
Child Partition
VM Network Adapter (Index 0)
Virtual Machine Bus (VMBus)
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index 3)
Windows component
Third-party component

Each extensible switch destination port is specified by an **NDIS_SWITCH_PORT_DESTINATION** element within the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. This array is contained in the out-of-band (OOB) forwarding context of the packet's **NET_BUFFER_LIST** structure. For more information on this context, see Hyper-V Extensible Switch Forwarding Context.

If a forwarding extension is bound and enabled in the extensible switch driver stack, it is responsible for determining the destination ports for every packet obtained from the extensible switch ingress data path, unless the packet is an NVGRE packet. For more information about this data path, see Overview of the Hyper-V Extensible Switch Data Path. For more information about NVGRE packets, see Hybrid Forwarding.

Note If a forwarding extension is not bound or enabled in the driver stack, the extensible switch determines the destination ports for packets it obtains from the ingress data path.

The forwarding extension must follow these guidelines when it determines destination ports for a packet obtained on the ingress data path:

- The extension must initialize an **NDIS_SWITCH_PORT_DESTINATION** structure within the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure with the destination port information.

If the destination port is not connected to the external network adapter, the extension must set the **NicIndex** member of the **NDIS_SWITCH_PORT_DESTINATION** structure to **NDIS_SWITCH_DEFAULT_NIC_INDEX**.

If the destination port is connected to the extensible switch external network adapter, the extension can specify the index of an underlying physical network adapter to forward the send request to. The extension does this by setting the **NicIndex** member to the nonzero NDIS_SWITCH_NIC_INDEX value of the destination network adapter that is bound to the external network adapter.

For more information, see Forwarding Packets to Physical Network Adapters.

- The extension must add destination ports to a packet's OOB data only for those ports that have active network adapter connections. If the extension had forwarded an OID_SWITCH_NIC_DISCONNECT request, it must not add a destination port that is associated with the disconnected network adapter.

- To improve performance, the extension must only add port destinations that are valid for packet delivery. In this case, the extension must set the **IsExcluded** member of the destination port's **NDIS_SWITCH_PORT_DESTINATION** structure to FALSE.

- To retain the 802.1Q virtual local area network (VLAN) data in a packet before it is delivered to a port, the extension sets the **PreserveVLAN** member to TRUE.

  To remove the 802.1Q virtual local area network (VLAN) data in a packet before it is delivered to a port, the extension sets the **PreserveVLAN** member to FALSE.

- To retain the 802.1Q priority data in a packet before it is delivered to a port, the extension sets the **PreservePriority** member to TRUE.

  To remove the 802.1Q priority data in a packet before it is delivered to a port, the extension sets the **PreservePriority** member to FALSE.

- If the forwarding extension adds multiple destination ports for a packet, it must follow these steps:

  1. The extension first accesses the packet's **NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO** structure by using the **NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL** macro. The extension then reads the **NumAvailableDestinations** member to determine how many unused destination port elements are available in the destination port array. If the extension requires more destination ports than are available

in the array, it must call the *GrowNetBufferListDestinations* function to allocate space for additional destination ports in the array.

    When the extension calls *GrowNetBufferListDestinations*, it sets the *NumberOfNewDestinations* parameter to the number of new destination ports to be added to the packet.

    The extension also sets the *NetBufferLists* parameter to a pointer to the packet's **NET_BUFFER_LIST** structure.

    Note If there are available destination ports in the array, the extension should not call *GrowNetBufferListDestinations*.

  2. If the *GrowNetBufferListDestinations* function returns successfully, it has added the additional destination ports to the end of the destination array in the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. A pointer to this structure is returned in the *Destinations* parameter.

     Note If the *GrowNetBufferListDestinations* function cannot allocate the requested number of destination ports, it returns NDIS_STATUS_RESOURCES.

  3. The extension specifies new destination port elements in the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. The extension initializes each new destination port as an **NDIS_SWITCH_PORT_DESTINATION** structure.

     The extension initializes new destination ports to the array starting at the **NumDestinations** offset. **NumDestinations** is a member of the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure.

  4. After the extension has finished adding or modifying destination port elements, it must call *UpdateNetBufferListDestinations* to commit those changes.

- If the extension adds a single destination port for a packet, it must follow these steps:

  1. The extension initializes the destination port information for the packet in an extension-allocated **NDIS_SWITCH_PORT_DESTINATION** structure.

  2. The extension calls *AddNetBufferListDestination* to commit the changes to the **NET_BUFFER_LIST** structure for the packet. The extension passes the address of the **NDIS_SWITCH_PORT_DESTINATION** structure in the *Destination* parameter.

Note The extension should not call the *UpdateNetBufferListDestinations* function to commit the changes to a packet with only one destination port.

- When the forwarding extension calls *AddNetBufferListDestination* or *UpdateNetBufferListDestinations* to commit the changes for destination ports, the extensible switch interface will not delete the extensible switch ports that are specified in the elements of the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. After the packet send or receive operation is complete, the interface is free to delete the port if it is necessary.

  Note After the forwarding extension commits the changes for destination ports to the forwarding context, destination ports cannot be removed and only the **IsExcluded** member of a destination port's **NDIS_SWITCH_PORT_DESTINATION** structure can be changed. For more information, see Excluding Packet Delivery to Extensible Switch Destination Ports.

- The forwarding extension must synchronize its handling of object identifier (OID) set requests of OID_SWITCH_NIC_DISCONNECT with its code that adds destination ports for the disconnected network adapter.

  If the forwarding extension's *FilterOidRequest* is called for an OID_SWITCH_NIC_DISCONNECT request, the extension can do one of the following:

  - If the extension called **NdisFOidRequest** to forward this OID request, it must not specify the port with the disconnected network adapter as a destination port for the packet.

    Note If the only destination port for the packet is the one with the disconnected network adapter, the extension must drop the packet.

  - The extension can return NDIS_STATUS_PENDING to complete the request asynchronously. This allows the extension to add the port with the disconnected network adapter as a destination port for the packet. This also allows the extension to call *AddNetBufferListDestination* or *UpdateNetBufferListDestinations* and complete the addition of destination ports to a packet.

    The extension may want to do this for packets that it has to forward to a port before it is torn down.

    Note If the extension returns NDIS_STATUS_PENDING, it can also call *ReferenceSwitchPort* to increment the reference counter for the port with the disconnected network adapter. However, the extension cannot forward the OID request until after it calls *DereferenceSwitchPort* to decrement the reference counter for the port.

- If the number of destination ports is zero, the forwarding extension must call **NdisMSendNetBufferListsComplete** to drop the packet. The extension must also call *ReportFilteredNetBufferLists* to notify the extensible switch interface about the dropped packet.

  Note If the forwarding extension obtained a linked list of **NET_BUFFER_LIST** structures for multiple packets from the ingress data path, it should create a separate list of dropped packets. By doing this, the extension can call **NdisMSendNetBufferListsComplete** and *ReportFilteredNetBufferLists* just once.

- If the number of destination ports is greater than zero, the forwarding extension must call **NdisFSendNetBufferLists** to forward the packet over the ingress data path to the miniport edge of the extensible switch.

  Note If the forwarding extension obtained a linked list of **NET_BUFFER_LIST** structures for multiple packets from the ingress data path, it should create a separate list of forwarded packets. By doing this, the extension can call **NdisFSendNetBufferLists** just once to forward the list of packets. In addition, the extension should maintain separate lists to forward packets that have the same destination ports. For more information, see Hyper-V Extensible Switch Send and Receive Flags.

Excluding Packet Delivery to Extensible Switch Destination Ports

Article • 12/15/2021

This topic describes how Hyper-V extensible switch extensions can exclude the delivery of packets to extensible switch ports. The destination ports for a packet are specified within the out-of-band (OOB) forwarding context within the packet's **NET_BUFFER_LIST** structure. For more information on this context, see Hyper-V Extensible Switch Forwarding Context.

Note This page assumes that you are familiar with the information and diagrams in Overview of the Hyper-V Extensible Switch and Hybrid Forwarding.

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*. For more information about the extensions, see Hyper-V Extensible Switch Extensions.

Filtering and forwarding extensions can exclude the delivery of packets obtained on the extensible switch ingress or egress data paths. Excluding packet delivery can be done in the following ways:

- The extension can drop the packet by completing the packet request or indication. This excludes the delivery of a packet to any extensible switch port. This method can be used on packets that have one or more destination ports.

  For packets obtained on the extensible switch ingress data path, the extension completes the packet send request by calling **NdisFSendNetBufferListsComplete**.

  For packets obtained on the extensible switch egress data path, the extension completes the packet receive indication by calling **NdisFReturnNetBufferLists**.

- For packets obtained on the egress data path with multiple destination ports, the extension can exclude packet delivery by modifying the data for one or more destination ports. The extension does this by setting the **IsExcluded** member of the destination port's **NDIS_SWITCH_PORT_DESTINATION** structure to a value of one. This method allows the packet to be delivered to those ports whose **IsExcluded** value is set to zero.

  Note Packets obtained on the ingress data path do not contain destination ports. This data is only available after the extensible switch forwards the packet up the egress data path.

After the extension has modified the destination port's **IsExcluded** value, it must forward the packet in the egress data path to overlying extensions. However, if the **IsExcluded** data for all the packet's destination ports is set to one, the extension should drop the packet by completing the packet receive indication instead of forwarding it.

Note After an extension has set the destination port's **IsExcluded** value to one, overlying extensions on the egress data path cannot change this value to zero.

Note Capturing extensions cannot exclude the delivery of packets to extensible switch ports.

Filtering and forwarding extensions must follow these guidelines for excluding packet delivery to extensible switch ports:

- On the extensible switch ingress data path, filtering and forwarding extensions can exclude packet delivery based on a policy criteria for a packet's source port or data.

  The source port information is stored in the **NDIS_SWITCH_FORWARDING_DETAIL_NET_BUFFER_LIST_INFO** union in the OOB data of the packet's **NET_BUFFER_LIST** structure. The extension obtains the data by using the **NET_BUFFER_LIST_SWITCH_FORWARDING_DETAIL** macro.

  If the extension excludes the delivery of a packet obtained from the ingress data path, it must drop the packet by completing the packet send request.

- On the extensible switch ingress data path, forwarding extensions determine a packet's destination ports and add this information to the packet's OOB data. Based on policy criteria enforced by the extension, it can exclude packet delivery to a port by not adding its destination port information to the OOB data.

  For more information about this procedure, see Adding Extensible Switch Destination Port Data to a Packet.

- On the extensible switch egress data path, filtering and forwarding extensions can exclude the delivery of the packet based on policy criteria. For example, filtering extensions can exclude packet delivery based on policy criteria for a packet's source port or destination ports.

  Extensions exclude the delivery of a packet to destination ports by following these steps:

  1. The extension obtains the packet's destination ports by calling *GetNetBufferListDestinations*. If the call returns NDIS_STATUS_SUCCESS, the *Destinations* parameter contains a pointer to an

NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure. This structure specifies the extensible switch destination ports of the packet. Each destination port is formatted as an NDIS_SWITCH_PORT_DESTINATION structure.

Note If the **NumDestinations** member of the NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY structure contains a value of zero, the packet has no data for destination ports.

2. The extension excludes the packet delivery to an extensible switch port by setting the **IsExcluded** member of the destination port's NDIS_SWITCH_PORT_DESTINATION structure to a value of one.

   Note If the extension excludes delivery of the packet to all of its destination ports, the extension must drop the packet by completing the packet's receive indication.

3. If the extension excludes delivery to one or all destination ports in a packet, it must do the following:

   - The extension must call *UpdateNetBufferListDestinations* to commit these changes to the packet's OOB data.

   - The extension must call *ReportFilteredNetBufferLists*. When this function is called, the extensible switch interface increments counters and logs events for the excluded packet. The extension must make this call before it forwards the packet in the extensible switch data path from which it obtained the packet.

   Similarly, if the extension completes the packet send request or indication to exclude delivery to all ports for the packet, it must also call *ReportFilteredNetBufferLists*.

   Note The extension can create a linked list of NET_BUFFER_LIST structures for packets that the extension is excluding. When the extension calls *ReportFilteredNetBufferLists*, it sets the *NetBufferLists* parameter to a pointer to the linked list.

For more information about the extensible switch ingress and egress data paths, see Hyper-V Extensible Switch Data Path.

Managing Hyper-V Extensible Switch Policies

Article • 12/15/2021

This section describes how a Hyper-V extensible switch extension manages port and switch policies. This section includes the following topics:

Managing Port Policies

Managing Switch Policies

For more information on extensible switch policies, see Overview of Hyper-V Extensible Switch Policies.

Managing Port Policies

Article • 12/15/2021

Hyper-V extensible switch filtering and forwarding extensions can be provisioned with the properties of standard and custom port properties. Once provisioned, these extensions enforce the policies when they filter packets obtained on the extensible switch ingress data path. For more information about these policies, see Port Policies.

The Hyper-V extensible switch interface uses the following object identifiers (OIDs) to provision filtering and forwarding extensions with the properties of standard and custom port policies:

OID_SWITCH_PORT_PROPERTY_ADD
This OID set request is issued by the protocol edge of the extensible switch to notify underlying extensions of the addition of a property at the WMI management layer. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_PORT_PROPERTY_PARAMETERS** structure.

Note Custom port properties are specified by an **NDIS_SWITCH_PORT_PROPERTY_TYPE** enumeration value of **NdisSwitchPortPropertyTypeCustom**. Standard port properties are specified by an **NDIS_SWITCH_PORT_PROPERTY_TYPE** enumeration value of **NdisSwitchPortPropertyTypeSecurity**, **NdisSwitchPortPropertyTypeVlan**, and **NdisSwitchPortPropertyTypeProfile**.

OID_SWITCH_PORT_PROPERTY_UPDATE
This OID set request is issued by the protocol edge of the extensible switch to inform underlying extensions of the update of a property at the WMI management layer. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_PORT_PROPERTY_PARAMETERS** structure.

OID_SWITCH_PORT_PROPERTY_DELETE
This OID set request is issued by the protocol edge of the extensible switch to inform underlying extensions of the deletion of a property at the WMI management layer. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_PORT_PROPERTY_DELETE_PARAMETERS** structure.

OID_SWITCH_PORT_PROPERTY_ENUM
This OID method request is sent by the extension to query the underlying miniport edge of the extensible switch about the currently configured properties for a specified port on the extensible switch. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PORT_PROPERTY_ENUM_PARAMETERS structure that specifies the parameters for the policy enumeration of a specified port.

- An array of NDIS_SWITCH_PORT_PROPERTY_ENUM_INFO structures. Each of these structures contains information about the properties of an extensible switch port policy.

  Note If the **NumProperties** member of the NDIS_SWITCH_PORT_PROPERTY_ENUM_PARAMETERS structure is set to zero, no NDIS_SWITCH_PORT_PROPERTY_ENUM_INFO structures are returned.

Note The extension must not originate OID set requests of OID_SWITCH_PORT_PROPERTY_ADD. OID_SWITCH_PORT_PROPERTY_UPDATE, or OID_SWITCH_PORT_PROPERTY_DELETE.

The extensible switch extension must follow these guidelines when it handles an OID set request of OID_SWITCH_PORT_PROPERTY_ADD, OID_SWITCH_PORT_PROPERTY_UPDATE, or OID_SWITCH_PORT_PROPERTY_DELETE:

- The extension must not modify the NDIS_SWITCH_PORT_PROPERTY_PARAMETERS or NDIS_SWITCH_PORT_PROPERTY_DELETE_PARAMETERS structure that is associated with the OID request.

- The extension must handle these OID requests if the extension manages the property. Depending on the OID request, the extension must inspect the following members of the NDIS_SWITCH_PORT_PROPERTY_PARAMETERS or NDIS_SWITCH_PORT_PROPERTY_DELETE_PARAMETERS structures to determine whether it manages the port property:

  - The **PropertyType** member. This member specifies the type of the port property. Custom port properties have a **PropertyType** member value of **NdisSwitchPortPropertyTypeCustom**. Standard port properties have other property type values. For example, standard VLAN port policies have a property type value of **NdisSwitchPortPropertyTypeVlan**.

  - The **PropertyId** member. This member specifies a proprietary GUID value for a custom port property. This GUID value is created by the independent software vendor (ISV) who also defines the format of the custom extensible switch property.

    Note The extension must ignore this member for standard port policies.

- The extension must handle an OID_SWITCH_PORT_PROPERTY_UPDATE set request if the extension was previously provisioned with a port property that matches the following members of the **NDIS_SWITCH_PROPERTY_PARAMETERS** structure:

  - The **PropertyType** member.

  - The **PropertyId** member.

    Note The extension must ignore this member for standard port policies.

  - The **PropertyVersion** member. This member specifies the version of the port property that the extension was provisioned with.

  - The **PropertyInstanceId** member. This member specifies the instance of the port property that the extension was provisioned with.

- The filtering or forwarding extension can veto the addition or update of a port policy that it manages. The extension does this by completing the OID request with STATUS_DATA_NOT_ACCEPTED.

  Note Capturing extensions must not veto the addition or update of a port policy. Instead, it must forward the OID request down the extensible switch control path.

- A forwarding extension can fail the OID request for standard port properties that it does not support or if the property conflicts with its own policy configuration. In this case, the extension must complete the OID request and return the appropriate NDIS status code to report the failure.

- If the extension successfully handles the OID set request for a standard port policy, it must not complete the OID request and must forward it down the extensible switch control path.

- If the capturing or filtering extension successfully handles the OID set request for a custom port policy, it must not complete the OID request and must forward it down the extensible switch control path.

  If the forwarding extension successfully handles the OID set request for a custom port policy, it must complete the OID request and return the appropriate NDIS_STATUS_*Xxx* value.

- If the extension does not complete the OID set request, it must call **NdisFOidRequest** to forward the OID request down the extensible switch driver stack. In this case, the extensions should monitor the completion status of the OID to detect whether an underlying extension has failed the OID request.

Managing Switch Policies

Article • 12/15/2021

Hyper-V extensible switch filtering and forwarding extensions can be provisioned with the properties of custom switch properties. Once provisioned, these extensions enforce the policies when they filter packets obtained on the extensible switch ingress data path. For more information about these policies, see Switch Policies.

The Hyper-V extensible switch interface uses the following object identifiers (OIDs) to provision filtering and forwarding extensions with the properties of custom switch policies:

OID_SWITCH_PROPERTY_ADD
This OID set request is issued by the protocol edge of the extensible switch to notify underlying extensions of the addition of a property at the WMI management layer. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_PROPERTY_PARAMETERS** structure.

Note Custom switch properties are specified by an **NDIS_SWITCH_PROPERTY_TYPE** enumeration value of **NdisSwitchPropertyTypeCustom**.

OID_SWITCH_PROPERTY_UPDATE
This OID set request is issued by the protocol edge of the extensible switch to notify underlying extensions of the update of a property at the WMI management layer. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_PROPERTY_PARAMETERS** structure.

OID_SWITCH_PROPERTY_DELETE
This OID set request is issued by the protocol edge of the extensible switch to notify underlying extensions of the deletion of a property at the WMI management layer. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_PROPERTY_DELETE_PARAMETERS** structure.

OID_SWITCH_PROPERTY_ENUM
This OID method request is sent by the extension to query the underlying miniport edge of the extensible switch about the currently configured switch properties on the extensible switch. The **InformationBuffer** of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer contains the following data:

- An **NDIS_SWITCH_PROPERTY_ENUM_PARAMETERS** structure that specifies the parameters for the property enumeration of a switch policy.

- An array of **NDIS_SWITCH_PROPERTY_ENUM_INFO** structures. Each of these structures contains information about the properties of a switch policy.

  Note If the **NumProperties** member of the **NDIS_SWITCH_PROPERTY_ENUM_PARAMETERS** structure is set to zero, no **NDIS_SWITCH_PROPERTY_ENUM_INFO** structures are returned.

Note The extension must not originate OID set requests of OID_SWITCH_PROPERTY_ADD. OID_SWITCH_PROPERTY_UPDATE, or OID_SWITCH_PROPERTY_DELETE.

The extensible switch extension must follow these guidelines when it handles an OID set request of OID_SWITCH_PROPERTY_ADD, OID_SWITCH_PROPERTY_UPDATE, or OID_SWITCH_PROPERTY_DELETE:

- The extension must not modify the **NDIS_SWITCH_PROPERTY_PARAMETERS** or **NDIS_SWITCH_PROPERTY_DELETE_PARAMETERS** structure that is associated with the OID request.

- The extension must handle an OID_SWITCH_PROPERTY_UPDATE or OID_SWITCH_PROPERTY_DELETE set request if the extension has been previously provisioned with a switch property that matches the following members of the **NDIS_SWITCH_PROPERTY_PARAMETERS** or **NDIS_SWITCH_PROPERTY_DELETE_PARAMETERS** structure:

  - The **PropertyType** member that specifies the type of switch property.

    Note Starting with NDIS 6.30, only switch properties of **NdisSwitchPropertyTypeCustom** are specified by the **NDIS_SWITCH_PROPERTY_PARAMETERS** or **NDIS_SWITCH_PROPERTY_DELETE_PARAMETERS** structures.

  - The **PropertyId** member that specifies a proprietary GUID value that the extension recognizes. This GUID value is created by the independent software vendor (ISV) who also defines the format of the custom extensible switch policy property.

    Note A custom extensible switch policy property is contained within an **NDIS_SWITCH_PROPERTY_CUSTOM** structure.

- If the extension handles these OID set requests, the extension must update or delete the switch policy that matches the following members of the **NDIS_SWITCH_PROPERTY_PARAMETERS** structure:

  - The **PropertyVersion** member that specifies the version of the extensible switch policy.

  - The **PropertyInstanceId** member that specifies the instance of the extensible switch policy.

  If the values of these members do not match a switch policy property for which the extension has been previously provisioned, the extension must fail the OID set request with NDIS_STATUS_INVALID_PARAMETER. Otherwise, the extension must complete the OID set request and return NDIS_STATUS_SUCCESS.

- The filtering or forwarding extension can veto the addition, deletion, or update of a switch policy. The extension does this by completing the OID request with STATUS_DATA_NOT_ACCEPTED.

  Note Capturing extensions must not veto the addition or update of a switch policy. Instead, it must forward the OID request down the extensible switch control path.

- If the capturing or filtering extension successfully handles the OID set request for a custom switch policy, it must not complete the OID request and must forward it down the extensible switch control path.

  If the forwarding extension successfully handles the OID set request for a custom switch policy, it must complete the OID request and return the appropriate NDIS_STATUS_*Xxx* value.

- If the extension does not complete the OID set request, it must call **NdisFOidRequest** to forward the OID request down the extensible switch driver stack. In this case, the extensions should monitor the completion status of the OID to detect whether an underlying extension has failed the OID request.

Managing Hyper-V Extensible Switch Feature Status Information

Article • 12/15/2021

The Hyper-V extensible switch interface supports the ability to obtain custom status information for an extensible switch or one of its ports. This status information is known as a feature status information.

This section includes the following topics:

Managing Custom Port Feature Status Information

Managing Custom Switch Feature Status Information

For more information on feature status information, see Hyper-V Extensible Switch Feature Status Information.

Managing Custom Port Feature Status Information

Article • 12/15/2021

The Hyper-V extensible switch interface uses the following object identifier (OID) to query custom status information for an extensible switch port. This status information is known as *port feature status* information:

OID_SWITCH_PORT_FEATURE_STATUS_QUERY
This OID method request is issued by the protocol edge of the extensible switch to obtain the custom feature status information for a specified port property.

After a successful return from this OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer contains the following data:

- An **NDIS_SWITCH_PORT_FEATURE_STATUS_PARAMETERS** structure that specifies the custom feature status information that is to be returned.

  Note For a custom feature status, the **FeatureStatusType** member is set to **NdisSwitchPortPropertyTypeCustom**.

- An **NDIS_SWITCH_PORT_FEATURE_STATUS_CUSTOM** structure that contains the status information about a custom property assigned to an extensible switch port.

  When the protocol edge of the extensible switch issues the OID_SWITCH_PORT_FEATURE_STATUS_QUERY request, it sets the **FeatureStatusCustomBufferLength** and **FeatureStatusCustomBufferOffset** members to a location in the **InformationBuffer** member that the extension can use to return the feature status information.

The extensible switch extension must follow these guidelines when it receives an OID method request of OID_SWITCH_PORT_FEATURE_STATUS_QUERY:

- The extension must handle the OID request if it manages a custom extensible switch port property that matches the **FeatureStatusId** member of the **NDIS_SWITCH_PORT_FEATURE_STATUS_PARAMETERS** structure.

- If the extension handles the OID method request, it must return the feature status information that matches the parameters specified by the **NDIS_SWITCH_PORT_FEATURE_STATUS_PARAMETERS** structure.

  If the feature status buffer is too small, the extension must fail the OID request with NDIS_STATUS_INVALID_LENGTH. The extension must set the **DATA.SET_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.

  Otherwise, the extension must return the feature status information and complete the OID request with NDIS_STATUS_SUCCESS.

- If the extension does not manage the custom extensible switch property, it must call **NdisFOidRequest** to forward the OID request down the extensible switch driver stack.

  For more information about how to forward OID requests, see Filtering OID Requests in an NDIS Filter Driver.

For more information about how to define and register port feature status information, see Custom Port Feature Status.

Managing Custom Switch Feature Status Information

Article • 12/15/2021

The Hyper-V extensible switch interface uses the following object identifier (OID) to query custom status information for the extensible switch. This status information is known as *switch feature status* information:

OID_SWITCH_FEATURE_STATUS_QUERY
This OID method request is issued by the protocol edge of the extensible switch to obtain the custom feature status information for a specified switch property.

After a successful return from this OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer contains the following data:

- An **NDIS_SWITCH_FEATURE_STATUS_PARAMETERS** structure that specifies the custom feature status information that is to be returned.

  Note For a custom feature status, the **FeatureStatusType** member is set to **NdisSwitchPropertyTypeCustom**.

- An **NDIS_SWITCH_FEATURE_STATUS_CUSTOM** structure that contains the status information about a custom property assigned to an extensible switch port.

  When the protocol edge of the extensible switch issues the OID_SWITCH_FEATURE_STATUS_QUERY request, it sets the **FeatureStatusCustomBufferLength** and **FeatureStatusCustomBufferOffset** members to a location in the **InformationBuffer** member that the extension can use to return the feature status information.

The extensible switch extension must follow these guidelines when it receives an OID method request of OID_SWITCH_FEATURE_STATUS_QUERY:

- The extension must handle the OID request if it manages a custom extensible switch feature status that matches the **FeatureStatusId** member of the **NDIS_SWITCH_FEATURE_STATUS_PARAMETERS** structure.

- If the extension handles the OID method request, it must return the feature status information that matches the parameters specified by the **NDIS_SWITCH_FEATURE_STATUS_PARAMETERS** structure.

If the feature status buffer is too small, the extension must fail the OID request with NDIS_STATUS_INVALID_LENGTH. The extension must set the **DATA.SET_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.

Otherwise, the extension must return the feature status information and complete the OID request with NDIS_STATUS_SUCCESS.

- If the extension does not manage the custom extensible switch feature status, it must call **NdisFOidRequest** to forward the OID request down the extensible switch driver stack.

  For more information about how to forward OID requests, see Filtering OID Requests in an NDIS Filter Driver.

For more information about how to define and register switch feature status information, see Custom Switch Feature Status.

Managing Hyper-V Extensible Switch Run-Time Data

Article • 12/15/2021

This topic describes save and restore operations for Hyper-V extensible switch extensions. These operations allow an extension to save and restore run-time data for individual extensible switch network adapters(NICs). These operations are performed when a Hyper-V child partition that has a network adapter connection to an extensible switch port is being stopped or started.

Saving Hyper-V Extensible Switch Run-Time Data

This section describes the operation by which a Hyper-V Extensible Switch extension can save run-time data for individual network adapters (NICs). This operation is performed when a Hyper-V child partition with a network adapter connection to an extensible switch port is being stopped or its state is being saved.

Handling the OID_SWITCH_NIC_SAVE Request

When a Hyper-V child partition with a network adapter connection to an extensible switch port is stopped or its state is saved, the Hyper-V extensible switch interface is notified. This causes the protocol edge of the extensible switch to issue an object identifier (OID) method request of OID_SWITCH_NIC_SAVE down the extensible switch driver stack. When an extensible switch extension receives this OID request, it can save its run-time data for the specified network adapter connection that is attached to the child partition.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for the OID_SWITCH_NIC_SAVE request contains a pointer to an **NDIS_SWITCH_NIC_SAVE_STATE** structure. This structure is allocated by the protocol edge of the extensible switch and initialized in the following way:

- The **Header** member is initialized to contain the current type, revisionof the **NDIS_SWITCH_NIC_SAVE_STATE** structure. Size is set to the full buffer size.
- The **PortId** member contains the unique identifier of the extensible switch port for which the save operation is being performed.

When it receives the OID_SWITCH_NIC_SAVE method request, the extension does the following:

1. The extension reads the **PortId** member of the **NDIS_SWITCH_NIC_SAVE_STATE** structure.

2. If the extension has run-time data to save for the specified NIC, it saves its data within the **NDIS_SWITCH_NIC_SAVE_STATE** structure starting with *SaveDataOffset* bytes from the start of the structure. The extension then completes the OID method request with NDIS_STATUS_SUCCESS.

3. If the **NDIS_SWITCH_NIC_SAVE_STATE** structure does not provide a sufficient buffer to hold the runtime state, the extension the extension sets the method structure's *BytesNeeded* field to **NDIS_SIZEOF_NDIS_SWITCH_NIC_SAVE_STATE_REVISION_1** plus the amount of buffer necessary to hold the save data, and completes the OID with **NDIS_STATUS_BUFFER_TOO_SHORT**. The OID will be re-issued with the required size.

4. If the extension does not have run-time data to save for the specified NIC, it must call **NdisFOidRequest**. This forwards the OID method request to underlying drivers in the extensible switch driver stack. For more information about this procedure, see Filtering OID Requests in an NDIS Filter Driver.

If the extension has run-time port data to save, it must follow these guidelines when it saves run-time port data within the **NDIS_SWITCH_NIC_SAVE_STATE** structure:

1. The extension sets the **ExtensionId** member to the GUID value that uniquely identifies the driver.

2. The extension sets the **ExtensionFriendlyName** member to the name of the driver.

   Note The NDIS_SWITCH_EXTENSION_FRIENDLYNAME data type is type-defined by the **IF_COUNTED_STRING** structure. A string that is defined by this structure does not have to be null-terminated. However, the length of the string must be set in the **Length** member of this structure. If the string is NULL-terminated, the **Length** member must not include the terminating NULL character.

3. If a feature class is associated with the saved run-time data, the extension sets the **FeatureClassId** with the GUID that uniquely identifies the class.

   Note If a feature class is not associated with the saved run-time data, the extension sets the **FeatureClassId** to zero.

4. The extension copies the run-time data to the **SaveData** member and sets the **SaveDataSize** member to the size, in bytes, of the run-time data.

Note The extension must not change the **Header** or **PortId** members of the **NDIS_SWITCH_NIC_SAVE_STATE** structure.

OID method requests of OID_SWITCH_NIC_SAVE are ultimately handled by the underlying miniport edge of the extensible switch. Once this OID method request has been forwarded to the miniport driver through the extensible switch driver stack, the miniport driver completes the OID request with NDIS_STATUS_SUCCESS. This notifies the protocol edge of the extensible switch that all extensions in the extensible switch driver stack have been queried for run-time port data. The protocol edge of the extensible switch then issues an OID set request of OID_SWITCH_NIC_SAVE_COMPLETE to complete the save operation.

Handling the OID_SWITCH_NIC_SAVE_COMPLETE Request

When a Hyper-V child partition that has a network adapter connection to an extensible switch port is paused or its state is being saved, the Hyper-V extensible switch interface is notified. This causes the protocol edge of the extensible switch to issue an object identifier (OID) method request of OID_SWITCH_NIC_SAVE down the extensible switch driver stack.

When every Hyper-V extensible switch extension has saved its run-time data, the protocol edge of the extensible switch notifies underlying extensions that the save operation has completed. The protocol edge does this by issuing an OID set request of OID_SWITCH_NIC_SAVE_COMPLETE down the extensible switch driver stack.

Note When a run-time save operation is started for an extensible switch network adapter connection, another save operation for the same network adapter connection will not be performed until the OID_SWITCH_NIC_SAVE_COMPLETE request is issued. However, save operations for other network adapter connections could occur during this time.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for the OID_SWITCH_NIC_SAVE_COMPLETE request contains a pointer to an **NDIS_SWITCH_NIC_SAVE_STATE** structure. This structure is allocated by the protocol edge of the extensible switch.

When it receives the OID set request of OID_SWITCH_NIC_SAVE_COMPLETE, the extension must follow these guidelines:

- The extension must not modify the **NDIS_SWITCH_NIC_SAVE_STATE** structure that is associated with the OID request.

- The extension must call **NdisFOidRequest** to forward this OID request through the extensible switch extension stack. The extension must not fail the OID request.

  Note The extension should monitor the completion status of this OID request. The extension does this to detect whether the save operation has completed successfully.

OID method requests of OID_SWITCH_NIC_SAVE_COMPLETE are ultimately handled by the underlying miniport edge of the extensible switch. Once this OID method request has been received by the miniport edge, it completes the OID request with NDIS_STATUS_SUCCESS. This notifies the protocol edge of the extensible switch that all extensions in the extensible switch driver stack have completed the save operation.

Restoring Hyper-V Extensible Switch Run-Time Data

When a Hyper-V child partition that has a network adapter connection to an extensible switch port is resumed from a pause, the Hyper-V extensible switch interface is notified. This causes the protocol edge of the extensible switch to issue an object identifier (OID) set request of OID_SWITCH_NIC_RESTORE down the extensible switch driver stack. When an extension receives this OID request, it can restore its run-time data for the extensible switch port that is used by the child partition.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure for the OID_SWITCH_NIC_RESTORE request contains a pointer to an **NDIS_SWITCH_NIC_SAVE_STATE** structure. This structure is allocated by the protocol edge of the extensible switch.

When it receives the OID set request of OID_SWITCH_NIC_RESTORE, the extensible switch extension must first determine whether it owns the run-time data. The extension does this by comparing the value of the **ExtensionId** member of the **NDIS_SWITCH_NIC_SAVE_STATE** structure to the GUID value that the extension uses to identify itself.

If the extension owns the run-time data for an extensible switch NIC, it restores this data in the following way:

1. The extension copies the run-time data in the **SaveData** member to driver-allocated storage.

> **Note** The value of the **PortId** member of the **NDIS_SWITCH_NIC_SAVE_STATE** structure may be different from the **PortId** value at the time that the run-time data was saved. This can occur if run-time data was saved during a Live Migration from one host to another. However, the configuration of the extensible switch NIC is retained during the Live Migration. This enables the extension to restore the run-time data to the extensible switch NIC by using the new **PortId** value.

2. The extension completes the OID set request with NDIS_STATUS_SUCCESS.

If the extension does not own the specified run-time data to save, the extension calls **NdisFOidRequest**. This forwards the OID set request to underlying drivers in the extensible switch driver stack. In this case, the extension must not modify the **NDIS_SWITCH_NIC_SAVE_STATE** structure that is associated with the OID request. For more information on how to forward OID requests, see Filtering OID Requests in an NDIS Filter Driver.

If the OID set request of OID_SWITCH_NIC_RESTORE is completed with NDIS_STATUS_SUCCESS, the protocol edge of the extensible switch issues another OID set request. When it receives this new OID set request, the extension can do one of the following:

- If it owns the run-time data in the new OID request, the extension restores the additional run-time data within the **NDIS_SWITCH_NIC_SAVE_STATE** structure. The extension then completes the OID request with NDIS_STATUS_SUCCESS.

- If it does not own the run-time data in the new OID request, the extension calls **NdisFOidRequest** to forward this OID set request to underlying drivers.

OID_SWITCH_NIC_RESTORE_COMPLETE
The extensible switch interface signals the protocol edge of the extensible switch to issue this OID at the completion of the restore operation of run-time data for an extensible switchnetwork adapter.

This OID request notifies the extension that the restore operation has completed only for a specified extensible switch NIC.

For more information about this OID request, see OID_SWITCH_NIC_RESTORE_COMPLETE.

Note If the OID_SWITCH_NIC_RESTORE set request is received by the miniport edge of the extensible switch, it completes the OID request with NDIS_STATUS_SUCCESS. This notifies the protocol edge of the extensible switch that no extension owns the run-time data. If this happens, the extensible switch interface logs an event that documents the

ExtensionId and **PortId** member values for the extension that originally saved the run-time port data.

Managing Physical Network Adapters

Article • 12/15/2021

This section describes the operations that a Hyper-V extensible switch extension can perform on underlying physical adapters that are bound to the extensible switch external network adapter.

These operations allow an extension to forward or originate object identifier (OID) requests to an underlying physical network adapter. The extension can also forward or originate NDIS status indications from a physical network adapter up the extensible switch driver stack.

Note Operations of this sort can only be performed by an extensible switch forwarding extension. For more information about this type of extension, see Forwarding Extensions.

This section includes the following topics:

Managing Physical Network Adapter Connection Status

Forwarding Packets to Physical Network Adapters

Managing OID Requests to Physical Network Adapters

Managing NDIS Status Indications from Physical Network Adapters

Managing Physical Network Adapter Connection Status

Article • 12/15/2021

The Hyper-V extensible switch architecture supports the connection to a single external network adapter for access to the underlying physical medium. The external network adapter can be bound to one or more underlying physical network adapters in a variety of configurations. For more information about these configurations, see Types of Physical Network Adapter Configurations.

The extensible switch interface notifies extensions of each physical network adapter connection status through the following steps:

1. The protocol edge of the extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_CREATE. This OID request notifies underlying extensible switch extensions about the creation of network connection to the extensible switch external network adapter.

   When the network connection is created, it is assigned an NDIS_SWITCH_NIC_INDEX value. This index value identifies the network connection of the adapter on an extensible switch port. The network connection to the external network adapter is assigned an NDIS_SWITCH_NIC_INDEX value of **NDIS_SWITCH_DEFAULT_NIC_INDEX**.

2. For every network adapter that is bound directly or indirectly to the external network adapter, the protocol edge of the extensible switch issues a separate OID set request of OID_SWITCH_NIC_CREATE. This OID request notifies the extension about the creation of a network connection to an underlying network adapter.

   Each physical or virtual network adapter that is bound to the external network adapter is assigned an identifier in the following way:

   - If a single physical adapter is bound to the external network adapter, it is assigned a NDIS_SWITCH_NIC_INDEX value of one.

   - If an load balancing fail over (LBFO) team of network adapters is bound to the external network adapter, it is assigned a NDIS_SWITCH_NIC_INDEX value of one.

     Note In the LBFO team configuration, only the virtual miniport edge of the LBFO provider that supports the LBFO team is considered to be bound to the external network adapter.

- If an extensible switch team of network adapters is bound to the external network adapter, each physical network adapter in the team is assigned a unique NDIS_SWITCH_NIC_INDEX value that is greater than or equal to one.

  Note In the extensible switch team configuration, every physical network adapter in the team is considered to be bound to the external network adapter.

3. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT for the external network adapter.

   Note At this point, the connection to the external network is not operational and cannot be used for packet traffic.

4. For every network adapter that is bound to the external network adapter, the protocol edge of the extensible switch issues a separate OID set request of OID_SWITCH_NIC_CONNECT. This OID request is issued after the OID set request of OID_SWITCH_NIC_CREATE is completed successfully.

   The OID_SWITCH_NIC_CONNECT OID request notifies the extension that the extensible switch network connection is now operational. If the external network adapter is bound to the virtual miniport edge of the MUX driver, the protocol edge issues a separate OID_SWITCH_NIC_CONNECT request.

   Note As soon as an OID_SWITCH_NIC_CONNECT request is issued for a physical network adapter with a NDIS_SWITCH_NIC_INDEX value greater than or equal to one, the connection to the external network is operational. At this point, packet traffic can be sent or received over the external network.

5. If the external network connection is being torn down, the protocol edge of the extensible switch first issues a separate OID set request of OID_SWITCH_NIC_DISCONNECT for every network adapter that is bound to the external network adapter. Once these OID requests are completed, the protocol edge of the extensible switch then issues separate OID set request of OID_SWITCH_NIC_DELETE for every physical network adapter that is bound to the external network adapter,

   Once all network connections to the underlying physical adapters have been disconnected and deleted, the protocol edge of the extensible switch issues OID_SWITCH_NIC_DISCONNECT and OID_SWITCH_NIC_DELETE requests to disconnect and delete the external network adapter connection.

For more information on NDIS_SWITCH_NIC_INDEX values, see Network Adapter Index Values.

Forwarding Packets to Physical Network Adapters

Article • 12/15/2021

Note This page assumes that you are familiar with the information and diagrams in the following pages:

- Forwarding Extensions
- Hybrid Forwarding
- Hyper-V Extensible Switch Extensions
- Overview of the Hyper-V Extensible Switch
- Teaming Provider Extensions

This page describes how Hyper-V extensible switch forwarding extensions can forward send requests of packets to underlying physical adapters. One or more physical network adapters can be bound to the extensible switch external network adapter.

For example, the extensible switch external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX intermediate driver itself can be bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*. For more information about extensible switch teams, see Types of Physical Network Adapter Configurations.

In this configuration, the extensible switch extensions are exposed to every network adapter in the extensible switch team. This allows a forwarding extension in the extensible switch driver stack to manage the configuration and use of individual network adapters in the team. For example, the extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. Such as extension is known as a *teaming provider*. For more information about teaming providers, see Teaming Provider Extensions.

If a forwarding extension is installed and enabled in the extensible switch driver stack, it is responsible for making forwarding decisions for each packet that it obtains on the extensible switch ingress data path, unless the packet is an NVGRE packet. (For more information about NVGRE packets, see Hybrid Forwarding.) Based on these forwarding decisions, the extension can add destination ports into the out-of-band (OOB) data of the packet's **NET_BUFFER_LIST** structure. After the packet has completed its traversal of the extensible switch data path, the extensible switch interface delivers the packet to the specified destination ports.

Note If a forwarding extension is not installed or enabled, the extensible switch itself makes the forwarding decisions for packets it obtains from ingress data path. The switch adds the destination ports to the OOB data of the packet's **NET_BUFFER_LIST** structure before it forwards the packet up the extensible switch egress data path.

When the forwarding extension's *FilterSendNetBufferLists* function is called, the *NetBufferList* parameter contains a pointer to a linked list of **NET_BUFFER_LIST** structures. Each of these structures specifies a packet obtained from the ingress data path. Within the OOB data of each packet's **NET_BUFFER_LIST** structure, the data for destination ports are contained in an **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. The extension obtains the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure and its elements by calling *GetNetBufferListDestinations*.

Note To improve performance, a forwarding extension can call the *GrowNetBufferListDestinations* function instead of *GetNetBufferListDestinations* to obtain a pointer to the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** structure. The extension does this if it determines that it needs additional array elements in the packet's OOB data for destination ports. For more information, see Adding Extensible Switch Destination Port Data to a Packet.

Each element in the **NDIS_SWITCH_FORWARDING_DESTINATION_ARRAY** array defines a destination port and is formatted as an **NDIS_SWITCH_PORT_DESTINATION** structure. This structure contains the following members:

- The **PortId** member contains a value that specifies the destination port on the extensible switch.

- The **NicIndex** member specifies the index of the network adapter that is connected to the extensible switch port specified by the **PortId** member.

  For more information on these index values, see Network Adapter Index Values.

If the forwarding extension adds a destination port that is connected to the external network adapter, the extension can specify the index of an underlying physical network adapter. For example, the extension could operate as a teaming provider for LBFO support over an extensible switch team. This allows the extension to balance the traffic overhead by forwarding send requests to different adapters of the team.

The forwarding extension must follow these guidelines when it adds or modifies an **NDIS_SWITCH_PORT_DESTINATION** structure to forward send requests to an underlying physical network adapter:

- If the **PortId** member specifies the extensible switch port to which the external network adapter is connected, the extension must set the **NicIndex** member to one of the following index values:
  - If only one physical network adapter is bound to the external network adapter, the extension must set the **NicIndex** member to **NDIS_SWITCH_DEFAULT_NIC_INDEX** or one.
  - If multiple physical network adapters are bound to the external network adapter, the extension must set the **NicIndex** member to the nonzero index value of the destination network adapter in the extensible switch team.

  Note If the **PortId** member does not specify the extensible switch port to which the external network adapter is connected, the extension must set the **NicIndex** member to **NDIS_SWITCH_DEFAULT_NIC_INDEX**.

- After the extension has added all of the destination ports for the packet, it must call **NdisFSendNetBufferLists** to forward the packet on the ingress data path.

For more information on how to add destination ports to a packet, see Forwarding Packets to Hyper-V Extensible Switch Ports.

For more information on the egress data path, see Hyper-V Extensible Switch Data Path.

Managing OID Requests to Physical Network Adapters

Article • 12/15/2021

This section discusses how Hyper-V extensible switch extensions manage object identifier (OID) requests for underlying physical network adapters over the extensible switch control path.

This section includes the following topics:

Forwarding OID Requests to Physical Network Adapters

Managing Hardware Offload OID Requests to Physical Network Adapters

For more information on how to manage OID requests over the Hyper-V extensible switch control path, see Hyper-V Extensible Switch Control Path for OID Requests.

Forwarding OID Requests to Physical Network Adapters

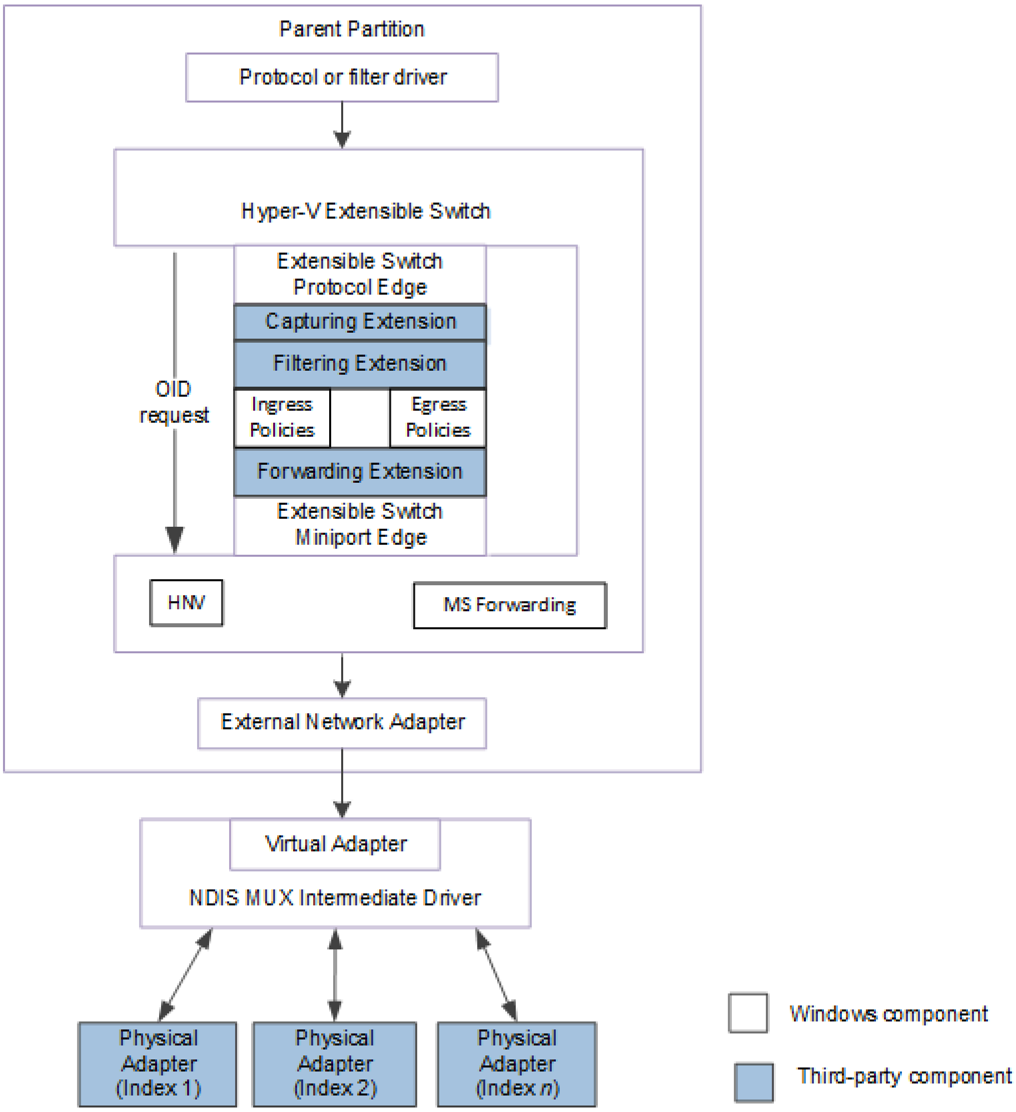
Article • 12/15/2021

This topic discusses how Hyper-V extensible switch extensions forward object identifier (OID) requests for underlying physical adapters over the Hyper-V extensible switch control path. The extension can also originate OID requests to underlying physical network adapters by following the methods described in this topic.

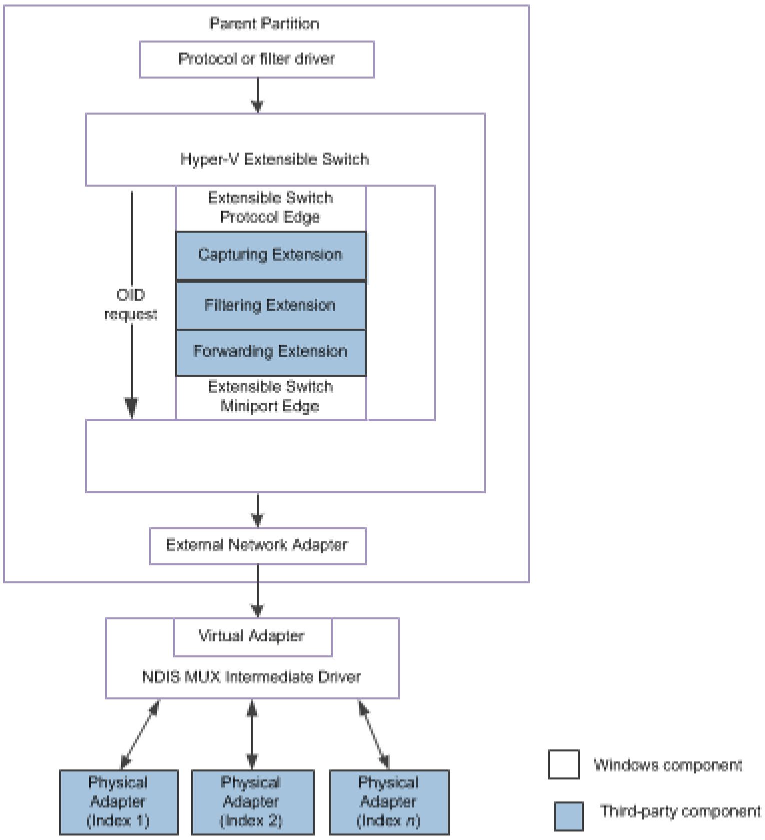
For example, the external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX driver is bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*.

In this configuration, an extensible switch extension is exposed to every network adapter in the team. This allows the extension to manage the configuration and use of individual network adapters in the team. For example, a forwarding extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. A forwarding extension that manages an extensible switch team is known as a *teaming provider*. For more information about teaming providers, see Teaming Provider Extensions.

The following figure shows an example of an extensible switch team for NDIS 6.40 (Windows Server 2012 R2) and later.


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Ingress Policies
Egress Policies
Forwarding Extension
Extensible Switch Miniport Edge
OID request
HNV
MS Forwarding
External Network Adapter
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index n)
Windows component
Third-party component

The following figure shows an example of an extensible switch team for NDIS 6.30 (Windows Server 2012).


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension
Extensible Switch Miniport Edge
OID request
External Network Adapter
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index n)
Windows component
Third-party component

Note In the Hyper-V extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

OID requests must be encapsulated to forward the request to an underlying physical network adapter. OID requests are first encapsulated inside an **NDIS_SWITCH_NIC_OID_REQUEST** structure. Then, the OID requests are forwarded through the extensible switch control path by an OID set request of OID_SWITCH_NIC_REQUEST.

OID requests to the underlying physical adapters are issued by the following:

The extensible switch interface.
OID requests, such as requests for hardware offloads, are issued by overlying protocol or filter drivers that run in one of the following:

- The management operating system that runs in the Hyper-V parent partition.
- The guest operating system that runs in the Hyper-V child partition.

When these OID requests are received by the extensible switch, they are encapsulated and forwarded over the extensible switch control path. When a forwarding extension receives the encapsulated OID request, it can forward the request to an underlying physical adapter. This ability is especially useful for configuring the extensible switch team for hardware offloads.

For example, the MUX driver advertises the common capabilities of the entire extensible switch team. However, the forwarding extension can issue OID requests to query or set the individual capabilities of adapters within the team. Then, the forwarding extension can originate an NDIS status indication from the external network adapter to notify overlying drivers about the capabilities that apply to the entire team. For more information on this procedure, see Originating NDIS Status Indications from Physical Network Adapters.

When the forwarding extension forwards the OID request over the control path, it is received by the external network adapter. At this point, the OID request is decapsulated and forwarded to the specified physical network adapter.

Note Starting with Windows Server 2012, only hardware offload OID requests are encapsulated and forwarded in this manner. For example, offload OID requests for virtual machine queue (VMQ) or Internet Protocol security (IPsec) are encapsulated and forwarded over the extensible switch control path. For more information, see Managing Hardware Offload OID Requests to Physical Network Adapters.

A forwarding extension.
The forwarding extension can originate its own encapsulated OID requests and forward them to an underlying physical network adapter. The forwarding extension can encapsulate standard NDIS OID requests. The forwarding extension can also encapsulate private OID requests that are defined by the independent hardware vendor (IHV) for the physical network adapters. This allows a forwarding extension that was also developed by the IHV to enable or disable proprietary attributes on individual physical adapters in the team.

In addition, the forwarding extension can originate encapsulated hardware offload OID requests to allocate resources for a specified Hyper-V child partition. For example, the forwarding extension can originate encapsulated OID requests of OID_RECEIVE_FILTER_ALLOCATE_QUEUE to allocate a VMQ for a specified child partition. In this case, the extension encapsulates the request as originating from the extensible switch port and network adapter connection associated with the partition.

Note The forwarding extension can only originate its own encapsulated hardware offload OID request if it is filtering the same OID request that was issued by overlying drivers. In this case, the extension must not forward the original OID request. Instead, the extension must call **NdisFOidRequestComplete** to complete this request when NDIS calls its *FilterOidRequestComplete* to complete the originated OID request.

Filtering or capturing extensions
A filtering or capturing extension can originate its own encapsulated OID query requests and forward them to an underlying physical network adapter. These extensions can

encapsulate standard NDIS OID query requests or private OID query requests that are defined by the independent hardware vendor (IHV) for the physical network adapters.

Note Only forwarding extensions can originate encapsulated OID set requests to underlying physical adapters.

The forwarding extension must follow these steps when it forwards, redirects, or originates an encapsulated OID request for an underlying physical adapter:

1. If the forwarding extension is originating an OID request, it must initialize an extension-allocated **NDIS_OID_REQUEST** structure with the information related to the request.

   If the extension is forwarding an OID request, it must not change the existing **NDIS_OID_REQUEST** structure referenced by the *OidRequest* parameter of the *FilterOidRequest* function. Instead, the extension must call **NdisAllocateCloneOidRequest** to allocate memory for a new **NDIS_OID_REQUEST** structure and copy all the information from the existing **NDIS_OID_REQUEST** structure.

2. The extension sets the members of an extension-allocated **NDIS_SWITCH_NIC_OID_REQUEST** structure to the following values:

   - The **DestinationPortId** member must be set to the identifier of the extensible switch port to which the external network adapter is connected.

   - The **DestinationNicIndex** member must be set to the nonzero index value of the underlying physical network adapter.

     For more information on these index values, see Network Adapter Index Values.

   - If the forwarding extension is originating a hardware offload OID request for a Hyper-V child partition, the **SourcePortId** member must be set to the identifier of the port that is used by the partition. Also, the **SourceNicIndex** member must be set to the network adapter index for the network connection to that port.

     If the forwarding extension is originating a standard or private OID request for its own purposes, the **SourcePortId** and **SourceNicIndex** members must be set to zero.

     If the forwarding extension is forwarding or redirecting a hardware offload OID request, it must retain the values of the **SourcePortId** and **SourceNicIndex** members that were set by the extensible switch interface.

- The **OidRequest** member must be set to a pointer to an initialized NDIS_OID_REQUEST structure for the encapsulated OID request. The forwarding extension either allocates and initializes this structure or uses the cloned copy of the structure.

3. The extension sets the members of an extension-allocated NDIS_OID_REQUEST structure to the following values:

   - The **Oid** member must be set to OID_SWITCH_NIC_REQUEST.

   - The **InformationBuffer** member must contain a pointer to a buffer that contains the generated or filtered OID request data.

   - The **InformationBufferLength** member must contain the length, in bytes, of the buffer that contains the generated or filtered OID request data.

   The extension sets the other members to values that are valid for the NDIS_OID_REQUEST structure.

4. The extension calls *ReferenceSwitchNic* to increment a reference counter for the index of the destination physical network adapter. This guarantees that the extensible switch interface will not delete the physical network adapter connection while its reference counter is nonzero.

   When the extension calls *ReferenceSwitchNic*, it sets the *SwitchPortId* parameter to the value specified for the **DestinationPortId** member. The extension also sets the *SwitchNicIndex* parameter to the value specified for the **DestinationNicIndex** member.

   Note If *ReferenceSwitchNic* does not return NDIS_STATUS_SUCCESS, the OID request cannot be forwarded to the destination physical network adapter.

5. If the forwarding extension is originating a hardware offload OID request for a Hyper-V child partition, it also calls *ReferenceSwitchNic* to increment a reference counter for the index of the source network adapter connection that is associated with the partition. This guarantees that the extensible switch interface will not delete the physical network adapter connection while its reference counter is nonzero.

   When the extension calls *ReferenceSwitchNic*, it sets the *SwitchPortId* parameter to the value specified for the **SourcePortId** member. The extension also sets the *SwitchNicIndex* parameter to the value specified for the **SourceNicIndex** member.

   Note If *ReferenceSwitchNic* does not return NDIS_STATUS_SUCCESS, the OID request cannot be forwarded to the destination physical network adapter.

6. The extension calls **NdisFOidRequest** to forward the encapsulated OID request to the specified destination extensible switch port and network adapter.

   Note If the extension is forwarding a filtered OID request, it must call **NdisFOidRequest** within the context of the call to its *FilterOidRequest* function. If the extension is forwarding OID requests that it has generated, it calls **NdisFIndicateStatus** while it is in the *Running*, *Restarting*, *Paused*, and *Pausing* states. For more information on these states, see Filter Module States and Operations.

7. When NDIS calls the *FilterOidRequestComplete* function, the extension calls *DereferenceSwitchNic* to clear the reference counter for the index of the destination physical network adapter.

   If the forwarding extension originated a hardware offload OID request for a Hyper-V child partition, it also calls *DereferenceSwitchNic* to clear the reference counter for the index of the source network adapter connection for the adapter.

   In both cases, the extension sets the *SwitchPortId* and *SwitchNicIndex* parameters to the same values that it used in the call to *ReferenceSwitchNic*.

For more information on how the extension issues OID requests, see Generating OID Requests from an NDIS Filter Driver.

For more information on MUX drivers, see NDIS MUX Intermediate Drivers.

Managing Hardware Offload OID Requests to Physical Network Adapters

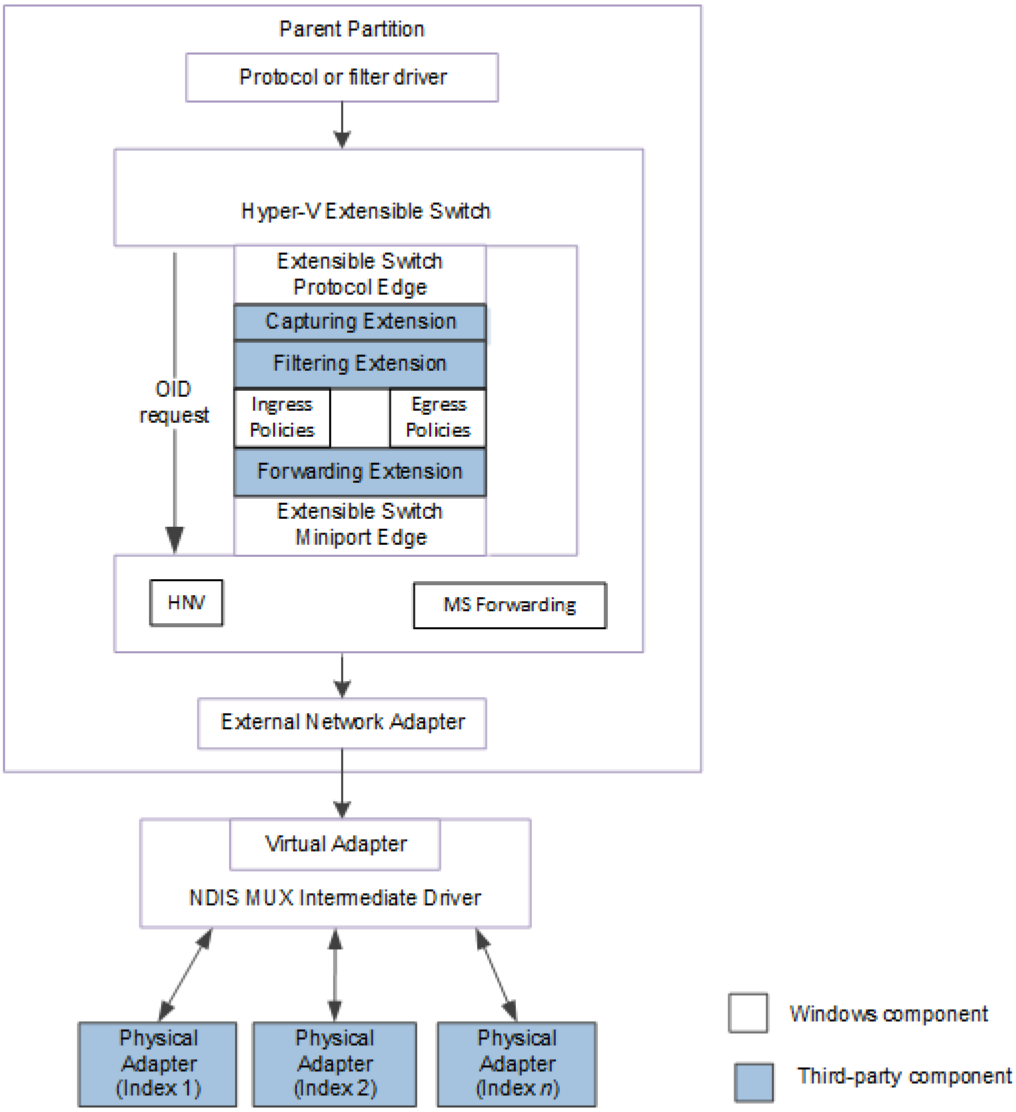
Article • 07/07/2022

This topic discusses how a Hyper-V extensible switch forwarding extension manages object identifier (OID) requests for hardware offload technologies on underlying physical adapters over the extensible switch control path.

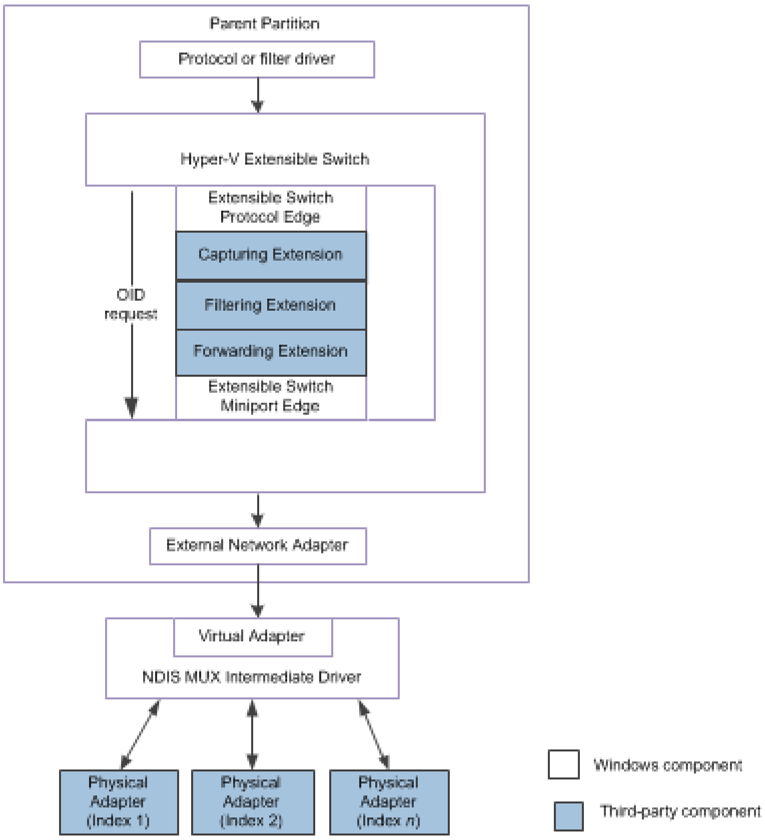
For example, the external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX driver is bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*.

In this configuration, an extensible switch extension is exposed to every network adapter in the team. This allows the extension to manage the configuration and use of individual network adapters in the team. For example, a forwarding extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. A forwarding extension that manages an extensible switch team is known as a *teaming provider*. For more information about teaming providers, see Teaming Provider Extensions.

The following figure shows an example of an extensible switch team for NDIS 6.40 (Windows Server 2012 R2) and later.


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Ingress Policies
Egress Policies
Forwarding Extension
Extensible Switch Miniport Edge
OID request
HNV
MS Forwarding
External Network Adapter
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index n)
Windows component
Third-party component

The following figure shows an example of an extensible switch team for NDIS 6.30 (Windows Server 2012).


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension
Extensible Switch Miniport Edge
OID request
External Network Adapter
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index n)
Windows component
Third-party component

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

By handling the OID request of OID_SWITCH_NIC_REQUEST, a forwarding extension can participate in the configuration of the extensible switch team for hardware offloads. For example, if the extension manages the physical network adapters of an extensible switch team, it can forward the OID_SWITCH_NIC_REQUEST request to a physical adapter that supports the hardware offload.

NDIS and overlying protocol and filter drivers can issue OID requests for hardware offload technologies to the underlying physical network adapter. When these OID requests arrive at the extensible switch interface, it encapsulates the OID request inside an **NDIS_SWITCH_NIC_OID_REQUEST**. Then, the protocol edge of the extensible switch issues an OID request of OID_SWITCH_NIC_REQUEST that contains this structure.

The extensible switch interface encapsulates OIDs for the following hardware offload technologies:

Internet Protocol security (IPsec) offload (version 2)
The following IPsec OID requests are encapsulated:

- OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA

- OID_TCP_TASK_IPSEC_OFFLOAD_V2_UPDATE_SA

The forwarding extension must not fail, or *veto*, these OID requests.

For more information about version 2 of the IPsec hardware offload technology, see IPsec Offload Version 2.

Single root I/O virtualization (SR-IOV)
The following SR-IOV OID requests are encapsulated:

- OID_NIC_SWITCH_ALLOCATE_VF
- OID_NIC_SWITCH_CREATE_VPORT
- OID_NIC_SWITCH_DELETE_VPORT
- OID_NIC_SWITCH_FREE_VF
- OID_RECEIVE_FILTER_CLEAR_FILTER
- OID_RECEIVE_FILTER_MOVE_FILTER

The forwarding extension can veto OID requests of OID_NIC_SWITCH_ALLOCATE_VF and OID_NIC_SWITCH_CREATE_VPORT by completing the request with a status code other than NDIS_STATUS_SUCCESS. However, the extension must not veto the other SR-IOV OID requests.

For more information about the SR-IOV hardware offload technology, see Single Root I/O Virtualization (SR-IOV).

Virtualized machine queue (VMQ)
The following VMQ OID requests are encapsulated:

- OID_RECEIVE_FILTER_ALLOCATE_QUEUE
- OID_RECEIVE_FILTER_CLEAR_FILTER
- OID_RECEIVE_FILTER_FREE_QUEUE
- OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE
- OID_RECEIVE_FILTER_SET_FILTER

The forwarding extension can veto OID requests of OID_RECEIVE_FILTER_ALLOCATE_QUEUE and OID_RECEIVE_FILTER_SET_FILTER by completing the request with a status code other than NDIS_STATUS_SUCCESS. However, the extension must not veto the other VMQ OID requests.

For more information about the VMQ hardware offload technology, see Virtual Machine Queue (VMQ).

The forwarding extension must follow these guidelines for handling hardware offload OID requests:

- The Microsoft IM platform advertises only the common offload capabilities for the overall team. However, the extension can generate OID requests to query the capabilities of each adapter in the team.

  Once the extension has determined the hardware capabilities of the physical adapters in the team, it can forward OID set requests for hardware offloads to an adapter that is best suited for the offload.

- All hardware offload OID requests that are originated by overlying protocol or filter drivers will be encapsulated within a **NDIS_SWITCH_NIC_OID_REQUEST** structure. All hardware offload OID requests that are originated by the forwarding extension must also be encapsulated in an **NDIS_SWITCH_NIC_OID_REQUEST** structure.

  The extension forwards the encapsulated OID request to an underlying physical network adapter through an OID set request of OID_SWITCH_NIC_REQUEST. For more information on this procedure, see Forwarding OID Requests to Physical Network Adapters.

- The extension must not modify or fail hardware offload OID requests to clear, free, or complete the allocation of offload resources. For example, the extension must not fail OID requests of OID_RECEIVE_FILTER_CLEAR_FILTER or OID_NIC_SWITCH_DELETE_VPORT. The extensible switch interface must handle these OID requests to clean up state information for these resources.

  The extension can modify or fail hardware offload OID requests to allocate, move, or set offload resources. For example, the extension can fail or modify OID requests of OID_NIC_SWITCH_ALLOCATE_VF or OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA.

- The extension can originate any hardware offload OIDs to an underlying physical network adapter. However, the extension must not originate a hardware offload OID that clears or frees offload resources that the extension did not allocate.

  For example, the extension must not originate a hardware offload OID request of OID_RECEIVE_FILTER_FREE_QUEUE if it did not originate an OID_RECEIVE_FILTER_ALLOCATE_QUEUE request for the same queue.

Note The extension can only originate its own encapsulated hardware offload OID request if it is filtering the same OID request that was issued by overlying drivers. In this case, the extension must not forward the original OID request. Instead, the extension must call **NdisFOidRequestComplete** to complete this request when NDIS calls its *FilterOidRequestComplete* to complete the originated OID request.

- If the extension is forwarding a hardware offload OID request to an underlying physical network adapter, the **DestinationNicIndex** member of the **NDIS_SWITCH_NIC_OID_REQUEST** structure must be set to the nonzero index value of the adapter. For more information on these index values, see Network Adapter Index Values.

  Also, the **DestinationPortId** member must be set to the identifier of the extensible switch port to which the external network adapter is connected.

- If the extension is originating a hardware offload OID request to allocate resources for a Hyper-V child partition, the **SourcePortId** member of the **NDIS_SWITCH_NIC_OID_REQUEST** structure must be set to the identifier of the extensible switch port to which the partition is connected.

  The **SourceNicIndex** member must be set to **NDIS_SWITCH_DEFAULT_NIC_INDEX**.

- When the extension calls **NdisFOidRequest** to forward the OID request, it must set the *OidRequest* parameter to a pointer to an **NDIS_OID_REQUEST** structure for an OID_SWITCH_NIC_REQUEST OID request.

For more information on how the extension filters OID requests, see Filtering OID Requests in an NDIS Filter Driver.

For more information on MUX drivers, see NDIS MUX Intermediate Drivers.

Managing NDIS Status Indications from Physical Network Adapters

Article • 03/14/2023

This section discusses how Hyper-V extensible switch extensions manage NDIS status indications from underlying physical network adapters over the extensible switch control path.

This section includes the following topics:

Forwarding NDIS Status Indications from Physical Network Adapters

Originating NDIS Status Indications from Physical Network Adapters

For more information on how to manage NDIS status indications over the Hyper-V extensible switch control path, see Hyper-V Extensible Switch Control Path for NDIS Status Indications.

Forwarding NDIS Status Indications from Physical Network Adapters

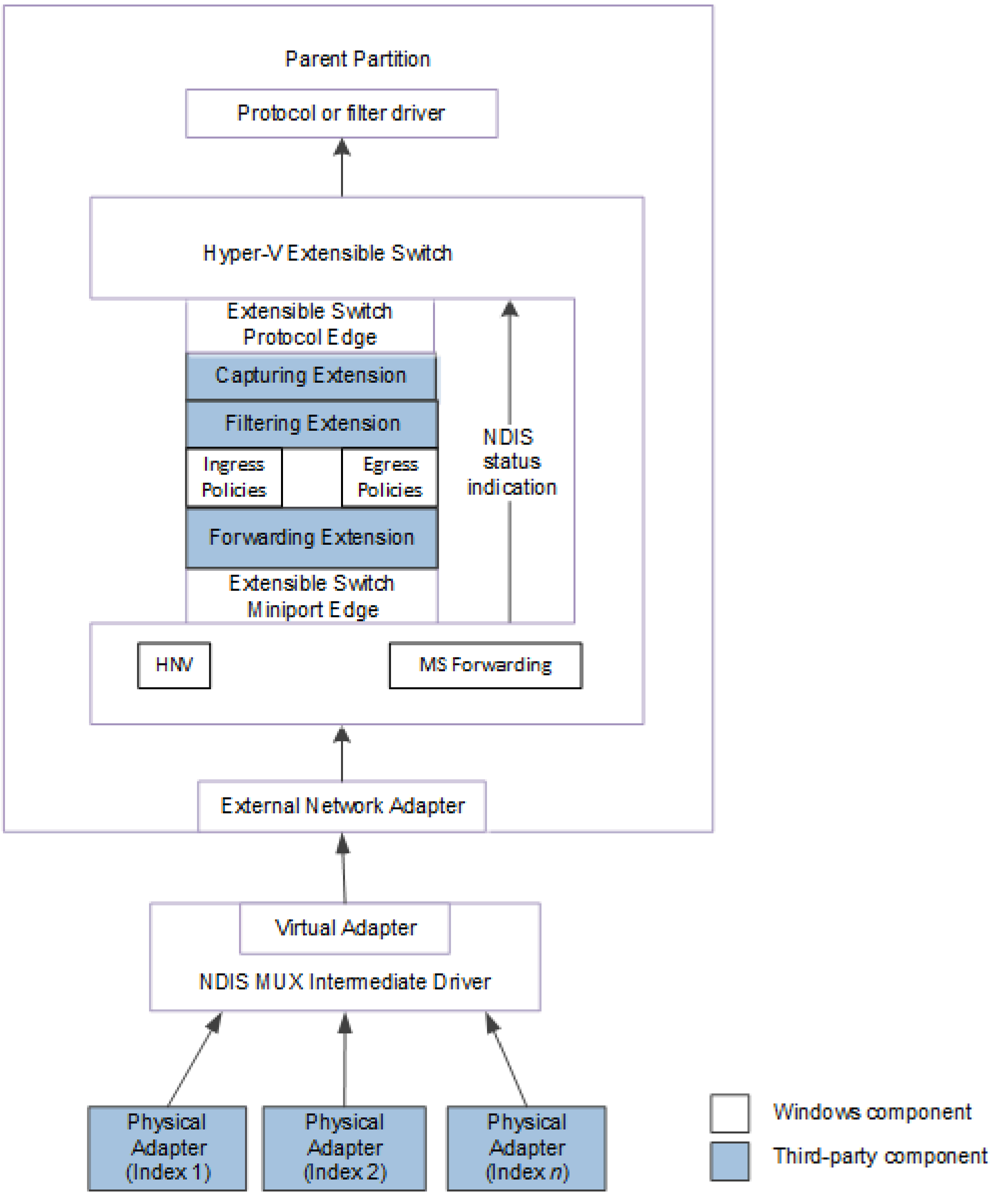
Article • 03/14/2023

This topic discusses the method that is used by extensible switch forwarding extensions to forward NDIS status indications from an underlying physical adapter. One or more underlying physical adapters can be bound to the external network adapter of the Hyper-V extensible switch.

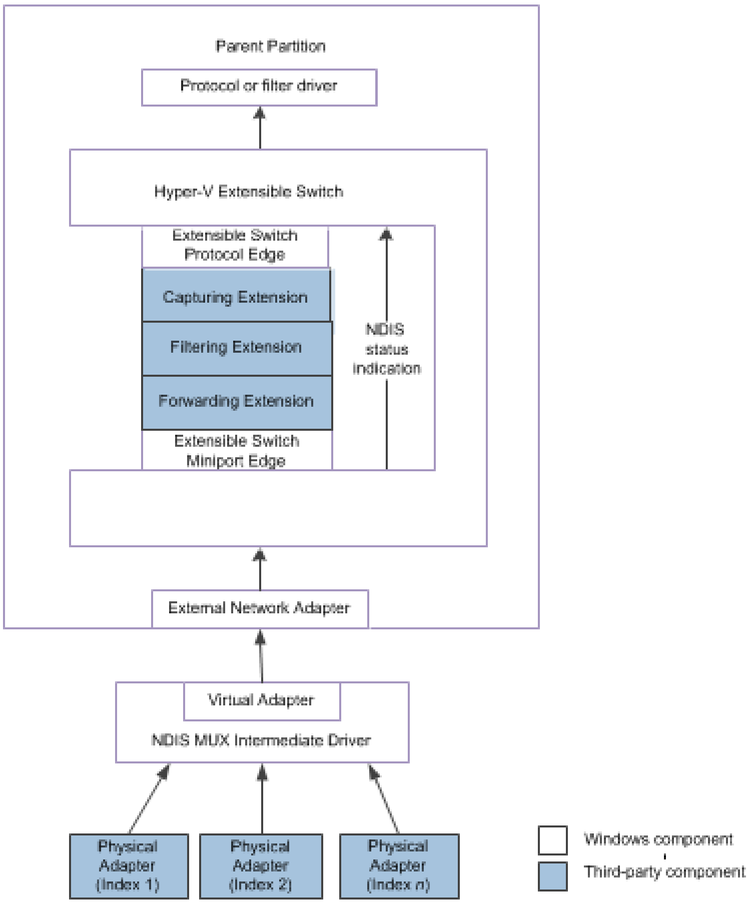
For example, the external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX driver is bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*.

In this configuration, an extensible switch extension is exposed to every network adapter in the team. This allows the extension to manage the configuration and use of individual network adapters in the team. For example, a forwarding extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. A forwarding extension that manages an extensible switch team is known as a *teaming provider*. For more information about teaming providers, see Teaming Provider Extensions.

The following figure shows the Hyper-V extensible switch control path for NDIS status indications from underlying physical network adapters for NDIS 6.40 (Windows Server 2012 R2) and later.


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Ingress Policies
Egress Policies
Forwarding Extension
Extensible Switch Miniport Edge
NDIS status indication
HNV
MS Forwarding
External Network Adapter
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index *n*)
Windows component
Third-party component

The following figure shows the Hyper-V extensible switch control path for NDIS status indications from underlying physical network adapters for NDIS 6.30 (Windows Server 2012).


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension
Extensible Switch Miniport Edge
NDIS status indication
External Network Adapter
Virtual Adapter
NDIS MUX Intermediate Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index *n*)
Windows component
Third-party component

Note In the extensible switch interface, NDIS filter drivers are known as *extensible switch extensions* and the driver stack is known as the *extensible switch driver stack*.

The extensible switch interface forwards NDIS status indications that were generated by the underlying physical adapters. If an external network adapter is bound to an extensible switch team, the NDIS status indication is originated by the virtual adapter edge of a MUX driver. Otherwise, the status indication is originated by the single physical network adapter that is bound to the external network adapter.

When an NDIS status indication arrives at the extensible switch interface, it encapsulates the indication inside an NDIS_SWITCH_NIC_STATUS_INDICATION structure. Then, the miniport edge of the extensible switch issues an NDIS_STATUS_SWITCH_NIC_STATUS indication that contains this structure.

Once the forwarding extension receives the NDIS status indication, it can forward the original indication data or modify the data before it forwards the indication.

Note Only forwarding extensions can modify the data before forwarding the status indication. For more information about this type of extension, see Forwarding Extension.

A forwarding extension can modify and forward status indications from any underlying physical adapter that is bound to the external network adapter of the extensible switch. Typically, the extension issues these status indications to change the advertised hardware offload capabilities of the underlying physical adapter. For example, the extension can modify and forward status indications for the following types of hardware offloads:

- Internet Protocol security (IPsec)
- Virtualized machine queue (VMQ)
- Single root I/O virtualization (SR-IOV)

If the forwarding extension is forwarding an NDIS status indication, it must set the members of the NDIS_SWITCH_NIC_STATUS_INDICATION structure in the following way:

- The **SourcePortId** member must be set to the identifier of the port to which the external network adapter is connected. The external network adapter is bound to one or more physical adapters. For more information, see External Network Adapters.
- The **SourceNicIndex** member must be set to NDIS_SWITCH_DEFAULT_NIC_INDEX. This allows the status indication to be interpreted as originating from the entire extensible switch team that is bound to the external network adapter.
- The **DestinationPortId** member must be set to **NDIS_SWITCH_DEFAULT_PORT_ID**.
- The **DestinationNicIndex** member must be set to **NDIS_SWITCH_DEFAULT_NIC_INDEX**.
- The **StatusIndication** member must be set to a pointer to an NDIS_STATUS_INDICATION structure. This structure contains the data for the encapsulated NDIS status indication.

When a forwarding extension issues the encapsulated NDIS status indication, it must follow these steps:

1. The extension calls *ReferenceSwitchNic* to increment a reference counter for the external network adapter. This guarantees that the extensible switch interface will not delete the network adapter connection while its reference counter is nonzero.

   When the extension calls *ReferenceSwitchNic*, it sets the *SwitchPortId* parameter to the value specified for the **SourcePortId** member. The extension also sets the *SwitchNicIndex* parameter to the value specified for the **SourceNicIndex** member.

   Note If *ReferenceSwitchNic* does not return NDIS_STATUS_SUCCESS, the encapsulated NDIS status indication cannot be issued.

2. The extension calls **NdisFIndicateStatus** to forward the encapsulated status notification.

   Note If the extension is forwarding an encapsulated NDIS status indication, it must call **NdisFIndicateStatus** within the context of the call to its *FilterStatus* function.

3. After **NdisFIndicateStatus** returns, the extension calls *DereferenceSwitchNic* to clear the reference counter for the source or destination network adapter connection. The extension sets the *SwitchPortId* and *SwitchNicIndex* parameters to the same values that it used in the call to *ReferenceSwitchNic*.

For more information on MUX drivers, see NDIS MUX Intermediate Drivers.

Originating NDIS Status Indications from Physical Network Adapters

Article • 03/14/2023

This topic discusses the method that is used by an extensible switch forwarding extension to originate NDIS status indications for a network adapter that is connected to the switch. The extension can originate an NDIS status indication for the following types of adapters:

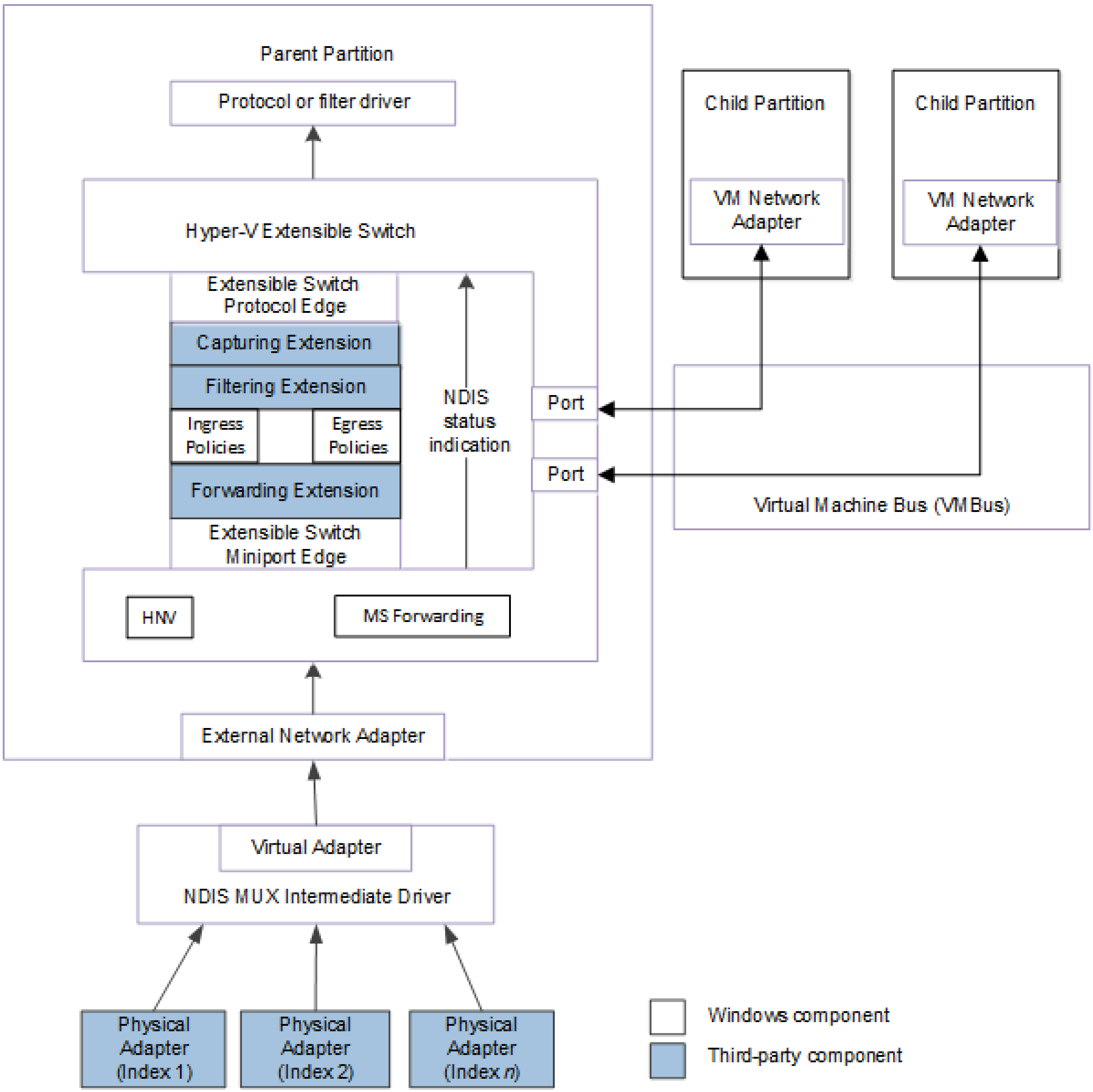
- One or more underlying physical adapters that are bound to the external network adapter of the extensible switch.

  For example, the external network adapter can be bound to the virtual miniport edge of an NDIS multiplexer (MUX) intermediate driver. The MUX driver is bound to a team of one or more physical networks on the host. This configuration is known as an *extensible switch team*.

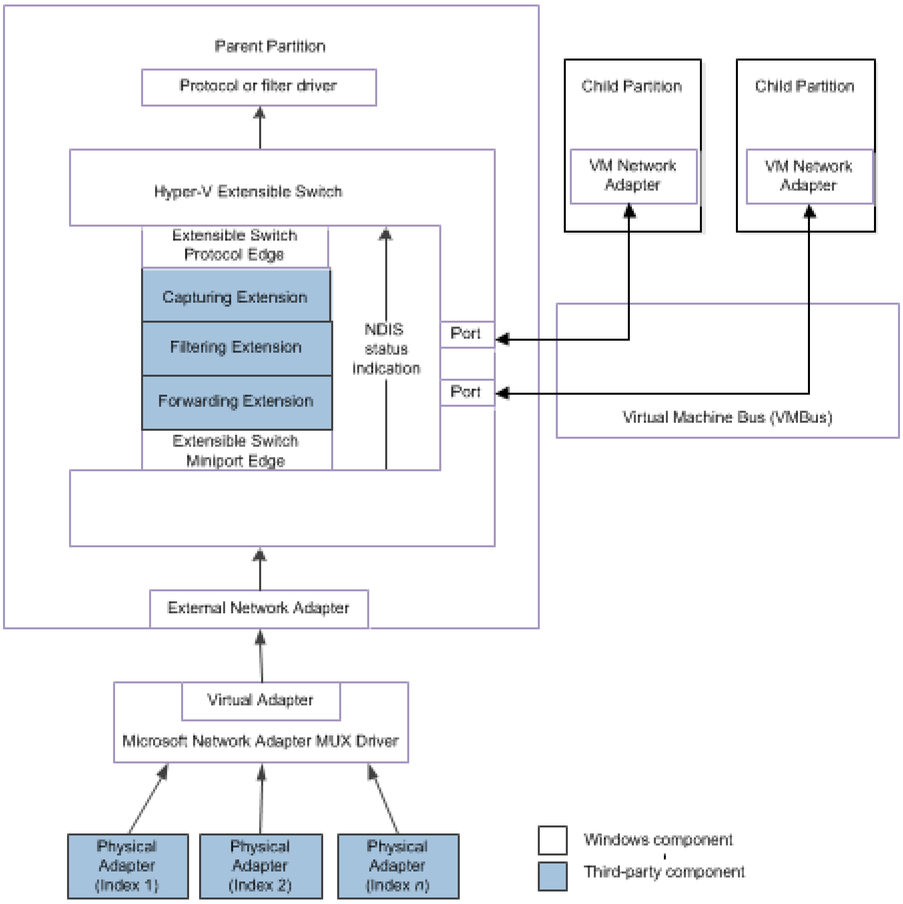
  In this configuration, an extensible switch extension is exposed to every network adapter in the team. This allows the extension to manage the configuration and use of individual network adapters in the team. For example, a forwarding extension can provide support for a load balancing failover (LBFO) solution over the team by forwarding outgoing packets to individual adapters. A forwarding extension that manages an extensible switch team is known as a *teaming provider*. For more information about teaming providers, see Teaming Provider Extensions.

- A virtual machine (VM) network adapter that is exposed within a Hyper-V child partition and connected to an extensible switch port.

The following figure shows the Hyper-V extensible switch control path for NDIS status indications from physical and VM network adapters for NDIS 6.40 (Windows Server 2012 R2) and later.



The following figure shows the Hyper-V extensible switch control path for NDIS status indications from physical and VM network adapters for NDIS 6.30 (Windows Server 2012).


Parent Partition
Protocol or filter driver
Hyper-V Extensible Switch
Extensible Switch Protocol Edge
Capturing Extension
Filtering Extension
Forwarding Extension
Extensible Switch Miniport Edge
NDIS status indication
Port
Port
External Network Adapter
Child Partition
VM Network Adapter
Child Partition
VM Network Adapter
Virtual Machine Bus (VMBus)
Virtual Adapter
Microsoft Network Adapter MUX Driver
Physical Adapter (Index 1)
Physical Adapter (Index 2)
Physical Adapter (Index *n*)
□ Windows component
▨ Third-party component

Note In the extensible switch interface, NDIS filter drivers are known as *extensions* and the driver stack is known as the *extensible switch driver stack*.

A forwarding extension can originate encapsulated hardware offload status indications to overlying drivers in the extensible switch driver stack. This also allows the extension to change the current offload capabilities of the underlying team of physical adapters that are bound to the external network adapter of the extensible switch. When a team of adapters are bound to the external network adapter, only the common capabilities of the team are advertised to NDIS or overlying protocol and filter drivers. The extension can extend the advertised capabilities by originating encapsulated status indications to advertise capabilities that are supported by some adapters in the team. For example, the extension can issue an encapsulated NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES indication to change the currently enabled receive filter capabilities for the entire team.

Note Only forwarding extensions can originate encapsulated status indications. For more information about this type of extension, see Forwarding Extension.

Typically, the forwarding extension originates encapsulated NDIS status indications to change the advertised hardware offload capabilities of the underlying physical adapter. For example, the extension can originate status indications for the following types of hardware offloads:

- Internet Protocol security (IPsec).

- Virtualized machine queue (VMQ).
- Single root I/O virtualization (SR-IOV).

The forwarding extension can also originate encapsulated NDIS status indications to change the hardware offload resources that are allocated for a Hyper-V child partition. Starting with NDIS 6.30, the extension can issue an encapsulated NDIS_STATUS_SWITCH_PORT_REMOVE_VF indication to remove the binding between a VM network adapter and a PCI Express (PCIe) virtual function (VF). The VF is exposed and supported by an underlying physical network adapter that supports the single root I/O virtualization (SR-IOV) interface.

If the forwarding extension originates an encapsulated NDIS status indication for the hardware offload resources of an underlying physical adapter, it must set the members of the NDIS_SWITCH_NIC_STATUS_INDICATION structure in the following way:

- The **DestinationPortId** member must be set to **NDIS_SWITCH_DEFAULT_PORT_ID**.
- The **DestinationNicIndex** member must be set to **NDIS_SWITCH_DEFAULT_NIC_INDEX**
- The **SourcePortId** member must be set to the identifier of the extensible switch port to which the external network adapter is connected.
- The **SourceNicIndex** member must be set to **NDIS_SWITCH_DEFAULT_NIC_INDEX**. This allows the status indication to be interpreted as originating from the entire extensible switch team that is bound to the external network adapter.

  Note The forwarding extension must also set this member to **NDIS_SWITCH_DEFAULT_NIC_INDEX** if only a single physical network adapter is bound to the external network adapter.

- The **StatusIndication** member must be set to a pointer to an NDIS_STATUS_INDICATION structure. This structure contains the data for the encapsulated NDIS status indication.

If the forwarding extension is originating an NDIS status indication for the hardware offload resources of a Hyper-V child partition, it must set the members of the NDIS_SWITCH_NIC_STATUS_INDICATION structure in the following way:

- The **DestinationPortId** and **DestinationNicIndex** members must be set to the corresponding values of the port and network adapter index for the network connection that is used by the partition.
- The **SourcePortId** member must be set to **NDIS_SWITCH_DEFAULT_PORT_ID**.

- The **SourceNicIndex** member must be set to **NDIS_SWITCH_DEFAULT_NIC_INDEX**.

- The **StatusIndication** member must be set to a pointer to an **NDIS_STATUS_INDICATION** structure. This structure contains the data for the encapsulated NDIS status indication.

When the extension issues the encapsulated NDIS status indication, it must follow these steps:

1. The extension calls *ReferenceSwitchNic* to increment a reference counter for the source or destination network adapter connection. This guarantees that the extensible switch interface will not delete the network adapter connection while its reference counter is nonzero.

   When the extension calls *ReferenceSwitchNic*, it sets the parameters in the following ways:

   - If the forwarding extension is originating an encapsulated NDIS status indication for an underlying physical adapter, it sets the *SwitchPortId* parameter to the value specified for the **SourcePortId** member. The extension also sets the *SwitchNicIndex* parameter to the value specified for the **SourceNicIndex** member.

   - If the forwarding extension is originating an NDIS status indication for a Hyper-V child partition, it sets the *SwitchPortId* parameter to the value specified for the **DestinationPortId** member. The extension also sets the *SwitchNicIndex* parameter to the value specified for the **DestinationNicIndex** member.

   Note If *ReferenceSwitchNic* does not return NDIS_STATUS_SUCCESS, the encapsulated NDIS status indication cannot be issued.

2. The extension calls **NdisFIndicateStatus** to forward the encapsulated status notification.

   Note If the extension is forwarding a filtered OID request, it must call **NdisFIndicateStatus** within the context of the call to its *FilterStatus* function.

3. After **NdisFIndicateStatus** returns, the extension calls *DereferenceSwitchNic* to clear the reference counter for the source or destination network adapter connection. The extension sets the *SwitchPortId* and *SwitchNicIndex* parameters to the same values that it used in the call to *ReferenceSwitchNic*.

Installing Hyper-V Extensible Switch Extensions

Article • 12/15/2021

This section describes the installation of Hyper-V extensible switch extensions and includes the following topics:

INF Requirements for Hyper-V Extensible Switch Extensions

Extension Driver MSI Packaging Requirements

Managing Installed Hyper-V Extensible Switch Extensions

INF Requirements for Hyper-V Extensible Switch Extensions

Article • 12/15/2021

Hyper-V extensible switch extensions are developed as NDIS filter drivers. As a result, the INF requirements for extensions are based on the INF requirements for all NDIS filter drivers. When you create an INF file for an extensible switch extension, you should use the INF settings for a modifying or monitoring filter driver. For more information on these settings, see INF File Settings for Filter Drivers.

In addition, you must follow these guidelines for INF files for extensible switch extensions:

- An extensible switch extension must be installed as a modifying filter driver.

  For more information on the INF requirements for a modifying filter driver, see Configuring an INF File for a Modifying Filter Driver.

  Note An extension with a filter class of **ms_switch_capture** can perform the same tasks as a monitoring filter driver. For more information, see Types of Filter Drivers.

- The **FilterMediaTypes** entry in the filter INF file defines the driver's bindings to other drivers and interfaces. The **FilterMediaTypes** entry for an extensible switch extension must include the **vmnetextension** value. This value specifies a binding to the extensible switch extension miniport adapter.

  The **FilterMediaTypes** entry allows a comma-delimited list of media types to be specified. This allows the extension to be bound to a physical interface or to the extensible switch interface.

  The following example shows a **FilterMediaTypes** entry that allows an extension to be bound to either the physical Ethernet network adapter or an extensible switch virtual network adapter.

  INF

  ```
  HKR, Ndi\Interfaces, FilterMediaTypes, , "ethernet, vmnetextension"
  ```

  If the **FilterMediaTypes** entry only specifies the **vmnetextension** value, the extension will only bind to the driver stacks for all extensible switches on the system.

If the **FilterMediaTypes** entry specifies **vmnetextension** as well as other media types, the extension can determine whether it is bound within an extensible switch driver stack by calling **NdisFGetOptionalSwitchHandlers**. If the function returns NDIS_STATUS_SUCCESS, the extension is bound within the extension driver stack. If the function returns NDIS_STATUS_NOT_SUPPORTED, the extension is bound within the driver stack for a different physical network interface.

For more information about the **FilterMediaTypes** entry, see Intermediate Driver UpperRange And LowerRange INF File Entries.

- The **FilterClass** value in the INF file for an extension determines its order in a stack of filters. The **FilterClass** entry must contain one of the values from the following table.

FilterClass value	Description
ms_switch_capture	An extension of this class monitors packet traffic. However, this class of extension cannot apply port policies or alter destination ports for a packet. For more information about this class of extension, see Capturing Extensions.
ms_switch_filter	An extension of this class filters packet traffic and enforces port or switch policy for packet delivery through the extensible switch. This class of driver can also inspect and remove destination ports for each packet based on policy settings. For more information about this class of extension, see Filtering Extensions.

FilterClass value	Description
ms_switch_forward	An extension of this class has the same capabilities as the **ms_switch_filter** class. This class of extension can also forward packets to other extensible switch ports, as well as inject packet traffic to any extensible switch port. On the ingress data path, this class of extension is invoked after the **ms_switch_filter** class of extension. On the egress data path, this class of extension is invoked before the **ms_switch_filter** class of extension. For more information about this class of extension, see Forwarding Extensions. **Note** Only one extension of this class is allowed in the extensible switch driver stack.

When the extension is installed with these INF settings, it will be configured to bind to every extensible switch instance. However, the binding will be disabled and must be explicitly enabled through a PowerShell cmdlet. For more information on this procedure, see Enabling Hyper-V Extensible Switch Extensions.

Extension Driver MSI Packaging Requirements

Article • 12/15/2021

Switch extensions must be packaged in a silently installable MSI file. This file can then be deployed to the computer where the extensions are used by management applications automatically.

The MSI file must meet the following requirements:

- Drivers must be packaged and distributed in the standard MSI package format.
- The MSI package must be silently uninstallable.
- The MSI package can contain only one extension.
- The MSI package must contain the required table fields described in the MSI table fields listed below. In addition, the MSI file must be able to silently install the driver .sys, .inf and any supplemental files required for the driver to operate using the parameters described in the *DriverInstallParams* field of the MSI Properties table fields list below.

Field	Required	Type	Details
Description	Required	**String**	Description for the extension that is displayed.
Manufacturer	Required	**String**	Name of the company publishing the extension driver. Localized strings can be stored.
ProductVersion	Required	**String**	The version of the this MSI package. Example: 1.0.0.0
ProductName	Required	**String**	Name of the driver.
DriverID	Required	**String**	Must match the Msvm_InstalledEthernetSwitchExtension.Name field that is available after the driver is installed and the driver ID in the driver's INF file.
DriverVersion	Required	**String**	The version of the driver contained in this package. Example: 1.0.0.0
ExtensionType	Required	**String**	Type of the extension. Values: Forwarding, Capture, Monitoring, Filter

Field	Required	Type	Details
DriverInstallParams	Required	**String**	Parameters used to install this driver silently. Example: /q
IsManagedByExtensionManager	Optional	**String**	Present and non-zero = Yes, 0 or not present = No
MinApplicableOSVersion	Required	**String**	The minimum version of the Windows operating system that this extension will run on. See Operating System Version for operating system version numbers. Note that the Hyper-V Extensible Switch feature was added in Windows Server 2012, so the lowest valid value for this field is "6.2".
MaxApplicableOSVersion	Optional	**String**	The maximum version of the Windows operating system that this extension will run on. See Operating System Version for operating system version numbers. Note that the Hyper-V Extensible Switch feature was added in Windows Server 2012, so the lowest valid value for this field is "6.2" or the value of **MinApplicableOSVersion**, whichever is higher. This field is optional. If no value is specified, the extension will run on **MinApplicableOSVersion** and later.

Managing Installed Hyper-V Extensible Switch Extensions

Article • 12/15/2021

Starting with Windows Server 2012, Hyper-V extensible switch extensions within each extensible switch instance can be managed through PowerShell cmdlets. Through these cmdlets, extensions can be enabled or disabled, and the order of extensions within the same class can be changed in the extensible switch driver stack. This allows each extensible switch instance to have a different set of enabled extensions and a unique ordering of extensions of the same class.

This section includes the following topics that describe how you use PowerShell cmdlets to manage Hyper-V extensible switch extensions:

- Enumerating Hyper-V Extensible Switch Instances
- Enumerating Hyper-V Extensible Switch Extensions
- Enabling Hyper-V Extensible Switch Extensions
- Disabling Hyper-V Extensible Switch Extensions
- Reordering Hyper-V Extensible Switch Extensions

Enumerating Hyper-V Extensible Switch Instances

Article • 12/15/2021

The Get-VMSwitch PowerShell cmdlet enumerates the Hyper-V virtual networks that have been created. One or more Hyper-V child partitions can be assigned to each virtual network. The Hyper-V virtualization stack creates an instance of a Hyper-V extensible switch for a virtual network when the first Hyper-V child partition that is assigned to the network is started.

The Get-VMSwitch cmdlet uses the following syntax:

syntax

```
Get-VMSwitch [[-Name] <string>] [-SwitchType <VMSwitchType[]>] [[-
ResourcePoolName] <string[]>] [-ComputerName
    <string[]>] [<CommonParameters>]

Get-VMSwitch [[-Id] <Guid[]>] [-SwitchType <VMSwitchType[]>] [[-
ResourcePoolName] <string[]>] [-ComputerName
    <string[]>] [<CommonParameters>]
```

The following example shows the output from the Get-VMSwitch cmdlet.

syntax

```
PS C:\Windows\system32> Get-VMSwitch

Name                           Learnable Status
                               Addresses
----                           --------- ------
Virtual Network - 1            2048      {OK}
Virtual Network - 2            2048      {OK}
```

Related topics

Get-VMSwitch

Msvm_VirtualEthernetSwitch

Enumerating Hyper-V Extensible Switch Extensions

Article • 12/15/2021

The Get-VMSwitchExtension PowerShell cmdlet enumerates the Hyper-V extensible switch extensions that are currently bound to an instance of an extensible switch. This cmdlet also reports whether the extension is enabled in the extensible switch instance.

The Get-VMSwitchExtension cmdlet uses the following syntax:

syntax

```
Get-VMSwitchExtension [[-VMSwitchName] <string[]>] [[-Name] <string[]>] [-
ComputerName <string[]>]
    [<CommonParameters>]

Get-VMSwitchExtension [[-VMSwitch] <VMSwitch[]>] [-ComputerName <string[]>]
[<CommonParameters>]
```

The following example shows the output from the Get-VMSwitchExtension cmdlet.

syntax

```
PS C:\Windows\system32> Get-VMSwitchExtension PrivateNetwork | fl -property
@("Name","ExtensionType", "SwitchName","Enabled")

Name          : NDIS Capture LightWeight Filter
ExtensionType : Capture
SwitchName    : PrivateNetwork
Enabled       : False

Name          : Switch Extensibility Test Extension 2
ExtensionType : Filter
SwitchName    : PrivateNetwork
Enabled       : False

Name          : Switch Extensibility Test Extension 1
ExtensionType : Filter
SwitchName    : PrivateNetwork
Enabled       : False

Name          : WFP extensible switch Layers LightWeight Filter
ExtensionType : Filter
SwitchName    : PrivateNetwork
Enabled       : True
```

Note In order to minimize the amount of information, the example pipes the returned extension objects through the filter command "fl". This causes a subset of information to be displayed that matches the attributes of the **-property** switch.

Related topics

Get-VMSwitchExtension

Msvm_EthernetSwitchExtension

Enabling Hyper-V Extensible Switch Extensions

Article • 12/15/2021

When Hyper-V extensible switch extensions are installed, they are bound to each instance of an extensible switch. However, the extensions are disabled by default and must be explicitly enabled on each extensible switch instance.

The Enable-VMSwitchExtension PowerShell cmdlet enables an extension on a specific instance of an extensible switch. This cmdlet uses the following syntax:

syntax

```
Enable-VMSwitchExtension [-Name] <string[]> [-ComputerName <string[]>]
[<CommonParameters>]

Enable-VMSwitchExtension [-Name] <string[]> [-VMSwitchName] <string[]> [-
ComputerName <string[]>]
    [<CommonParameters>]

Enable-VMSwitchExtension [-Name] <string[]> [-VMSwitch] <VMSwitch[]> [-
ComputerName <string[]>]
    [<CommonParameters>]

Enable-VMSwitchExtension [-VMSwitchExtension] <VMSwitchExtension[]> [-
ComputerName <string[]>] [<CommonParameters>]
```

The following shows an example of how to use the Enable-VMSwitchExtension cmdlet.

syntax

```
PS C:\Windows\system32> Enable-VMSwitchExtension "Switch Extensibility Test
Extension 1" PrivateNetwork

PS C:\Windows\system32> Get-VMSwitchExtension PrivateNetwork "Switch
Extensibility Test Extension 1" | fl -property @("Name","ExtensionType",
"SwitchName","Enabled")

Name          : Switch Extensibility Test Extension 1
ExtensionType : Filter
SwitchName    : PrivateNetwork
Enabled       : True
```

Note The Windows Filtering Platform (WFP) in-box filtering extension (Wfplwfs.sys) is enabled by default on each extensible switch instance.

Related topics

Enable-VMSwitchExtension

Get-VMSwitchExtension

Msvm_EthernetSwitchExtension

Disabling Hyper-V Extensible Switch Extensions

Article • 12/15/2021

The Disable-VMSwitchExtension PowerShell cmdlet disables an extension on a specific instance of an extensible switch.

The Disable-VMSwitchExtension cmdlet uses the following syntax:

syntax

```
Disable-VMSwitchExtension [-VMSwitchExtensionName] <string[]> [-ComputerName
<string[]>] [<CommonParameters>]

Disable-VMSwitchExtension [-VMSwitchExtensionName] <string[]> [-
VMSwitchName] <string[]> [-ComputerName
    <string[]>] [<CommonParameters>]

Disable-VMSwitchExtension [-VMSwitchExtensionName] <string[]> [-VMSwitch]
<VMSwitch[]> [-ComputerName <string[]>]
    [<CommonParameters>]

Disable-VMSwitchExtension [-VMSwitchExtension] <VMSwitchExtension[]> [-
ComputerName <string[]>]
    [<CommonParameters>]
```

The following shows an example of how to use the Disable-VMSwitchExtension cmdlet.

syntax

```
PS C:\Windows\system32> Disable-VMSwitchExtension "Switch Extensibility Test
Extension 1" PrivateNetwork

PS C:\Windows\system32> Get-VMSwitchExtension PrivateNetwork "Switch
Extensibility Test Extension 1" | fl -property @("Name","ExtensionType",
"SwitchName","Enabled")

Name          : Switch Extensibility Test Extension 1
ExtensionType : Filter
SwitchName    : PrivateNetwork
Enabled       : False
```

Related topics

Disable-VMSwitchExtension

Get-VMSwitchExtension

Msvm_EthernetSwitchExtension

Reordering Hyper-V Extensible Switch Extensions

Article • 12/15/2021

Multiple Hyper-V extensible switch capturing or filtering extensions can be enabled in each instance of an extensible switch.

Note Only one forwarding extension can be enabled in each instance of an extensible switch.

By default, multiple capturing or filtering extensions are ordered based on their type and when they were installed. For example, multiple capturing extensions are layered in the extensible switch driver stack with the most recently installed extension closest to the protocol edge of the switch.

When multiple capturing or filtering extensions are installed, you can use PowerShell cmdlets to reorder the drivers in the extensible switch driver stack. The following example shows the commands that you enter from a PowerShell window to do this.

syntax

```
# Show the current order. The ExtensionOrder field contains paths to WMI
extension instances.
# The [wmi] operator can be used to convert the paths to full WMI objects.
PS C:\Windows\system32> $privateNetwork = Get-VMSwitch PrivateNetwork
PS C:\Windows\system32> $extensionOrder = $privateNetwork.ExtensionOrder
PS C:\Windows\system32> $extensionOrder | ForEach-Object { Write-Host
"Name:" ([wmi]$_).ElementName }
Name: NDIS Capture LightWeight Filter
Name: Switch Extensibility Test Extension 2
Name: Switch Extensibility Test Extension 1
Name: WFP extensible switch Layers LightWeight Filter

# Place “Test Extension 1” above “Test Extension 2” in the ordered list of
extensions.
PS C:\Windows\system32> $tmp = $extensionOrder[1]
PS C:\Windows\system32> $extensionOrder[1] = $extensionOrder[2]
PS C:\Windows\system32> $extensionOrder[2] = $tmp

# Commit the updated order.
PS C:\Windows\system32> $privateNetwork.ExtensionOrder = $extensionOrder

# Retrieve the switch information again to validate the order.
PS C:\Windows\system32> $privateNetwork = Get-VMSwitch PrivateNetwork
PS C:\Windows\system32> $privateNetwork.ExtensionOrder | ForEach-Object {
Write-Host "Name:" ([wmi]$_).ElementName }
```

```
Name: NDIS Capture LightWeight Filter
Name: Switch Extensibility Test Extension 1
```

Related topics

Get-VMSwitch

Msvm_EthernetSwitchExtension

Msvm_VirtualEthernetSwitchSettingData

Hyper-V Extensible Switch OIDs

Article • 12/15/2021

This section describes the Hyper-V extensible switch object identifiers (OIDs). These OIDs may be issued by either the extensible switch extension or a Hyper-V extensible switch extension.

The following table defines the characteristics of the extensible switch OIDs. The following abbreviations are used to specify the OIDs' characteristics in the table.

- Q
  The OID is used only in query requests.
- S
  The OID is used only in set requests.
- M
  The OID is used only in method requests. These requests could be issued for set or query operations.
- P
  The OID request is issued by the protocol edge of the extensible switch. The extension can inspect the results of the OID request to obtain information about the extensible switch, its ports, or virtual network adapters connected to the ports.
- E
  The OID request is issued by an extension.

Name	**Q**	**S**	**M**	**P**	**E**
OID_SWITCH_FEATURE_STATUS_QUERY			X	X	
OID_SWITCH_NIC_ARRAY	X				X
OID_SWITCH_NIC_CONNECT		X		X	
OID_SWITCH_NIC_CREATE		X		X	
OID_SWITCH_NIC_DELETE		X		X	
OID_SWITCH_NIC_DISCONNECT		X		X	
OID_SWITCH_NIC_REQUEST			X		X
OID_SWITCH_NIC_RESTORE		X		X	
OID_SWITCH_NIC_SAVE	X			X	
OID_SWITCH_NIC_SAVE_COMPLETE		X		X	

Name	Q	S	M	P	E
OID_SWITCH_PARAMETERS	X				X
OID_SWITCH_PORT_ARRAY	X				X
OID_SWITCH_PORT_CREATE		X		X	
OID_SWITCH_PORT_DELETE		X		X	
OID_SWITCH_PORT_FEATURE_STATUS_QUERY			X	X	
OID_SWITCH_PORT_PROPERTY_ADD		X		X	
OID_SWITCH_PORT_PROPERTY_DELETE		X		X	
OID_SWITCH_PORT_PROPERTY_ENUM			X		X
OID_SWITCH_PORT_PROPERTY_UPDATE		X		X	
OID_SWITCH_PORT_TEARDOWN		X		X	
OID_SWITCH_PROPERTY_ADD		X		X	
OID_SWITCH_PROPERTY_DELETE		X		X	
OID_SWITCH_PROPERTY_ENUM			X		X
OID_SWITCH_PROPERTY_UPDATE		X		X	

Hyper-V Extensible Switch Status Indications

Article • 12/15/2021

This section describes the following NDIS status indications that can be issued or handled by a Hyper-V extensible switch extension:

- **NDIS_STATUS_SWITCH_NIC_STATUS**
- **NDIS_STATUS_SWITCH_PORT_REMOVE_VF**

For more information on how extensions issue or handle extensible switch extension status indications, see Hyper-V Extensible Switch Control Path for NDIS Status Indications.

NDIS_STATUS_SWITCH_NIC_STATUS

Article • 03/14/2023

The **NDIS_STATUS_SWITCH_NIC_STATUS** status indication is used to encapsulate a status indication from a physical network adapter that is bound to the external network adapter of the Hyper-V extensible switch. Through this encapsulation, the status indication is forwarded up the extensible switch driver stack.

The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure for this indication contains a pointer to an NDIS_SWITCH_NIC_STATUS_INDICATION structure.

Remarks

When an underlying physical network adapter issues an NDIS status indication, it is received by the external network adapter. When this happens, the extensible switch interface performs these steps:

1. The interface encapsulates the status indication inside an NDIS_SWITCH_NIC_STATUS_INDICATION structure.

2. The interface issues an **NDIS_STATUS_SWITCH_NIC_STATUS** status indication to forward the encapsulated status indication up the extensible switch driver stack. This allows extensible switch extensions to modify the encapsulated status indication.

   Typically, the extension modifies an encapsulated status indication to change the current offload capabilities of the underlying team of physical adapters that are bound to the external network adapter.

   For more information about the different configurations in which physical network adapters can be bound to the external network adapter, see Types of Physical Network Adapter Configurations.

3. When the **NDIS_STATUS_SWITCH_NIC_STATUS** status indication is received by the overlying extensible switch protocol driver in the stack, the interface forwards the decapsulated status indication to overlying protocol or filter drivers.

An extension can also originate encapsulated hardware offload status indications to overlying drivers in the extensible switch driver stack. This also allows the driver to change the current offload capabilities of the underlying team of physical adapters that are attached to the external network adapter. When a team of adapters are bound to the external network adapter, only the common capabilities of the team are advertised to NDIS or the overlying protocol and filter drivers. The extension can extend the advertised capabilities by originating encapsulated status indications to advertise capabilities that are supported by some adapters in the team.

For example, the extension can issue an encapsulated NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES indication to change the currently-enabled receive filter capabilities for the entire team.

For more information on how to forward or originate **NDIS_STATUS_SWITCH_NIC_STATUS** indications, see Managing NDIS Status Indications from Physical Network Adapters.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NDIS_STATUS_RECEIVE_FILTER_CURRENT_CAPABILITIES

NDIS_STATUS_SWITCH_PORT_REMOVE_VF

Article • 03/14/2023

The **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** status indication is issued by a Hyper-V extensible switch forwarding extension to remove the binding between a virtual machine (VM) network adapter and a PCI Express (PCIe) virtual function (VF). The VF is exposed and supported by an underlying physical network adapter that supports the single root I/O virtualization (SR-IOV) interface.

In order to issue the **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** status indication, the forwarding extension must encapsulate the indication in an **NDIS_SWITCH_NIC_STATUS_INDICATION** structure and issue an **NDIS_STATUS_SWITCH_NIC_STATUS** status indication.

For more information on this process, see Guidelines for Issuing an **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** Status Indication.

Remarks

A PCIe VF is created and allocated by an underlying physical adapter that supports the SR-IOV interface. Once created, the virtualization stack attaches, or *assigns*, the VF to a Hyper-V child partition. The guest operating system that runs in this partition exposes a virtual machine (VM) network adapter that is bound to the VF of the underlying SR-IOV physical adapter.

After the virtual and physical network adapters are assigned, packets are routed directly between the VF and the VM network adapter. However, because the extensible switch is not involved in packet delivery, extensible switch port policies are not applied to these packets. This includes port policies for access control lists (ACLs) and quality of service (QoS).

An extensible switch forwarding extension can remove the assignment of the VF to the child partition by issuing an **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** status indication. This indication causes the packets to be delivered through an extensible switch port instead of directly between the VM network adapter and the VF of the underlying SR-IOV physical adapter. This allows the extensible switch port policies to be applied to packets that are received or sent over the extensible switch port.

When the forwarding extension makes the **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** status indication, it specifies the extensible switch port to which the VM network adapter is connected.

For more information about extensible switch forwarding extensions, see Forwarding Extensions.

Guidelines for Issuing an NDIS_STATUS_SWITCH_PORT_REMOVE_VF Status Indication

In order to issue the **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** status indication, the forwarding extension must follow these steps:

1. The forwarding extension initializes an NDIS_STATUS_INDICATION structure for the **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** indication. For this indication, the forwarding extensions sets the following members of the **NDIS_STATUS_INDICATION** structure:

   - The **StatusCode** member must be set to **NDIS_STATUS_SWITCH_PORT_REMOVE_VF**.

   - The **StatusBuffer** member must be set to **NULL**.

   - The **StatusBufferSize** must be set to zero.

2. The forwarding extension initializes an NDIS_SWITCH_NIC_STATUS_INDICATION structure. In order to remove a VF assignment, the forwarding extension must set the members in the following way:

   - The **DestinationPortId** member must be set to the identifier of an extensible switch port to which the VM network adapter is connected.

   - The **DestinationNicIndex** member must be set to the index value of the VM network adapter that is connected to the specified port.

   - The **SourcePortId** member must be set to **NDIS_SWITCH_DEFAULT_PORT_ID**.

   - The **SourceNicIndex** member must be set to **NDIS_SWITCH_DEFAULT_NIC_INDEX**.

   - The **StatusIndication** member must be set to the address of the NDIS_STATUS_INDICATION structure for the **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** indication.

3. The forwarding extension initializes an NDIS_STATUS_INDICATION structure for the NDIS_SWITCH_NIC_STATUS_INDICATION indication. For this indication, the forwarding extension sets the following members of the **NDIS_STATUS_INDICATION** structure:

   - The **StatusCode** member must be set to NDIS_STATUS_SWITCH_NIC_STATUS.

   - The **StatusBuffer** member must be set to the address of the NDIS_SWITCH_NIC_STATUS_INDICATION structure.

   - The **StatusBufferSize** must be set to the length, in bytes, of the NDIS_SWITCH_NIC_STATUS_INDICATION structure and the NDIS_STATUS_INDICATION structure for the **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** indication.

4. The forwarding extension must call *ReferenceSwitchNic* to increment a reference counter for the VM network adapter. If *ReferenceSwitchNic* does not complete with NDIS_STATUS_SUCCESS, the forwarding extension must not forward the status indication.

   Note If the forwarding extension has received an OID_SWITCH_NIC_DISCONNECT set request for the VM adapter, it must not call *ReferenceSwitchNic* nor forward the status indication.

5. The forwarding extension calls **NdisFIndicateStatus** to forward the **NDIS_STATUS_INDICATION** to overlying extensions in the extensible switch driver stack. When the forwarding extension calls this function, it sets the *StatusIndication* parameter to a pointer to the **NDIS_STATUS_INDICATION** structure for the **NDIS_STATUS_SWITCH_NIC_STATUS** indication.

6. After **NdisFIndicateStatus** returns, the forwarding extension must call *DereferenceSwitchNic* to decrement the reference counter for the VM network adapter.

Note The forwarding extension must follow the previous steps for each VF assignment that the forwarding extension is removing.

For more information on how a forwarding extension forwards status indications, see Filter Module Status Indications.

Guidelines for Determining VF Assignments

The forwarding extension can enumerate the current VF assignments for virtual network adapters by issuing an OID query request of OID_SWITCH_NIC_ARRAY. This request returns an **NDIS_SWITCH_NIC_ARRAY** structure that contains an array of **NDIS_SWITCH_NIC_PARAMETERS** structures. Each **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the parameters of a network adapter that is exposed in one of the following environments:

- The management operating system that runs in the Hyper-V parent partition.

  Network adapters that are exposed in this operating system are specified with an **NDIS_SWITCH_NIC_TYPE** enumeration value of **NdisSwitchNicTypeExternal** or **NdisSwitchNicTypeInternal**.

- The guest operating system that runs in a Hyper-V child partition.

  Network adapters that are exposed in this operating system are specified with an **NDIS_SWITCH_NIC_TYPE** enumeration value of **NdisSwitchNicTypeSynthetic** or **NdisSwitchNicTypeEmulated**.

If the OID query request of OID_SWITCH_NIC_ARRAY completes with a status of NDIS_STATUS_SUCCESS, the forwarding extension can determine VF assignments by inspecting each **NDIS_SWITCH_NIC_PARAMETERS** structure in the returned array. If the **VFAssigned** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure is set to **TRUE**, the network adapter that corresponds to the **NDIS_SWITCH_NIC_PARAMETERS** structure is assigned to a VF.

The forwarding extension can remove the assignment by issuing an **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** status indication. In this case, the forwarding extension must set the **DestinationPortId** member of the **NDIS_SWITCH_NIC_STATUS_INDICATION** to the value of the **PortId** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure.

For more information on how to issue an **NDIS_STATUS_SWITCH_PORT_REMOVE_VF** status indication, see Guidelines for Issuing an NDIS_STATUS_SWITCH_PORT_REMOVE_VF Status Indication.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h (include Ndis.h)

See also

NdisFIndicateStatus

NDIS_STATUS_INDICATION

NDIS_STATUS_SWITCH_NIC_STATUS

NDIS_SWITCH_NIC_ARRAY

NDIS_SWITCH_NIC_PARAMETERS

NDIS_SWITCH_NIC_TYPE

OID_SWITCH_NIC_ARRAY

NDIS_STATUS_ISOLATION_PARAMETERS_CHANGE

Article • 03/14/2023

A VM network adapter miniport driver generates an **NDIS_STATUS_ISOLATION_PARAMETERS_CHANGE** status indication whenever the routing domain configuration is updated on the network adapter's port. This triggers the TCP layer to re-query the multi-tenancy configuration by issuing an OID_GEN_ISOLATION_PARAMETERS OID. This status indication does not have a status buffer.

Requirements

Version	Supported in NDIS 6.40 and later.
Header	Ndis.h

See also

NDIS_STATUS_INDICATION

OID_GEN_ISOLATION_PARAMETERS

NDIS_STATUS_RECEIVE_FILTER_QUEUE_PARAMETERS

Article • 03/14/2023

The **NDIS_STATUS_RECEIVE_FILTER_QUEUE_PARAMETERS** status indicates to NDIS and overlying drivers that the current virtual machine (VM) queue parameters have changed on the network adapter.

Remarks

The miniport driver must issue an **NDIS_STATUS_RECEIVE_FILTER_QUEUE_PARAMETERS** status indication when the current VM queue parameters have changed on the network adapter. The VM queue parameters could change when one of the following conditions is true:

- The VM queue parameters are changed through a management application developed by the independent hardware vendor (IHV).
- The VM queue parameters change for one or more network adapters that belong to a load balancing failover (LBFO) team managed by a MUX intermediate driver. For more information, see NDIS MUX Intermediate Drivers.

When the miniport driver issues the **NDIS_STATUS_RECEIVE_FILTER_QUEUE_PARAMETERS** status indication, it must follow these steps:

1. The miniport driver initializes an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure with the current VM queue parameters on the network adapter. The driver must also set the **Flags** member of this structure with the appropriate NDIS_RECEIVE_QUEUE_PARAMETERS_*Xxx*_CHANGED flags to report on **NDIS_RECEIVE_QUEUE_PARAMETERS** member values that have changed.

   Note Starting with NDIS 6.30, the miniport driver can only issue an **NDIS_STATUS_RECEIVE_FILTER_QUEUE_PARAMETERS** status indication to report on changes to the **InterruptCoalescingDomainId** member.

When the miniport driver initializes the **Header** member of this structure, it sets the **Type** member of **Header** to NDIS_OBJECT_TYPE_DEFAULT. The miniport driver sets the **Revision** member of **Header** to NDIS_RECEIVE_QUEUE_PARAMETERS_REVISION_2 and the **Size** member to NDIS_SIZEOF_RECEIVE_QUEUE_PARAMETERS_REVISION_2.

2. The miniport driver initializes an NDIS_STATUS_INDICATION structure in the following way:

   - The **StatusCode** member must be set to **NDIS_STATUS_RECEIVE_FILTER_QUEUE_PARAMETERS**.

   - The **StatusBuffer** member must be set to the pointer to a NDIS_RECEIVE_QUEUE_PARAMETERS structure. This structure contains the currently enabled hardware capabilities of the NIC switch.

   - The **StatusBufferSize** member must be set to sizeof(NDIS_RECEIVE_QUEUE_PARAMETERS).

3. The miniport driver issues the status notification by calling NdisMIndicateStatusEx. The driver must pass a pointer to the NDIS_STATUS_INDICATION structure to the *StatusIndication* parameter.

Overlying drivers can use the **NDIS_STATUS_RECEIVE_FILTER_QUEUE_PARAMETERS** status indication to determine the current VM queue parameters on the network adapter. Alternatively, these drivers can also issue object identifier (OID) query requests of OID_RECEIVE_FILTER_QUEUE_PARAMETERS to obtain these parameters at any time.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ndis.h

See also

NDIS_RECEIVE_QUEUE_PARAMETERS

NDIS_STATUS_INDICATION

OID_RECEIVE_FILTER_QUEUE_PARAMETERS

NDIS_STATUS_RECEIVE_QUEUE_STATE

Article • 03/14/2023

The NDIS_STATUS_RECEIVE_QUEUE_STATE status indicates to overlying drivers that the queue state of a virtual machine queue (VMQ) receive queue has changed.

Remarks

NDIS 6.20 and later miniport drivers that support the virtual machine queue interface generate this status indication.

The miniport driver supplies an **NDIS_RECEIVE_QUEUE_STATE** structure in the **StatusBuffer** member of the **NDIS_STATUS_INDICATION** structure.

The change to the *DMA Stopped* state is the only queue state change indication that is required. A miniport driver must indicate this state after it receives an OID_RECEIVE_FILTER_FREE_QUEUE set request and stops the DMA. In this case, the miniport driver sets the **QueueState** member of the **NDIS_RECEIVE_QUEUE_STATE** structure to **NdisReceiveQueueOperationalStateDmaStopped**.

After the miniport driver receives the OID_RECEIVE_FILTER_FREE_QUEUE set request, it must stop DMA to any shared memory that was allocated for the specified queue.

If the miniport driver stopped the DMA for some other reason (for example, it freed the last filter on a queue), the queue should not enter the *DMA Stopped* state. However, the DMA can be stopped in the *Paused* or *Running* states if there are no filters set on the queue.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ndis.h

See also

NDIS_RECEIVE_QUEUE_STATE

NDIS_STATUS_INDICATION

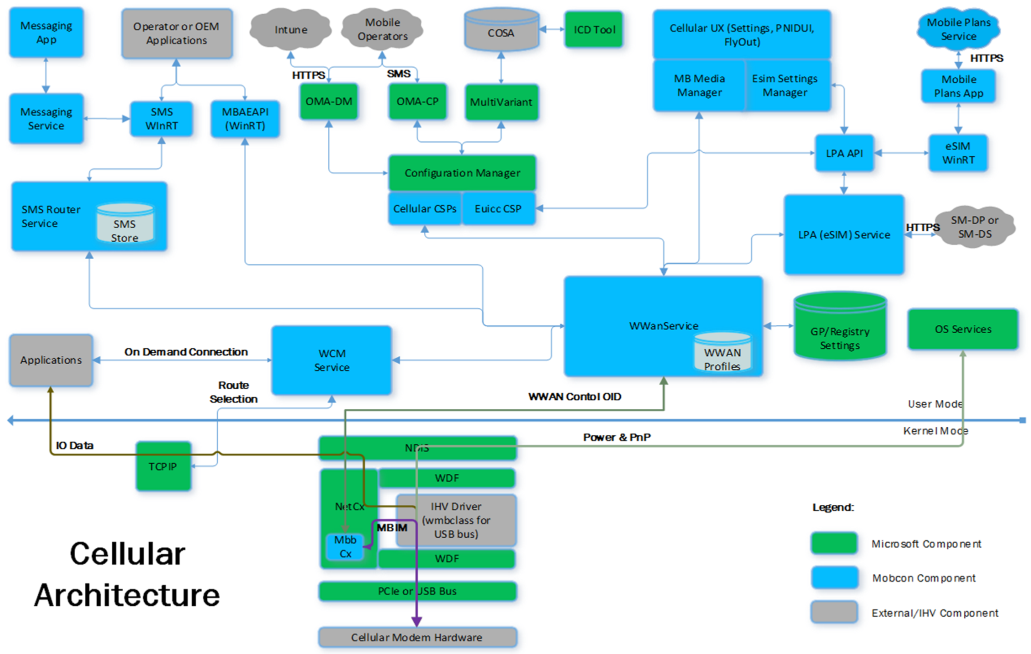
OID_RECEIVE_FILTER_FREE_QUEUE

Cellular architecture

Article • 03/14/2023

This section describes the elements of the cellular architecture for Windows 10 and how they interact. It also includes the implementation requirements for making cellular modem hardware compatible with Windows 10.

Windows 10 cellular architecture



The following describes the elements shown in the Windows 10 cellular architecture:

User Mode

WWAN Service and MBAE WinRT API

The Wireless Wide Area Network Service (WwanSvc) is responsible for handling modem initialization, registration, power state changes, and automatic and manual connection for default and on-demand cellular connection. WWAN Service also handles the modem access interface for SAR, PCO, Scan, SMS, USSD, LTE configuration, SIM File, SIM PIN, and low level SIM card access. The Mobile Broadband Account Experience Windows Runtime (MBAE WinRT) API allows programmatic access to these interfaces for original equipment manufacturer (OEM)/Mobile Operator (MO) applications.

WCM Service

The Windows Connection Manager (WCM) Service controls L3 connectivity and dynamically selects which specific L2 media (Ethernet, Wi-Fi, or Cellular) should be connected or disconnected at any given time.

SMS Router Service and SMS WinRT API

The SMS Router Service is responsible for decoding the SMS Packet Data Unit (PDU) and routing SMS messages to associated applications. The SMS WinRT API allows applications to subscribe to SMS messages and launch when the matching messages are received. Apps can also send SMS messages. The SMS messages are temporarily stored for concatenation while decoding the messages and for reliable delivery to services and applications.

Messaging Service and Messaging App

The messaging service stores user text messages for persistent access and the application displays the messages to users.

LPA (eSIM) Service and eSIM WinRT API

The Local Profile Assistant (LPA) Service implements GSMA specification for remote SIM profile management by interacting with the Subscription Manager – Device Provisioning server (SM-DP+) to download eSIM profiles for the user. The WinRT API allows accessing eSIM profiles, enabling, disabling, and deleting profiles, and sending low level Application Protocol Data Unit (APDU) for firmware update via smartcard interface.

Cellular CSPs

Cellular Configuration Service Providers (CSPs) allow configuration management through Intune (Enterprise), Multivariant, and Open Mobile Alliance – Device Management and Client Provisioning (OMA-DM/CP). Enterprise uses EnterpriseAPN, eUICC, and MultiSIM CSPs to override the APN connectivity settings, download and activate eSIM profiles, and switch to preferred SIM slot. CM CellularEntries CSP is used to configure the default connectivity for the modem. Cellular Settings CSP is used to control roaming and automatic connection configurations. CSPLte is used for Verizon-specific configurations.

Mobile Plans Service and Mobile Plans App

The mobile plans service and application offers users a simplified mechanism to purchase and install eSIM profiles.

Cellular UX

Cellular UX is a settings application and VANUI network flyout that allows users to view and control the cellular settings, control connectivity, and change radio state. PNIDUI shows the default network connection and signal bars for the network. Quick actions and airplane mode controls allow radio state control.

COSA/MultiVariant Service

Country & Operator Settings Asset (COSA) is an OEM configurable database with settings that are applied through the MultiVariant service that are specific to the SIM inserted by the user.

Kernel Mode

NDIS

Network Driver Interface Specification (NDIS) is a driver model that abstracts network hardware from network drivers and specifies a standard interface between layered network drivers.

NetCx

Network Adapter WDF Class Extension (NetAdapterCx) is a driver model that enables you to write a KMDF-based client driver for a Network Interface Controller (NIC). NetAdapterCx gives you the power and flexibility of WDF and the networking performance of NDIS, and makes it easy to write a NIC driver.

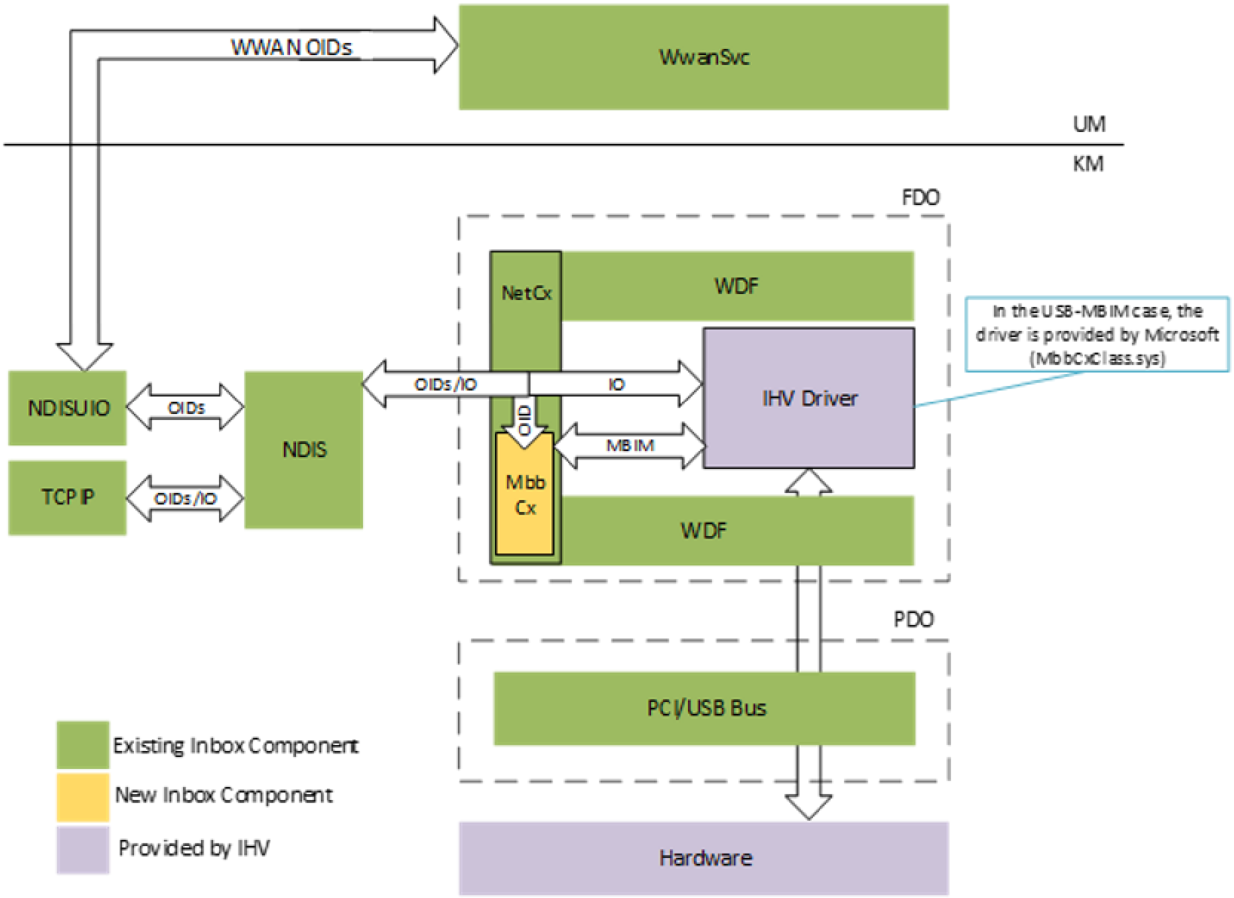
MBBCx

Mobile Broadband WDF Class Extension (MBBCx) extends the NetAdatperCx Driver Framework with cellular-specific functionalities, and implementing the "upper edge" which is common across different modems. MbbCx handles the control OIDs from NDIS and converts them to MBIM commands for the IHV driver.

IHV Driver (wmbclass)

The IHV-implemented "lower edge" cellular device driver implements all of the adapter-specific cellular driver functionalities that are specified by MBIM. For USB based modems, the interfaces are standardized and handled by the inbox wmbclass driver. For PCIe cellular modem devices, the IHV vendors are expected to provide an IHV client driver that translates the MBIM commands to be transmitted over the PCIe bus.

MBB and MBIM Driver Interactions



Windows 10 cellular implementation requirements

For Windows 10, the following is required.

- Implement the MBIM protocol interface in your modem hardware.
- Implement a USB interface to the modem hardware. This can be a removable USB dongle or another interface that presents itself as a USB host controller.

Roadmap to Develop MB Miniport Drivers

Article • 03/14/2023

To create an MB miniport driver, follow these steps:

- **Step 1**: Learn about Windows architecture and miniport drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- **Step 2**: Learn the fundamentals of MB miniport drivers.

  MB miniport drivers are supported in Windows 7 and later versions of Windows and conform to the *NDIS 6.20 Specification*. To understand the miniport driver design decisions you must make, see Introduction to NDIS 6.20.

- **Step 3**: Determine additional Windows driver design decisions.

  For information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- **Step 4**: Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user mode application. For information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Lab Kit (HLK) testing, see Building, Debugging, and Testing Drivers. For information about building, testing, verifying, and debugging tools, see Driver Development Tools.

- **Step 5**: Make design decisions about your MB miniport driver.

  For more information, see MB Interface Overview.

- **Step 6**: Develop, build, test, and debug your MB miniport driver.

  For information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process will help ensure that you build a miniport driver that works.

- **Step 7**: Create a driver package for your MB miniport driver.

For more information, see Providing a Driver Package.

- **Step 8**: Sign and distribute your MB miniport driver.

  The final step is to sign (optional) and distribute the miniport driver. If your miniport driver meets the quality standards that are defined for the Windows Hardware Lab Kit (HLK), you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual miniport driver.

MB Interface Model Overview

Article • 03/14/2023

This section provides information for mobile broadband devices that are implemented based on the Mobile Broadband Interface Model (MBIM) specification.

Starting with Windows 8, Microsoft provides an inbox class driver, referred to as MBCD, for MBIM functions. Microsoft already provides an inbox driver, USBCCGP, for composite devices. This section describes the requirements for mobile broadband devices to load USBCCGP and MBCD in Windows 8.

Mobile broadband composite devices that use WMC UFD for grouping interfaces into functions should implement Microsoft OS descriptors to load USBCCGP on Windows 8 and instruct USBCCGP to parse WMC UFD to create functions. Mobile broadband composite devices that use Interface Association Descriptors (IADs) for grouping interfaces into functions do not need to implement Microsoft OS descriptors to load USBCCGP.

MBIM functions that are backward compatible should implement Microsoft OS descriptors to load MBCD. MBIM functions that are not backward compatible do not need to implement Microsoft OS descriptors to load MBCD.

Mobile broadband devices that exhibit identity morphing should also implement Microsoft OS descriptors.

These scenarios are discussed in more detail throughout the MB Interface Model topics. The following table summarizes all of the Microsoft OS compatible IDs mentioned in these subtopics. For more information see Microsoft OS Descriptors.

Microsoft OS compatible IDs

Microsoft OS Compatible ID	Microsoft OS Sub Compatible ID	Required for Scenario
"CDC_WMC"		Loading USBCCGP on composite devices that use WMC UFD for grouping interfaces into functions
"MBIM"		Loading MBCD on MBIM backward-compatible function
"ALTRCFG"	Configuration number in ASCII	Identity morphing with IADs

Microsoft OS Compatible ID	Microsoft OS Sub Compatible ID	Required for Scenario
"WMCALTR"	Configuration number in ASCII	Identity morphing with WMC UFD

The MB Interface Model in described further in the following subtopics:

MB Interface Terms MB Union Function Descriptors MB Identity Morphing MB Interface Model Supplement

MB Interface Terms

Article • 03/14/2023

The following terminology is used throughout the Mobile Broadband (MB)documentation:

Term	Description
MBIM	Mobile Broadband Interface Model, a USB Device Working Group (DWG) specification for mobile broadband devices.
MBIM function	A USB function within a USB device that is compliant with the MBIM specification.
Mobile broadband device	A USB device that is either single function or multi-functional. In the single function case, the function should be an MBIM function. In the multi-function case, one of the functions is the MBIM function. This may also be a multi-configuration device in which at least one of the configurations contains the MBIM function.
NCM2	The earlier name for the MBIM specification. Some diagrams still refer to the MBIM functions as NCM2 functions.
Virtual CD-ROM	A CD-ROM function that does not have a physical CD-ROM drive.
IAD	Interface association descriptors (IADs) used to group interfaces into functions.
WMC UFD	Union function descriptors (UFDs) described in the Wireless Mobile Communication (WMC) specifications. UFDs are used to group interfaces into functions. This is an alternative to using IADs.
Morphing	The ability of a USB device to expose a different set of USB functions than what is currently exposed.
Driver	Software required by Windows to work with a USB function.
Inbox driver	A driver supplied by Microsoft for USB functions. These drivers are present in Windows.
IHV driver	A driver supplied by the independent hardware vendor (IHV) for USB functions that do not have inbox drivers.
IHV driver package	A collection of all IHV drivers supplied by the IHV.
USBHUB	A Microsoft USB hub driver.
USBCCGP	A Microsoft driver for USB composite devices.

Term	Description
MBCD	Mobile Broadband Class Driver, the inbox driver in Windows 8 for USB functions that conform to the MBIM specification.

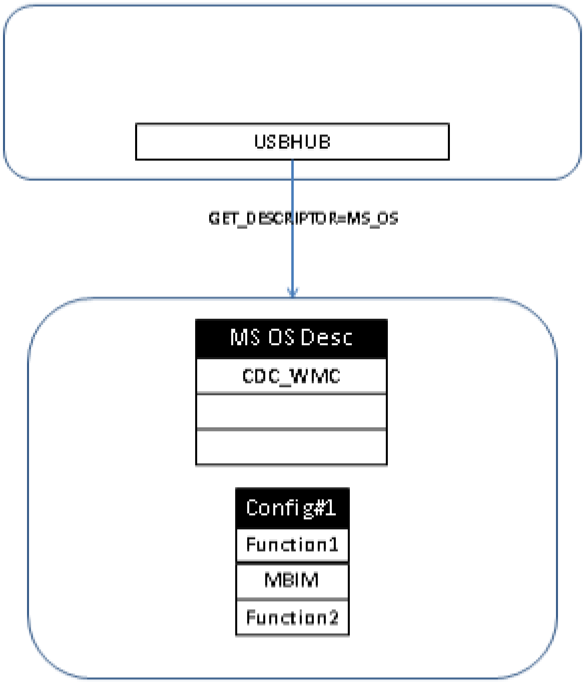
MB Union Function Descriptors

Article • 03/14/2023

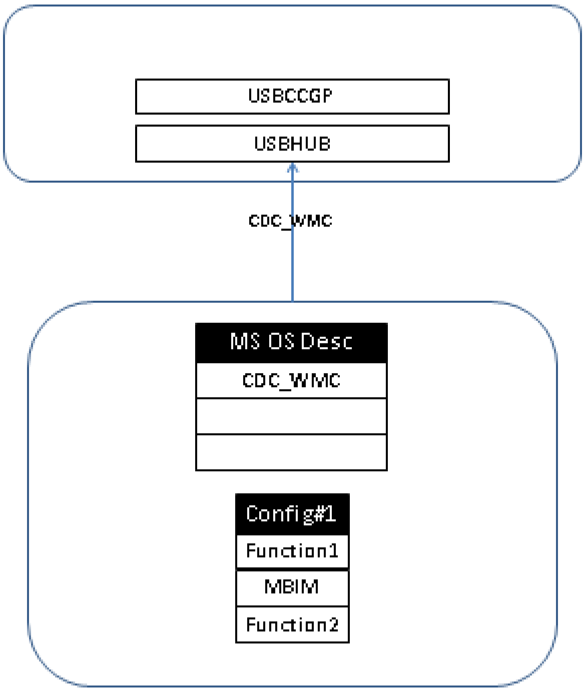
Union Function Descriptors

Mobile broadband devices that implement UFDs have Device Class / Subclass / Protocol of 2 / 0 / 0 as required for CDC devices. This prevents Windows from loading USBCCGP on the device. For information on how Windows loads USBCCGP on composite devices, see USB Generic Parent Driver (Usbccgp.sys).

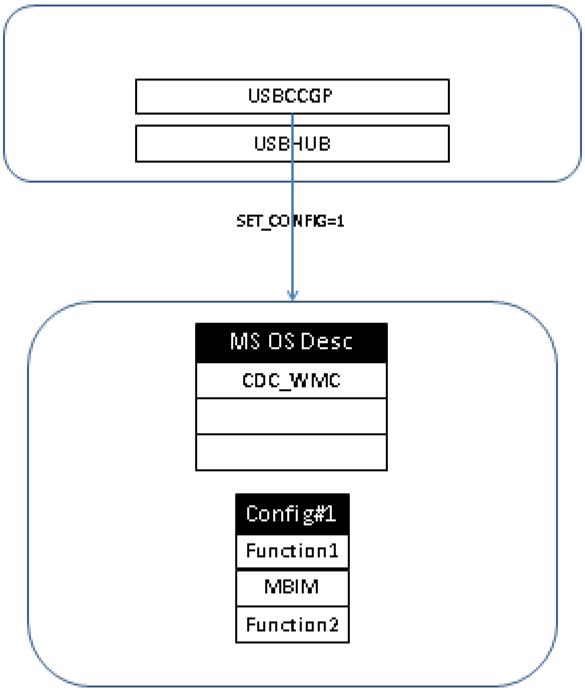
To allow Windows to load USBCCGP, the device needs to report a Microsoft OS compatible ID of "CDC_WMC" when the device is not configured. After detecting the compatible ID of "CDC_WMC", Windows loads USBCCGP, and USBCCGP sets the configuration on the device to 1. USBCCGP will then query again for the Microsoft OS compatible IDs. This time, however, the device should not report the Microsoft OS compatible ID of "CDC_WMC". The device may report Microsoft OS compatible IDs for functions in the selected configuration.



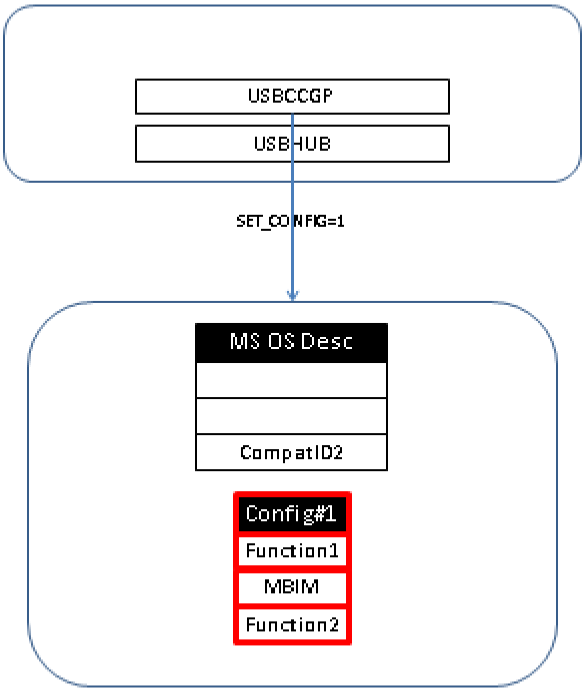
USBHUB queries for the Microsoft OS descriptor when the device is not configured



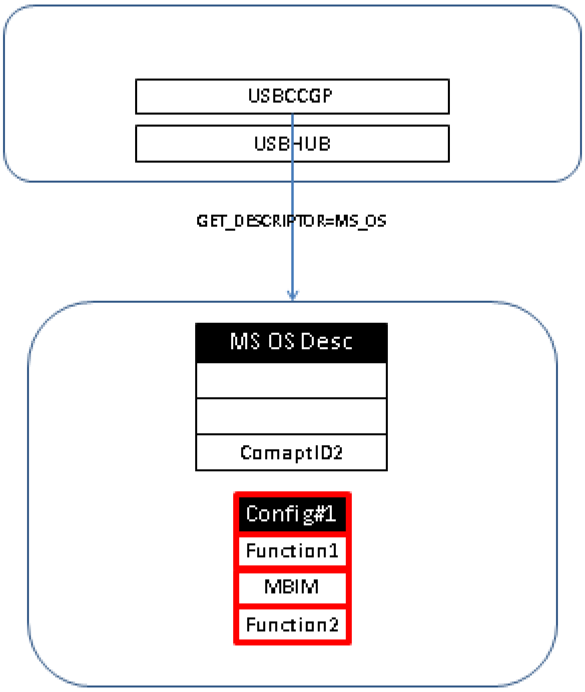
The device responds with "CDC_WMC", which causes Windows to load USBCCGP



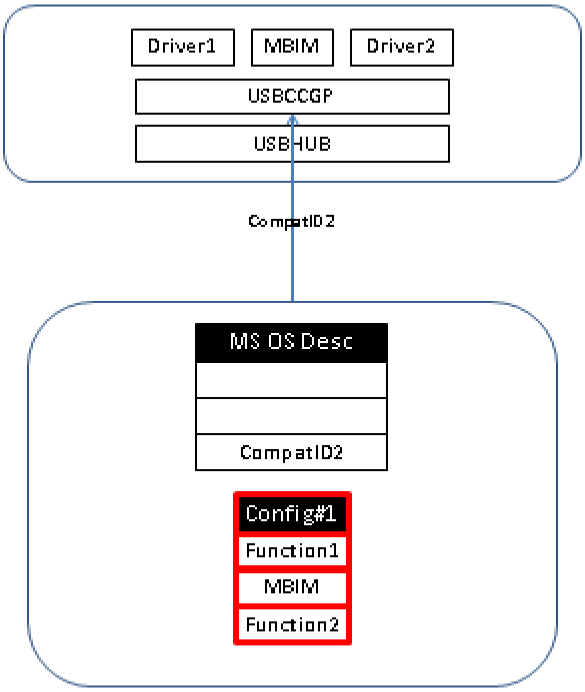
USBCCGP selects Configuration #1 on the device.



The device selects the configuration and morphs the list of compatible IDs. The device may include CompatID2, which is necessary for Function2.



After loading, USBCCGP queries for Microsoft OS compatible IDs again.



The device reports any compatible ID that it has for its function. USBCCGP then creates child device nodes for each function in the device.

MBIM Backward-Compatible Functions

MBIM functions that are backward compatible with the NCM 1.0 specification will come up as NCM 1.0 functions by default. Mobile broadband devices that consist of an MBIM backward-compatible function should report a Microsoft OS compatible ID of "MBIM" for the MBIM function. This allows Windows 8 to detect the NCM 1.0 function as the MBIM function and load MBCD as the function driver.

Introduction to MB Identity Morphing

Article • 03/14/2023

Identity Morphing

Mobile broadband USB dongle solutions eliminate the need for distributing the driver package for mobile broadband and other IHV functions through separate media (such as CD-ROM) by having a storage function in the USB device itself that contains the driver package.

Upon first time insertion of such a device in Windows, the device presents itself as mass storage, which results in the Windows AutoPlay dialog displayed to the user. At this point, the device exposes no other functions to the host except the mass storage function to prevent the other functions appearing to the user as non-functional due to missing driver software. The user can run the IHV-supplied software that installs the driver package. In addition to installing the driver package, the IHV-supplied software also morphs the device to expose the other functions to the user.

Mobile broadband devices that use the previously described mechanism when inserted in Windows 8 would come up as mass storage. Because Windows 8 has native support for mobile broadband functions that conform to the MBIM specification, installation of the driver package is not necessary for the user to use the mobile broadband function. The subtopics in this section provide guidance to IHVs on how to implement this solution for Windows 8 to allow the user to use the mobile broadband device without the need to install the driver package.

Mobile broadband devices that exhibit morphing behavior are referred to as morphing devices throughout the subtopics in this section.

MB Identity Morphing Solution Overview MB Identity Morphing Solution Details

MB Identity Morphing Solution Overview

Article • 03/14/2023

The solution maps the morphing device’s USB configuration to a set of USB functions. At any point in time, a single set of functions (by way of a configuration) are exposed to the host. The solution achieves morphing by switching between these configurations.

Logical configurations

The functions present in the device are grouped into the following logical sets.

Logical Set of Functions

Logical Set of Functions	Description
Windows-7-Configuration	Configuration selected by Windows 7 and older versions of Windows when the morphing device is inserted into the host for the first time.
Windows-8-Configuration	Configuration selected by Windows 8 when the morphing device is inserted into the host.
IHV-NCM-1.0-Configuration	Configuration selected by the IHV software installed on Windows 7 and older versions of Windows after the user installs the driver package.
IHV-NCM-2.0-Configuration	Configuration selected by the IHV software installed on Windows 8 after the user installs the driver package.

The following table shows the USB configurations listed in the previous table along with possible interfaces and functions. Additional requirements for each configuration are described in the remaining subtopics.

USB Configurations

Configuration 1 (Windows-7-Configuration)	Configuration 2(IHV–NCM-10-Configuration)	Configuration 3(Windows-8-Configuration)	Configuration 4(IHV–NCM-20-Configuration)

Configuration 1 (Windows-7-Configuration)	Configuration 2(IHV–NCM-10-Configuration)	Configuration 3(Windows-8-Configuration)	Configuration 4(IHV–NCM-20-Configuration)
Mass CD-ROM	Mass CD-ROM	Mass CD-ROM	Mass CD-ROM
Mass SD	Mass SD	Mass SD	Mass SD
	NCM1.0	MBIM	NCM2.0
	Modem		Modem
	TV		TV
	GPS		GPS
	FP		FP
	PC/SC smart card		PC/SC smart card
	Voice		Voice
	Diag		Diag

Goals of the solution

- In Windows 7, users need to perform the extra step of installing driver packages before being able to use the mobile broadband function on morphing devices.
- In Windows 8, users should not have to perform extra steps for installing driver packages to use the mobile broadband function on morphing devices that conform to the MBIM specification.
- In Windows 8, users need to perform the extra step of installing driver packages before being able to use IHV functions on morphing devices that do not have inbox drivers.

Assumptions

MBIM also includes backward compatibility for NCM 1.0.

Supported Transitions

For Windows 8

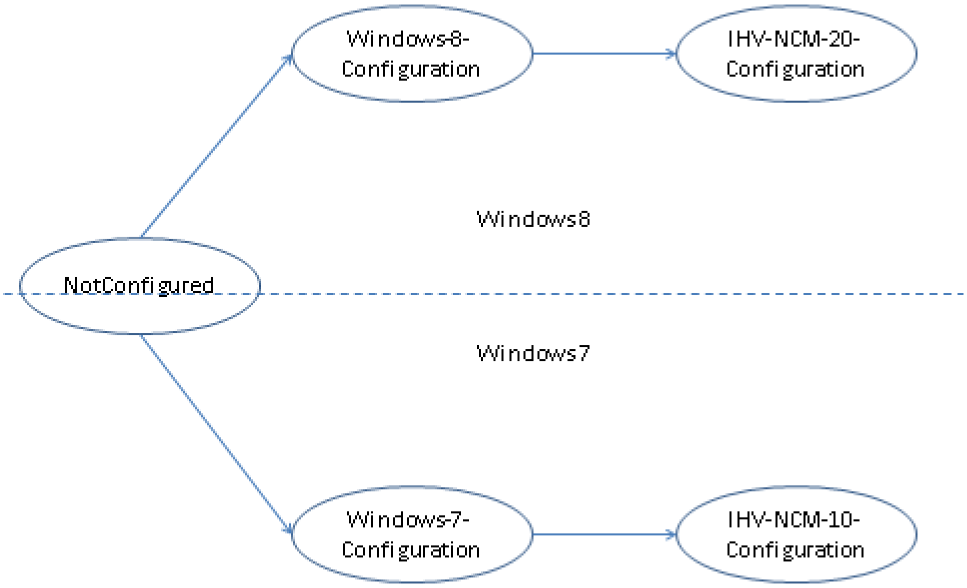
Not-Configured -> Windows-8-Configuration

Windows-8-Configuration -> IHV-NCM-2.0-Configuration

For Windows 7

Not-Configured -> Windows-7-Configuration

Windows-7-Configuration -> IHV–NCM-1.0-Configuration

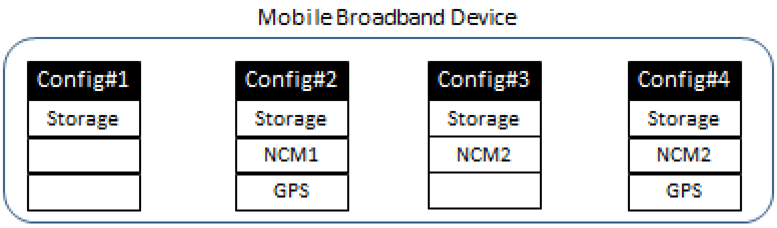


The configuration transition paths for Windows 7 and Windows 8

Note that any transition not shown previously is not supported.

Transition Details

Consider a sample USB morphing device with the following functions in its configurations.


Mobile Broadband Device

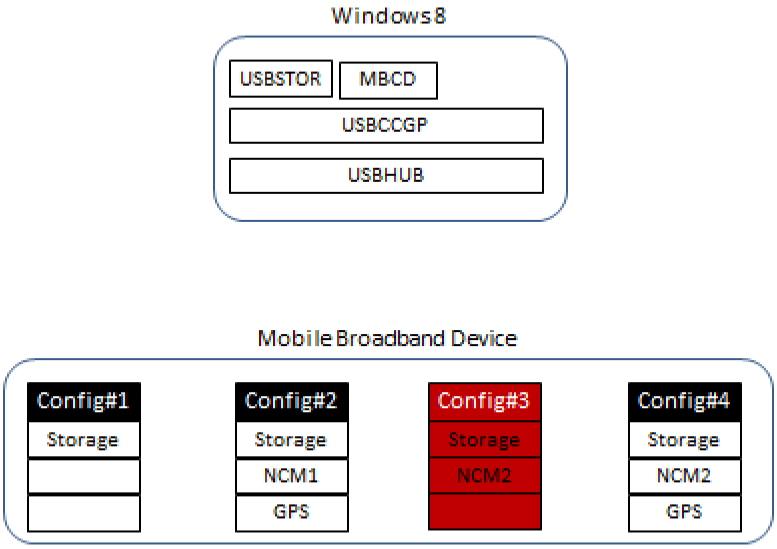
Config#1	Config#2	Config#3	Config#4
Storage	Storage	Storage	Storage
	NCM1	NCM2	NCM2
	GPS		GPS

USB device with multiple functions

Windows 8

Windows-8-Configuration

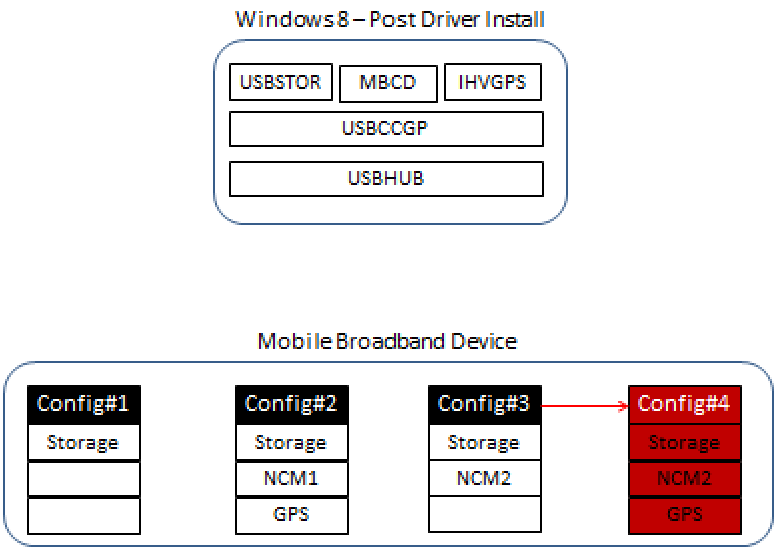
When the morphing device is plugged into a computer running Windows 8, the Windows-8-Configuration would be selected, which exposes the MBIM function. The Windows 8 Mobile Broadband Class Driver (MBCD) will be loaded on the MBIM function. In the following example, Configuration 3 is the Windows-8-Configuration containing the MBIM function.



Driver stack and device configuration on Windows 8 after device is plugged in

IHV-NCM-2.0-Configuration

In the Windows-8-Configuration, the morphing device also has a mass storage function that will allow the user to install the IHV driver package. After installation of the driver package from the mass storage function, the device will morph to expose the functions in the IHV-NCM-2.0-Configuration. This configuration has an additional IHV function such as GPS, diagnostics, and so on. Configuration 4 in the following diagram represents the IHV-NCM-2.0-Configuration.



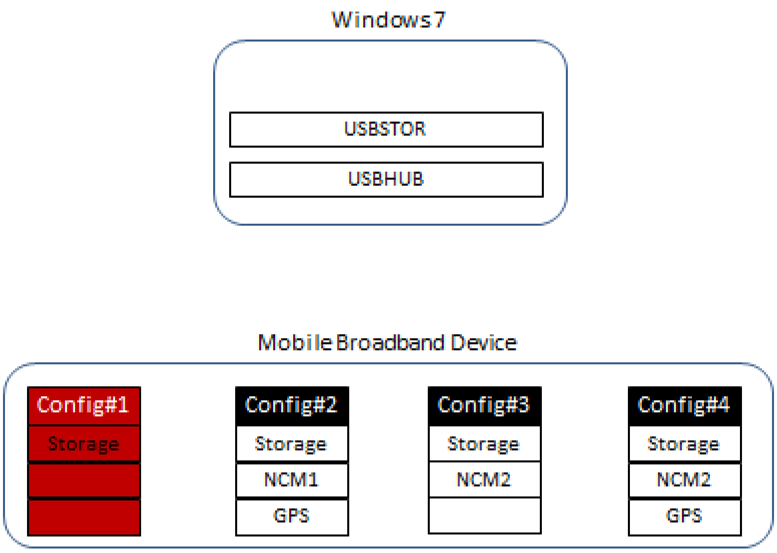
Driver stack and device configuration on Windows 8 after user installs IHV driver package

Windows 7

Windows-7-Configuration

When the morphing device is plugged into a computer running Windows 7 or an earlier version of Windows, the Windows-7-Configuration would be selected, which exposes the mass storage function. This will allow the user to install the IHV driver package from the mass storage function.

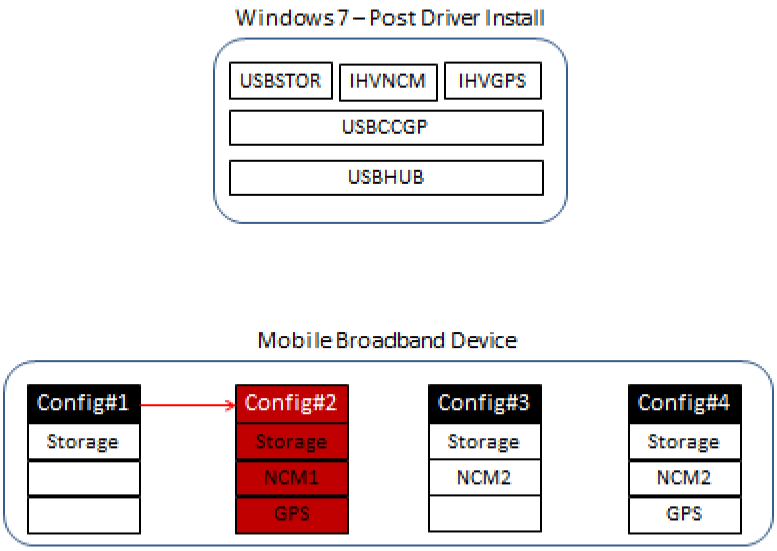
In the following example, Configuration 1 is the Windows-7-Configuration



Driver stack and device configuration on Windows 7 when the user has not installed the IHV driver package

IHV-NCM-1.0-Configuration

In Windows 7, the user can install the driver package from the mass storage function. Along with installing the driver software, the IHV software will also morph the device from the Windows-7-Configuration to the IHV-NCM-1.0-Configuration.



Driver stack and device configuration in Windows 7 after user installs IHV driver package

MB Identity Morphing Solution Details

Article • 03/14/2023

Configuration requirements

The order of the functions across transitions in Windows 8 needs to be maintained. For example, if MBIM is the third function in the Windows-8-Configuration, it should also be the third function in the IHV-NCM-2.0-Configuration.

Windows-7-Configuration

The Windows-7-Configuration should be the first configuration in the morphing device. This configuration should have the mass storage function as one of the functions. Windows 8 will not select this configuration. In Windows 7 and earlier versions of Windows, the Windows-7-Configuration is the default configuration selected. This configuration is used to expose a USB mass storage function where IHVs put their driver package, which allows users to install the IHV’s driver.

Windows-8-Configuration

The Windows-7-Configuration exposes the MBIM function as one of the functions on which MBCD is loaded. In Windows 8, the value of this configuration is used in the subCompatibleID value returned to USBCCGP. USBCCGP selects this configuration when it is loaded. The Windows-8-Configuration should be either Configuration 2, 3, or 4. No other configuration is supported as the Windows-8-Configuration. This configuration also exposes the mass storage function as the first function to allow a user to install the IHV’s driver package.

IHV-NCM-2.0-Configuration

The IHV-NCM-2.0-Configuration exposes IHV-specific functions along with MBIM and mass storage functions. This configuration is not set or used by Windows. The IHV software, after installation by the user, can morph to this configuration. Note that the order of the functions in this configuration should be the same as in the Windows-8-Configuration. Although extra functions can be added to the Windows-8-Configuration, the existing functions should be retained in the same order.

IHV-NCM-1.0-Configuration

The IHV-NCM-1.0-Configuration exposes IHV-specific functions along with NCM 1.0 and mass storage functions. This configuration is not set or used by Windows 8. This configuration is used only in Windows 7 and earlier versions of Windows after the IHV software is installed by the user. The IHV software morphs the morphing device from the Windows-7-Configuration to this configuration.

Compatible IDs

Compatible IDs are 8-character or smaller strings used by the device to indicate the driver loading preference to Windows. Devices can define compatible IDs by using Microsoft OS descriptors. Compatible and subcompatible IDs apply to individual functions. Each configuration can have a separate set of compatible IDs, which map to the set of functions within that configuration. Although compatible and subcompatible IDs apply to individual functions, the morphing device can have a single compatible ID when no configuration is selected. This compatible and subcompatible ID logically applies to the whole morphing device.

Loading USBCCGP

In Windows 8, a USBCCGP driver is required to automatically select the Windows-8-Configuration on the morphing device.

To load the USBCCGP driver, the morphing device needs to report the following compatible and subcompatible IDs when no configuration is selected on the morphing device:

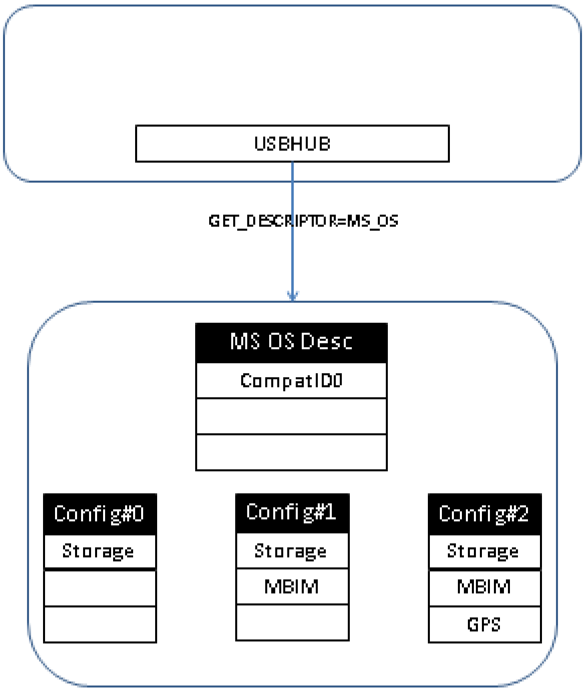
- If the morphing device uses IADs for grouping interfaces into functions, the compatible ID should be reported as "ALTRCFG" and the subcompatible ID as the number of the Windows-8-Configuration.
- If the morphing device uses WCM UFDs for grouping interfaces into functions, the compatible ID should be reported as "WMCALTR" and the subcompatible ID as the number of the Windows-8-Configuration.

For example, if the Windows-8-Configuration is Configuration 3, the subcompatible ID would be "3" in both of these cases.

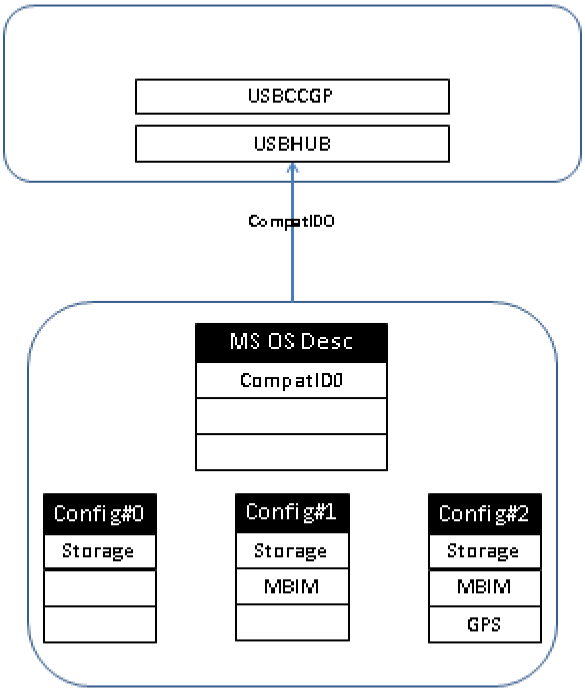
Morphing compatible IDs

During USB device enumeration, USBHUB queries the morphing device for the compatible ID when no configuration is selected on the morphing device. The morphing device should return the compatible and subcompatible ID used to load USBCCGP, as described in MB Identity Morphing Solution Overview.

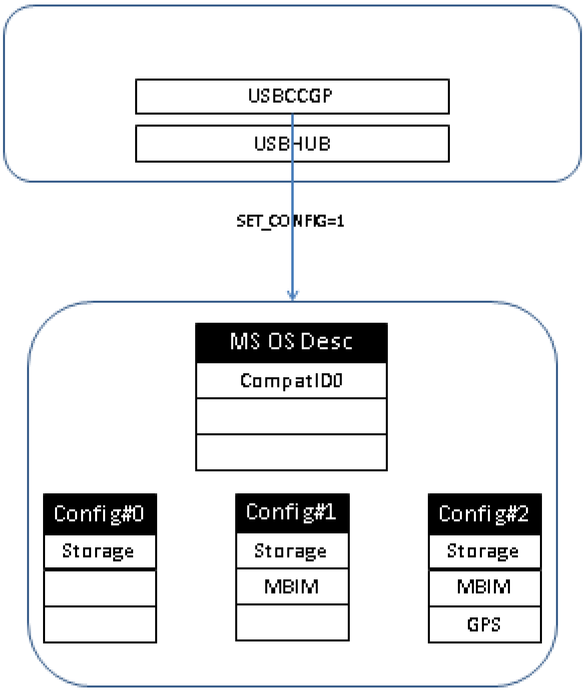
After USBHUB loads USBCCGP, USBCCGP selects the configuration indicated by the subcompatible ID reported earlier. USBCCGP then queries the compatible and subcompatible ID a second time. At this point, the morphing device should return the compatible and subcompatible IDs for the configuration that is currently selected. Therefore, after USBCCGP loads and selects a particular configuration, the morphing device needs to morph the compatible and subcompatible IDs that are reported. The morphing device must not report the compatible and subcompatible IDs that are used to load USBCCGP after a configuration is selected.



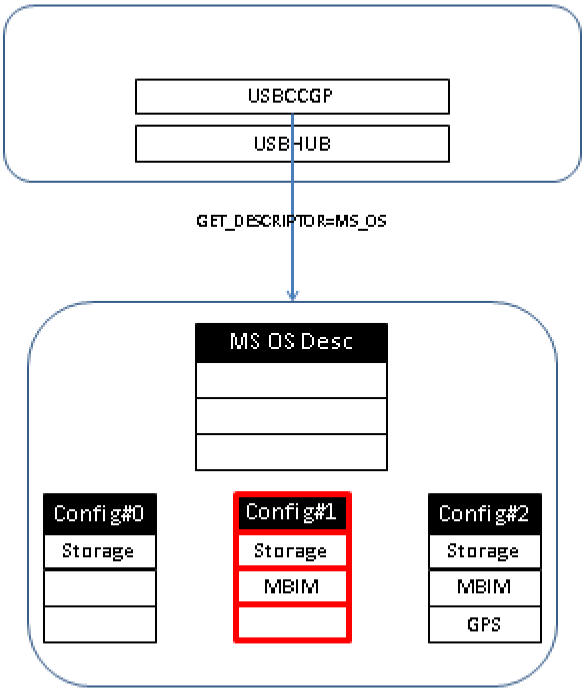
USBHUB querying the Microsoft OS descriptor from the device during enumeration.



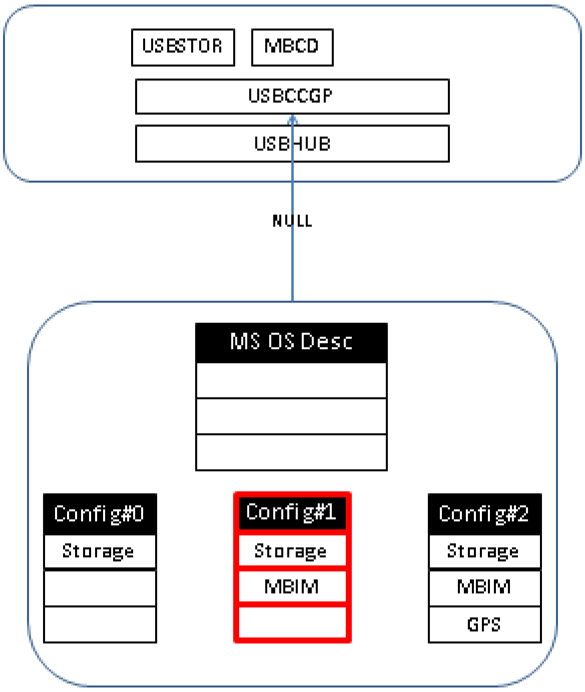
Device returns CompatId in the not-configured state. This CompatId is used to load USBCCGP.



USBCCGP selects the configuration reported in the subcompatible ID.



Device morphs its Microsoft OS descriptor based on the new configuration. USBCCGP queries for the Microsoft OS descriptor.



Device does not return any CompatID. Based on the Class / Subclass / Protocol, USBCCGP loads USBSTOR and MBCD.

MB Interface Model Supplement

Article • 03/14/2023

The Microsoft OS descriptor is broken up into the following segments:

- One Microsoft OS string descriptor
- One or more Microsoft OS feature descriptors

To support the OS descriptor, the device must implement the string descriptor. **String Descriptor**

The Microsoft OS string descriptor is a string that is stored at string index 0xEE. The format of this string is well defined.

The Microsoft OS String Descriptor is used to achieve the following objectives

- The presence of the Microsoft OS string descriptor indicates to the operating system that the device has information embedded in it, in the form of Microsoft OS feature descriptors.
- The Microsoft OS string descriptor has an embedded signature field that is used to differentiate it from random strings that might happen to be on a device at string index 0xEE.
- The Microsoft OS string descriptor also has an embedded version number that allows for future revisions of the Microsoft OS descriptor.

Only one Microsoft OS string descriptor is stored on a device. The following sections describe the structure of the Microsoft OS string descriptor and its retrieval procedure. **Structure of the OS string**

Here is the structure of the string descriptor:

String Descriptor Structure

Field	Length (Bytes)	Value	Description
bLength	1	0x12	Length of the descriptor
bDescriptorType	1	0x03	String descriptor
qwSignature	14	"MSFT100"	Signature field (4D00530046005400310030003000)
bMS_VendorCode	1	Vendor Code	Vendor code to fetch other OS feature descriptors

Field	Length (Bytes)	Value	Description
bPad	1	0x00	Pad field

The structure of the Microsoft OS string descriptor is fixed for version 1.00 and has an overall length of 18 bytes. The version number of the Microsoft OS string descriptor is listed in the **qwSignature** field. The information stored in the **bMS_VendorCode** field must be a single byte value. It will be used to retrieve Microsoft OS feature descriptors, and this byte value is used in the **bmRequestType** field described as follows:

Retrieving the OS string descriptor

To retrieve the information stored in the string, a standard GET_DESCRIPTOR request must be issued to the device. Here is the format of the request:

Standard Get_Descriptor String Request

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0000b	GET_DESCRIPTOR	0x03EE	0x0000	0x12	Returns the string

The **bmRequestType** field is a bitmap composed of three parts—direction of data transfer, descriptor type, and recipient. According to the USB specification, the value of **bmRequestType** is set to 1000 0000b (0x80).

For a GET_DESCRIPTOR request, the **wValue** field is split into two parts. The high byte stores the descriptor type and the low byte stores the descriptor index. To retrieve the Microsoft OS string descriptor, the high byte should be set to retrieve a string descriptor—0x03. Because the Microsoft OS string descriptor is always stored at index 0xEE, this string index should be stored in the lower byte of the **wValue** field.

The **wIndex** is used to store the language ID, but it must be set to zero for a Microsoft OS string descriptor.

The **wLength** field is used to indicate the length of the string descriptor to be retrieved. The device should respond to any valid range from 0x02–0xFF.

If a device does not have a valid descriptor at the corresponding address (0xEE), it will respond with a request error or stall. When devices do not respond with a stall, a single-ended zero reset will be issued to the device (to recover it, if it should go into an unknown state).

Verifying the integrity of the OS descriptor

Because vendors are allowed to use any string ID to store information, the operating system must verify that the string stored in index 0xEE is indeed the Microsoft OS string descriptor. To verify this, the following tests will be conducted. Failure of either will inhibit retrieval of the Microsoft OS feature descriptors.

- If a vendor stores a string in index location 0xEE, the operating system will retrieve the string and query it to see if it is the Microsoft OS string. This can be verified by comparing the signature field in the string to the signature field entry specified previously. A mismatch would prevent further parsing of the string.
- The second test will include a verification of the length of the string based on the version number specified in the signature field. The version number specified (in the string "MSFT100") is 1.00. This corresponds to an 18-byte string descriptor.

Microsoft OS string descriptor constraints

The following constraints apply to Microsoft OS string descriptors and their retrieval:

- To store information in compliance with the Microsoft OS descriptor specification, the device must have one and only Microsoft OS string descriptor that is in compliance with the information outlined in Microsoft OS Descriptors.
- A device vendor is free to use any value in the **bMS_VendorCode** field in the Microsoft OS string descriptor

Feature Descriptor

A feature descriptor is a fixed-format descriptor that has been defined for a specific purpose.

Retrieving an OS feature descriptor

To retrieve a Microsoft OS feature descriptor, a special GET_MS_DESCRIPTOR request needs to be issued to the device. Here is the format of the request:

Standard device request format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1100 0000b	GET_MS_DESCRIPTOR	X	Feature Index	Length	Returns descriptor

The **bmRequestType** field is a bitmap composed of three parts—direction of data transfer, descriptor type, and recipient—and is in accordance with the USB specification. The Microsoft OS feature descriptor is a vendor-specific descriptor and the direction of data transfer is from the device to the host. Therefore, the value of **bmRequestType** is set to 1100 0000b (0xC0).

The **bRequest** field is used to indicate the format of the request. To retrieve a Microsoft OS feature descriptor, the **bRequest** field should be populated with a special GET_MS_DESCRIPTOR byte. The value of this byte is indicated by the **bMS_VendorCode**, which is retrieved from the Microsoft string descriptor. For more information about the retrieval of the Microsoft OS string descriptor, see **Retrieving the OS string descriptor**.

The **wValue** field is put to special use and is broken up into a high byte and a low byte. The high byte is used to store the interface number. This is essential for storing feature descriptors on a per interface basis, especially for composite devices, or devices with multiple interfaces. In most cases, interface 0 will be used. The low byte is used to store a page number. This feature prevents descriptors from having a size boundary of 64 KB (a limit set by the size of the **wLength** field). A descriptor will be fetched with the page value initially set to zero. If a full descriptor (size is 64 KB) is received, the page value will be incremented by one and the request for the descriptor will be sent again (this time with the incremented page value). This process will repeat until a descriptor with a size less than 64 KB is received. Note that the maximum number of pages is 255, which places a limit of 16 MB on the descriptor size.

The **wIndex** field stores the feature index number for the Microsoft OS feature descriptor being retrieved. Microsoft will maintain this list of Microsoft OS feature descriptors and indexes. To learn more about Microsoft OS feature descriptors, see Microsoft OS Descriptors.

The **wLength** field specifies the length of the descriptor to be fetched. If the descriptor is longer than the number of bytes stated in the **wLength** field, only the initial bytes of the descriptor are returned. If it is shorter than the value specified in the **wLength** field, a short packet is returned.

If a particular OS descriptor is not present, the device will issue a request error or stall.

Microsoft OS feature descriptor constraints

The following constraints apply to Microsoft OS feature descriptors and their retrieval.

- All Microsoft OS feature descriptors are defined and standardized. Vendors are not allowed to modify, append, or create Microsoft OS feature descriptors without direct consent from Microsoft.
- All Microsoft OS feature descriptors will have a size and version number embedded in them. These values should always report correct information to the operating system.
- A device can have more than one Microsoft OS feature descriptor embedded in its firmware.

- Some Microsoft OS feature descriptors are stored on a per-interface level, while others are unique to the device. Device-level Microsoft OS feature descriptors should set the high byte of the wValue field as zero.

Structure of the feature descriptor

To identify itself as capable of supporting MBIM, a device must also support the extended configuration descriptor, which is one of the defined feature descriptors. The structure of this descriptor is as follows.

Header section

The header section stores information about the rest of the extended configuration descriptor. The **dwLength** field contains the length of the entire extended configuration descriptor. The header section also contains a version number, which will be initially set to 1.00 (0100H). Future revisions of this descriptor may be released at a later stage. Note that future versions of the extended configuration descriptor might also need to increase the number of entries in the header section, so please verify that this number is accurately stored in the device and read by the operating system.

Extended configuration descriptor header section

Offset	Field	Size	Value	Description
0	dwLength	4	Unsigned DWORD	The length field describes the length of the extended configuration descriptor, in bytes.
4	bcdVersion	2	BCD	Extended configuration descriptor release number in Binary Coded Decimal (for example, version 1.00 is 0100H).
6	wIndex	2	WORD	Fixed = 0x0004

Offset	Field	Size	Value	Description
8	bCount	1	BYTE	Total number of function sections that follow the header section = 0x01
9	RESERVED	7		RESERVED

Function section

The function section provides two important pieces of information. It groups consecutive interfaces that serve a similar purpose into function groups, and provides compatible and subcompatible IDs for each function.

Here is the format of the function section, including values that should be used by an MBIM device:

Extended configuration descriptor function section

Offset[1]	Field	Size	Value	Description
0	bFirstInterfaceNumber	1	Byte	Starting interface number for this function = 0x00
1	bInterfaceCount	1	Byte	Total number of Interfaces that must be included to from this function = 0x01
2	compatibleID	8	Bytes	Compatible ID
10	subCompatibleID	8	Bytes	Subcompatible ID
18	RESERVED	6		RESERVED = 0

[1]Offset of the custom property section has been reset to zero. To calculate the offset of a field from the beginning of the extended configuration descriptor, add the length of the sections that precede it.

Compatible and Subcompatible IDs based on the configuration exposing the MBIM function

bConfiguration	compatibleID	subCompatibleID

bConfiguration	compatibleID	subCompatibleID
2	ALTRCFG (0x41 0x4C 0x54 0x52 0x43 0x46 0x47 0x00)	20000000 (0x32 0x00 0x00 0x00 0x00 0x00 0x00 0x00)
3	ALTRCFG (0x41 0x4C 0x54 0x52 0x43 0x46 0x47 0x00)	30000000 (0x33 0x00 0x00 0x00 0x00 0x00 0x00 0x00)
4	ALTRCFG (0x41 0x4C 0x54 0x52 0x43 0x46 0x47 0x00)	40000000 (0x34 0x00 0x00 0x00 0x00 0x00 0x00 0x00)

- **bConfiguration** refers to the **bConfiguration** value within the USB configuration descriptor of the configuration that exposes the MBIM function. **bConfiguration** cannot be 1 because that is the default configuration exposing only the CDROM function. **bConfiguration** cannot be greater than 4; that is, the MBIM function should be exposed within the first four configurations.
- compatibleID remains the same for all configurations. The subcompatibleID changes based on the configuration

Example

This table shows a sample multi-configuration scenario. The table lists the functions available in each configuration and the actions that different versions of the operating system takes for each of these configurations:

Example of a multi-configuration mobile broadband device

bConfiguration	1 (Windows-7-Configuration)	2 (IHV-NCM-1.0-Configuration)	3 (Windows-8-Configuration)	3 (IHV-NCM-2.0-Configuration)

bConfiguration	1 (Windows-7-Configuration)	2 (IHV-NCM-1.0-Configuration)	3 (Windows-8-Configuration)	3 (IHV-NCM-2.0-Configuration)
Functions exposed	CDROM SD	CD-ROM SD NCM1.0 Modem TV GPS FP PC/SC smart card Voice Diag	CD-ROM SD MBIM	CD-ROM SD NCM2.0 Modem TV GPS FP PC/SC smart card Voice Diag

The following tables show the values used by the Microsoft OS string descriptor and the Microsoft OS extended configuration feature descriptor for the previous sample’s multi-configuration scenario.

Example of a multi-configuration mobile broadband device

Field	Length (Bytes)	Value
bLength	1	0x12
bDescriptorType	1	0x03
qwSignature	14	‘MSFT100’ 0x4D 0x00 0x53 0x00 0x46 0x00 0x54 0x00 0x31 0x00 0x30 0x00 0x30 0x00
bMS_VendorCode	1	0xA5
bPad	1	0x00

Example Microsoft OS extended configuration feature descriptor header

Offset	Field	Size	Value
0	dwLength	4	16

Offset	Field	Size	Value
4	bcdVersion	2	0100H
6	wIndex	2	0x0004
8	bCount	1	1
9	RESERVED	7	

Example Microsoft OS extended configuration feature descriptor function

Offset[2]	Field	Size	Value
0	bFirstInterfaceNumber	1	
1	bInterfaceCount	1	
2	compatibleID	8	
10	subCompatibleID	8	
18	RESERVED	6	

[2] Offset of the custom property section has been reset to zero. To calculate the offset of a field from the beginning of the extended configuration descriptor, add the length of the sections that precede it.

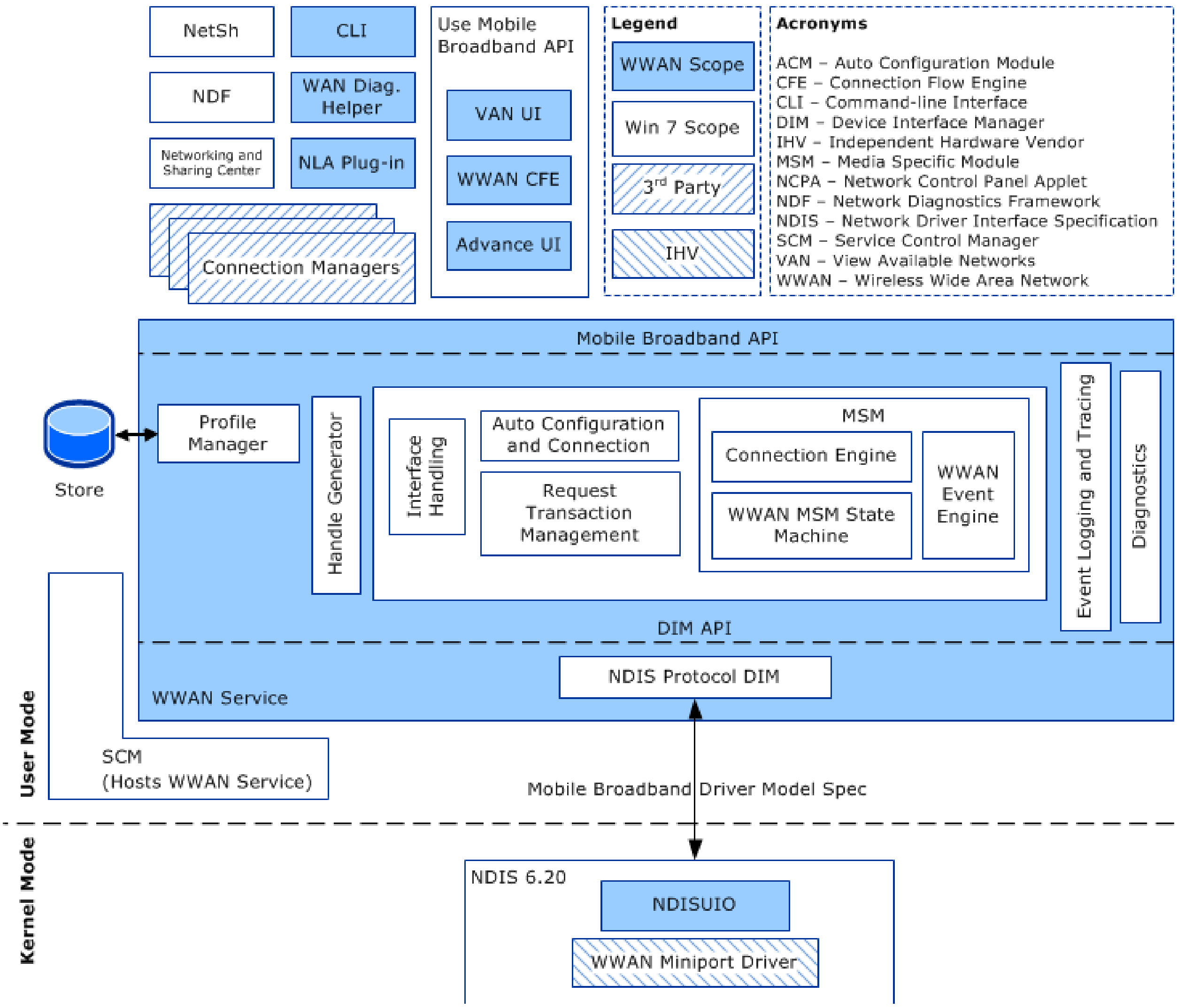
MB Interface Overview

Article • 03/14/2023

This documentation describes the Mobile Broadband (MB) driver model. The MB driver model is a software architecture provided with Windows 7 and later versions of Windows. It provides a framework for an integrated set of networking features based on CDMA-based (1xRTT/1xEV-DO/1xEV-DO RevA/1xEvDO RevB) and GSM-based (GPRS/EDGE/UMTS/HSDPA/TD-SCDMA) cellular technologies.

Windows 8 MB miniport drivers are based on the NDIS 6.30 interface. Windows 7 MB miniport drivers are based on the NDIS 6.20 interface. MB miniport drivers must follow the appropriate "NDIS 6.x Specification" in addition to the device driver interfaces (DDIs) described throughout this documentation.

The following diagram represents the architecture of the MB driver model.


NetSh
CLI
NDF
WAN Diag. Helper
Networking and Sharing Center
NLA Plug-in
Connection Managers
Use Mobile Broadband API
VAN UI
WWAN CFE
Advance UI
Legend
WWAN Scope
Win 7 Scope
3rd Party
IHV
Acronyms
ACM – Auto Configuration Module
CFE – Connection Flow Engine
CLI – Command-line Interface
DIM – Device Interface Manager
IHV – Independent Hardware Vendor
MSM – Media Specific Module
NCPA – Network Control Panel Applet
NDF – Network Diagnostics Framework
NDIS – Network Driver Interface Specification
SCM – Service Control Manager
VAN – View Available Networks
WWAN – Wireless Wide Area Network
Mobile Broadband API
Store
Profile Manager
Handle Generator
Interface Handling
Auto Configuration and Connection
Request Transaction Management
MSM
Connection Engine
WWAN MSM State Machine
WWAN Event Engine
Event Logging and Tracing
Diagnostics
DIM API
NDIS Protocol DIM
WWAN Service
SCM (Hosts WWAN Service)
Mobile Broadband Driver Model Spec
User Mode
Kernel Mode
NDIS 6.20
NDISUIO
WWAN Miniport Driver

Terminology

Be aware that the terms *Wireless Wide Area Network* (WWAN) and *Mobile Broadband* (MB) are used interchangeably throughout this documentation to represent the Mobile Broadband technology.

The terms *activate* and *activation* have two different meanings in this documentation. The term *activate* is related to service activation, such as when a network provider must explicitly enable the MB subscription before the provider's services can be used. The term *activation* is specific to setting up a connection in GSM-based (GPRS/EDGE/UMTS/HSDPA/TD-SCDMA) technologies. For example, *PDP context activation* refers to setting up an MB connection according to the parameters specified in the PDP context.

SIM access refers to accessing the Subscriber Identity Module (SIM, also known as the R-UIM). If the MB device does not have a SIM/R-UIM as a physical entity but instead has a logical equivalent embedded in the device, this term is applicable to that logical circuit equivalent as well. When SIM access is not required, the miniport driver is not expected to retrieve the information from the SIM in order to complete the request.

Semantics

The MB Service component in user mode cannot directly exchange data with MB miniport drivers in kernel mode. Intermediaries such as WMI or NDIS filter drivers are required. For simplicity, these intermediaries are not explicitly discussed in this documentation. However, this omission does not mean that the MB Service and MB miniport drivers can engage in direct data exchanges.

The following topics provide a summary of NDIS 6.20 and MB OID semantics, the procedures that miniport drivers must follow to perform synchronous and asynchronous operations, and an overview of the operations supported by the Mobile Broadband driver model:

MB / NDIS 6.20 Interfacing Overview

MB Data Model

MB Operational Semantics

MB Driver Model Versioning

MB Miniport Driver INF Requirements

MB Miniport Driver Types

MB Adapter General Attribute Requirements

MB Raw IP Packet Processing Support

Guidelines for MB Miniport Driver IP Address Notifications

MB Miniport Driver Error Logging

MB Miniport Driver Performance Requirements

MB / NDIS 6.20 Interfacing Overview

Article • 03/14/2023

This topic is designed to provide enough background about the *NDIS 6.20 Specification* to put the MB driver model into perspective. It is not intended to be a reference for NDIS 6.20. In the case of discrepancies between this content and the *NDIS 6.20 Specification*, see the NDIS 6.20 documentation for complete information.

In NDIS 6.20, the MB Service calls **NdisOidRequest** to issue OID requests to the miniport driver. Then, miniport drivers call **NdisMIndicateStatusEx** to return data back to the MB Service.

NDIS 6.20 supports the following types of OID operations:

- *Set* operations that send data from the service to a miniport driver.
- *Query* operations that request miniport drivers to return data to the service.
- *Method* operations, equivalent to a function call, that have both input parameters and output parameters.

Finally, miniport drivers may send *indications* that contain data to notify the service about state changes in the MB device.

Receiving *Set* and *Query* Requests

MB miniport drivers implement the *MiniportOidRequest* NDIS handler to respond to both *set* and *query* requests.

Sending Status Indications

Miniport drivers provide status indications to the MB Service by calling **NdisMIndicateStatusEx**. See the **NDIS_STATUS_INDICATION** structure for more details about status indications.

Connection State Indications

NDIS 6.20 miniport drivers must use the **NDIS_STATUS_LINK_STATE** status indication to notify NDIS and overlying drivers that there has been a change in the physical characteristics of a transmission medium.

The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure is an NDIS_LINK_STATE structure, which specifies the physical state of the transmission medium.

MB miniport drivers should avoid sending the NDIS_STATUS_LINK_STATUS status indication if there have been no changes in the physical state of the medium. However, miniport drivers are not necessarily required to avoid sending this status indication.

MB miniport drivers must report the maximum data rate of the currently connected data-class. A change in data-class while connected must result in a Connection State Indication with the corresponding data rate reported. The following is a recommended implementation of this rule:

1. MB miniport drivers that conform to this specification must use NDIS_STATUS_LINK_STATE to indicate connection status changes instead of NDIS_STATUS_MEDIA_CONNECT, NDIS_STATUS_MEDIA_DISCONNECT, or NDIS_STATUS_LINK_SPEED_CHANGE (as in NDIS 5.1) for connection status indications.

2. The **XmitLinkSpeed** and **RcvLinkSpeed** members of the NDIS_LINK_STATE structure must not report NDIS_LINK_SPEED_UNKNOWN. Miniport drivers must report the speed by using the information in the following tables.

For GSM-based MB device speed links

Data class	XmitLinkSpeed	RcvLinkSpeed
GPRS	8 to 48 kbps	8 to 48 kbps
EDGE	8 to 220 kbps	8 to 220 kbps
UMTS	64 to 384 kbps	64 to 384 kbps
HSDPA	64 to 5.76 mbps	1.8 to 14.4 mbps
HSUPA	1.4 to 5.76 mbps	64 kbps to 7.2 mbps

For CDMA-based MB device speed links

Data Class	XmitLinkSpeed	RcvLinkSpeed
1xRTT	115.2 kbps to 307.2 kbps	153.6 kbps to 3 mbps
3xRTT	614 kbps to 1.04 mbps	307.2 kbps to 1.04 mbps
1xEV-DO	153.6 kbps	2.4 mbps

Data Class	XmitLinkSpeed	RcvLinkSpeed
1xEvDO Rev. A.	1.8 mbps	3.1 mbps
1xEV-DV	1.8 mbps	3.1 mbps
1xEvDO Rev. B.	27 mbps	3.1 mbps to 73.5 mbps

Note MB devices should report the speed in the range of speed shown in the previous tables.

Unlike NDIS 5.1, different link state change indications are consolidated into a single NDIS_STATUS_LINK_STATE indication by using the NDIS_LINK_STATE data structure. NDIS 5.1 indications can be mapped to this structure according to the information in the following table. In the case of link speed change, the consumer of the indication should compare the transmitting and receiving speed values with the ones it recorded for a previous indication to decide whether the link speed change has occurred or not.

Connection status indication mapping from NDIS 5.1 to 6.x

NDIS 5.1 indication NDIS 6.x NDIS_LINK_STATE data structure Parameter Value NDIS_STATUS_MEDIA_CONNECT

MediaConnectState

MediaConnectStateConnected

NDIS_STATUS_MEDIA_DISCONNECT

MediaConnectState

MediaConnectStateDisconnected

NDIS_STATUS_LINK_SPEED_CHANGE

XmitLinkSpeed

Transmitting speed (bps)

RcvLinkSpeed

Receiving speed (bps)

MB Data Model

Article • 03/14/2023

The MB driver model uses a data model that consists of a set of objects defined as abstractions of MB device features. Each object is identified by a unique object identifier (OID) and is defined by a set of corresponding attributes. The set of attributes is organized into a data structure. To manage the device, the MB Service and the MB miniport driver exchange OIDs and their associated data structures based on OID requests and indications provided by the Network Driver Interface Specification (NDIS).

In the MB driver model, only *set* and *query* operations are used for OID requests. The MB driver model does not use *method* operations. For indications, the MB driver model uses both event and transactional notifications to indicate state changes in the objects of the MB device. Transactional notifications also signal completion of an asynchronous transaction.

The following tables list the OIDs and status indications defined for MB miniport drivers, as well as the associated data structures. MB miniport drivers must implement all mandatory general OIDs that the NDIS 6.20 Specification requires. For a list of general OIDs for NDIS 6.x, see General Operational OIDs.

In addition, MB miniport drivers must implement OID_GEN_PHYSICAL_MEDIUM even though the NDIS Specification describes it as optional to implement.

The syntax and semantics of the MB OIDs listed in the following table are described in MB Operational Semantics.

WWAN-Specific OIDs

OID and Corresponding Data Structure	Set, Windows 7	Set, Windows 8	Query, Windows 7	Query, Windows 8	GSM/CDMA
OID_WWAN_DRIVER_CAPS uses **NDIS_WWAN_DRIVER_CAPS**	Not supported	Not supported	S	S	GSM, CDMA
OID_WWAN_DEVICE_CAPS has no corresponding structure	Not supported	Not supported	A	A	GSM, CDMA
OID_WWAN_READY_INFO has no corresponding structure	Not supported Not supported	A	A	GSM, CDMA	
OID_WWAN_SERVICE_ACTIVATION† uses **NDIS_WWAN_SERVICE_ACTIVATION**	A	A	Not supported	Not supported	GSM, CDMA
OID_WWAN_RADIO_STATE uses **NDIS_WWAN_SET_RADIO_STATE**	A	A	A	A	GSM, CDMA
OID_WWAN_PIN uses **NDIS_WWAN_SET_PIN**	A	Not supported	A	Not supported	GSM, CDMA
OID_WWAN_PIN_LIST has no corresponding structure	Not supported	Not supported	A	A	GSM, CDMA
OID_WWAN_PIN_EX uses **NDIS_WWAN_SET_PIN_EX**	Not supported	A	Not supported	A	GSM, CDMA
OID_WWAN_HOME_PROVIDER has no corresponding structure	Not supported	Not supported	A	A	GSM, CDMA

OID and Corresponding Data Structure	Set, Windows 7	Set, Windows 8	Query, Windows 7	Query, Windows 8	GSM/CDMA
OID_WWAN_PREFERRED_PROVIDERS† uses **NDIS_WWAN_SET_PREFERRED_PROVIDERS**	A	A	A	A	GSM only
OID_WWAN_VISIBLE_PROVIDERS has no corresponding structure	Not supported	Not supported	A	A	GSM
OID_WWAN_REGISTER_STATE uses **NDIS_WWAN_SET_REGISTER_STATE**	A	A	A	A	CDMA
OID_WWAN_SIGNAL_STATE uses **NDIS_WWAN_SET_SIGNAL_INDICATION**	A	A	A	A	GSM, CDMA
OID_WWAN_PACKET_SERVICE uses **NDIS_WWAN_SET_PACKET_SERVICE**	A	A	A	A	GSM, CDMA
OID_WWAN_PROVISIONED_CONTEXTS†† uses **NDIS_WWAN_SET_PROVISIONED_CONTEXT**	A	A	A	A	GSM, CDMA
OID_WWAN_CONNECT uses **NDIS_WWAN_SET_CONTEXT_STATE**	A	A	A	A	GSM, CDMA
OID_WWAN_SMS_CONFIGURATION uses **NDIS_WWAN_SET_SMS_CONFIGURATION**	A	A	A	A	GSM, CDMA
OID_WWAN_SMS_READ uses **NDIS_WWAN_SMS_READ**	Not supported	A	A	A	GSM, CDMA
OID_WWAN_SMS_SEND uses **NDIS_WWAN_SMS_SEND**	A	A	Not supported	Not supported	GSM, CDMA
OID_WWAN_SMS_DELETE uses **NDIS_WWAN_SMS_DELETE**	A	A	Not supported	Not supported	GSM, CDMA
OID_WWAN_SMS_STATUS uses **NDIS_WWAN_SMS_STATUS**	Not supported	Not supported	A	A	GSM, CDMA
OID_WWAN_VENDOR_SPECIFIC† uses a vendor-defined structure	A	A	Not supported	Not supported	GSM, CDMA
OID_WWAN_DEVICE_SERVICES has no corresponding structure	Not supported	Not supported	Not supported	A	GSM, CDMA
OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS uses **NDIS_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS**	Not supported	A	Not supported	Not supported	GSM, CDMA
OID_WWAN_AUTH_CHALLENGE uses **NDIS_WWAN_AUTH_CHALLENGE**	Not supported	Not supported	Not supported	A	GSM, CDMA
OID_WWAN_USSD uses **NDIS_WWAN_USSD_REQUEST**	Not supported	A	Not supported	Not supported	GSM
OID_WWAN_DEVICE_SERVICE_COMMAND uses **NDIS_WWAN_DEVICE_SERVICE_COMMAND**	Not supported	A	Not supported	A	GSM, CDMA

ⓘ Note

The following notes apply to the preceding table: † represents optional OIDs that miniport drivers may support. Miniport drivers that do not support the optional OIDs must not return them in OID_GEN_SUPPORTED_LIST.

†† represents miniport drivers that support GSM-based devices which can optionally support OID_WWAN_PROVISIONED_CONTEXTS set and query operations. Miniport drivers that support CDMA-based devices can optionally support OID_WWAN_PROVISIONED_CONTEXTS query operations for CDMA-based devices that report Simple IP (WWAN_CTRL_CAPS_CDMA_SIMPLE_IP).

Miniport drivers must support all non-optional OIDs. The MB Service may ignore any miniport driver that does not report all of the mandatory OIDs.

"A" and "S" in the Set and Query operation columns in the preceding table reflect the nature of the transaction for completing the OID request: "A" stands for an asynchronous transaction and "S" for a synchronous transaction.

The data structures in the preceding table correspond to set operation OIDs and to return data for synchronous query operation OIDs.

The following OIDs share a common variable length list data structure called **WWAN_LIST_HEADER** in their corresponding data structures:

- OID_WWAN_READY_INFO
- OID_WWAN_PREFERRED_PROVIDERS
- OID_WWAN_VISIBLE_PROVIDERS
- OID_WWAN_PROVISIONED_CONTEXTS
- OID_WWAN_SMS_READ

WWAN-Specific Indications, Corresponding Data Structures, and OS Revisions

Indication and **Corresponding Data Structure**	**Windows 7 Revision** **Windows 8 Revision**
NDIS_STATUS_WWAN_DEVICE_CAPS uses NDIS_WWAN_DEVICE_CAPS	NDIS_WWAN_DEVICE_CAPS_REVISION_1 NDIS_WWAN_DEVICE_CAPS_REVISION_2
NDIS_STATUS_WWAN_READY_INFO uses NDIS_WWAN_READY_INFO	NDIS_WWAN_READY_INFO_REVISION_1 NDIS_WWAN_READY_INFO_REVISION_1
NDIS_STATUS_WWAN_RADIO_STATE uses NDIS_WWAN_RADIO_STATE	NDIS_WWAN_RADIO_STATE_REVISION_1 NDIS_WWAN_RADIO_STATE_REVISION_1
NDIS_STATUS_WWAN_PIN_INFO uses NDIS_WWAN_PIN_INFO	NDIS_WWAN_PIN_INFO_REVISION_1 NDIS_WWAN_PIN_INFO_REVISION_1
NDIS_STATUS_WWAN_PIN_LIST uses NDIS_WWAN_PIN_LIST	NDIS_WWAN_PIN_LIST_REVISION_1 NDIS_WWAN_PIN_LIST_REVISION_1
NDIS_STATUS_WWAN_SERVICE_ACTIVATION† uses NDIS_WWAN_SERVICE_ACTIVATION_STATUS	NDIS_WWAN_SERVICE_ACTIVATION_STATUS_REVISION_1 NDIS_WWAN_SERVICE_ACTIVATION_STATUS_REVISION_1
NDIS_STATUS_WWAN_HOME_PROVIDER uses NDIS_WWAN_HOME_PROVIDER	NDIS_WWAN_HOME_PROVIDER_REVISION_1 NDIS_WWAN_HOME_PROVIDER_REVISION_1

NDIS_STATUS_WWAN_PREFERRED_PROVIDERS†	NDIS_WWAN_PREFERRED_PROVIDERS_REVISION_1
uses NDIS_WWAN_PREFERRED_PROVIDERS	NDIS_WWAN_PREFERRED_PROVIDERS_REVISION_1
NDIS_STATUS_WWAN_VISIBLE_PROVIDERS	NDIS_WWAN_VISIBLE_PROVIDERS_REVISION_1
uses NDIS_WWAN_VISIBLE_PROVIDERS	NDIS_WWAN_VISIBLE_PROVIDERS_REVISION_1
NDIS_STATUS_WWAN_REGISTER_STATE	NDIS_WWAN_REGISTRATION_STATE_REVISION_1
uses NDIS_WWAN_REGISTRATION_STATE	NDIS_WWAN_REGISTRATION_STATE_REVISION_2
NDIS_STATUS_WWAN_SIGNAL_STATE	NDIS_WWAN_SIGNAL_STATE_REVISION_1
uses NDIS_WWAN_SIGNAL_STATE	NDIS_WWAN_SIGNAL_STATE_REVISION_1
NDIS_STATUS_WWAN_PACKET_SERVICE	NDIS_WWAN_PACKET_SERVICE_STATE_REVISION_1
uses NDIS_WWAN_PACKET_SERVICE_STATE	NDIS_WWAN_PACKET_SERVICE_STATE_REVISION_1
NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS	NDIS_WWAN_PROVISIONED_CONTEXTS_REVISION_1
uses NDIS_WWAN_PROVISIONED_CONTEXTS	NDIS_WWAN_PROVISIONED_CONTEXTS_REVISION_1
NDIS_STATUS_WWAN_CONTEXT_STATE	NDIS_WWAN_CONTEXT_STATE_REVISION_1
uses NDIS_WWAN_CONTEXT_STATE	NDIS_WWAN_CONTEXT_STATE_REVISION_1
NDIS_STATUS_WWAN_SMS_CONFIGURATION	NDIS_WWAN_SMS_CONFIGURATION_REVISION_1
uses NDIS_WWAN_SMS_CONFIGURATION	NDIS_WWAN_SMS_CONFIGURATION_REVISION_1
NDIS_STATUS_WWAN_SMS_RECEIVE	NDIS_WWAN_SMS_RECEIVE_REVISION_1
uses NDIS_WWAN_SMS_RECEIVE	NDIS_WWAN_SMS_RECEIVE_REVISION_1
NDIS_STATUS_WWAN_SMS_SEND	NDIS_WWAN_SMS_SEND_STATUS_REVISION_1
uses NDIS_WWAN_SMS_SEND_STATUS	NDIS_WWAN_SMS_SEND_STATUS_REVISION_1
NDIS_STATUS_WWAN_SMS_DELETE	NDIS_WWAN_SMS_DELETE_STATUS_REVISION_1
uses NDIS_WWAN_SMS_DELETE_STATUS	NDIS_WWAN_SMS_DELETE_STATUS_REVISION_1
NDIS_STATUS_WWAN_SMS_STATUS	NDIS_WWAN_SMS_STATUS_REVISION_1
uses NDIS_WWAN_SMS_STATUS	NDIS_WWAN_SMS_STATUS_REVISION_1
NDIS_STATUS_WWAN_VENDOR_SPECIFIC†	N/A
uses a vendor-defined structure	
NDIS_STATUS_WWAN_USSD	NDIS_WWAN_USSD_EVENT_REVISION_1
uses NDIS_WWAN_USSD_EVENT	NDIS_WWAN_USSD_EVENT_REVISION_1
NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS	NDIS_WWAN_DEVICE_SERVICES_REVISION_1
uses NDIS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS	NDIS_WWAN_DEVICE_SERVICES_REVISION_1
NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE	NDIS_WWAN_DEVICE_SERVICE_RESPONSE_REVISION_1
uses NDIS_WWAN_DEVICE_SERVICE_RESPONSE	NDIS_WWAN_DEVICE_SERVICE_RESPONSE_REVISION_1
NDIS_STATUS_WWAN_DEVICE_SERVICE_EVENT	NDIS_WWAN_DEVICE_SERVICE_EVENT_REVISION_1
uses NDIS_WWAN_DEVICE_SERVICE_EVENT	NDIS_WWAN_DEVICE_SERVICE_EVENT_REVISION_1
NDIS_STATUS_WWAN_DEVICE_SERVICE_SUBSCRIPTION	NDIS_WWAN_DEVICE_SERVICE_SUBSCRIPTION_REVISION_1

uses NDIS_WWAN_DEVICE_SERVICE_SUBSCRIPTION	NDIS_WWAN_DEVICE_SERVICE_SUBSCRIPTION_REVISION_1
NDIS_STATUS_WWAN_AUTH_RESPONSE uses NDIS_WWAN_AUTH_RESPONSE	NDIS_WWAN_AUTH_RESPONSE_REVISION_1 NDIS_WWAN_AUTH_RESPONSE_REVISION_1
NDIS_STATUS_WWAN_SET_HOME_PROVIDER_COMPLETE uses NDIS_WWAN_SET_HOME_PROVIDER	N/A NDIS_WWAN_HOME_PROVIDER_REVISION_2

ⓘ Note

The following notes apply to the preceding table: † represents optional indications that miniport drivers may support. Be aware that if a miniport driver supports an optional OID, the miniport driver should also support the corresponding indication.

WWAN-Specific Indication Support for GSM, CDMA, and Unsolicited Indications

Indication	GSM	CDMA	Unsolicited indication allowed?
NDIS_STATUS_WWAN_DEVICE_CAPS	X	X	N
NDIS_STATUS_WWAN_READY_INFO	X	X	Y
NDIS_STATUS_WWAN_RADIO_STATE	X	X	Y
NDIS_STATUS_WWAN_PIN_INFO	X	X	N
NDIS_STATUS_WWAN_PIN_LIST	X	X	N
NDIS_STATUS_WWAN_SERVICE_ACTIVATION	X	X	N
NDIS_STATUS_WWAN_HOME_PROVIDER	X	X	N
NDIS_STATUS_WWAN_PREFERRED_PROVIDERS	X		Y
NDIS_STATUS_WWAN_VISIBLE_PROVIDERS	X	X	N
NDIS_STATUS_WWAN_REGISTER_STATE	X	X	Y
NDIS_STATUS_WWAN_SIGNAL_STATE	X	X	Y
NDIS_STATUS_WWAN_PACKET_SERVICE uses NDIS_WWAN_PACKET_SERVICE_STATE	X	X	Y
NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS	X	X	Y
NDIS_STATUS_WWAN_CONTEXT_STATE	X	X	Y
NDIS_STATUS_WWAN_SMS_CONFIGURATION	X	X	Y
NDIS_STATUS_WWAN_SMS_RECEIVE	X	X	Y
NDIS_STATUS_WWAN_SMS_SEND uses NDIS_WWAN_SMS_SEND_STATUS	X	X	N

NDIS_STATUS_WWAN_SMS_DELETE	X	X	N
NDIS_STATUS_WWAN_SMS_STATUS	X	X	Y
NDIS_STATUS_WWAN_VENDOR_SPECIFIC	X	X	Y
NDIS_STATUS_WWAN_USSD	X		Y
NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS	X	X	N
NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE	X	X	N
NDIS_STATUS_WWAN_DEVICE_SERVICE_EVENT	X	X	Y
NDIS_STATUS_WWAN_DEVICE_SERVICE_SUBSCRIPTION	X	X	N
NDIS_STATUS_WWAN_AUTH_RESPONSE	X	X	N
NDIS_STATUS_WWAN_SET_HOME_PROVIDER_COMPLETE	X	X	N

Multi-carrier Specific OIDs

The following changes apply to NDIS 6.30 miniport drivers that support multi-carrier mode. If the miniport driver does not support multi-carrier mode then please refer to the preceding table.

OID and **Windows 8 Corresponding Data Structure**	**Query Operation**	**Set Operation**	**GSM/CDMA**
OID_WWAN_HOME_PROVIDER uses **NDIS_WWAN_SET_HOME_PROVIDER**	A	A	GSM, CDMA
OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS uses **NDIS_WWAN_SET_PREFERRED_MULTICARRIER_PROVIDERS**. The **PreferredListHeader.ElementType** should be set to **WwanStructProvider2** and the structure is WWAN_PROVIDER2.	A	A	GSM, CDMA

Multi-carrier Specific Indications, Corresponding Data Structures, and OS Revisions

Indication and **Corresponding Data Structure**	**Windows 8 Revision**
NDIS_STATUS_WWAN_HOME_PROVIDER uses **NDIS_WWAN_HOME_PROVIDER2**	NDIS_WWAN_HOME_PROVIDER_REVISION_2
NDIS_STATUS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS uses **NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS**	NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS_REVISION_1. The **PreferredListHeader.ElementType** should be set to **WwanStructProvider2** and the list should contain WWAN_PROVIDER2 structure.
NDIS_STATUS_WWAN_VISIBLE_PROVIDERS uses **NDIS_WWAN_VISIBLE_PROVIDERS**	NDIS_WWAN_VISIBLE_PROVIDERS_REVISION_1. The **VisibleListHeader.ElementType** should be set to **WwanStructProvider2** and the list should contain WWAN_PROVIDER2 structure.

Multi-carrier Specific Indication Support for GSM, CDMA, and Unsolicited Indications

Indication and Corresponding Data Structure	GSM	CDMA	Unsolicited indication allowed?
NDIS_STATUS_WWAN_HOME_PROVIDER	X	X	N
NDIS_STATUS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS	X	X	Y
NDIS_STATUS_WWAN_VISIBLE_PROVIDERS uses NDIS_WWAN_VISIBLE_PROVIDERS	X	X	N

MB Operational Semantics

Article • 03/14/2023

Asynchronous Transactions

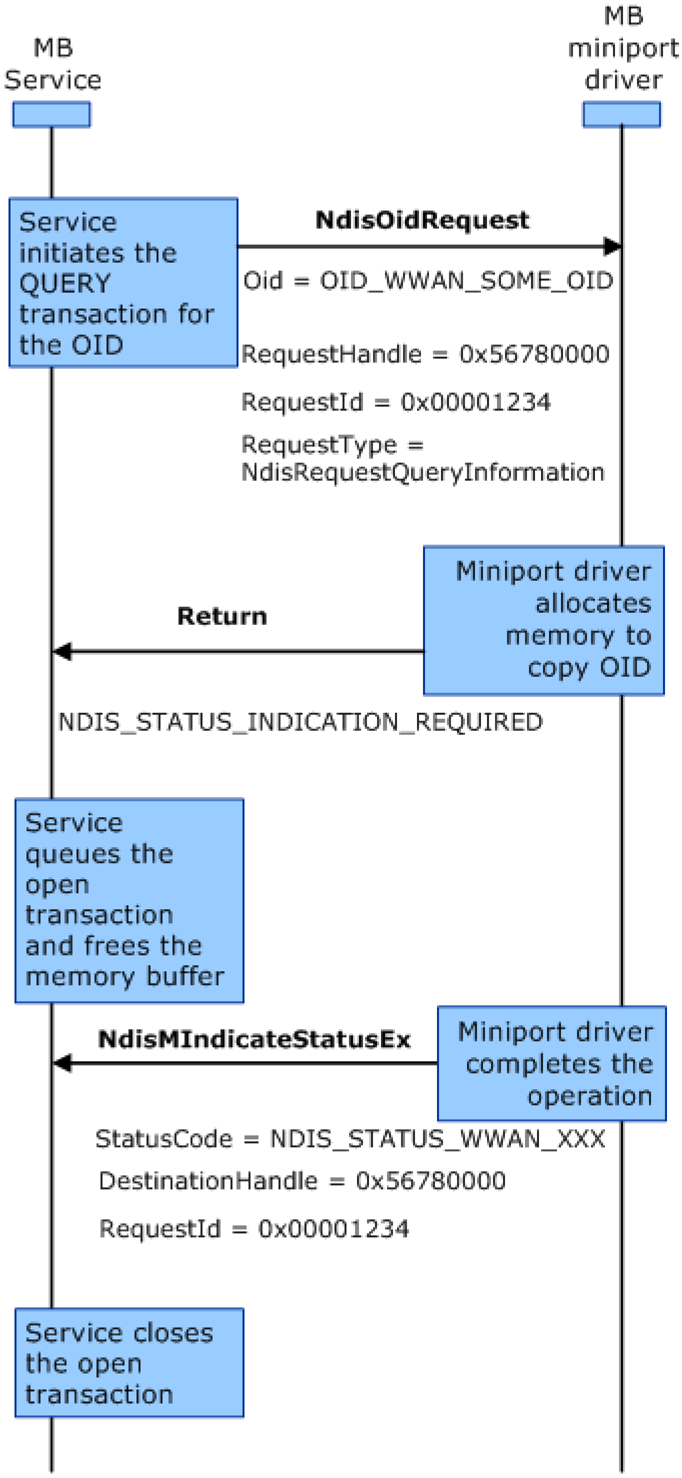
The MB driver model assumes non-blocking operational semantics between the MB Service and miniport drivers by using the asynchronous notification mechanism provided in NDIS 6.x. This mechanism allows the MB Service to continue to send OID requests to the miniport driver for processing without waiting for the current operation to complete.

An asynchronous transaction is a three-way handshake that starts with the initial request, followed by a request status response, and then completed by a final transactional indication. The request status response is provisional in that it only acknowledges that the miniport driver has received the request. The follow-up asynchronous indication is transactional in that it signals the completion of the transaction. The miniport driver returns the status code as well as the resulting data in the transactional indication.

Asynchronous *Set* and *Query* Requests

Many of the *set* and *query* OID requests that are used by the MB Service are processed asynchronously. For more information about *set* and *query* OID requests, see **NDIS_OID_REQUEST**. The "WWAN-specific OIDs" table in the MB Data Model topic identifies which OIDs are processed asynchronously.

The following diagram represents the interaction sequence for an asynchronous *query* transaction between the MB Service and the miniport driver. The labels in bold represent OID identifiers, or transactional flow control, and the labels in regular text represent the important flags within the OID structure.


MB Service
MB miniport driver
Service initiates the QUERY transaction for the OID
NdisOidRequest
Oid = OID_WWAN_SOME_OID
RequestHandle = 0x56780000
RequestId = 0x00001234
RequestType = NdisRequestQueryInformation
Miniport driver allocates memory to copy OID
Return
NDIS_STATUS_INDICATION_REQUIRED
Service queues the open transaction and frees the memory buffer
NdisMIndicateStatusEx
Miniport driver completes the operation
StatusCode = NDIS_STATUS_WWAN_XXX
DestinationHandle = 0x56780000
RequestId = 0x00001234
Service closes the open transaction

The three-way handshake is the same for both *query* and *set* requests.

Except for OID_WWAN_DRIVER_CAPS, all other MB-specific OID requests follow the asynchronous transaction mechanism for information exchange between miniport drivers and the MB Service, with the following additional notes:

- Miniport drivers should immediately fail an OID request on any error condition, such as an invalid OID request.
- Miniport drivers must return any WWAN-specific error conditions with the correct error code (for example, WWAN_STATUS_XXX) specified in the **uStatus** member of the event notification structure. Miniport drivers should also appropriately fill in

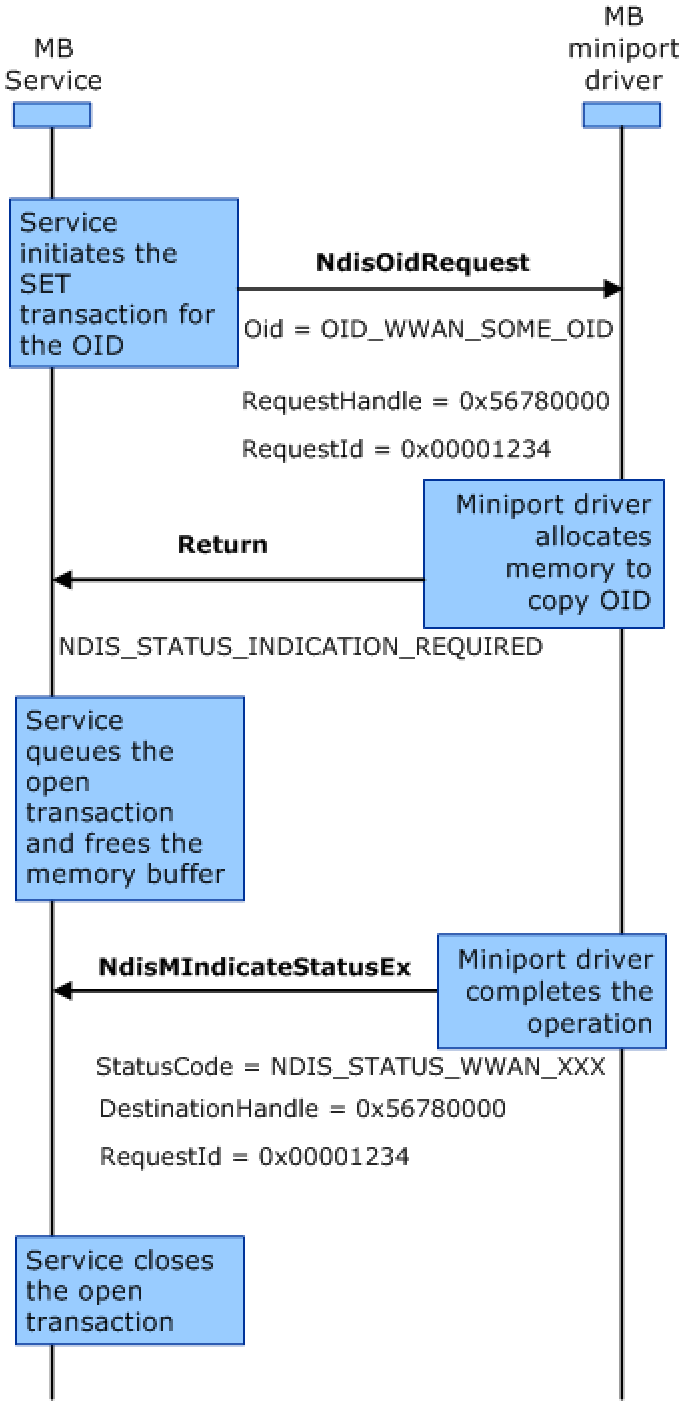
the members that follow the **uStatus** member, as needed. For example, miniport drivers should fill in the **ContextState.uNwError** member of the NDIS_WWAN_CONTEXT_STATE structure, if available. However, in the case of a failure when processing OIDs related to PINs, miniport drivers may not necessarily have the current PIN state information to specify in the **PinInfo.PinState** member of NDIS_WWAN_PIN_INFO.

- Miniport drivers should return NDIS_STATUS_INDICATION_REQUIRED as a provisional response for all asynchronous OID requests.

- Miniport drivers should be able to distinguish device state changes caused by an OID request from other causes. Miniport drivers should send transactional notifications for state changes resulting from OID requests, and they should send unsolicited event notifications for state changes from other causes.

- Miniport drivers are responsible for managing kernel-mode memory, although the MB Service initially allocates the memory for requests. After the MB Service receives a response from a miniport driver, the service may release the user-mode memory that it allocated for the OID request.

The following diagram represents the interaction sequence for an asynchronous *set* transaction between the MB Service and the miniport driver. The labels in bold represent OID identifiers, or transactional flow control, and the labels in regular text represent the important flags within the OID structure.


MB Service
MB miniport driver
Service initiates the SET transaction for the OID
NdisOidRequest
Oid = OID_WWAN_SOME_OID
RequestHandle = 0x56780000
RequestId = 0x00001234
Miniport driver allocates memory to copy OID
Return
NDIS_STATUS_INDICATION_REQUIRED
Service queues the open transaction and frees the memory buffer
NdisMIndicateStatusEx
Miniport driver completes the operation
StatusCode = NDIS_STATUS_WWAN_XXX
DestinationHandle = 0x56780000
RequestId = 0x00001234
Service closes the open transaction

Asynchronous Response

The *NDIS 6.0 Specification* (released with Windows Vista) introduced a new status code, NDIS_STATUS_INDICATION_REQUIRED, for miniport drivers to convey the asynchronous nature of a transaction to the MB Service in a miniport driver's provisional response to an OID request.

As mentioned in MB Interface Overview, the MB Service does not have direct access to kernel-mode memory that is allocated by an MB miniport driver. The execution result stored in the kernel-mode memory is assumed to be copied and made available to the MB Service by some intermediary, such as WMI or an NDIS filter driver. Hence, miniport drivers can release the allocated kernel-mode memory after the NdisMIndicateStatusEx function call returns in the transactional indication.

The handshake procedures that miniport drivers and the MB Service must follow are described in the following procedure.

MB miniport driver procedure

Upon receiving an OID request, miniport drivers should perform the following steps:

1. Allocate memory in kernel mode to copy the contents of the NDIS_OID_REQUEST data structure associated with the OID request.

2. Among the request's parameters, ensure that the **RequestId** and **RequestHandle** members of the OID request structure are also copied. These members will be used later in the transactional *indication*.

3. Return a provisional NDIS_STATUS_INDICATION_REQUIRED status response to inform the MB Service that the miniport driver will complete the request asynchronously.

4. Upon completion of the operation, store the result in local or driver-allocated memory, as appropriate.

5. Call the NdisMIndicateStatusEx function to notify the MB Service that the outstanding operation has been completed. Miniport drivers should fill in the members of the NDIS_STATUS_INDICATION structure as follows:
   a. Set the **StatusCode** member to the type of status notification. For example, NDIS_STATUS_WWAN_XXX.
   b. Set the **DestinationHandle** member to the **RequestHandle** member that was received in the NDIS_OID_REQUEST data structure when the miniport driver received the corresponding OID request.
   c. Set the **RequestId** member to match the **RequestId** member of the NDIS_OID_REQUEST status structure when the miniport driver received the corresponding OID request.
   d. Set the **StatusBuffer** and **StatusBufferSize** members to point to the miniport driver-allocated memory and the size of the memory buffer, respectively. This memory buffer contains the result of the completed operation.
   e. If the operation completes successfully, set the **uStatus** member to WWAN_STATUS_SUCCESS. Otherwise, set the **uStatus** member to the appropriate WWAN_STATUS_XXX value to indicate the type of failure.

6. When the function call returns, the miniport driver should release the memory it allocated for the OID request.

MB Service procedure

The MB Service processes asynchronous transactions by using the following procedure:

1. Allocate buffer memory for the request based on the OID data structure. Fill in the data structure members with appropriate values.

2. Call the NdisOidRequest function with the **InformationBuffer** member pointing to the OID data structure for the OID request and wait for the miniport driver to respond.

3. Upon receipt of an NDIS_STATUS_INDICATION_REQUIRED provisional response from the miniport driver, the MB Service saves the **RequestId**, releases the allocated memory, and marks the transaction as open. At this point, the MB Service is free to process subsequent OID requests and notifications.

4. Upon receipt of a notification with NDIS_STATUS_WWAN_XXX as the **StatusCode** value, check whether the **RequestId** matches that of any transaction marked as open. If there is a match, the service closes the transaction. If no match is found, treat the notification as an unsolicited event notification.

5. Process the data returned in the **StatusBuffer** member and make state changes to the MB Service as appropriate.

Indications

There are two types of WWAN-specific *indications* that miniport drivers can generate:

- Event notifications that result from an object state change in the MB device.

- Transactional notifications that signal the completion of an asynchronous operation.

In both cases, miniport drivers should call the NdisMIndicateStatusEx function.

Event Notification

Event notification is unsolicited in the sense that the miniport driver proactively sends the indication to the MB Service as a state change event. The state change is caused by an action from some entity other than the MB Service. The MB Service assumes miniport drivers are able to detect the cause of the change.

For any WWAN-specific event notification, miniport drivers must set the **RequestId** member of the NDIS_STATUS_INDICATION structure to zero. The **StatusCode** member specifies which object in the MB device has changed. The miniport driver can set this object to any of the following values:

NDIS_STATUS_WWAN_DEVICE_CAPS

NDIS_STATUS_WWAN_READY_INFO

NDIS_STATUS_WWAN_RADIO_STATE

NDIS_STATUS_WWAN_PIN_INFO

NDIS_STATUS_WWAN_PIN_LIST

NDIS_STATUS_WWAN_HOME_PROVIDER

NDIS_STATUS_WWAN_PREFERRED_PROVIDERS

NDIS_STATUS_WWAN_VISIBLE_PROVIDERS

NDIS_STATUS_WWAN_REGISTER_STATE

NDIS_STATUS_WWAN_PACKET_SERVICE

NDIS_STATUS_WWAN_SIGNAL_STATE

NDIS_STATUS_WWAN_CONTEXT_STATE

NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS

NDIS_STATUS_WWAN_SERVICE_ACTIVATION

NDIS_STATUS_WWAN_SMS_CONFIGURATION

NDIS_STATUS_WWAN_SMS_RECEIVE

NDIS_STATUS_WWAN_SMS_SEND

NDIS_STATUS_WWAN_SMS_DELETE

NDIS_STATUS_WWAN_SMS_STATUS

NDIS_STATUS_WWAN_VENDOR_SPECIFIC

The MB Service may also process other event notifications from NDIS. These non-MB event notifications are not necessarily subject to the requirement that their **RequestId** values be set to zero.

Transactional Notifications

Miniport drivers use transactional notifications to inform the MB Service that an asynchronous transaction has completed, and the MB Service uses transactional notifications to close open transactions and to update its state machine.

The MB Service expects transactional notifications so that it can close open transactions. It is the final exchange of the three-way handshake between the MB Service and the miniport driver in an asynchronous transaction. The value of **RequestId** member of the NDIS_STATUS_INDICATION in any transactional notification must be nonzero, which is copied from the corresponding request in the same transaction.

You must set the **RequestId** member of the NDIS_STATUS_INDICATION structure correctly for the asynchronous mechanism to function properly. The MB Service ensures that the **RequestId** value is unique and nonzero among all outstanding requests. Miniport drivers must return the same **RequestId** value in the corresponding *indication* in order for the MB Service to correlate the indication with an open transaction.

Status Indication Structure

Both the asynchronous response for a given OID request and the unsolicited event notification structures share the following structure members that are pointed to by **StatusBuffer** member of the *StatusIndication* parameter to NdisMIndicateStatusEx:

C++

```cpp
typedef struct _NDIS_WWAN_XXX {
  NDIS_OBJECT_HEADER Header;
  WWAN_STATUS uStatus;
  ULONG uNwError;//Optional. Only used for network operations.
  WWAN_XXX XxxStruct;
} NDIS_WWAN_XXX, *PNDIS_WWAN_XXX;
```

A value of zero in the **RequestId** member of the NDIS_STATUS_INDICATION structure means it is an unsolicited event notification and can occur any time.

If the **uStatus** member in the returned indication of any *set* or *query* OID request does not equal WWAN_STATUS_SUCCESS the members of the associated NDIS_WWAN_XXX structure do not need to be valid.

In the case of unsolicited event notifications based on network events, miniport drivers must fill in the **uNwError** member as appropriate, if applicable.

The following table shows registration, packet-attach, and packet-detach cause code failure values that are defined in the *3GPP TS 24.008 Specification* for GSM-based networks:

3GPP 24.008 Cause code	Interpretation of cause code
2 - International Mobile Subscriber Identity (IMSI) unknown in HLR	Either the SIM or the device is not activated, or the subscription has expired, which caused a network deactivation.
4 - IMSI unknown in VLR	Roaming feature is not subscribed to.
6 - Illegal ME	MS blocked by network due to stolen report.
7 - GPRS services not allowed	User does not have a GPRS subscription. User has only a voice connection subscription.
8 - GPRS and non-GPRS services not allowed	GPRS and non-GPRS services are not allowed.
11 - PLMN not allowed	Service is blocked by the network due to an expired subscription or another cause.
12 - Location area not allowed	User subscription does not allow access in the present location area.
13 - Roaming not allowed in this location area	The subscription permits roaming, but roaming is not allowed in the present location area.
14 - GPRS services not allowed in this PLMN	Selected network provider does not provide GPRS service to the MS.
15 - No suitable cells in location area	No subscription for the service.
17 - Network failure	Registration failed.
22 - Congestion	Registration failed due to network congestion.

For example, if the network initiates a deactivate context event because roaming is not allowed in the location area, miniport drivers should set the **uNwError** member to 13 as per the 3GPP TS 24.008 Cause codes for GSM-based networks.

Similar logic should be applied to CDMA-based networks as well. However, there is no standard for CDMA-based network error codes. CDMA-based devices should use the network -specific or device-specific error codes.

In the case of a miniport driver's asynchronous response to OID requests, the **RequestId** member of the NDIS_STATUS_INDICATION structure is a non-zero number that was passed to the miniport driver as part of a *set* or *query* request. The miniport driver must fill the **uStatus** member as appropriate. For example, WWAN_STATUS_SUCCESS, or any of the appropriate error values listed in the following section. In addition to this, the miniport driver must fill in the **uNwError** member where appropriate and available.

Event Notification Status

The following table lists the WWAN_STATUS codes that MB miniport drivers can specify in the **uStatus** member of the NDIS_WWAN_XXX event notification structures.

Value	Meaning
WWAN_STATUS_SUCCESS	The operation succeeded.
WWAN_STATUS_FAILURE	The operation failed (a generic failure).
WWAN_STATUS_BUSY	The operation failed because the device is busy.
WWAN_STATUS_SIM_NOT_INSERTED	The operation failed because the SIM card was not inserted fully into the device.
WWAN_STATUS_BAD_SIM	The operation failed because the SIM card is bad and cannot be used any further.
WWAN_STATUS_PIN_REQUIRED	The operation failed because a PIN must be entered to proceed.
WWAN_STATUS_PIN_DISABLED	The operation failed because the PIN is disabled.
WWAN_STATUS_NOT_REGISTERED	The operation failed because the device is not registered with any network.
WWAN_STATUS_PROVIDERS_NOT_FOUND	The operation failed because no network providers could be found.
WWAN_STATUS_NO_DEVICE_SUPPORT	The operation failed because the device does not support the operation.
WWAN_STATUS_PROVIDER_NOT_VISIBLE	The operation failed because the service provider is not currently visible.
WWAN_STATUS_DATA_CLASS_NOT_AVAILABLE	The operation failed because the requested data-class was not available.
WWAN_STATUS_PACKET_SVC_DETACHED	The operation failed because packet service is detached.

Value	Meaning
WWAN_STATUS_MAX_ACTIVATED_CONTEXTS	The operation failed because the maximum number of activated contexts has been reached.
WWAN_STATUS_NOT_INITIALIZED	The operation failed because the device is in the process of initializing. Retry the operation after the ready-state of the device changes to **WwanReadyStateInitialized**.
WWAN_STATUS_VOICE_CALL_IN_PROGRESS	The operation failed because a voice call is in progress.
WWAN_STATUS_CONTEXT_NOT_ACTIVATED	The operation failed because the context is not activated.
WWAN_STATUS_SERVICE_NOT_ACTIVATED	The operation failed because service is not activated.
WWAN_STATUS_INVALID_ACCESS_STRING	The operation failed because the access string is invalid.
WWAN_STATUS_INVALID_USER_NAME_PWD	The operation failed because the user name and/or password supplied are invalid.
WWAN_STATUS_RADIO_POWER_OFF	The operation failed because the radio is currently powered off.
WWAN_STATUS_INVALID_PARAMETERS	The operation failed because of invalid parameters.
WWAN_STATUS_READ_FAILURE	The operation failed because of a read failure.
WWAN_STATUS_WRITE_FAILURE	The operation failed because of a write failure.

The following table shows SMS specific status values.

Value	Meaning
WWAN_STATUS_SMS_OPERATION_NOT_ALLOWED	The SMS operation failed because the operation is not allowed.
WWAN_STATUS_SMS_MEMORY_FAILURE	The SMS operation failed because of a memory failure.
WWAN_STATUS_SMS_INVALID_MEMORY_INDEX	The SMS operation failed because of an invalid memory index-- *WwanSmsFlagIndex* for OID_WWAN_SMS_READ.

Value	Meaning
WWAN_STATUS_SMS_UNKNOWN_SMSC_ADDRESS	The SMS operation failed because the service center number is either invalid or unknown.
WWAN_STATUS_SMS_NETWORK_TIMEOUT	The SMS operation failed because of a network timeout.
WWAN_STATUS_SMS_MEMORY_FULL	The SMS operation failed because the SMS message store is full.
WWAN_STATUS_SMS_UNKNOWN_ERROR	The SMS operation failed because of an unknown error (a generic error).
WWAN_STATUS_SMS_FILTER_NOT_SUPPORTED	The SMS operation failed because the filter type requested is not supported.
WWAN_STATUS_SMS_MORE_DATA	This transaction is not yet complete. Some data has been returned and there is more data to be returned.
WWAN_STATUS_SMS_LANG_NOT_SUPPORTED	The SMS operation failed because the SMS language is not supported. This applies to CDMA-based devices only.
WWAN_STATUS_SMS_ENCODING_NOT_SUPPORTED	The SMS operation failed because the SMS encoding is not supported. This applies to CDMA-based devices only.
WWAN_STATUS_SMS_FORMAT_NOT_SUPPORTED	The SMS operation failed because the SMS format is not supported.

Note These WWAN-specific status codes are used only for asynchronous transactions in the **uStatus** member of the NDIS_WWAN_XXX structures.

Miniport drivers use event notifications to inform the MB Service about an object state change in their MB device without first having received an OID request. The MB Service uses event notifications to update its state machine only.

Be aware that while NDIS serializes all requests that are sent to miniport drivers, miniport drivers might not return the responses in the same order. This is because the queued requests in the miniport driver might be processed in parallel. Hence the MB Service ensures that if two requests are dependent upon each other, it will not send the second request until the miniport driver completes the first request.

State Change Notification

In general, miniport drivers should always notify the MB Service about the updated state of their MB device either through transactional notifications or through unsolicited event notifications. The following scenarios are some exceptions where miniport drivers are not supposed to respond with updated state information. The MB Service can determine the updated state from the completion status of other operations:

1. Miniport drivers do not need to send an NDIS_STATUS_WWAN_PIN_LIST event indication when PIN state changes occur because the MB Service requested to enable or disable the PIN.

2. Miniport drivers do not need to return the updated list of the provisioned contexts in transactional responses to OID_WWAN_PROVISIONED_CONTEXT *set* operations.

3. Miniport drivers do not need to respond with the updated list of the preferred providers in transactional responses to OID_WWAN_PREFERRED_PROVIDERS *set* operations. The MB Service can determine this information based on the initial list and success status of the *set* operation.

4. Miniport drivers do not need to respond with the current WWAN_SMS_CONFIGURATION value for OID_WWAN_SMS_CONFIGURATION *set* operations.

MB Driver Model Versioning

Article • 03/14/2023

MB driver model versioning is accomplished by having the driver model version and individual OID data structure revisions. This is consistent with the versioning paradigm used in NDIS 6.x.

The driver model version defines the interface evolution between the MB Service and the MB miniport driver. The individual OID revisions keep track of the changes made to OIDs in different MB driver model versions. That is, the driver model version defines a set of OIDs whose data structures are identified by specific revision numbers.

Consistent with the *NDIS Specification*, the MB driver model evolution is *additive*. That is, new OIDs and new members can only be added to existing OID data structures. This ensures that the MB Service can support backward compatibility for miniport drivers.

Important Only under extremely rare circumstances will existing OIDs be deprecated or members of existing OID data structures not be used in the next version. If that happens, these changes and their impacts on backward compatibility shall be clearly documented in subsequent documentation about newer versions of the MB driver model specification.

This documentation covers the Windows 8 release of the MB driver model. The driver model version has been incremented to version 2.0. Some OID revisions continue to be revision number 1, while some have been updated to revision 2. For more information about which revisions to use with respective OIDs, see MB Data Model.

This documentation covers the initial release of the MB driver model, so both the driver model version and individual OID revisions start with revision number 1.

When the driver model moves to the next version, its version number is increased by 1. Any new OIDs added to the driver model will start at revision 1; any existing OIDs whose data structures have changed will increase their corresponding revision by 1, and any existing OIDs that do not change will keep their respective revision numbers.

The driver model version is conveyed by OID_WWAN_DRIVER_CAPS. The MB Service sends an OID_WWAN_DRIVER_CAPS query request to the miniport driver during MB Miniport Driver Initialization. Individual OID revisions are described by the **Revision** member of the **NDIS_OBJECT_HEADER** structure that is included as part of the data structure for each individual OID.

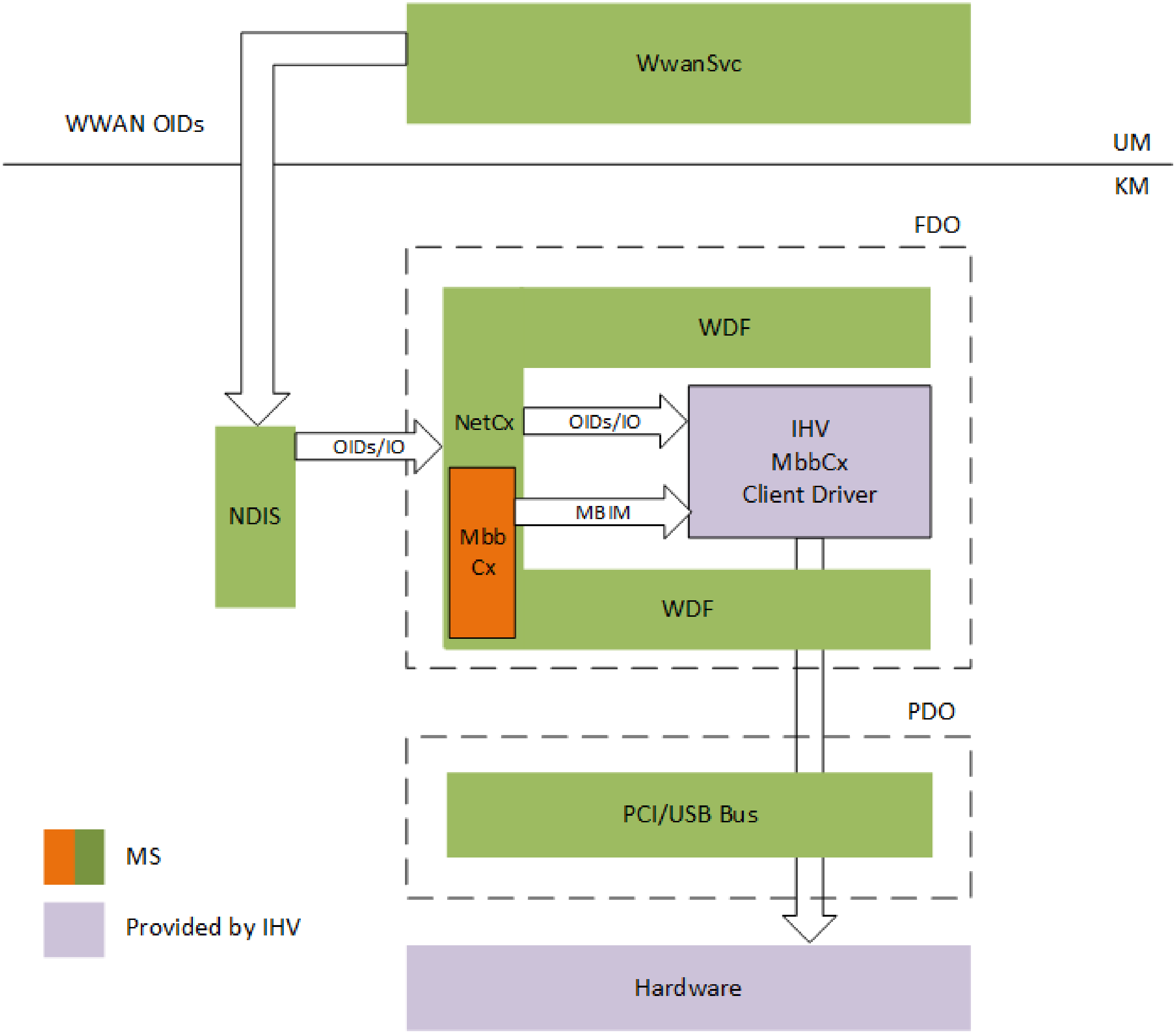
See also

MBIM extension 2.0 versioning for 5G

Introduction to the Mobile Broadband (MBB) WDF class extension (MBBCx)

Article • 12/15/2021

Starting in the next release of Windows 10, the Windows Driver Kit (WDK) includes a Mobile Broadband (MBB) WDF class extension that works with NetAdapterCx. MBB-NetAdapter client drivers are first and foremost fully fledged WDF client drivers, then they're NetAdapterCx client drivers just like other NIC drivers, and finally they're client drivers of the MBB class extension (MBBCx) that provides MBB media-specific functionality. The following block diagram illustrates the MBBCx architecture:



An MBB-NetAdapter client driver performs 3 categories of tasks based on its relationships with the framework:

- Call standard WDF APIs for common device tasks like Pnp and Power management.
- Call NetAdapterCx APIs for common network device operations like transmitting or receiving network packets.

- Call MbbCx APIs for MBB-specific control path operations like MBIM message handling.

Before you begin, you should familiarize yourself with these concepts:

- Windows Driver Foundation (WDF)
- NetAdapter class extension (NetAdapterCx)

The topics in this section assume you already know how to write a NetAdapterCx client driver for a basic NIC, so they focus only on MBBCx-specific code.

This section contains the following topics:

- Writing an MBBCx client driver

MB Driver Stack, Suspend, and Resume

Article • 03/14/2023

Overview

Microsoft provides an inbox class driver for mobile broadband (MBB) devices called the Mobile Broadband Class Driver (MBCD). This driver is based on the Mobile Broadband Interface Model (MBIM) specification, which is an interface for MBB devices (also known as modems) to communicate with Windows. The MBIM specification is based on USB. MBCD provides support for USB modems and modems that emulate USB through a technology called USB Device Emulation (UDE).

MBCD is a miniport driver that combines with the Network Driver Interface Specification (NDIS) port driver to form a single function driver. In the OSI Network Model, this driver logically sits on the top half of the Data Link Layer (layer 2). Network protocol drivers (such as IP) that logically sit on the Network layer (layer 3) receive data (SDUs) in segments (TCP) or datagrams (UDP) from the Transport Layer (layer 4) and send down data (PDUs) as packets to the Data Link Layer by invoking NDIS APIs. Generally, NDIS only involves a miniport driver when it is necessary.

OSI Network Model				
Layer			Protocol Data Unit (PDU)	Function
Host Layer	7	Application	Data	High level APIs including resource sharing and remote file access
	6	Presentation		Translation of data between a networking service and an application including character encoding, data compression, and encryption
	5	Session		Management of communication sessions, for example continuous information exchange in the form of multiple back-and-forth transmissions between two nodes
	4	Transportation	Segment	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement, and multiplexing

Media Layer	3	Network	Packet	Management and structuring of multi-node networks including address mapping, routing, and traffic control
	2	Data Link	Frame	Reliable transmission of data frames between two nodes connected by the Physical layer
	1	Physical	Symbol	Transmission and reception of raw bit streams over a physical medium

The Network Layer is where network protocol drivers reside, including the NDIS Usermode I/O (NDISUIO) protocol driver. This driver serves an important role in the control and configuration of MBB devices. It is important to note that this layer is also conceptually where the IP portion of TCP/IP resides. You may think of these as siblings.

WwanSvc is the service primarily responsible for control of the modems, enumerating their capabilities, and their configuration. WwanSvc uses WWAN OIDs to issue commands to NDISUIO, which will pass these OIDs to NDIS. The MBCD miniport driver defines the OIDs that it supports and provides this to NDIS as part of the initialization of the function driver. Therefore, when NDIS receives an OID from NDISUIO it will involve the miniport as necessary.

The flow of a command from an application (such as the cellular UI) looks like this:

Application -> WwanSvc ---(OID)---> NDISUIO ----(OID)---> NDIS ----(OID)---> MBCD ---(MBIM)---> MBB Device.

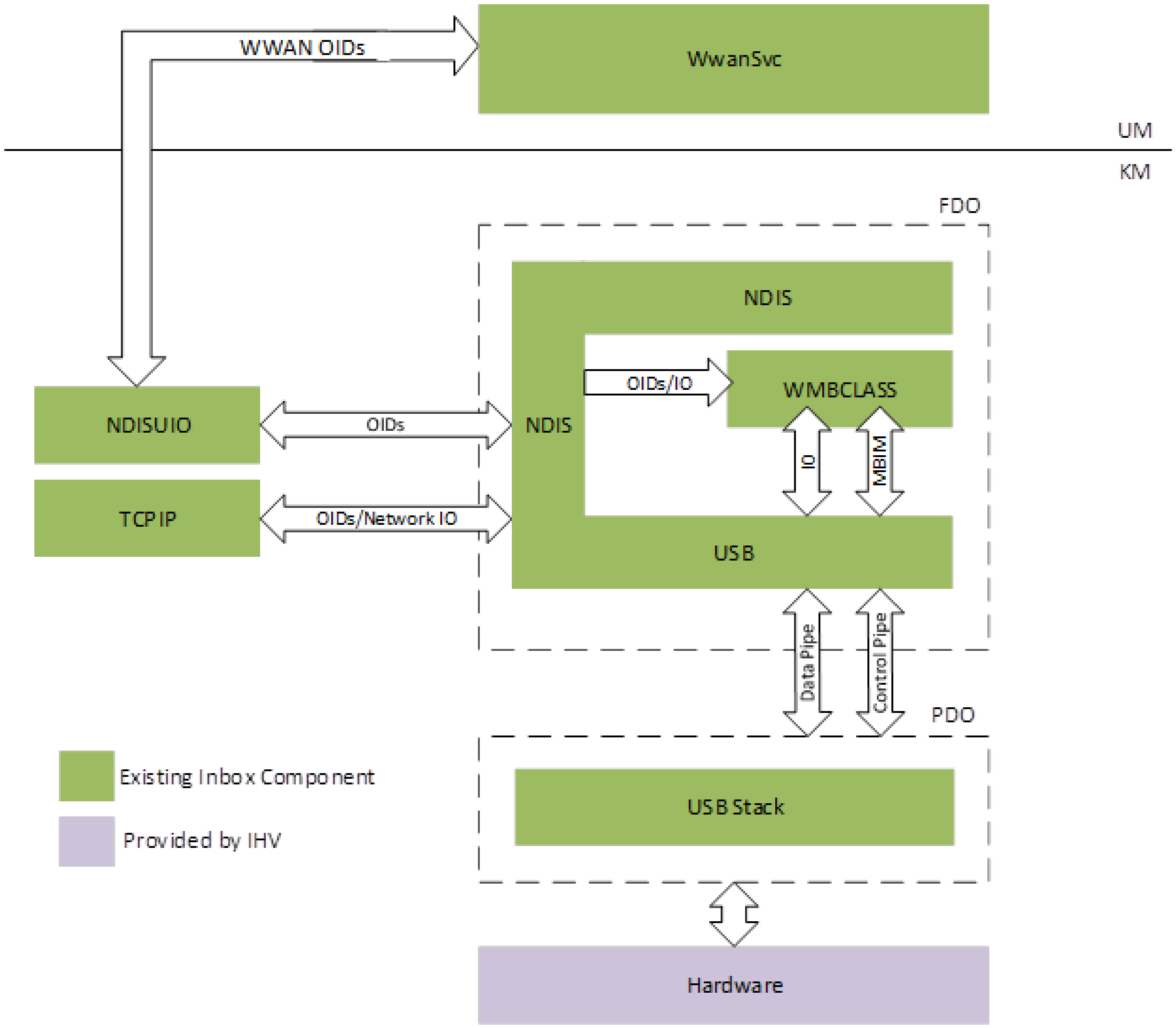
The above provides an overview of the technologies involved for the control path. The data path is more complicated as there are several solutions in place. However, we can generalize the data path as:

Application -> TCP/IP --(packets)--> NDIS ----(frames)---> [Driver] ---> MBB Device.

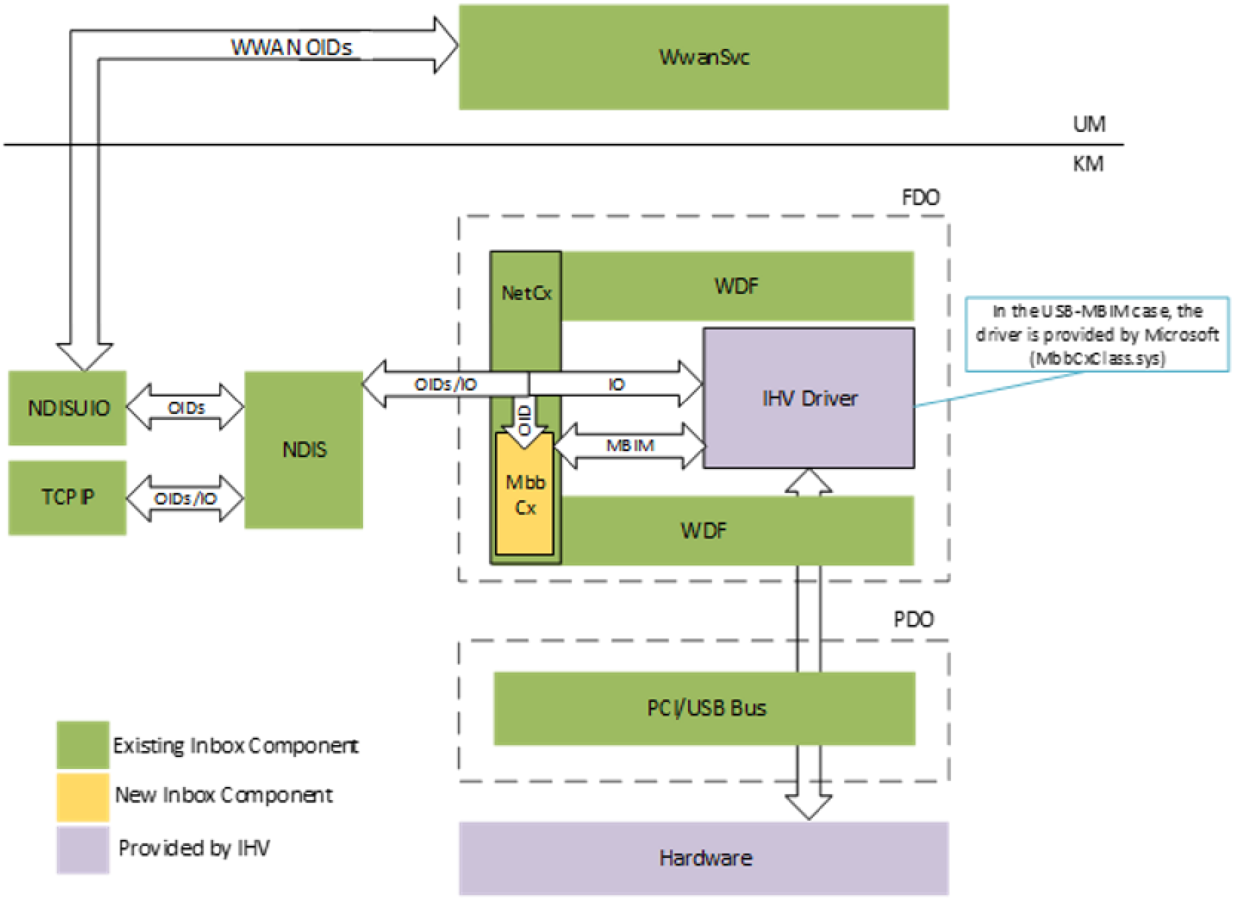
[Driver] might be the legacy driver, the new modern driver, or a 3rd party IHV driver.

Driver Architecture

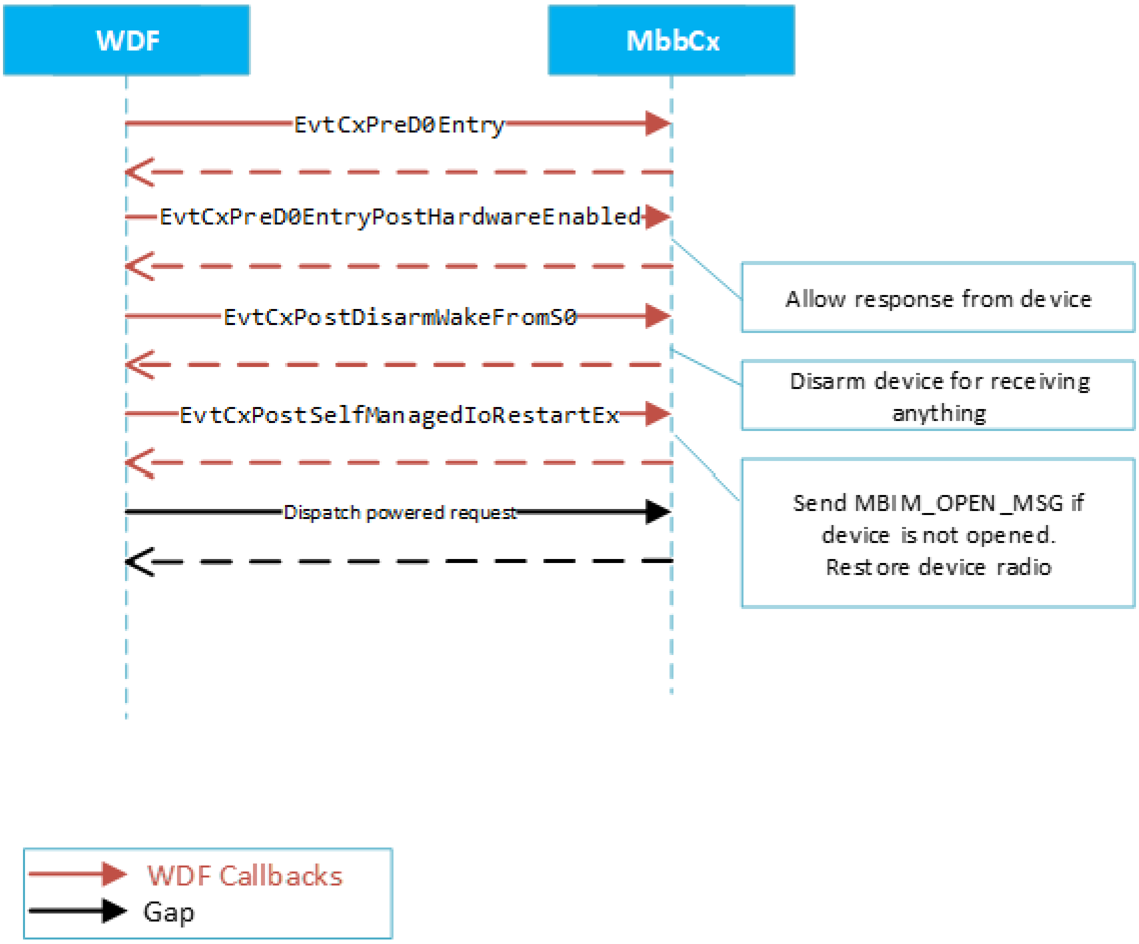
Legacy



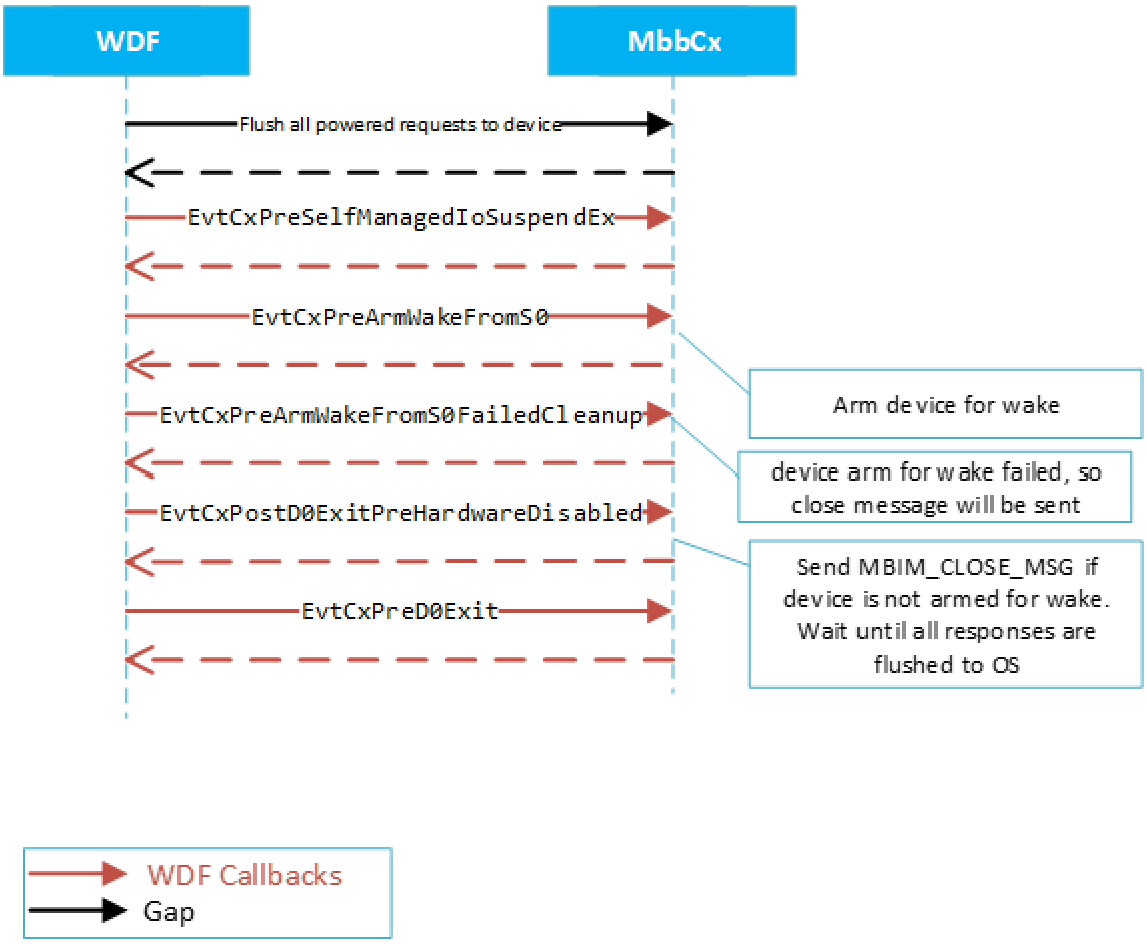
Current (Since RS5 OSBuild 17763)



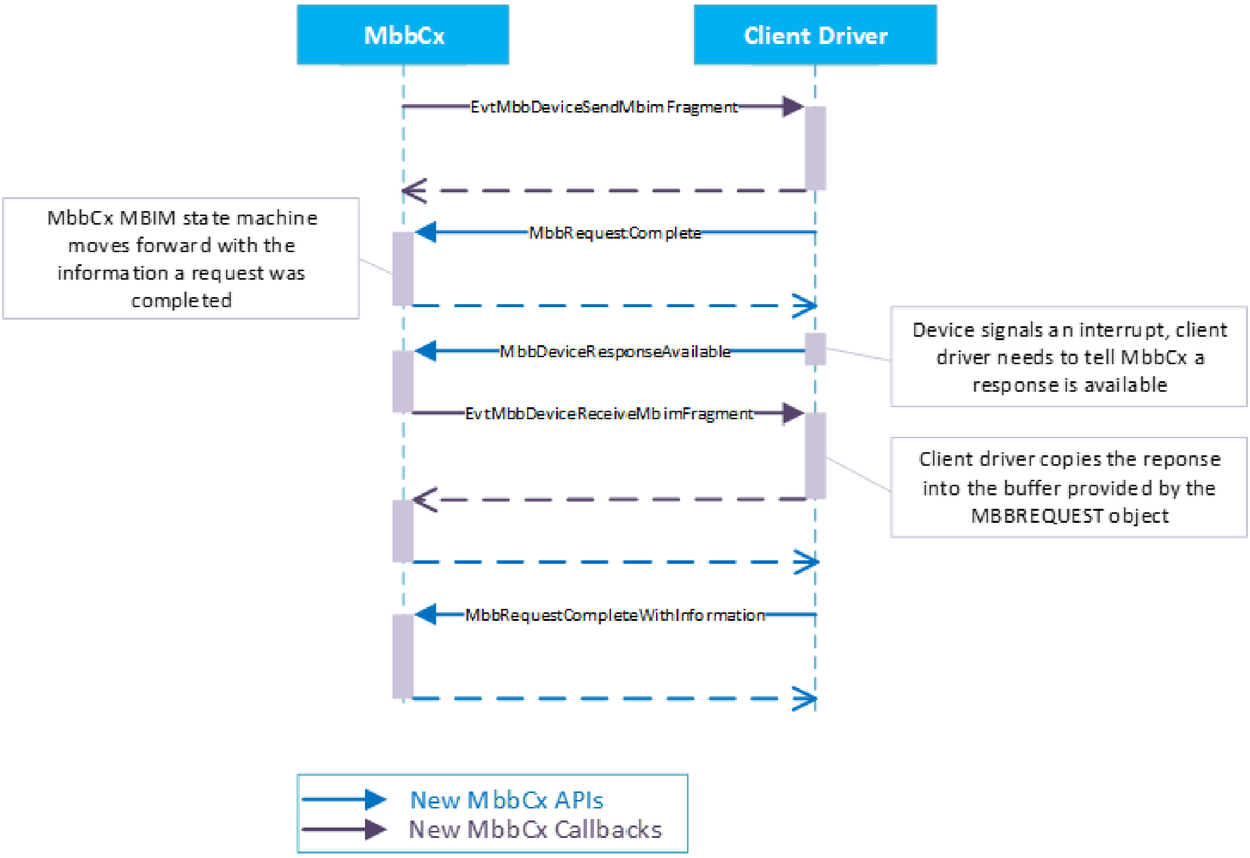
Device Power Up


WDF
MbbCx
EvtCxPreD0Entry
EvtCxPreD0EntryPostHardwareEnabled
EvtCxPostDisarmWakeFromS0
EvtCxPostSelfManagedIoRestartEx
Dispatch powered request
Allow response from device
Disarm device for receiving anything
Send MBIM_OPEN_MSG if device is not opened. Restore device radio
WDF Callbacks
Gap

Device Power Down


WDF
MbbCx
Flush all powered requests to device
EvtCxPreSelfManagedIoSuspendEx
EvtCxPreArmWakeFromS0
EvtCxPreArmWakeFromS0FailedCleanup
EvtCxPostD0ExitPreHardwareDisabled
EvtCxPreD0Exit
Arm device for wake
device arm for wake failed, so close message will be sent
Send MBIM_CLOSE_MSG if device is not armed for wake. Wait until all responses are flushed to OS
WDF Callbacks
Gap

MBBCx interface

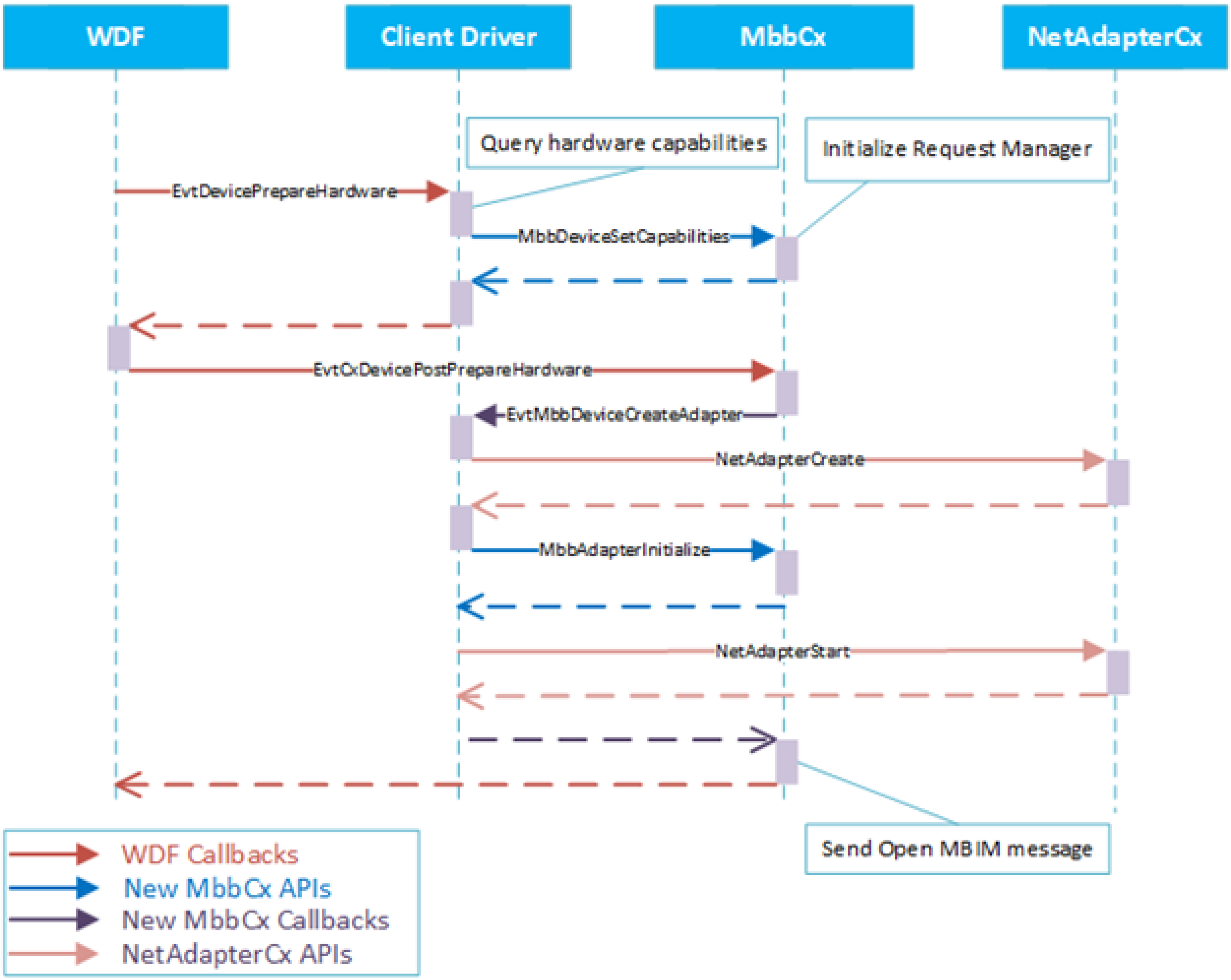

MbbCx
Client Driver
EvtMbbDeviceSendMbimFragment
MbbCx MBIM state machine moves forward with the information a request was completed
MbbRequestComplete
Device signals an interrupt, client driver needs to tell MbbCx a response is available
MbbDeviceResponseAvailable
EvtMbbDeviceReceiveMbimFragment
Client driver copies the reponse into the buffer provided by the MBBREQUEST object
MbbRequestCompleteWithInformation
New MbbCx APIs
New MbbCx Callbacks

See Also

EvtMbbDeviceSendMBIMFragment

MbbRequestComplete

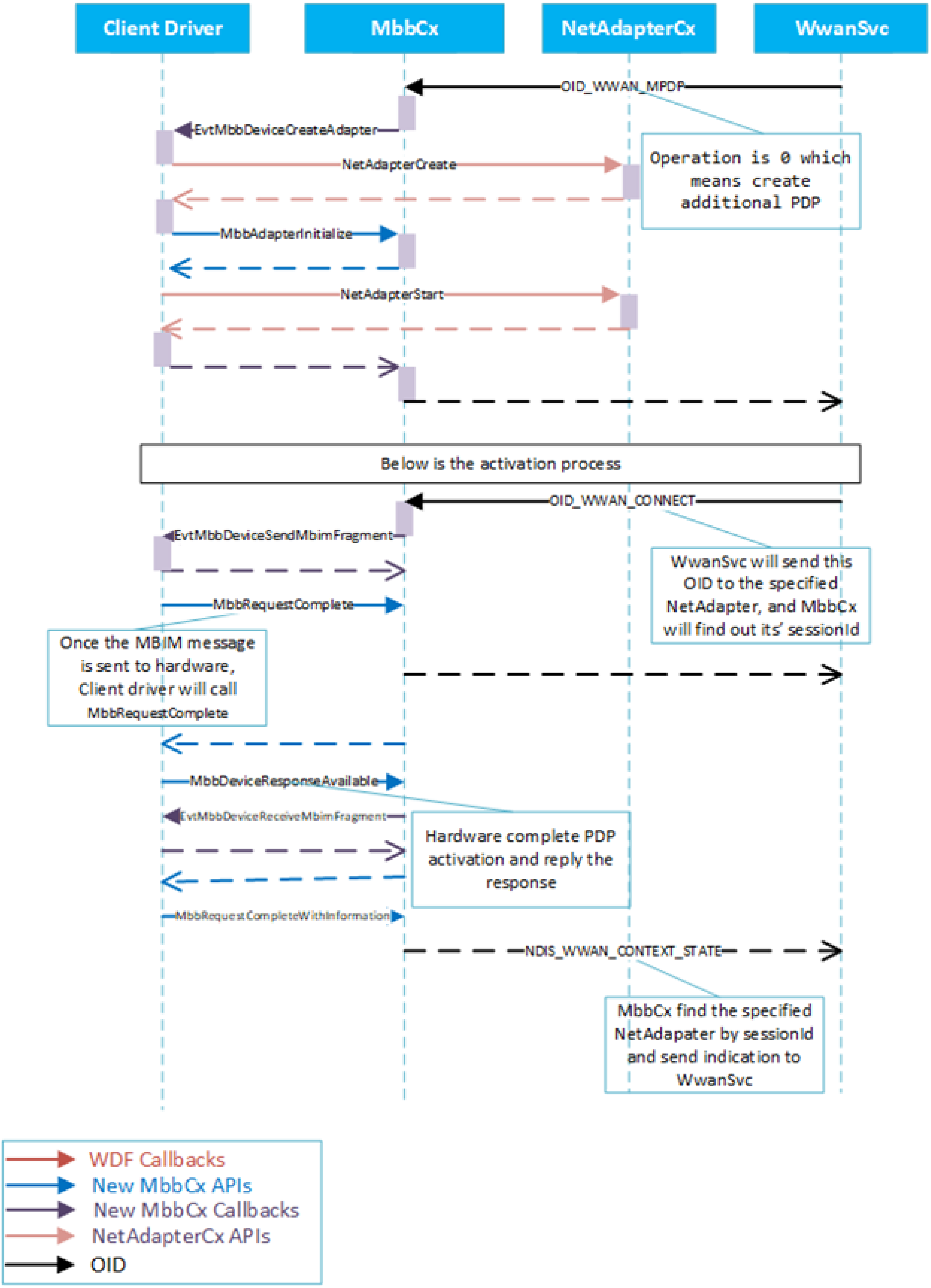
Default NetAdapter Initialization



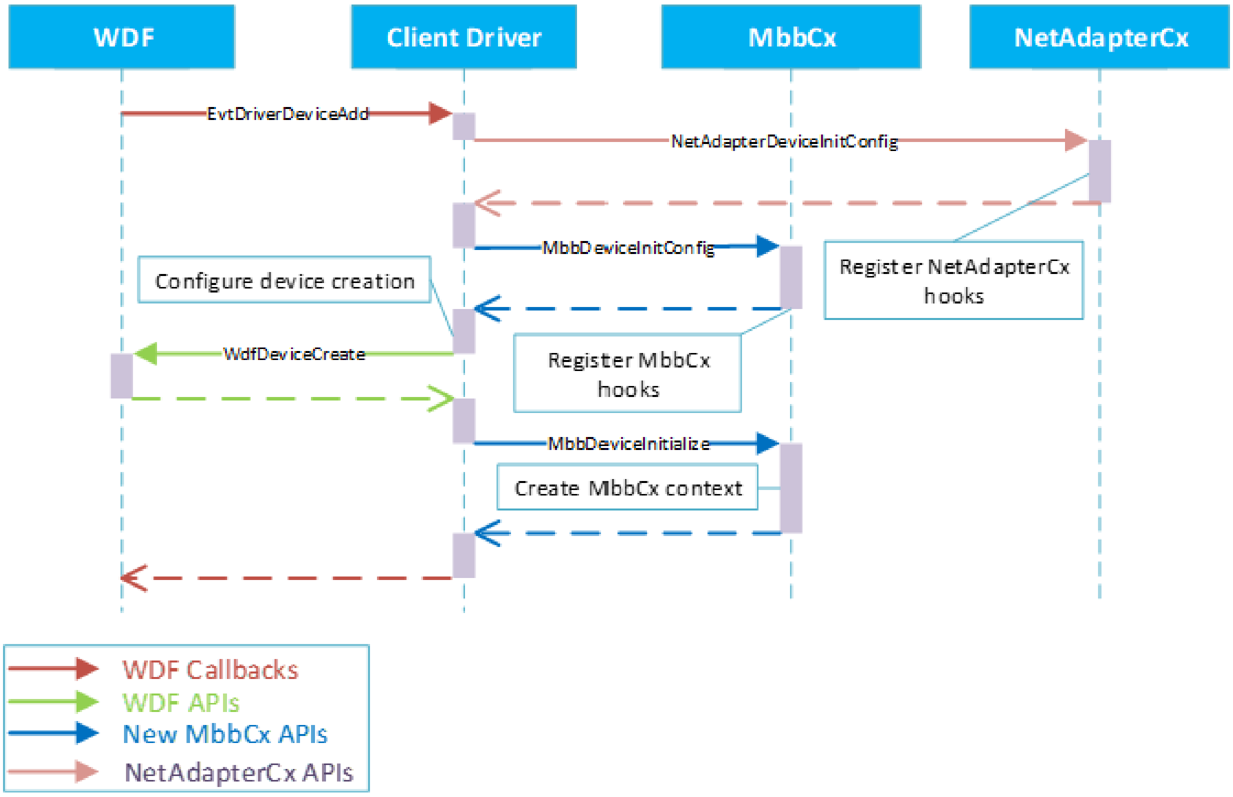
See Also

MbbAdapterInitialize

Additional NetAdapter Initialization


Client Driver
MbbCx
NetAdapterCx
WwanSvc
OID_WWAN_MPDP
Operation is 0 which means create additional PDP
EvtMbbDeviceCreateAdapter
NetAdapterCreate
MbbAdapterInitialize
NetAdapterStart
Below is the activation process
OID_WWAN_CONNECT
WwanSvc will send this OID to the specified NetAdapter, and MbbCx will find out its' sessionId
EvtMbbDeviceSendMbimFragment
MbbRequestComplete
Once the MBIM message is sent to hardware, Client driver will call MbbRequestComplete
MbbDeviceResponseAvailable
EvtMbbDeviceReceiveMbimFragment
Hardware complete PDP activation and reply the response
MbbRequestCompleteWithInformation
NDIS_WWAN_CONTEXT_STATE
MbbCx find the specified NetAdapater by sessionId and send indication to WwanSvc
WDF Callbacks
New MbbCx APIs
New MbbCx Callbacks
NetAdapterCx APIs
OID

Device Initialization



Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to the device Cellular modem driver and run test: TestPowerStates.

Via netsh, we can run the **TestPowerStates** HLK testlist. For more information on using the netsh tool, see **netsh-mbn** and **netsh-mbn-test-installation**.

```
netsh mbn test feature=power testpath="C:\\data\\test\\bin"
taefpath="C:\\data\\test\\bin"
```

This file showing the HLK test results should have been generated in the directory that the 'netsh mbn test' command was ran from: `TestPowerStates.htm`.

Manual Tests

Auto-connect after wake from hibernation (S4)

1. Ensure "Let Windows manage this connection" is checked in Cellular settings.
2. Put DUT into S4.

3. Wake DUT. Verify it automatically establishes a cellular connection and the user is able to browse the internet.

Connect Cellular manually after wake from hibernation (S4)

1. With Ethernet unplugged and Wi-Fi toggled off, uncheck "Let Windows manage this connection" in Cellular settings.
2. In an admin CMD prompt run the command: shutdown -h
3. Machine will hibernate. After more than 30 seconds press the machine's power button to wake from hibernation. Log back in, open Cellular settings, and click Connect to Cellular. Cellular should connect and the user should be able to browse the internet.

Auto-connect after wake from screen sleep

1. With Ethernet unplugged and Wi-Fi toggled off, verify an active cellular connection.
2. (Optional Step) Allow screen to sleep. You can set screen sleep to 1 minute under Settings -> System -> Power & sleep. The setting should not be set to "Never".
3. Wake the screen by using the mouse or keyboard and log back in. Cellular should stay connected and the user should be able to browse internet, including under the VAIL/WCOS system.

Log Analysis

Tips

- Ensure necessary ETW providers are included in the log, including MbbCx, NetAdapterCx, WwanSvc, and NdisUio.
- Check the device power state (Dx state) and the device power capabilities first
- Check the logs with power flows above
- OID and indication pair

Sample log

```
597454 [2]1020.115C::2018-08-31 01:05:12.669792000 [WwanService]INFO:
CWwanDataExecutor::OnNdisNotification - current device power state 3
```

```
(WaitForDeviceD0AfterSleep 1 systemPowerState 0)
679337 [6]1020.115C::2018-08-31 01:07:36.343312200 [WwanService]INFO:
CWwanManager::OnSystemPowerStateChange - system resuming from sleep
(fWaitForDeviceD0AfterSleep 1)
2422155 [7]1020.1150::2018-08-31 01:07:37.878446100 [WwanService]INFO:
CWwanDataExecutor::OnNdisNotification - current device power state 0
(WaitForDeviceD0AfterSleep 1 systemPowerState 1)
2437098 [3]1020.115C::2018-08-31 01:07:37.893061200 [WwanService]INFO:
CWwanDeviceEnumerator::onDeviceRemoval: MBB device removed [9d33b700-d66d-
4c0a-807f-6a328690dafa].
2678588 [5]1020.2E30::2018-08-31 01:07:40.765642800 [WwanService]INFO:
CWwanDeviceEnumerator::onDeviceArrival: MBB device arrived [9d33b700-d66d-
4c0a-807f-6a328690dafa]. Parent Interface = [00000000-0000-0000-0000-
000000000000].
2679204 [6]1020.2E30::2018-08-31 01:07:40.766278700 [sys]Ref
WwanprotGetD3ColdCapability:0x6a2 \DEVICE\{9D33B700-D66D-4C0A-807F-
6A328690DAFA} 0x2
2679205 [6]1020.2E30::2018-08-31 01:07:40.766280200 [sys]Sending
IRP_MN_QUERY_INTERFACE for interface GUID_D3COLD_SUPPORT_INTERFACE
2679211 [6]1020.2E30::2018-08-31 01:07:40.766287400
[sys]IRP_MN_QUERY_INTERFACE for interface GUID_D3COLD_SUPPORT_INTERFACE
succeeded
2679212 [6]1020.2E30::2018-08-31 01:07:40.766289500 [sys]Successfully
queried the D3 cold capability of device. D3ColdCapability = 0
2679213 [6]1020.2E30::2018-08-31 01:07:40.766290000 [sys]DeRef
WwanprotGetD3ColdCapability:0x6a8 \DEVICE\{9D33B700-D66D-4C0A-807F-
6A328690DAFA} 0x2
2679214 [6]1020.2E30::2018-08-31 01:07:40.766290500 [sys]Returning D3 cold
capability as 0. Status = c0000225
2679219 [6]1020.2E30::2018-08-31 01:07:40.766294100
[WwanService]CWwanNetworkInterface::InitializeInterface: Getting D3 cold
capability for interface 9d33b700-d66d-4c0a-807f-6a328690dafa failed [1168]
2679220 [6]1020.2E30::2018-08-31 01:07:40.766294600
[WwanService]CWwanNetworkInterface::InitializeInterface:
fIsEmbedded:0x00000001(true) fIsD3ColdSupported:0x00000000(false)
```

See Also

UDE Architecture

Introduction to NDIS 6.20

MBIM Overview

MBIM Compliance Testing Revision 1.0

Mobile Broadband Implementation Guidelines for USB Devices

NetAdapterCx

MB Miniport Driver INF Requirements

Article • 03/14/2023

MB miniport drivers must have the following entries in their INF file:

INF

```
*IfType  = 243; IF_TYPE_WWANPP
*MediaType  = 9; <mark type="enumval">NdisMediumWirelessWan</mark>
*PhysicalMediaType  = 8; NdisPhysicalMediumWirelessWan
EnableDhcp  = 0; Disable DHCP

;Entries to be put in add-registry-section for NdisMediumWirelessWan
HKR, Ndi\Interfaces, UpperRange, 0, "flpp4, flpp6"
HKR, Ndi\Interfaces, LowerRange, 0, "ppip"
```

All the entries mentioned in the preceding code example, except UpperRange and LowerRange, should be under the same INF section as that of keywords such as AddReg and CopyFiles. UpperRange and LowerRange should be put in the add-registry-section of the INF file.

*IfType

Dual-mode devices can specify either of the *IfType* values from the following table:

Description	Name	IfType
GSM-based MB devices	IF_TYPE_WWANPP	243
CDMA-based MB devices	IF_TYPE_WWANPP2	244

*MediaType

MB miniport drivers must specify one of the MediaType values from the following table based on the type of packet framing the miniport driver is capable of interpreting in its send and receive data path.

Description	Name	MediaType
MB miniport drivers that interpret 802.3 packets must report this media type. This framework is only for migration of old miniport	NdisMedium802_3	0

drivers and is not recommended for production-quality miniport drivers.		
MB miniport drivers that are able to handle raw IP traffic must set this media type. This is the recommended media type to be used in production-quality miniport drivers.	NdisMediumWirelessWan	9

EnableDhcp

MB miniport drivers must specify one of the EnableDhcp values from the following table based on whether they implement DHCP server emulation.

Value	Description
0	Disable DHCP for this interface. The miniport driver does not implement DHCP server spoofing. This is the recommended value to be used in production-quality drivers.
1	Enable DHCP for this interface. The miniport driver implements DHCP server spoofing. That is, the miniport driver will need to spoof a DHCP server and ARP resolutions.

UpperRange

This keyword is set with one or more combinations of the following strings as applicable when media type is NdisMediumWirelessWan. NdisMedium802_3 miniport drivers should use the existing values in UpperRange.

Value	Description
"flpp4"	Miniport drivers specify "flpp4" if the MB device supports IPv4.
"flpp6"	Miniport drivers specify "flpp6" if the MB device supports IPv6. This value is needed only for devices that support IPv6.

LowerRange

This keyword must have, at the minimum, the following value when media type is NdisMediumWirelessWan. NdisMedium802_3 miniport drivers should use the existing values in LowerRange.

Value	Description
"ppip"	MB device type on the lower edge.

MB Miniport Driver Types

Article • 03/14/2023

Based on data packet handling, DHCP Server and ARP emulations, multiple MB miniport driver implementation types are possible. The following table represents the different possible implementation types and the required implementation for production quality miniport drivers.

Description	MediaType	EnableDhcp	ARP emulation
Ethernet emulation with DHCP emulation	NdisMedium802_3	1	Required
Ethernet emulation with no DHCP	NdisMedium802_3	0	Required
IP packet handling with DHCP emulation	NdisMediumWirelessWan	1	Not required
IP packet handling capability	NdisMediumWirelessWan	0	Not Required

During development or migration phases, miniport drivers can specify any of first three entries. However, production quality miniport drivers should use only the settings specified in the last entry of the table ("IP packet handling capability").

Production quality MB miniport drivers should specify the settings in the following table in the INF file.

Field in INF file	Recommended value(s)
*IfType	IF_TYPE_WWANPP / IF_TYPE_WWANPP2
MediaType	NdisMediumWirelessWan
EnableDhcp	0
UpperRange	"flpp4" and "flpp6" (if IPv6 supported)
LowerRange	"ppip"

MB Adapter General Attribute Requirements

Article • 03/14/2023

The following table describes the values that miniport drivers should set the member variables of the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure to. MB miniport drivers must use these values when they call NdisMSetMiniportAttributes from their *MiniportInitializeEx* function, during miniport driver initialization.

Field in INF file	Recommended values
IfType	GSM-based devices must specify IF_TYPE_WWANPP. CDMA-based devices specify IF_TYPE_WWANPP2. The value must match the *IfType value specified in the miniport driver's INF file.
MediaType	The value must match the *MediaType value specified in the miniport driver's INF file. For example, either **NdisMediumWirelessWan** or **NdisMedium802_3**.
PhysicalMediumType	The value must match the *PhysicalMediaType value specified in the miniport driver's INF file. The value must be **NdisPhysicalMediumWirelessWan**.
AccessType	If the value in MediaType is specified as **NdisMediumWirelessWan**, specify **NET_IF_ACCESS_POINT_TO_POINT** for AccessType. If MediaType is **NdisMedium802_3**, specify NET_IF_ACCESS_BROADCAST.

MB Raw IP Packet Processing Support

Article • 03/14/2023

MB miniport drivers that support Raw IP packet frames in their send/receive data path should observe the following guidelines:

Net buffer list (NBL) flags for RAW IP packet processing

- For IPv4 packets:

  The **NblFlags** member of the NET_BUFFER_LIST structure must be set to NDIS_NBL_FLAGS_IS_IPV4.

  The **NetBufferListFrameType** member of the NET_BUFFER_LIST structure must be set to 0x0800 (Ethertype IPv4) in network byte order.

- For IPv6 packets:

  The **NblFlags** member of NET_BUFFER_LIST structure must be set to NDIS_NBL_FLAGS_IS_IPV6.

  The **NetBufferListFrameType** member of the NET_BUFFER_LIST structure must be set to 0x86dd (Ethertype IPv6) in network byte order.

Miniport drivers can use the NdisSetNblFlag macro to set flags in the net buffer list. The following line demonstrates how to set IPv4 packet flag in the net buffer list:

```C++
NdisSetNblFlag(pNbl, NDIS_NBL_FLAGS_IS_IPV4);
```

Miniport drivers can use the NET_BUFFER_LIST_INFO to get and set information in a net buffer list. The following line demonstrates how to modify the **NetBufferListFrameType** OOB in the network buffer list for IPV4 packets:

```C++
Value = ConvertToNetworkByteOrder(0x0800);
```

```C++
NET_BUFFER_LIST_INFO(pNbl, NetBufferListFrameType) = Value;
```

Send Path Processing

The MB Service will set these flags in the NBL before passing the list to the miniport driver to send across the network. The miniport driver can verify the flags in the input NBL.

Receive Path Processing

Miniport drivers should set flags in the NBL before passing the NBL to the MB Service for received packets.

If your miniport driver implements Raw IP Packet Processing during its driver development phase, but still has DHCP server spoofing enabled (EnableDhcp = 1), your miniport driver should ensure following:

- The hardware address and its length set in DHCP response from the miniport driver should match the values of the **CurrentMacAddress** and **MacAddressLength** members specified by the miniport driver in the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure.
- Transaction ID (the **xid** member) of the DHCP response from the miniport driver should match exactly the transaction ID set in the DHCP request message from the client.

Guidelines for MB Miniport Driver IP Address Notifications

Article • 03/14/2023

MB miniport drivers that specify *EnableDhcp* equal to zero in their INF files can use the IP Helper and associated functions in kernel mode to create, change, and delete the IP address:

To use the IP Helper functions in kernel mode, miniport drivers must include the Netioapi.h header file, and link against Netio.lib.

When miniport drivers specify *EnableDhcp* equal to zero they are required to perform the following operations to notify the MB Service about any of the following events:

- Set IP address for the MB interface
- Set default gateway address
- Update DNS addresses

IP addresses and default gateways that are set by using the IP Helper API persist network connect or disconnect events, or both. Therefore, if the new IP address or default gateway, or both, values are different than the values currently set, the miniport driver should first clear the previous values before setting new values on a network connection event.

Note Miniport drivers can find the **LUID** and **Index** of the MB interface from the **NetLuid** or **IfIndex** members of **NDIS_MINIPORT_INIT_PARAMETERS** structure that is passed to the miniport driver's *MiniportInitializeEx* function.

Resetting the IP Address and Gateway Address

Certain changes to the TCP/IP stack, such as the loading of a mandatory filter driver, can remove the IP and gateway addresses set by the IP helper functions. Miniport drivers must reset the IP and gateway addresses if changes to the TCP/IP stack remove the settings.

Miniport drivers should use following procedure to be notified when the addresses are removed, and must be reset again.

1. During **driver initialization**, miniport drivers should specify a callback function to register for IP interface change notifications using **NotifyIpInterfaceChange**.

Windows will call the function wheneven an IP interface is added, deleted or changed.

2. During **adapter initialization**, miniport drivers should save in miniport driver's local adapter context the **LUID** value from the **NDIS_MINIPORT_INIT_PARAMETERS** structure that is passed to the miniport driver's *MiniportInitializeEx* function. The value contains the *NetLuid* which identifies adapter's interface, which is used in the notification callback.

3. In the **notification callback**, Windows passes the following parameters to the notification function registered with **NotifyIpInterfaceChange**:

   - A pointer to a **MIB_IPINTERFACE_ROW** structure, which contains the *NetLuid* of the miniport adapter's interface.
   - The type of notification, which can be **MibAddInstance**, **MibDeleteInstance** or **MibParameterNotification**.

   Miniport drivers should reset the IP and gateway addresses when the adapter is in a connected state, and the notification type is **MibAddInstance**, and the *NetLuid* in **MIB_IPINTERFACE_ROW** corresponds to one of the miniport driver's adapters, which was saved during adapter initialization.

   Miniport drivers should then follow the Setting the IP Address for the MB Interface and Setting Default Gateway Address procedures to reset the respective addresses.

4. During **driver unload**, miniport drivers should unregister the notification callback function using the **CancelMibChangeNotify2** IP helper function.

Setting the IP Address for the MB Interface

To set an IPv4 address, use the following procedure. You can use similar IP Helper functionality to set an IPv6 address.

1. Use the **GetUnicastIpAddressTable** IP Helper function to find all the IP address entries in the system.

2. For each entry whose **InterfaceLuid** value matches the **InterfaceLuid** of the MB interface:
   a. Find the IP address entry that matches the IP address used in previous connection. First time connections will not have a previous IP address.
   b. If the new IP address is different than the previous IP address, delete the IP address entry for previous connection IP addresses by using the **DeleteUnicastIpAddressEntry** IP Helper function.

   c. If the new IP address is the same as the previous IP address, verify that the desired entry already exists.

3. If the miniport driver did not find the desired IP address entry in the previous loop, it should add a new entry.

   a. Use the InitializeUnicastIpAddressEntry IP Helper function to initialize a MIB_UNICASTIPADDRESS_ROW structure and set the following members of the structure:
      i. Set the **InterfaceLuid** or **InterfaceIndex** members, as appropriate.
      ii. Set the **OnlinePrefixLength** member. This is the number of bits that have a value of one in the subnet mask. For example, if the subnet mask is 255.255.255.0, **OnlinePrefixLength** should be 24.
      iii. Set the **Address** member.
      iv. Set the **PrefixOrigin** member to **IpPrefixOriginManual**.

   b. Pass the initialized MIB_UNICASTADDRESS_ROW structure to the CreateUnicastIpAddressEntry IP Helper function to create the IP address entry.

Setting Default Gateway Address

To set an IPv4 gateway address, use the following procedure. You can use similar IP Helper functionality to set an IPv6 gateway address.

1. Use GetIpForwardTable2 IP Helper function to obtain all the routing entries in the system.

2. For each entry whose **InterfaceLuid** value matches the **InterfaceLuid** value of the MB interface and **DestinationPrefix** is "0.0.0.0/0", call the DeleteIpForwardEntry2 IP Helper function to delete the route if **NextHop** is not equal to the new gateway address. Otherwise, the routing entry is already in the system.

3. If the miniport driver did not find the desired routing entry in the previous loop, it should add a new entry by using the InitializeIpForwardEntry IP Helper function to initialize a MIB_IPFORWARD_ROW2 structure. Initialize the following members of the structure:

   InterfaceLuid or **InterfaceIndex** .

   Set **DestinationPrefix** to 0.0.0.0/0 for default gateway. (Prefix = 0.0.0.0 and PrefixLength = 0)

   Set **NextHop** to the IP address of the default gateway.

Other members are set to default values during initialization. Miniport drivers should use default values for those members.

4. Pass the **MIB_IPFORWARD_ROW2** structure to the **CreateIpForwardEntry2** IP Helper function to set a new default gateway address.

To Set DNS Addresses

- Set the **NameServer** registry key as described in MB DNS Updates to notify Windows about updated DNS addresses.

MB Miniport driver Error Logging

Article • 03/14/2023

MB miniport drivers should perform the following checks in their *MiniportInitializeEx* function, such as:

- The presence of the correct device firmware version required to support the MB driver model.
- An available COM port to communicate with the device.
- No resource conflicts.

If a miniport driver fails to obtain resources it requires, it should return NDIS_STATUS_RESOURCES from its MiniportInitializeEx function. Miniport drivers should call **NdisWriteErrorLogEntry** to log the details of failure to the Windows Event Log.

Miniport drivers should specify the error code in the first element of the last parameter in the call to NdisWriteErrorLogEntry (a variable-sized array of ULONGs) according to the information in the following table.

Error code	Description
WWAN_ERROR_UNSUPPORTED_FIRMWARE	The device is running an unsupported firmware version.
WWAN_ERROR_COM_PORT_CONFLICT	Unable to open COM port for communicating with the device.
WWAN_ERROR_RESOURCE_CONFLICT_OTHER	Any other resource conflict.

Miniport drivers can put other values in the remainder of the elements of variable-sized array.

MB Miniport Driver Performance Requirements

Article • 03/14/2023

The following table describes the expectations for MB miniport drivers to respond to different MB operations. For the best experience, miniport drivers should complete operations within the time identified in the "Best case time (sec)" column.

MB operation	Best case time (sec)	Worst case time (sec)
Time for device to initialize (to reach **WwanReadyStateInitialized**) after being inserted into the machine (OID_WWAN_READY_INFO)	1	5
Manual network registration (OID_WWAN_REGISTER_STATE)	0	50
Network scan operation (OID_WWAN_VISIBLE_PROVIDERS)	2	200
Packet-attach operation (OID_WWAN_PACKET_SERVICE)	1	5
Packet-detach operation (OID_WWAN_PACKET_SERVICE)	1	5
PDP activation (OID_WWAN_CONNECT)	2	10
PDP deactivation (OID_WWAN_CONNECT)	1	10
Update system with IP address, default gateway, and DNS address	1	5
NDIS_STATUS_LINK_STATE notification after PDP activation	2	10
Completion of the following PIN operations (OID_WWAN_PIN): Enter Enable Disable Change	1	4

MB operation	Best case time (sec)	Worst case time (sec)
Query OID_WWAN_PIN to get the current PIN state of the MB device	1	4
OID_WWAN_PIN_LIST response to get a list of supported PIN types	1	4
Time for SMS subsystem to be ready (should send the NDIS_STATUS_WWAN_SMS_CONFIGURATION unsolicited indication)	1	60
Time for miniport driver to complete all other WWAN OIDs except OID_WWAN_VENDOR_SPECIFIC which are not covered in this table	1	15

MB Device Readiness

Article • 03/14/2023

This topic describes the procedures to ensure that an MB device is accessible and ready to be used for network-related activities before the MB Service proceeds to setup data connections. The device is ready to use when the user subscription has been activated and subscriber-related information stored to the device or the Subscriber Identity Module (SIM card)

The MB Service assumes that a miniport driver automatically initializes its MB device's hardware (radio stack, SIM card or equivalent circuitry) after the system has loaded it, without waiting for any instruction from the service.

Miniport drivers set the initial ready-state of their MB device to **WwanReadyStateOff**. As they proceed with initializing, miniport drivers must send event notifications to inform the MB Service of changes to their device's ready state.

Miniport drivers must stop the initialization process if they run into any error conditions. After the error condition is cleared, miniport drivers can resume the initialization process until their device has reached the **WwanReadyStateInitialized** ready-state.

The following are examples of some error scenarios:

- If the device requires a SIM card and the miniport driver detects that no SIM card is present, the miniport driver must send a **WwanReadyStateSimNotInserted** ready-state event notification, and the miniport driver must remain in that state until the user inserts a SIM card into the device.

- If the device requires a SIM card and the miniport driver cannot read the SIM card that has been inserted (for example, a U-RIM is inserted into a GSM-based device or a USIM is inserted into a CDMA-based device) or the SIM card is not compatible with the device (for example, a 3G USIM is inserted into a 2G device, which cannot interpret the USIM format), the miniport driver must send a **WwanReadyStateBadSim** ready-state event notification, and the miniport driver must remain in that state until the user inserts a correct SIM card into the device.

- If the device is locked by the PIN (for devices that use SIM cards) or by a password (for devices that do not use SIM cards) that prevents further device initialization progress, the miniport driver must send a **WwanReadyStateDeviceLocked** ready-state event notification, and the miniport driver must remain in that state until the user enters the correct PIN or password.

- If the miniport driver detects that service activation is required to proceed, the miniport driver must send a **WwanReadyStateNotActivated** ready-state event notification, and it must remain in that state until the service has been activated. This is typical behavior for CDMA-based devices in North America.
- If the miniport driver runs into failures other than the ones mentioned previously, the miniport driver must send a **WwanReadyStateFailure** ready-state event notification, and it must remain in that state until the problem has been identified and corrected.

Be aware that the MB Service does not assume that miniport drivers can detect all these errors. Nor does the service assume the order in which miniport drivers detect these error conditions. However, it is best to implement the error scenarios in the order listed previously.

Until a miniport driver sends a **WwanReadyStateInitialized** ready-state event notification, the service will not proceed any further with network-related activities until the problem has been identified and corrected. However, the service may still send OIDs to the miniport driver.

Miniport drivers do not need to wait for the SMS subsystem to be ready before reporting the **WwanReadyStateInitialized** ready-state. Instead, miniport drivers should send a separate OID_WWAN_SMS_CONFIGURATION notification when the SMS subsystem is ready to send and receive SMS messages.

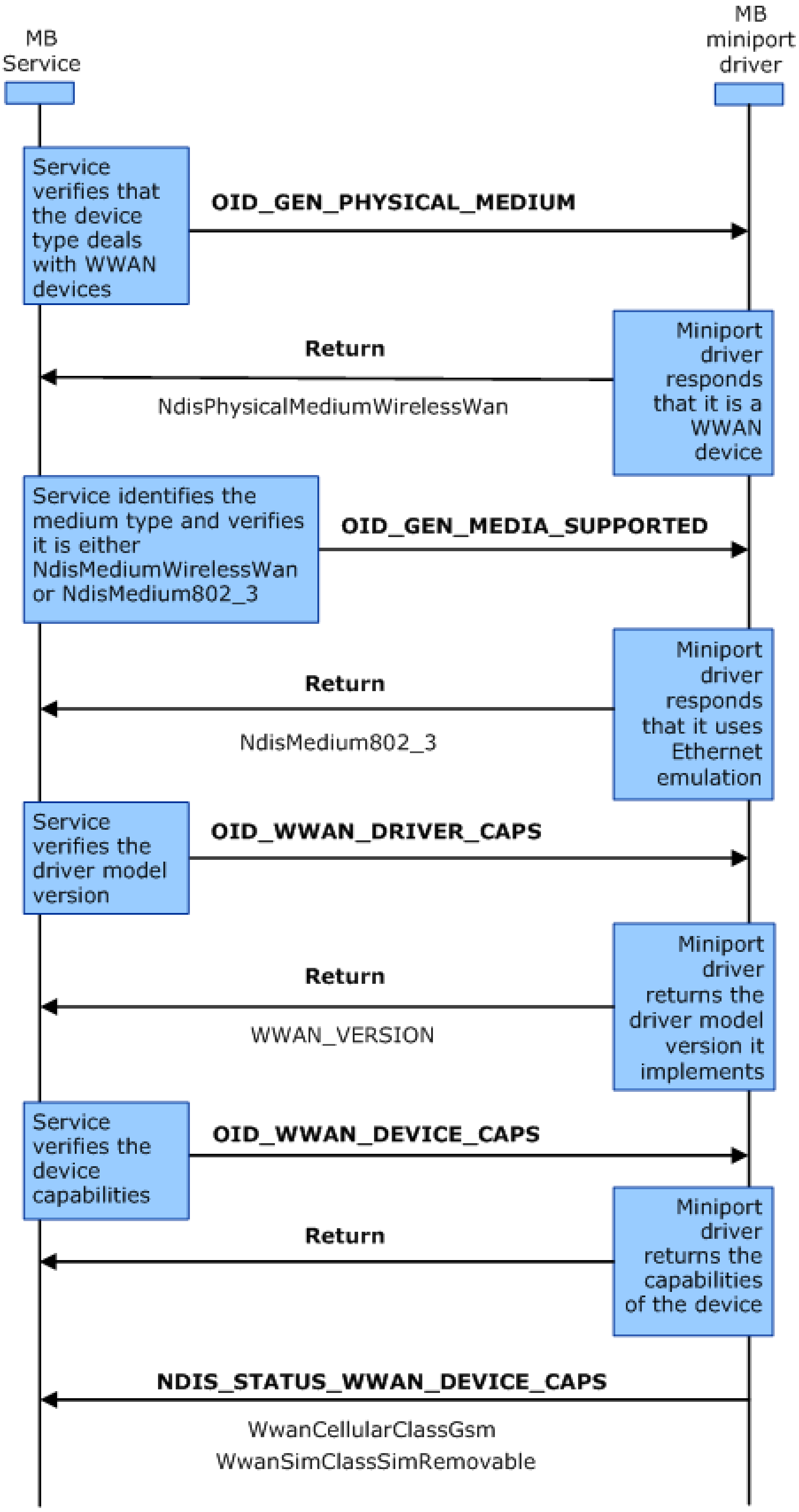
Emergency Mode Support

If the miniport driver indicates that it supports emergency call services while processing OID_WWAN_READY_INFO the miniport driver must set the **EmergencyMode** member of the **WWAN_READY_INFO** structure to **WwanEmergencyModeOn**. In this case, the miniport driver should continue to send registration notifications to the MB Service, but the service will not invoke any automatic configuration related functionalities.

Miniport drivers can specify that they support emergency call services even in scenarios where they detect that the SIM is no longer valid, perhaps because the subscription is unpaid, or service has been deactivated because the device has been reported stolen.

MB Miniport Driver Initialization

The following diagram represents the process taken to determine whether the interface is a qualified MB interface and to gather information about the device capabilities. These steps are performed for each enumerated MB interface when the MB Service starts up, as well as for each new interface arrival while the service is running. The labels in bold represent OID identifiers or transactional flow control. The labels in regular text represent the important flags within the OID structure.


MB Service
MB miniport driver
Service verifies that the device type deals with WWAN devices
OID_GEN_PHYSICAL_MEDIUM
Return
Miniport driver responds that it is a WWAN device
NdisPhysicalMediumWirelessWan
Service identifies the medium type and verifies it is either NdisMediumWirelessWan or NdisMedium802_3
OID_GEN_MEDIA_SUPPORTED
Return
Miniport driver responds that it uses Ethernet emulation
NdisMedium802_3
Service verifies the driver model version
OID_WWAN_DRIVER_CAPS
Return
Miniport driver returns the driver model version it implements
WWAN_VERSION
Service verifies the device capabilities
OID_WWAN_DEVICE_CAPS
Return
Miniport driver returns the capabilities of the device
NDIS_STATUS_WWAN_DEVICE_CAPS
WwanCellularClassGsm
WwanSimClassSimRemovable

To initialize an MB miniport driver, use the following procedure:

1. The MB Service sends a synchronous (blocking) OID_GEN_PHYSICAL_MEDIUM query request to identify the type of the MB device. The miniport driver responds with **NdisPhysicalMediumWirelessWan** to indicate that the MB device is a WWAN device.

2. The MB Service sends a synchronous (blocking) OID_GEN_MEDIA_SUPPORTED query request to the miniport driver to identify what kind of medium the MB device uses. The miniport driver responds with **NdisMedium802_3** to indicate that it uses Ethernet emulation.

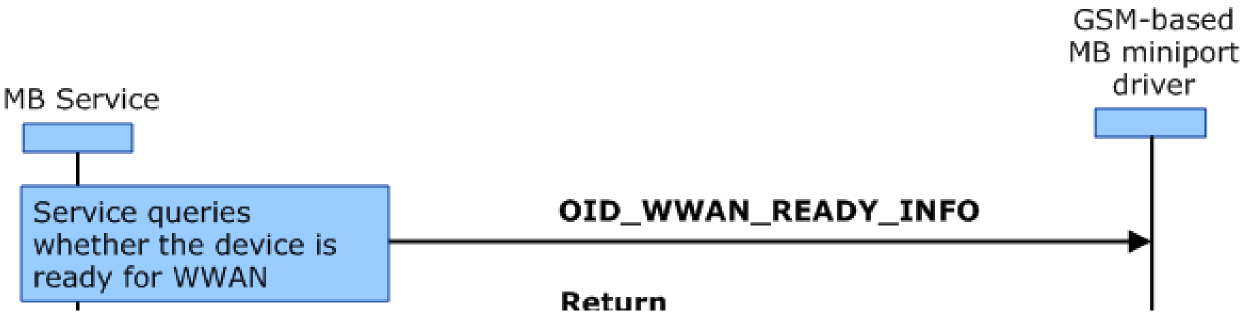
3. The MB Service sends a synchronous (blocking) OID_WWAN_DRIVER_CAPS query request to the miniport driver to identify what driver model version the miniport driver supports. The miniport driver responds with WWAN_VERSION.

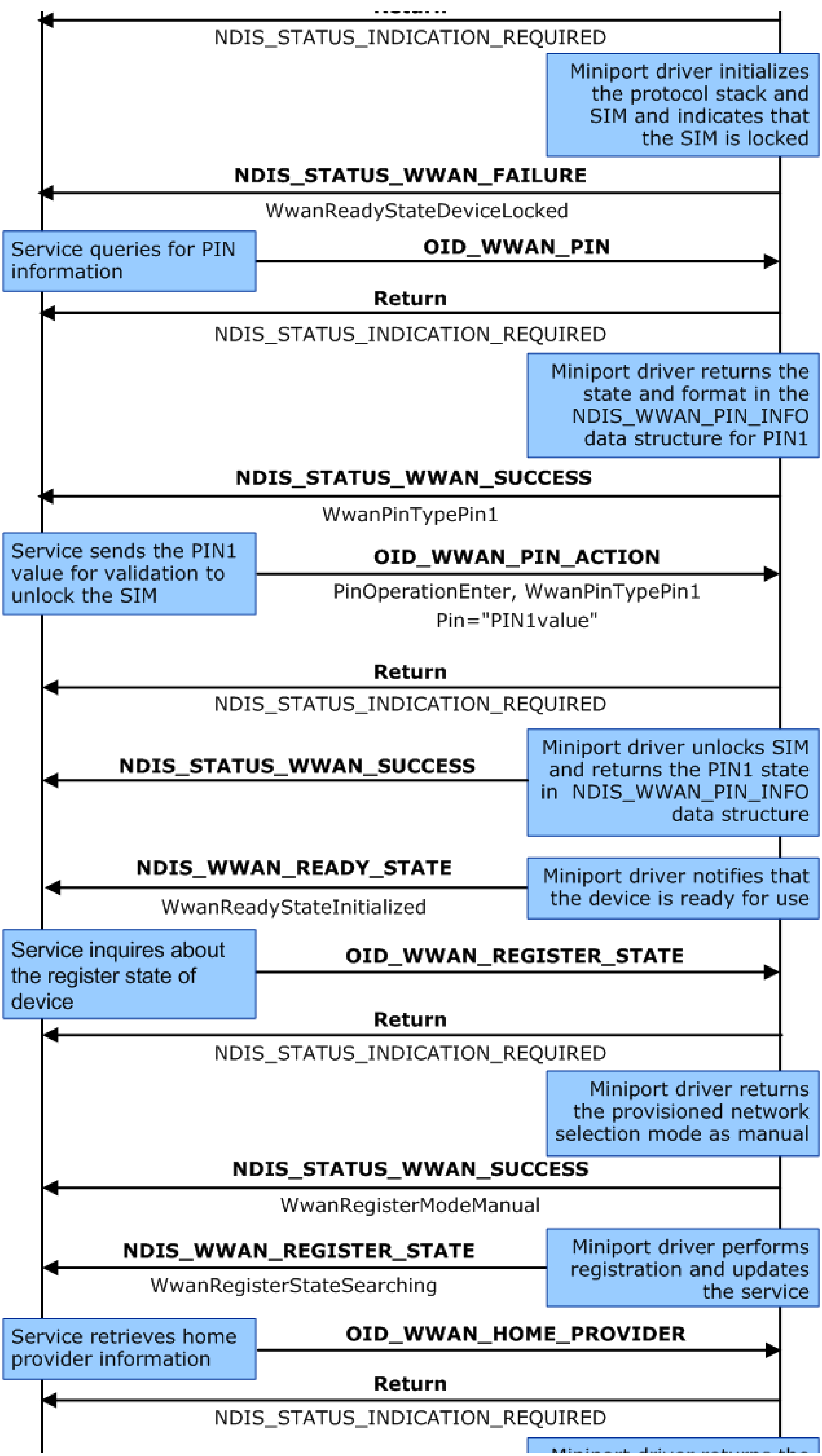
4. The MB Service sends an asynchronous (non-blocking) OID_WWAN_DEVICE_CAPS query request to the miniport driver to identify the capabilities of the MB device. The miniport driver responds with a provisional acknowledgement that it has received the request, and it will send a notification with the requested information in the future.

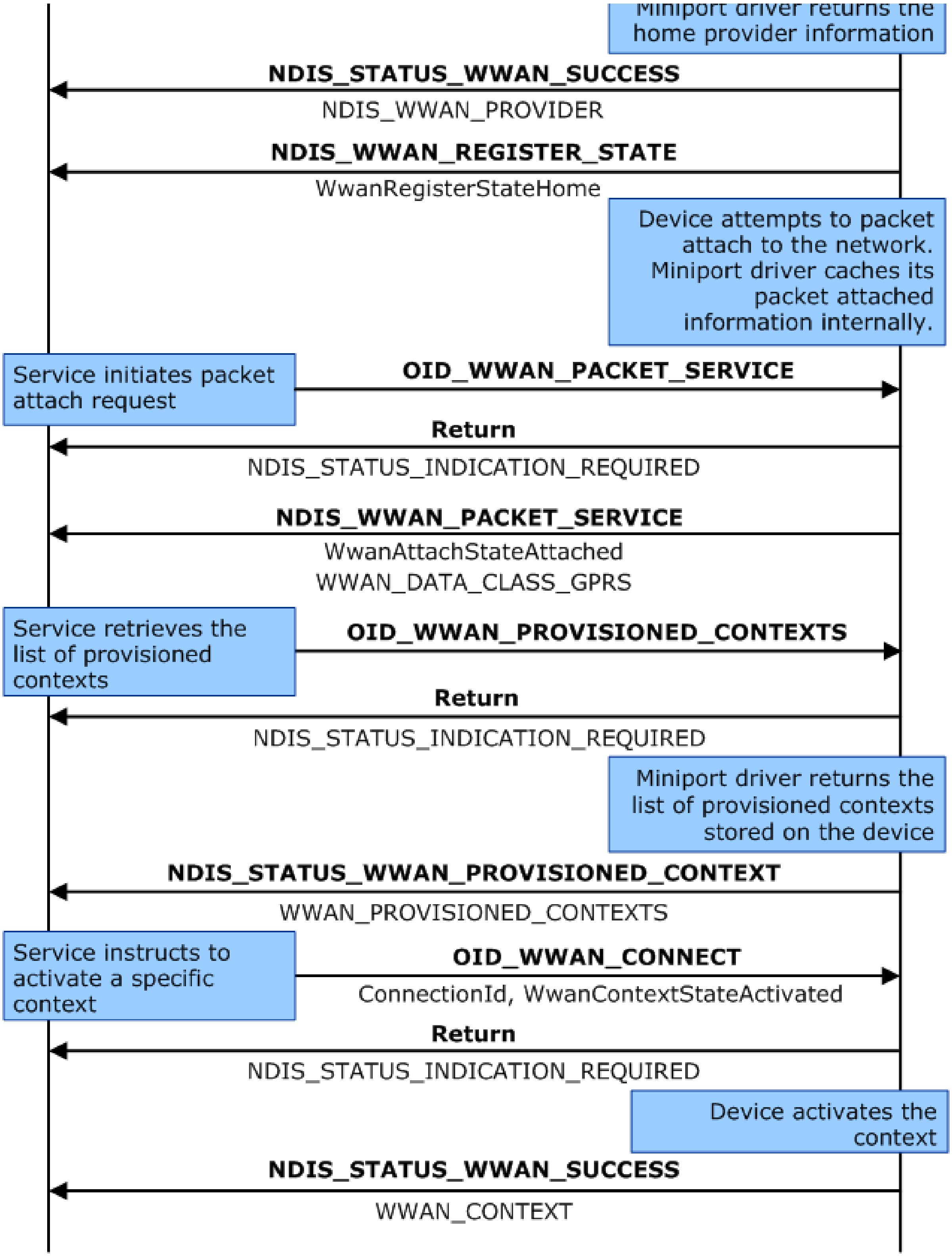
5. The miniport driver sends an **NDIS_STATUS_WWAN_DEVICE_CAPS** notification to the MB Service that indicates the capabilities of the MB device that the miniport driver supports. For example, if the miniport driver supports a GSM-based device, it should specify the **WwanCellularClassGsm** value in the **DeviceCaps.WwanCellularClass** member of the **NDIS_WWAN_DEVICE_CAPS** structure. If the miniport driver supports a CDMA-based device, it should specify **WwanCellularClassCdma**.

Initialization of SIM-Locked GPRS Device with a User-Defined Context

The following diagram illustrates the scenario in which the user enters a SIM PIN and manually configures an access point name string. The labels in bold are OID identifiers or transactional flow control, and the labels in regular text are the important flags within the OID structure.


GSM-based MB miniport driver
MB Service
Service queries whether the device is ready for WWAN
OID_WWAN_READY_INFO
Return


NDIS_STATUS_INDICATION_REQUIRED
Miniport driver initializes the protocol stack and SIM and indicates that the SIM is locked
NDIS_STATUS_WWAN_FAILURE
WwanReadyStateDeviceLocked
Service queries for PIN information
OID_WWAN_PIN
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver returns the state and format in the NDIS_WWAN_PIN_INFO data structure for PIN1
NDIS_STATUS_WWAN_SUCCESS
WwanPinTypePin1
Service sends the PIN1 value for validation to unlock the SIM
OID_WWAN_PIN_ACTION
PinOperationEnter, WwanPinTypePin1
Pin="PIN1value"
Return
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_SUCCESS
Miniport driver unlocks SIM and returns the PIN1 state in NDIS_WWAN_PIN_INFO data structure
NDIS_WWAN_READY_STATE
WwanReadyStateInitialized
Miniport driver notifies that the device is ready for use
Service inquires about the register state of device
OID_WWAN_REGISTER_STATE
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver returns the provisioned network selection mode as manual
NDIS_STATUS_WWAN_SUCCESS
WwanRegisterModeManual
NDIS_WWAN_REGISTER_STATE
WwanRegisterStateSearching
Miniport driver performs registration and updates the service
Service retrieves home provider information
OID_WWAN_HOME_PROVIDER
Return
NDIS_STATUS_INDICATION_REQUIRED


Miniport driver returns the home provider information
NDIS_STATUS_WWAN_SUCCESS
NDIS_WWAN_PROVIDER
NDIS_WWAN_REGISTER_STATE
WwanRegisterStateHome
Device attempts to packet attach to the network. Miniport driver caches its packet attached information internally.
Service initiates packet attach request
OID_WWAN_PACKET_SERVICE
Return
NDIS_STATUS_INDICATION_REQUIRED
NDIS_WWAN_PACKET_SERVICE
WwanAttachStateAttached
WWAN_DATA_CLASS_GPRS
Service retrieves the list of provisioned contexts
OID_WWAN_PROVISIONED_CONTEXTS
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver returns the list of provisioned contexts stored on the device
NDIS_STATUS_WWAN_PROVISIONED_CONTEXT
WWAN_PROVISIONED_CONTEXTS
Service instructs to activate a specific context
OID_WWAN_CONNECT
ConnectionId, WwanContextStateActivated
Return
NDIS_STATUS_INDICATION_REQUIRED
Device activates the context
NDIS_STATUS_WWAN_SUCCESS
WWAN_CONTEXT

To initialize a GSM-based device with PIN1 locked, implement the following steps:

1. The MB Service sends an asynchronous (non-blocking) OID_WWAN_READY_INFO query request to the miniport driver to identify the ready state of the device. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.

2. The miniport driver sends an NDIS_STATUS_WWAN_FAILURE notification to the MB Service to indicate to the MB Service that the subscriber identity module (SIM) is

locked.

3. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PIN query request to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.

4. The miniport driver sends an NDIS_STATUS_WWAN_SUCCESS notification to the MB Service.

5. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PIN set request to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.

6. The miniport driver sends an NDIS_STATUS_WWAN_SUCCESS notification to the MB Service.

7. The miniport driver sends an **NDIS_STATUS_WWAN_READY_INFO** notification to the MB Service that indicates to the MB Service that the state of the MB device is **WwanReadyStateInitialized**.

8. The MB Service sends an asynchronous (non-blocking) OID_WWAN_REGISTER_STATE query request to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

9. The miniport driver sends an NDIS_STATUS_WWAN_SUCCESS notification to the MB Service.

10. The miniport driver sends an **NDIS_STATUS_WWAN_REGISTER_STATE** notification to the MB Service.

11. The MB Service sends an asynchronous (non-blocking) OID_WWAN_HOME_PROVIDER query request to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

12. The miniport driver sends an NDIS_STATUS_WWAN_SUCCESS notification to the MB Service.

13. The miniport driver sends an **NDIS_STATUS_WWAN_REGISTER_STATE** notification to the MB Service.

14. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PACKET_SERVICE request to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.

15. The miniport driver sends an **NDIS_STATUS_WWAN_PACKET_SERVICE** notification to the MB Service.

16. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PROVISIONED_CONTEXTS query request to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

17. The miniport driver sends **NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS** to the MB Service.

18. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PROVISIONED_CONTEXTS set request to the MB Service. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

19. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.

See Also

For more information about device readiness, see OID_WWAN_READY_INFO.

For more information about device initialization with provisioned contexts, see MB Provisioned Context Operations.

MB Data Connectivity

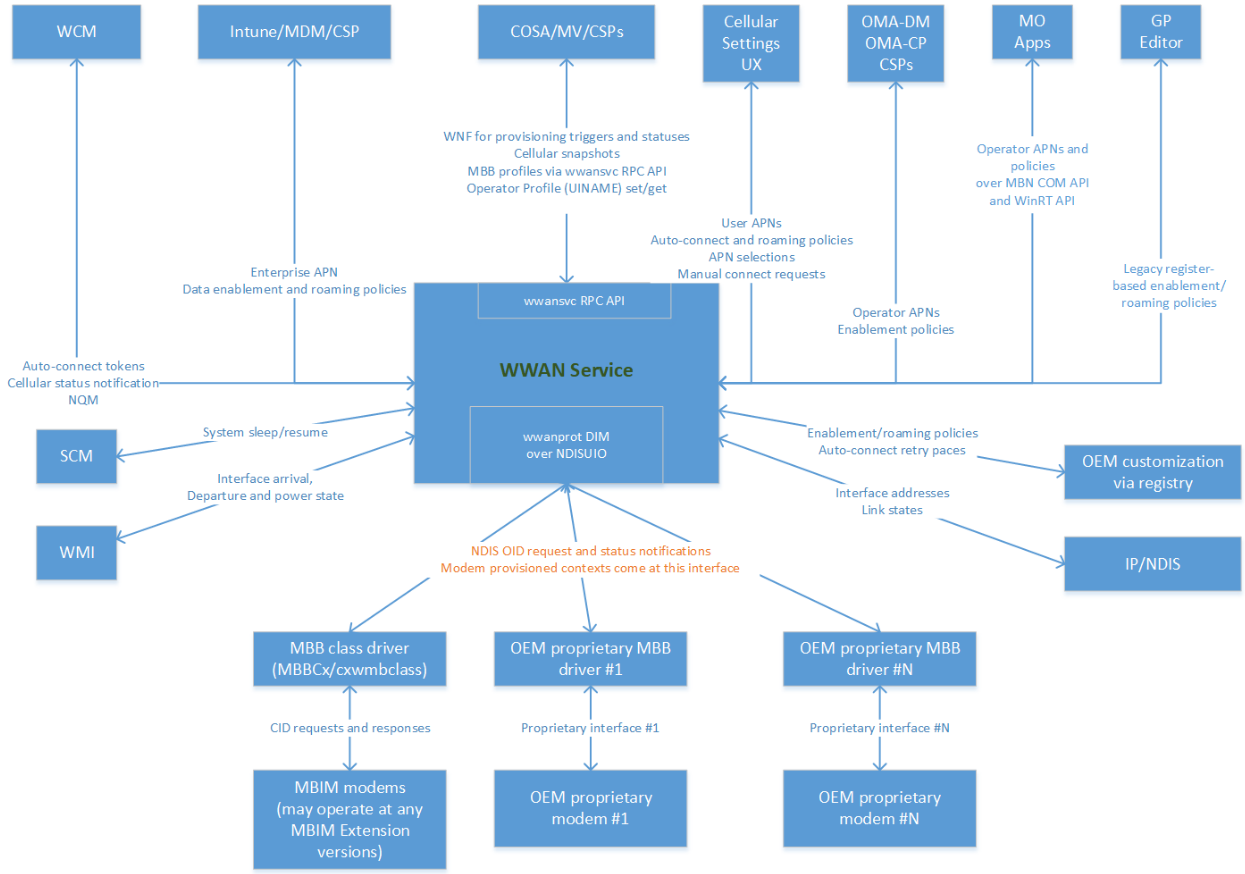
Article • 03/14/2023

Summary

- How data connectivity components interact
  - Cellular Architecture in Windows
  - General block diagram of components involved in basic data connectivity
  - Interactions between the Default Context Controller and its immediate neighbors
- How the Default Context Controller manages the internet data connection
- The data connectivity flows between WWAN Service and the modem
- Hardware Lab Kit (HLK) tests
- Manual Tests for cellular connection
- MB data connectivity troubleshooting guide

Cellular Architecture in Windows

The main component of the cellular stack in the OS is **WWAN Service (WwanSvc)** which controls and sets up all the data connection, states, and events. It interacts with a couple of client drivers to enable activities across the OS.


WCM
Intune/MDM/CSP
COSA/MV/CSPs
Cellular Settings UX
OMA-DM OMA-CP CSPs
MO Apps
GP Editor
WNF for provisioning triggers and statuses
Cellular snapshots
MBB profiles via wwansvc RPC API
Operator Profile (UINAME) set/get
User APNs
Auto-connect and roaming policies
APN selections
Manual connect requests
Operator APNs and policies over MBN COM API and WinRT API
Enterprise APN
Data enablement and roaming policies
wwansvc RPC API
Operator APNs
Enablement policies
Legacy register-based enablement/ roaming policies
Auto-connect tokens
Cellular status notification
NQM
WWAN Service
System sleep/resume
SCM
wwanprot DIM over NDISUIO
Interface arrival,
Departure and power state
WMI
Enablement/roaming policies
Auto-connect retry paces
OEM customization via registry
Interface addresses
Link states
IP/NDIS
NDIS OID request and status notifications
Modem provisioned contexts come at this interface
MBB class driver (MBBCx/cxwmbclass)
OEM proprietary MBB driver #1
OEM proprietary MBB driver #N
CID requests and responses
Proprietary interface #1
Proprietary interface #N
MBIM modems (may operate at any MBIM Extension versions)
OEM proprietary modem #1
OEM proprietary modem #N

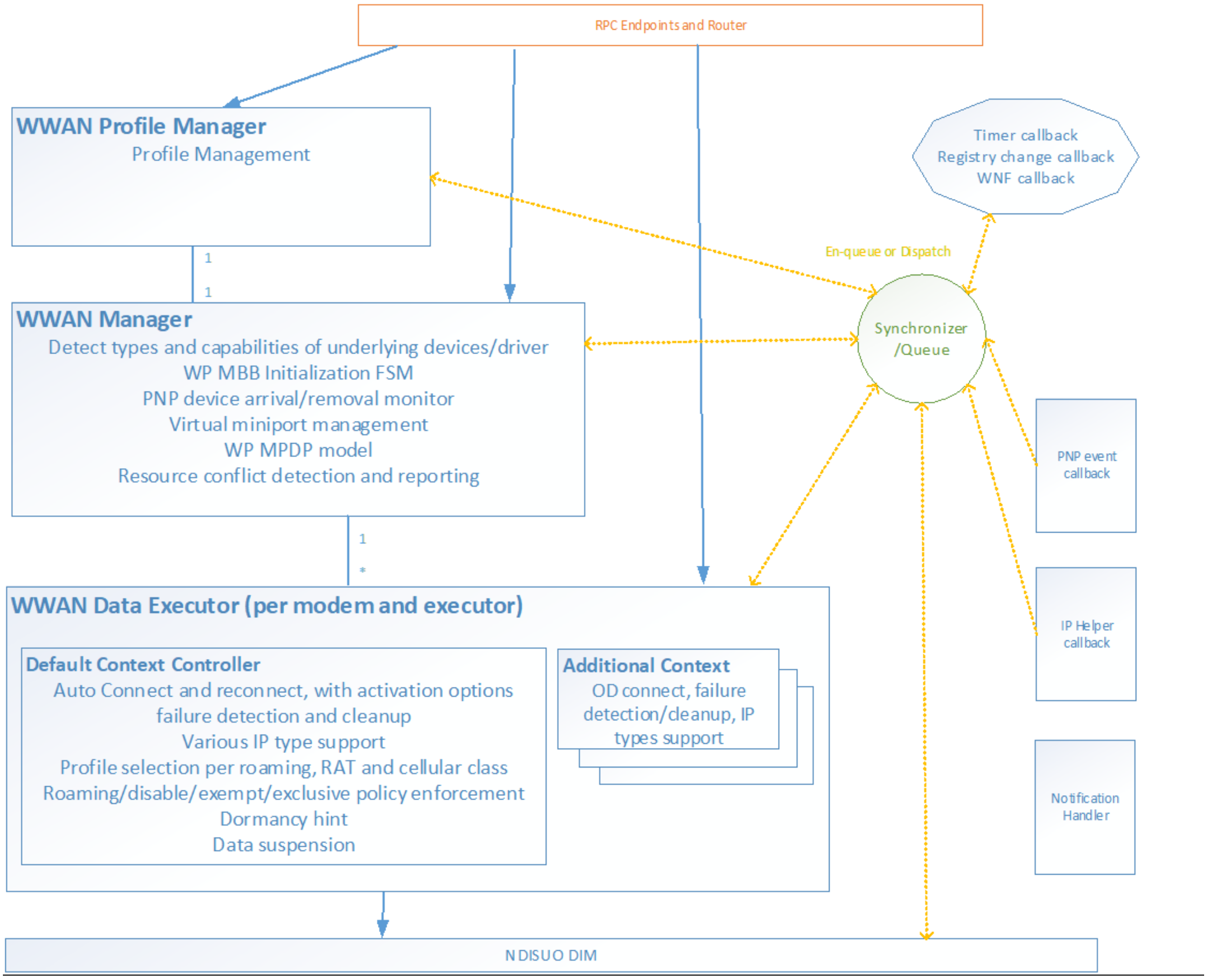
The acronyms in the preceding image:

- **COSA:** Country & Operator Settings Asset
- **CSP:** Configuration Service Provider
- **GP Editor:** Group Policy Editor
- **MDM:** Mobile Device Management
- **MBBCx:** Mobile Broadband WDF class extension
- **MO:** Mobile Operator
- **MV:** Multivariant (framework that associates SIMs with corresponding data from the COSA database)
- **NDISUIO:** NDIS Usermode I/O
- **NQM:** Network Quiet Mode
- **OEM:** Original Equipment Manufacturer
- **OMA-DM:** Open Mobile Alliance – Device Management
- **OMA-CP:** Open Mobile Alliance – Client Provisioning
- **SCM:** Service Control Manager
- **WCM:** Windows Connection Manager
- **WMI:** Windows Management Instrumentation
- **WNF:** Windows Notification Facility
- **wwanprot DIM:** WWAN Protocol Driver Interface Model
- **wwansvc:** WWAN Service

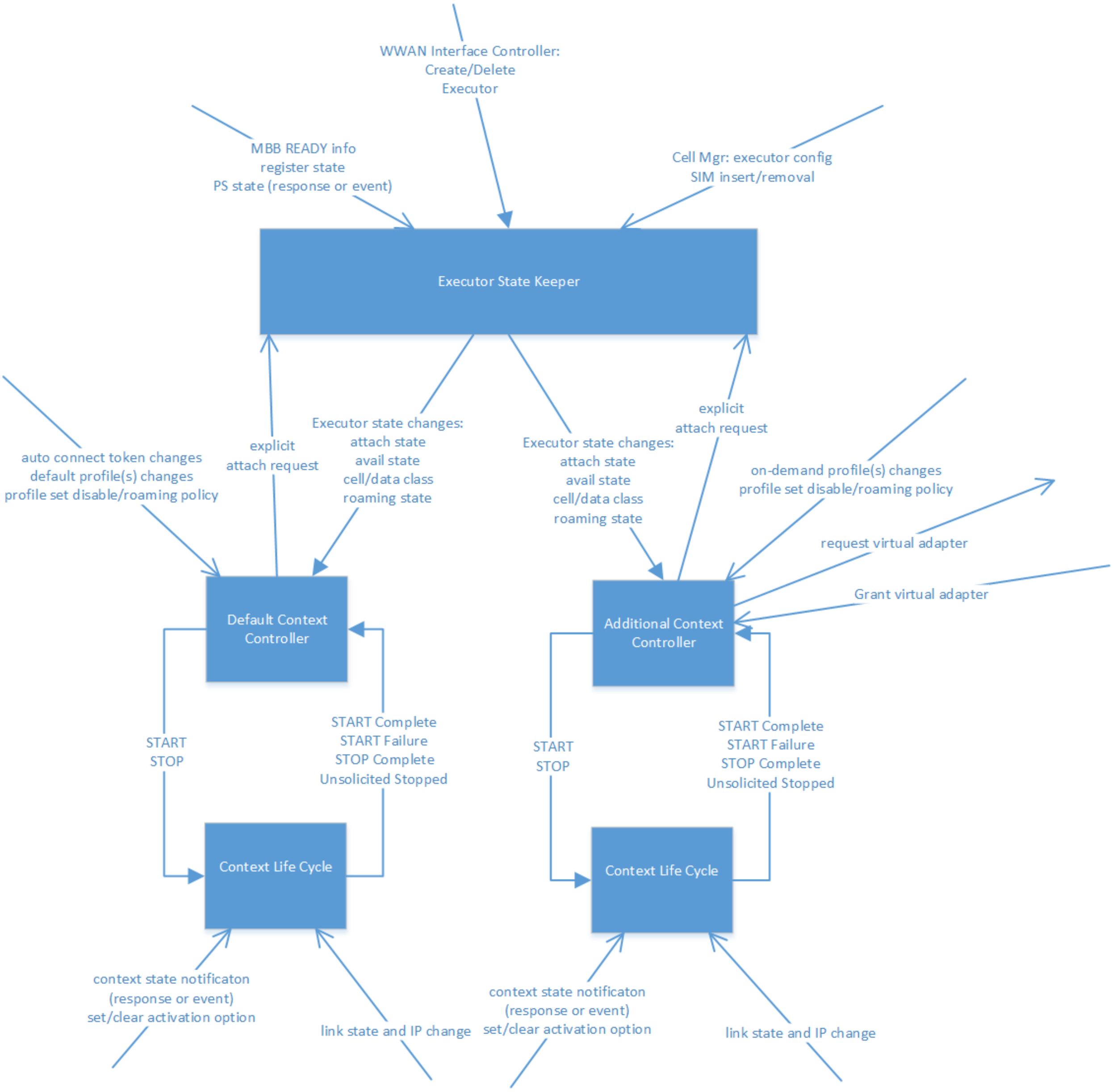
For more information on individual components, see Cellular architecture.

General block diagram of components involved in basic data connectivity

The main state machines resides in the Default Context Controller and its associated Context Life Cycle object.


RPC Endpoints and Router
WWAN Profile Manager
Profile Management
Timer callback
Registry change callback
WNF callback
En-queue or Dispatch
1
1
WWAN Manager
Detect types and capabilities of underlying devices/driver
WP MBB Initialization FSM
PNP device arrival/removal monitor
Virtual miniport management
WP MPDP model
Resource conflict detection and reporting
Synchronizer
/Queue
PNP event callback
1
*
WWAN Data Executor (per modem and executor)
Default Context Controller
Auto Connect and reconnect, with activation options
failure detection and cleanup
Various IP type support
Profile selection per roaming, RAT and cellular class
Roaming/disable/exempt/exclusive policy enforcement
Dormancy hint
Data suspension
Additional Context
OD connect, failure detection/cleanup, IP types support
IP Helper callback
Notification Handler
NDISUO DIM

Interactions between the Default Context Controller and its immediate neighbors



Default Context Controller

The Default Context Controller controls the internet data connection. It manages the cellular data connection base on either auto-connect or manual connect, with or without a profile.

The Default Context Controller performs the following tasks:

- Performs auto-connect, back-off, and auto-retry for cell internet connection
- There is one instance of the Default Context Controller for each primary/physical interface, where each instance:
  - Receives and keeps related policy settings from various sources
  - Receives and keeps related state information (SIM state, reg state, packet service state, iWLAN state, ICCID/IMSI, etc.)
- MBB profile evaluation

    - Evaluates whether an MBB profile is applicable for the current policy settings and cellular states

- In the Vibranium release or older:
    - Keeps track of add/delete/update of related MBB profiles and keeps a list of them
    - Selects profiles for activation (priority rings, previous profile, auto-connect order, LKG profile, purchase profile, provisioned context profile, etc.)

- In the Manganese release:
    - Profile Administrator handles the profile selection for activation

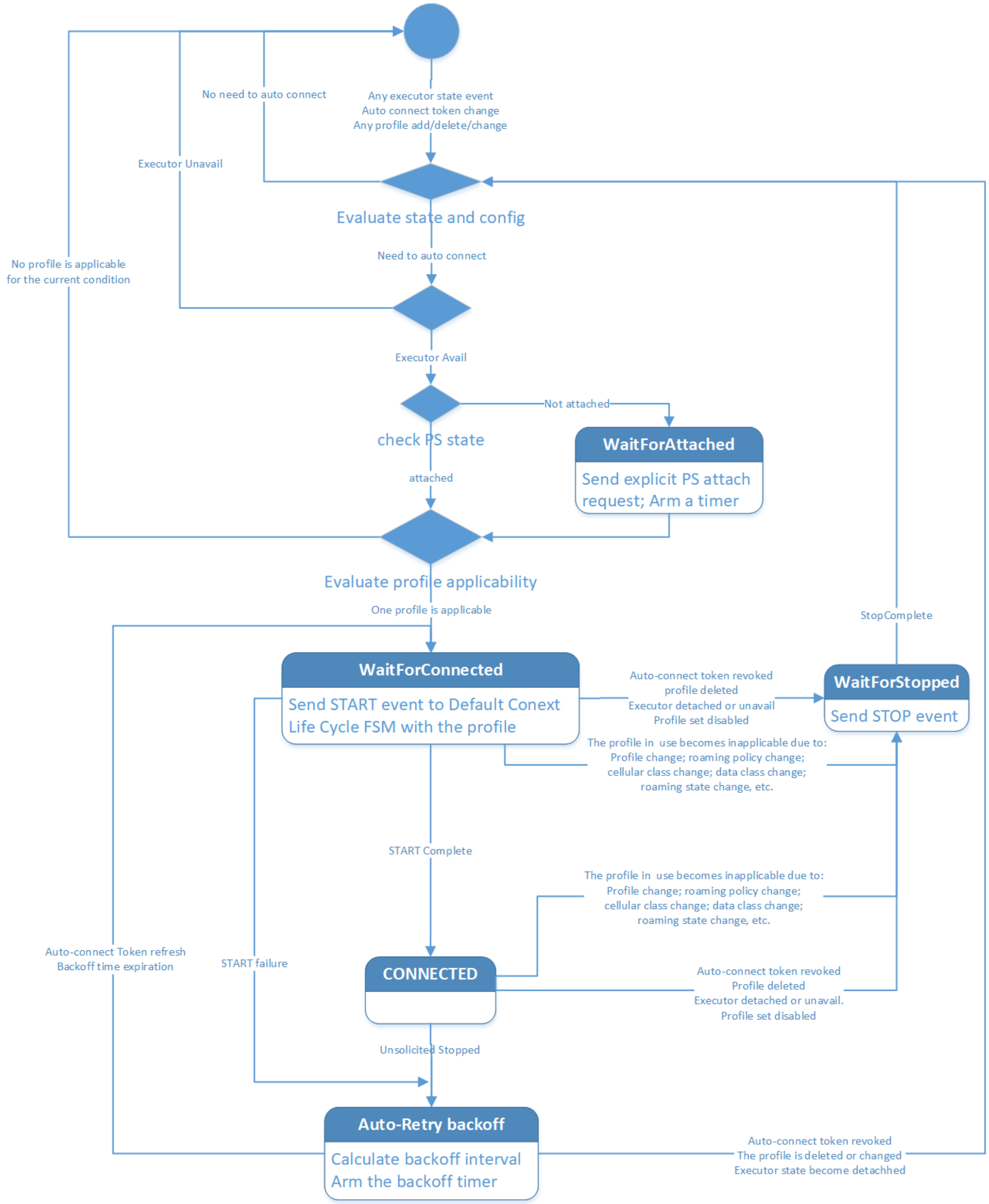
- Back-off interval calculation and timer

- Handles cellular Internet manual connect requests (profile or no-profile mode)

- Uses an instance of the class CWwanContextLifeCycle to activate a connection with an MBB profile

The Default Context Controller uses a finite state machine to manage its tasks.

Finite state machine transitions of the Default Context Controller


No need to auto connect
Any executor state event
Auto connect token change
Any profile add/delete/change
Executor Unavail
Evaluate state and config
No profile is applicable for the current condition
Need to auto connect
Executor Avail
Not attached
check PS state
WaitForAttached
Send explicit PS attach request; Arm a timer
attached
Evaluate profile applicability
One profile is applicable
StopComplete
WaitForConnected
Send START event to Default Conext Life Cycle FSM with the profile
Auto-connect token revoked
profile deleted
Executor detached or unavail
Profile set disabled
WaitForStopped
Send STOP event
The profile in use becomes inapplicable due to:
Profile change; roaming policy change;
cellular class change; data class change;
roaming state change, etc.
START Complete
The profile in use becomes inapplicable due to:
Profile change; roaming policy change;
cellular class change; data class change;
roaming state change, etc.
Auto-connect Token refresh
Backoff time expiration
START failure
CONNECTED
Auto-connect token revoked
Profile deleted
Executor detached or unavail.
Profile set disabled
Unsolicited Stopped
Auto-Retry backoff
Calculate backoff interval
Arm the backoff timer
Auto-connect token revoked
The profile is deleted or changed
Executor state become detachhed

Auto-connect

Policy settings that need to be met for auto-connect

Policy Setting	Config from	Config Unit
EnabledInternet	from users via UI in phones	per system

Policy Setting	Config from	Config Unit
highestConnCategory	from Admin/User/Operator/Device via UI	per interface
ClientDisableAutoConnect	from user via UI in desktops	per interface
OperatorServiceEnablement	from MO via OTA	per interface
GPolicyDisableAutoConnect	group policy via registry	per system
mdmDataEnablementPolicy	from MDM, notified via WNF (OnEnforced/OffEnforced/NoPolicy)	per system
mdmRoamingPolicy	from MDM, notified via WNF (DisabledEnforced/EnabledEnforced/NoPolicy)	per system

States that need to be concerned about auto-connect

State	Value
System power state	S0/S3/S4/D0/D3/D4
device power state	D0/D3/D4
Ready state	Initialized/ICCID
IMSI	affects applicability of IMSI-conditioned profiles
IWLAN state	affect applicability of IWLAN only/OK profiles
Registration state	Home/Roam/Partner
provider ID	may cancel back-off and trigger immediate retry
Packet service state	Detached/Attached
Current data class	may trip highestConnCategory policy and affect applicability of data class conditioned profiles
RnR State	RnR in progress

MB profile applicability for auto-connect

- SimIccID: Must match the ICCID of current SIM at the interface (except for AnyICCID)
- IsAdditionalPdpContextProfile: Must be false (except for purchase profile)
- ConnectionMode: Auto or auto-home
- ProfileCreationType: At or below the highestConnCategory (Admin/User/Operator/Device)
- CellularClass (v4): 3GPP/3GPP2
- RATApplicability (v4): LTE_eHRPD/3GPP_LEGACY
- RoamApplicability (v4): NonPartnerOnly/PartnerOnly/HomeOnly/ HomeAndPartner/PartnerAndNonpartner/AllRoaming; except for iWLAN profile and iWLAN available
- IMSI (v4): If present, must match current IMSI. For multi-app SIMs
- AdminEnable (v4): Is not administratively disabled
- AdminRoamControl (v4): Is not administratively roam-controlled out except for iWLAN profile and iWLAN available

Selection of MBB profiles for auto-connect in VB

- Priority rings:
  - Are based on ProfileCreationType: AdminProvisioned, UserProvisioned, OperatorProvisioned, and DeviceProvisioned.
  - An applicable profile in a higher priority ring excludes all profiles in lower priority rings.
- Modem provisioned profiles:
  - Are based on provisioned contexts.
  - Have the same ring as DeviceProvisioned profiles with subtle details.
- Purchase profiles are special.
- One round of auto-connect and retry attempts:
  - Will try all applicable profiles in the highest priority ring with any applicable profile, plus all applicable purchase profiles.
  - Each profile in one round has at most one chance.
  - If connection with a profile succeeds with a valid IP, the round stops and the profile is designated the last known good (LKG) profile.

Order of profiles in one round of attempts in VB

If one round of attempts has multiple MBB profiles, the order is:

- LKG profile if it is present and is a non-purchase profile.

- Non-purchase modem provisioned profiles. If there are more than one, the order of these profiles is unspecified.
- All non-purchase profiles with explicit AutoConnectOrder, in order of increasing AutoConnectOrder. If an AutoConnectOrder has more than one profiles, the order of these profiles is unspecified.
- All non-purchase profiles with no explicit AutoConnectOrder. If there are more than one, the order of these profiles is unspecified.
- All purchase profiles. If there are more than one, the order of these profiles is unspecified.

Exponential back-off

- Pause for a certain amount of time before retry after failures to activate all applicable MBB profiles in a retry round.
- Commonly used technique in random access media to avoid re-collision after a collision.
- Back-off happens after all profiles in one round of attempts fail to connect.
- There is no back-off between retries of two profiles inside one round.
- The base exponential back-off algorithm: Initial back-off 3 seconds, exponential factor 3, with cap of 24 hours. For example: 3, 9, 27, 81, ...
- Special network cause codes for slow-pace retry (initial back-off 300 seconds):
  - WWAN_ERR_3GPP_SO_NOT_SUBSCRIBED, // 33
  - WWAN_ERR_3GPP_AUTH_FAILURE, // 29
  - WWAN_ERR_3GPP_INSUFFICIENT_RESOURCES, // 26
  - WWAN_ERR_3GPP_UNKNOWN_PDP_ADDRESS_TYPE, // 28
  - WWAN_ERR_3GPP_ACTIVATION_REJECT /
- OEM can customize the initial back-off. Each code can have one of these three catagories:
  - Normal-pace: the same as the base case (3 seconds)
  - Slow-pace: 300 seconds
  - Glacier-pace: 24 hours (practically no retry)

Back-off cancelation or back-off timer expiration

- Back-off can be cancelled and retry commenced immediately in these situations:
  - Auto-connect hint from WCM
  - Auto-connect MBB profiles are added or updated
  - Device roams to a different MO
  - Highest connection category policy is changed

- If a manual connect request comes during back-off, back-off is canceled and manual connect procedure commences.

- Back-off will be cancelled and no auto-connect occurs in these situations:
  - The SIM is removed.
  - Cellular state is no longer available for connect (such as during deregistration or detachment).
  - Auto-connect token is revoked.
  - Cellular data is disabled.
  - Other policy settings are changed such that auto-connect is no longer possible.
  - Later events may re-trigger auto-connect in the event that back-off is cancelled and no auto-connect occurs.

- When the back-off timer expires naturally, retry starts and does the same thing as the initial auto-connect.

Manual Connect

- Bring-up of the data connection is initiated externally via the wwansvc RPC API:
  - In Cellular Setting UI or Network flyout, users uncheck the "let Windows keep this connected" box and then click the Connect button.
  - Starting with Windows 8, WCM may also bring-up the data connection.
  - Manual connect is only allowed if auto-connect is not in progress (idle or back-off).

- The connect request may be issued with or without a specific MBB profile. For Cellular UX since RS2:
  - If a specific MBB profile is given, only that MBB profile is used to connect.
  - If no specific MBB profile is given, the Default Context Controller picks MBB profiles and tries them one by one until the connection is either successfully activated with an MBB profile or all of them fail to connect.

- Is subject to similar set of policy settings as auto-connect.

- Is subject to similar set of cellular state information and restrictions as auto-connect.

- MBB profile applicability is subject to a similar set of rules as for auto-connect with one notable exception:
  - An MBB profile with a ConnectionMode of manual is applicable for manual connect.

- MBB profile selection and order are the same as for auto-connect.

- If no specific MBB profile is given and the MBB profiles in the a round all fail to connect successfully, then the manual connect request is completed with failure. There is no back-off and no retry.

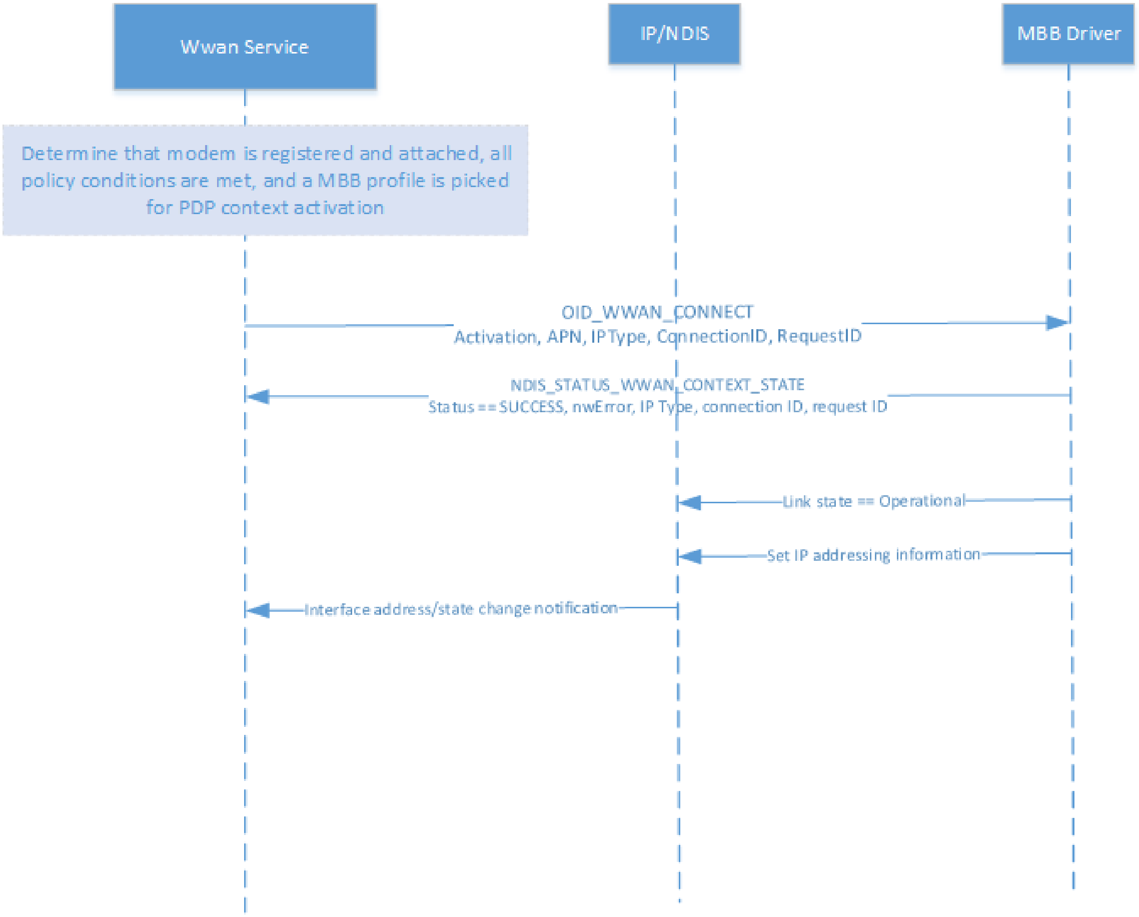
- If a specific MBB profile is given and the MBB profile fails to connect successfully, then the manual connect request is completed with failure. There is no back-off and no retry.

- If a successfully-connected manual connection gets disconnected later unsolicitedly, the state is reported but there is no back-off and no retry.

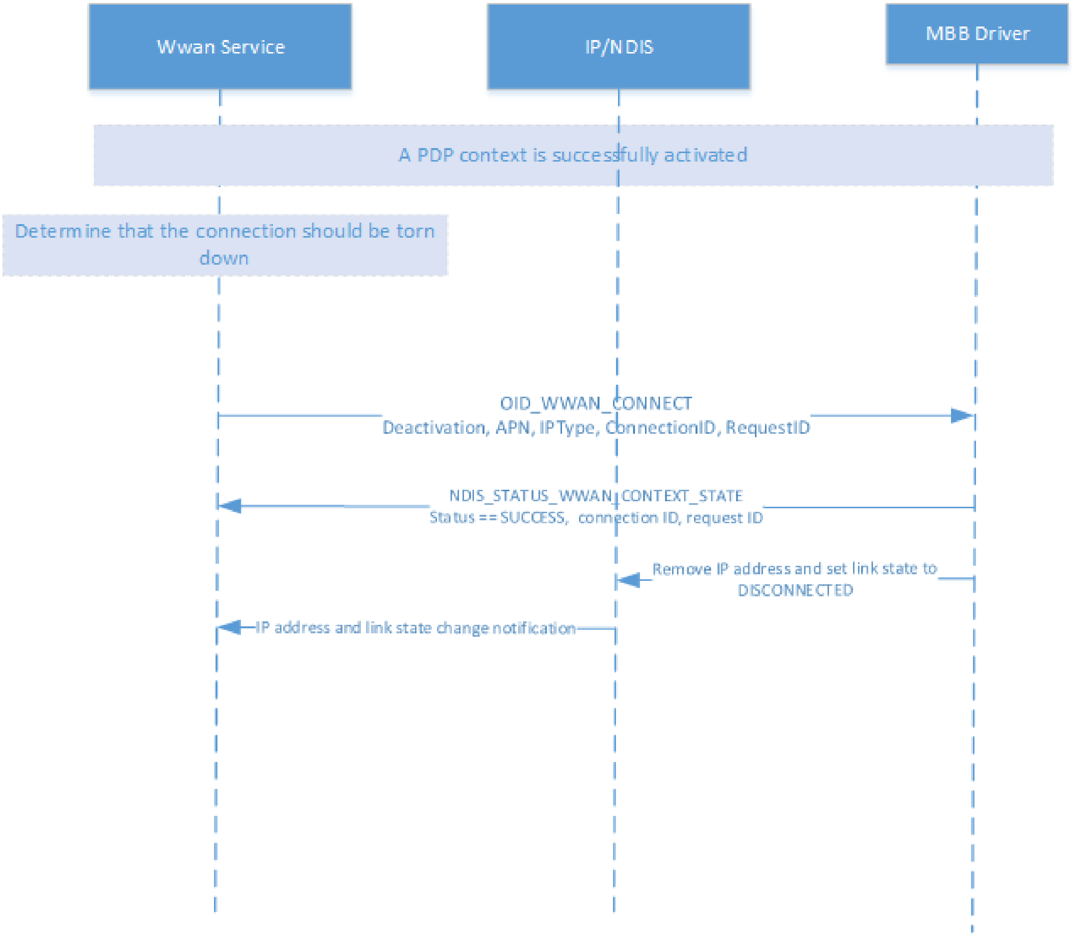
MB data connectivity flows

OID_WWAN_CONNECT is used to initiate the connection with the modem. Below are flows describing the data connection with the modem.

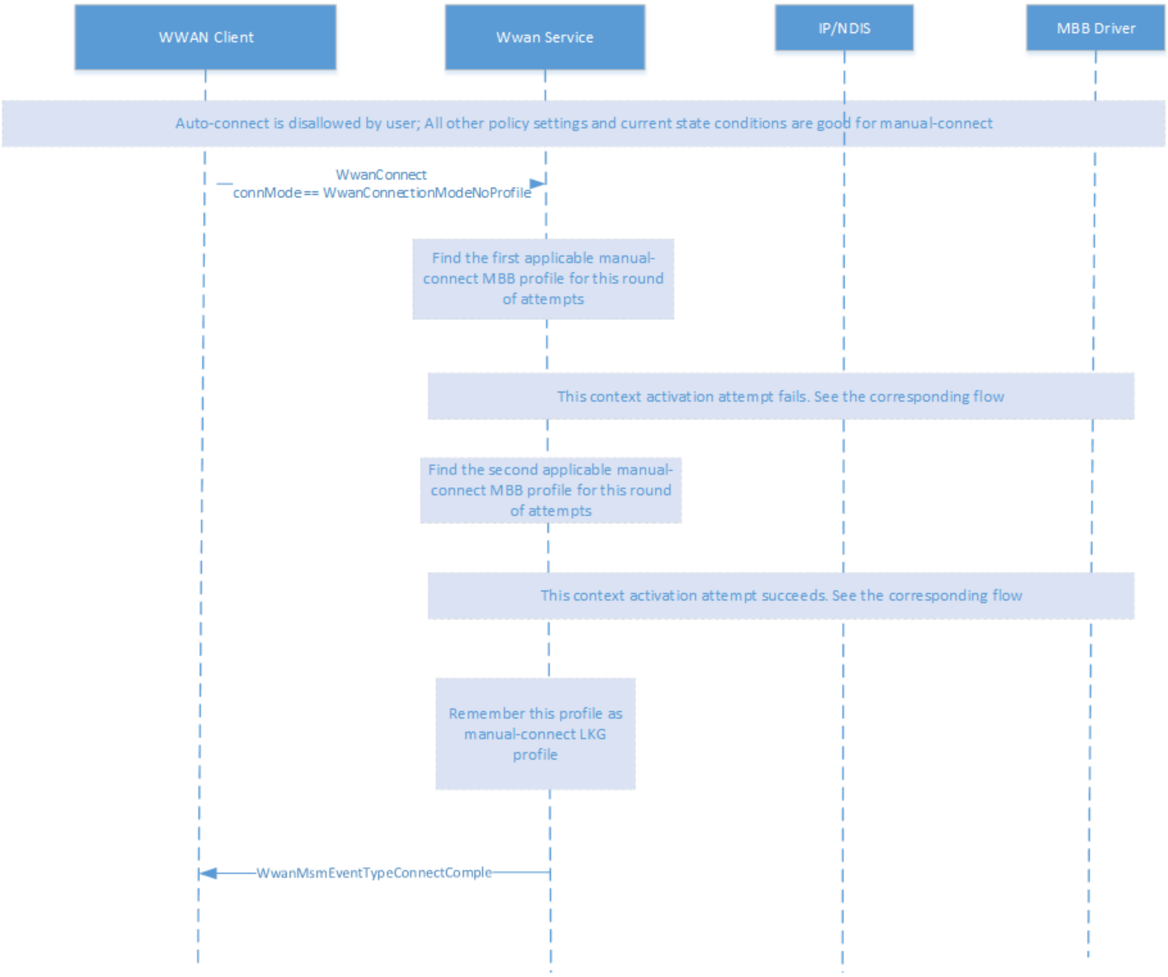
Successful Activation


Wwan Service
IP/NDIS
MBB Driver
Determine that modem is registered and attached, all policy conditions are met, and a MBB profile is picked for PDP context activation
OID_WWAN_CONNECT
Activation, APN, IPType, ConnectionID, RequestID
NDIS_STATUS_WWAN_CONTEXT_STATE
Status == SUCCESS, nwError, IP Type, connection ID, request ID
Link state == Operational
Set IP addressing information
Interface address/state change notification

Successful Deactivation


Wwan Service
IP/NDIS
MBB Driver
A PDP context is successfully activated
Determine that the connection should be torn down
OID_WWAN_CONNECT
Deactivation, APN, IPType, ConnectionID, RequestID
NDIS_STATUS_WWAN_CONTEXT_STATE
Status == SUCCESS, connection ID, request ID
Remove IP address and set link state to DISCONNECTED
IP address and link state change notification

Manual Connect


WWAN Client
Wwan Service
IP/NDIS
MBB Driver
Auto-connect is disallowed by user; All other policy settings and current state conditions are good for manual-connect
WwanConnect
connMode== WwanConnectionModeNoProfile
Find the first applicable manual-connect MBB profile for this round of attempts
This context activation attempt fails. See the corresponding flow
Find the second applicable manual-connect MBB profile for this round of attempts
This context activation attempt succeeds. See the corresponding flow
Remember this profile as manual-connect LKG profile
WwanMsmEventTypeConnectComple

Hardware Lab Kit (HLK) tests

Connect the test machine with ATT SIM to HLK Server.

See Steps for installing HLK .

In HLK Studio, connect to the device Cellular modem driver and run the test: Win6_4.MB.GSM.Data.TestConnect.

Alternatively, run the **TestConnect** HLK testlist by **netsh** and **netsh-mbn-test-installation**.

```
netsh mbn test feature=connectivity param="AccessString=internet"
```

The file showing the HLK test results should have been generated in the directory that the 'netsh mbn test' command was ran.

Manual tests

After reboot, Cellular auto-connects

1. With Wi-Fi toggled off, verify active cellular connection. Systray should show Cellular connection bars and internet browsing should work.
2. Reboot DUT. After reboot, verify there is an active cellular connection. Systray should show Cellular connection bars.

Browse Internet using Cellular data with new SIM

1. Insert SIM card with an active data plan. If the device already has a SIM card, pop out the SIM card and insert a different SIM card from another operator.
2. With Wi-Fi toggled off, verify an active cellular connection. Swipe down from top of screen to bring up quick action center and Systray should show Cellular connection bars and a data icon.

Connect Cellular manually

1. With Ethernet unplugged and Wi-Fi toggled off, uncheck "Let Windows manage this connection" in Cellular settings.
2. Reboot DUT.
3. After boot, open Cellular settings and click Connect to Cellular. Cellular should connect and internet browsing should work.

After wake from hibernation (S4), Cellular auto-connects

1. Ensure "Let Windows manage this connection" is checked in Cellular settings.
2. Put DUT into S4.
3. Wake DUT and verify it automatically establishes a cellular connection. The user should be able to browse the internet.

After wake from hibernation (S4), connect Cellular manually

1. With Ethernet unplugged and Wi-Fi toggled off, uncheck "Let Windows manage this connection" in Cellular settings.
2. In an admin CMD prompt run command: shutdown -h

3. Machine will hibernate. After more than 30 seconds press the machine's power button to wake from hibernation. Log back in, open Cellular settings, and click Connect to Cellular. Cellular should connect and the user should be able to browse the internet.

After wake from screen sleep, Cellular auto-connects

1. With Ethernet unplugged and Wi-Fi toggled off, verify an active cellular connection.
2. (Optional) Allow the screen to sleep. You can set the screen sleep to 1 minute under Settings -> System -> Power & sleep. The setting should not be set to "Never".
3. Wake the screen by using the mouse or keyboard and log back in. Cellular should stay connected and the user should be able to browse internet (also via container for VAIL/WCOS).

MB data connectivity troubleshooting guide

1. Logs can be collected and decoded using these instructions: MB Collecting Logs
2. Open the .txt file in TextAnalysisTool
3. Load the Bacis Connectivity filter

Sample log for disconnect success:

```
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanManager::EnumerateInterfaces Message:  Number of interfaces returned:
1"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanDataExecutor::WwanDisconnect InterfaceGuid:    {f1a7855c-27f0-433d-
9bcd-55e1068c4f41} Message:     connectionID 0x0"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanDefaultContextController::WwanDisconnect Message:  Disconnect
(connectionId:85) Invoked"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanDefaultContextController::fsmEventHandler InterfaceGuid:   {f1a7855c-
27f0-433d-9bcd-55e1068c4f41} Message:   ""entry with state: 4, event: 15
(EXEC 0)"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanDefaultContextController::fsmEventHandler_Connected Message:   manual
disconnecting"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
WwanNhTraceMsmNotification InterfaceGuid:   {f1a7855c-27f0-433d-9bcd-
55e1068c4f41} Message:  ""[NH] Dispatch
```

```
WwanNotificationSourceMsm\WwanMsmEventTypeConnectionIStreamUpdated
ConnectionIStream[Intf={F1A7855C-27F0-433D-9BCD-55E1068C4F41} Prfl[Name=
Guid= Conn=] State[Ready=1 Register=3 Activation=4] contextState NwError =
0x0, apiInfoResult = 0x0]"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanContextLifeCycle::fsmEventHandler Message:     entry with state 4 Event
1"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanContextLifeCycle::CleanUpFull Message:     Starting to Cleanup the
Context LifeCyle"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanContextLifeCycle::SetProfileIndex InterfaceGuid:   {f1a7855c-27f0-433d-
9bcd-55e1068c4f41} Message:     ""set profile index, profile index
20000006"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "   InterfaceGuid=
{f1a7855c-27f0-433d-9bcd-
55e1068c4f41},RequestId=0x8C,,cbPayload=131614,Payload=0x1C00000006000020011
8C01E340300000A000000C8000000983A0000,ErrorCode=The operation completed
successfully."
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
WwanTxSendReq Message:  OID (Code: 23 Type: 0) sent and completed"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
wwanTxmAoAcRefHandler InterfaceGuid:    {f1a7855c-27f0-433d-9bcd-
55e1068c4f41} Message:  Acquiring AoAc Ref for Parent Interface before
sending a TX [0x8d]"
TraceLog    Microsoft-Windows-wmbclass  24:09.5 "Instance:  1 request:
0xFFFFCD067126BF00 OID:     0xE01010C OID name:     OID_WWAN_CONNECT
RequestId:  0x8D RequestHandle:     0x0 Type:   1 InformationLength:
1260"
TraceLog    Microsoft-Windows-wmbclass  24:09.5 "Instance:  1 Request:
0xFFFFCD067126BF00 Status:  The operation that was requested is pending
completion." TraceLog   Microsoft-Windows-wmbclass  24:09.5
"CallerRequestId:   0x8D DriverRequestId:   0 ServiceId:    {00000274-cc33-
a289-bbbc-4f8bb6b0133e} CommandName:    ???¦????BASIC_CONNECT CommandId:
12 InBufferSize:    116 Payload:
0x0000000000000003C0000001A00000058000000A00000064000000100000000000000000
00000000000007E5E2A7E4E6F7272736B656E7E5E2A7E6D006900630072006F0073006F0066
0074002E0063006F006D000000610064006D0069006E0000007000610073007300770072006F0072
006400"
TraceLog    Microsoft-Windows-wmbclass  24:09.5 "Instance:  1MessageType:
0x3 MessageLength:  164 MessageTransactionId:   54TotalFragments:
1CurrentFragment:   0 ServiceId:    {33cc89a2-bbbc-4f8b-b6b0-133ec2aae6df}
CID:    12 CommandType:     1 InfoLength:   116"
TraceLog    Microsoft-Windows-wmbclass  24:09.5 "CallerRequestId:   0x8D
DriverRequestId:    0 ServiceId:    {00000274-cc33-a289-bbbc-4f8bb6b0133e}
CommandName:    ???¦????BASIC_CONNECT CommandId:    12 InBufferSize:    116
Payload:
0x0000000000000003C0000001A00000058000000A00000064000000100000000000000000
00000000000007E5E2A7E4E6F7272736B656E7E5E2A7E6D006900630072006F0073006F0066
0074002E0063006F006D000000610064006D0069006E0000007000610073007300770072006F0072
006400 NdisStatus:  STATUS_SUCCESS"
TraceLog    Microsoft-Windows-wmbclass  24:09.5 "Instance:  1 Request:
0xFFFFCD067126BF00 OID:     0xE01010C OID name:     OID_WWAN_CONNECT
RequestId:  0x8D RequestHandle:     0x0 Type:   1 BytesUsed:    1260
BytesNeeded:    0 Status:   The request will be completed later by NDIS
```

```
status indication."
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
WwanTxSendReq Message:  OID (Code: 12 Type: 0 timeoutInSec: 199) sent to dim
and pending solicited notif"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
WwanTimerWrapper::StartTimer Message:   Timer (ID = 0) Start Completed"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
WwanTxmEvaluateArmTimer InterfaceGuid:  {f1a7855c-27f0-433d-9bcd-
55e1068c4f41} Message:  ""TXM timer armed for 199 seconds expire 0x4e42f9,
TxmHandle=(0x2)"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
_sendReq Message:   ASYNC OID (pTx->handle: 000000000000008D Code: 12) sent
(time 0x4b39a1)"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanContextLifeCycle::SendMbbConnectReq InterfaceGuid:     {f1a7855c-27f0-
433d-9bcd-55e1068c4f41} Message:    OID_WWAN_CONNECT (Deactivate): ReqHandle
0x8d ReqID 0x60 ConnID 0x55 APN [microsoft.com] IPType (sent 0 confg 0) Auth
0 PwdP 1 MediaPref 1 PrefSrc 4"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanContextLifeCycle::StartTimer Message:  Timer Start Completed"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanContextLifeCycle::CleanUpFull Message:     Completed Cleanup of the
Context LifeCyle"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanContextLifeCycle::fsmEventHandler Message:     exit with state 6"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanDefaultContextController::fsmEventHandler InterfaceGuid:   {f1a7855c-
27f0-433d-9bcd-55e1068c4f41} Message:   exit with state 5 (EXEC 0)"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanResetRecovery::fsmEventHandler InterfaceGuid:  {f1a7855c-27f0-433d-
9bcd-55e1068c4f41} Message:     "" entry with state: 3, event: 0"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanResetRecovery::fsmEventHandler InterfaceGuid:  {f1a7855c-27f0-433d-
9bcd-55e1068c4f41} Message:     "" exit with state: 1, event: 0, RnR stage:
0 Potent RnR: 0"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "InterfaceGuid:
{f1a7855c-27f0-433d-9bcd-55e1068c4f41}"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
WwanNhTraceMsmNotification InterfaceGuid:   {f1a7855c-27f0-433d-9bcd-
55e1068c4f41} Message:  [NH] Dispatch
WwanNotificationSourceMsm\WwanMsmEventTypeIStreamChanged (RegistrationState:
3)"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "FunctionCall:
CWwanDataExecutor::GetConnectionInfo InterfaceGuid:     {f1a7855c-27f0-433d-
9bcd-55e1068c4f41} Message:     isPhysi 1 PS 2 isIWLANAvail 0 isConnected 0"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "interfaceGuid:
{f1a7855c-27f0-433d-9bcd-55e1068c4f41}"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "mbnInterface:
{F1A7855C-27F0-433D-9BCD-55E1068C4F41} info:    12301"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 "mbnInterface:
{F1A7855C-27F0-433D-9BCD-55E1068C4F41} info:    MS MBN"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   24:09.5 " Message:
WWAN_INTERFACE_OBJECT::readyObject.readyInfo.ReadyState=1"
```

Sample log for connect success:

```
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanManager::EnumerateInterfaces Message:  Number of interfaces returned:
1"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataExecutor::WwanConnect Message:     ""Connect (connMode:0,
str:!!##MBIMModemProvisionedContextV2InternetProfile##098765432109876)
Invoked"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataExecutor::WwanConnect Message:     ""Connect (flags 0x0,
apiStartTime 4996546 isUserStarted 1 isLowBoxMBAERequest 0"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "InterfaceGuid:
{f1a7855c-27f0-433d-9bcd-55e1068c4f41} ModemIndex:  0 ExecutorIndex:    0
ProfileName:
!!##MBIMModemProvisionedContextV2InternetProfile##098765432109876
ProfileSource:  WwanProfileModemProvisioned connMode:
WwanConnectionModeProfile"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDefaultContextController::IsAllowedByRoamingPolicies Message:  return
TRUE"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWWANContextControllerBase::FillProfileGuidInCIS Message:
[ConnectionIStream] Updated PrflGuid={64CFE041-9925-4109-B738-9C9F7EC95A92}"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDefaultContextController::WwanConnect Message:     manual connection
request: temp conn ID 0x61 APN [microsoft.com]"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDefaultContextController::fsmEventHandler InterfaceGuid:   {f1a7855c-
27f0-433d-9bcd-55e1068c4f41} Message:   ""entry with state: 0, event: 14
(EXEC 0)"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDefaultContextController::IsAllowedByRoamingPolicies Message:  return
TRUE"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataExecutor::DisconnectMatchingAdditionalPdpContexts Message:
""Looking for APN: microsoft.com, IPType: 0"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataResourceManager::CheckResourceMaxContextCountByOEM Message:    non-
CDMA"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataResourceManager::CheckResourceMaxContextCountByOEM Message:
""per IMSI OEM configred MaxNumberOfPDPContexts not found, trying device
settings."""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataResourceManager::CheckResourceMaxContextCountByOEM Message:
""device OEM configred MaxNumberOfPDPContexts not found, using default
settings."""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataResourceManager::SetPdpContextsOEMConfigured Message:  OEMConfig
using 8"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
```

```
CWwanDataResourceManager::UpdatePdpContexts Message:    ""OEMConfiged 8,
Modem supports 17, using 8"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataResourceManager::ExecutorAcquireResourceMessage:   Acquired
Resource Count 1"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
WwanNhTraceMsmNotification InterfaceGuid:   {f1a7855c-27f0-433d-9bcd-
55e1068c4f41} Message:  ""[NH] Dispatch
WwanNotificationSourceMsm\WwanMsmEventTypeConnectionIStreamUpdated
ConnectionIStream[Intf={F1A7855C-27F0-433D-9BCD-55E1068C4F41}
Prfl[Name=!!##MBIMModemProvisionedContextV2InternetProfile##098765432109876
Guid={64CFE041-9925-4109-B738-9C9F7EC95A92} Conn=] State[Ready=1 Register=3
Activation=2] contextState NwError = 0x0, apiInfoResult = 0x0]"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDefaultContextController::StartContextLifeCycleWrapper Message:
Manual connecting on profile
!!##MBIMModemProvisionedContextV2InternetProfile##098765432109876 ConnID 97"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanContextLifeCycle::fsmEventHandler Message:     entry with state 0 Event
0"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanContextLifeCycle::SetProfileIndex InterfaceGuid:   {f1a7855c-27f0-433d-
9bcd-55e1068c4f41} Message:     ""set profile index, profile index
20000006"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "   InterfaceGuid=
{f1a7855c-27f0-433d-9bcd-
55e1068c4f41},RequestId=0x8E,,cbPayload=131614,Payload=0x1C00000006000020011
8C01E340300000A000000C8000000983A0000,ErrorCode=The operation completed
successfully."
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
WwanTxSendReq Message:  OID (Code: 23 Type: 0) sent and completed"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
wwanTxmAoAcRefHandler InterfaceGuid:    {f1a7855c-27f0-433d-9bcd-
55e1068c4f41} Message:  Acquiring AoAc Ref for Parent Interface before
sending a TX [0x8f]"
TraceLog    Microsoft-Windows-wmbclass  25:16.1 "Instance:  1 Request:
0xFFFFCD06728F7160 OID:     0xE01010C OID name:     OID_WWAN_CONNECT
RequestId:  0x8F RequestHandle:     0x0 Type:   1 InformationLength:
1260"
TraceLog    Microsoft-Windows-wmbclass  25:16.1 "Instance:  1 Request:
0xFFFFCD06728F7160 Status:  The operation that was requested is pending
completion."
TraceLog    Microsoft-Windows-wmbclass  25:16.1 "CallerRequestId:   0x8F
DriverRequestId:    0 ServiceId:    {00000281-cc33-a289-bbbc-4f8bb6b0133e}
CommandName:    Ã,ÂªÃ¦ÃŸBASIC_CONNECT CommandId:    12 InBufferSize:    116
Payload:
0x00000000010000003C0000001A00000058000000A0000000640000001000000000000000000
00000000000007E5E2A7E4E6F7272736B656E7E5E2A7E6D00690063 0072006F0073006F0066
0074002E0063006F006D000000610064006D0069006E000000700061007300730077006F0072
006400"
TraceLog    Microsoft-Windows-wmbclass  25:16.1 "Instance:  1 MessageType:
0x3 MessageLength:  164 MessageTransactionId:   55 TotalFragments:  1
CurrentFragment:    0 ServiceId:    {33cc89a2-bbbc-4f8b-b6b0-133ec2aae6df}
CID:    12 CommandType:     1 InfoLength:   116"
TraceLog    Microsoft-Windows-wmbclass  25:16.1 "CallerRequestId:   0x8F
```

```
DriverRequestId:    0 ServiceId:    {00000281-cc33-a289-bbbc-4f8bb6b0133e}
CommandName:    Ã,ÂªÃ¦ÃŸBASIC_CONNECT CommandId:    12InBufferSize:
116Payload:
0x0000000001000000 3C0000001A000000580000000A000000640000001000000000000000000
00000000000007E5E2A7E4E6F7272736B656E7E5E2A7E6D006900630072006F0073006F0066
0074002E0063006F006D000000610064006D0069006E00000070006100730073077006F0072
006400 NdisStatus:  STATUS_SUCCESS"
TraceLog    Microsoft-Win dows-wmbclass 25:16.1 "Instance:  1 Request:
0xFFFFCD06728F7160 OID:     0xE01010C OID name:     OID_WWAN_CONNECT
RequestId:  0x8FRequestHandle:  0x0Type:    1BytesUsed:     1260
BytesNeeded:    0 Status:   The request will be completed later by NDIS
status indication."
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
WwanTxSendReq Message:  OID (Code: 12 Type: 0 timeoutInSec: 181) sent to dim
and pending solicited notif"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
WwanTimerWrapper::StartTimer Message:   Timer (ID = 0) Start Completed"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
WwanTxmEvaluateArmTimer InterfaceGuid:  {f1a7855c-27f0-433d-9bcd-
55e1068c4f41} Message:  ""TXM timer armed for 181 seconds expire 0x4f00ca,
TxmHandle=(0x2)"""
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
_sendReq Message:   ASYNC OID (pTx->handle: 000000000000008F Code: 12) sent
(time 0x4c3dc2)"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanContextLifeCycle::SendMbbConnectReq InterfaceGuid:     {f1a7855c-27f0-
433d-9bcd-55e1068c4f41} Message:    OID_WWAN_CONNECT (Activate): ReqHandle
0x8f ReqID 0x62 ConnID 0x61 APN [microsoft.com] IPType (sent 0 confg 0) Auth
0 PwdP 1 MediaPref 1 PrefSrc 4"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanContextLifeCycle::StartTimer Message:  Timer Start Completed"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanContextLifeCycle::fsmEventHandler Message:     exit with state 2"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDefaultContextController::fsmEventHandler InterfaceGuid:   {f1a7855c-
27f0-433d-9bcd-55e1068c4f41} Message:   exit with state 3 (EXEC 0)
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "InterfaceGuid:
{f1a7855c-27f0-433d-9bcd-55e1068c4f41}"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
WwanNhTraceMsmNotification InterfaceGuid:   {f1a7855c-27f0-433d-9bcd-
55e1068c4f41}Message:   [NH] Dispatch
WwanNotificationSourceMsm\WwanMsmEventTypeIStreamChanged (RegistrationState:
3)"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
CWwanDataExecutor::GetConnectionInfoInterfaceGuid:  {f1a7855c-27f0-433d-
9bcd-55e1068c4f41}Message:  isPhysi 1 PS 2 isIWLANAvail 0 isConnected 0"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "interfaceGuid:
{f1a7855c-27f0-433d-9bcd-55e1068c4f41}"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "mbnInterface:
{F1A7855C-27F0-433D-9BCD-55E1068C4F41}info:     12301"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "mbnInterface:
{F1A7855C-27F0-433D-9BCD-55E1068C4F41}info:     MS MBN"
TraceLog    Microsoft-Windows-WWAN-SVC-EVENTS   25:16.1 "FunctionCall:
_PublishSebNotificationMessage:
WWAN_INTERFACE_OBJECT::readyObject.readyInfo.ReadyState=1"
```

Basic Connectivity Log Filter

Article • 12/15/2021

To load a basic connectivity log filter:

1. Copy and paste the lines below and save them into a text file named "basicconnectivity.tat."

2. Load the filter file into the TextAnalysisTool by clicking File > Load Filters.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<TextAnalysisTool.NET version="2016-01-08" showOnlyFilteredLines="True">
  <filters>
    <filter enabled="y" excluding="n" description="" foreColor="800000"
type="matches_text" case_sensitive="n" regex="n" text="Globals Module
Initialization Completed" />
    <filter enabled="n" excluding="n" description="" foreColor="ff0000"
type="matches_text" case_sensitive="n" regex="n" text="ERROR: " />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="IsAutoConnectPossible" />
    <filter enabled="y" excluding="n" description="" foreColor="800080"
type="matches_text" case_sensitive="n" regex="n" text="SetAcState" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="SetEnablementPolicy" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="SetRoamControlPolicy" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="y" regex="n" text=": entry with state" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text=": exit with state" />
    <filter enabled="n" excluding="n" description="" foreColor="8b008b"
type="matches_text" case_sensitive="n" regex="n" text=" Dispatch
WwanNotificationSourceMsm\WwanMsmEventTypeConnectionIStream" />
    <filter enabled="y" excluding="n" description="" foreColor="006400"
type="matches_text" case_sensitive="n" regex="n" text="OID_WWAN_CONNECT ("
/>
    <filter enabled="y" excluding="n" description="" foreColor="4b0082"
type="matches_text" case_sensitive="n" regex="n" text=" Indicating
NDIS_STATUS_WWAN_CONTEXT_STATE with status=" />
    <filter enabled="y" excluding="n" description="" foreColor="008b8b"
type="matches_text" case_sensitive="n" regex="n"
text="CWwanDataExecutor::OnNdisNotification: NDIS_STATUS_WWAN_CONTEXT_STATE
Resp (" />
    <filter enabled="y" excluding="n" description="" foreColor="b22222"
type="matches_text" case_sensitive="y" regex="n" text="BASIC_CONNECT" />
    <filter enabled="y" excluding="n" description="" foreColor="008080"
type="matches_text" case_sensitive="n" regex="n"
text="NDIS_STATUS_WWAN_CONTEXT_STATE Event" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
```

```
case_sensitive="n" regex="n" text="ound valid v" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="OnIPAddrLinkStateChange" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="WwanContextLifeCycleFSMState_Idle" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="WwanContextLifeCycleFSMState_Connected"
/>
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="data call connected" />
    <filter enabled="n" excluding="y" description="" type="matches_text"
case_sensitive="n" regex="n" text="Qualcomm-WOS-mbb" />
    <filter enabled="n" excluding="n" description="" foreColor="00008b"
type="matches_text" case_sensitive="n" regex="n" text="::OnNdisNotification:
WwanEventCodeRegisterState" />
    <filter enabled="n" excluding="n" description="" foreColor="ff00ff"
type="matches_text" case_sensitive="n" regex="n" text="::OnNdisNotification:
WwanEventCodePacketService" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="-&gt;ActivOption DSResponse" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="-&gt;DataDormancyHint DSResponse" />
    <filter enabled="y" excluding="n" description="" foreColor="ff0000"
type="matches_text" case_sensitive="n" regex="n" text="InternalErrorReport"
/>
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="ModemDualSIMCap" />
    <filter enabled="y" excluding="n" description="" foreColor="ff0000"
type="matches_text" case_sensitive="n" regex="n"
text="CWwanDataExecutorState::SendMbbPsAttachDetachReq" />
    <filter enabled="n" excluding="n" description="" foreColor="b22222"
type="matches_text" case_sensitive="n" regex="n"
text="WwanDefaultContextControllerFSMEvent_PSStateChanged" />
    <filter enabled="n" excluding="n" description="" foreColor="ff0000"
type="matches_text" case_sensitive="n" regex="n"
text="WwanDefaultContextControllerFSMEvent_ContextStoppedUnsolictedly" />
    <filter enabled="n" excluding="n" description="" foreColor="ff0000"
type="matches_text" case_sensitive="n" regex="n" text="lossing link state or
IP addr" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="_debouncing" />
    <filter enabled="n" excluding="n" description="" foreColor="ff0000"
type="matches_text" case_sensitive="n" regex="n" text=" unsolicitedlt
deactivated. APN " />
    <filter enabled="n" excluding="n" description="" foreColor="ff1493"
type="matches_text" case_sensitive="n" regex="n" text="ims status update
received" />
    <filter enabled="y" excluding="n" description="" foreColor="800000"
type="matches_text" case_sensitive="n" regex="n" text="RetryBackoffT" />
    <filter enabled="n" excluding="n" description="" foreColor="006400"
type="matches_text" case_sensitive="n" regex="n"
text="CWWANContextControllerBase::StartTimer:  " />
    <filter enabled="n" excluding="n" description="" foreColor="800080"
type="matches_text" case_sensitive="n" regex="n"
text="CWwanDefaultContextController::CalculateArmBackoffTimer" />
```

```
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="WwanProtDim" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="wmbclass" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="MBIM" />
    <filter enabled="n" excluding="n" description="" foreColor="808000"
type="matches_text" case_sensitive="y" regex="n" text="CID " />
    <filter enabled="n" excluding="n" description="" foreColor="808000"
type="matches_text" case_sensitive="n" regex="n" text="CID=" />
    <filter enabled="n" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="AdaptersAddresses" />
  </filters>
</TextAnalysisTool.NET>
```

MB Service Detection and Activation

Article • 03/14/2023

This topic describes the procedures to detect whether an MB device must have its service activated, and how to gain access to a provider's network.

Service Activation Detection

Miniport drivers can determine whether they must perform service activation in a couple of ways:

- For CDMA-based devices, in North America or other places where U-RIM is not used, there should be a flag on the device to indicate activation status. Miniport drivers should be able to detect the activation status during initialization without contacting the provider network. Miniport drivers should perform service activation automatically when the device first connects over-the-air to the home network. After activation has been completed, miniport drivers should clear the flag so that they will not need to perform service activation again.

  Miniport drivers inform the MB Service about service activation progress by sending **NDIS_STATUS_WWAN_READY_INFO** notifications during MB device initialization. Alternatively, to determine service activation status, the service may send an OID_WWAN_READY_INFO query request to a miniport driver. In both cases, the initial ready-state should be **WwanReadyStateNotActivated**. After service has been activated, miniport drivers should resume the initialization process and notify the service as the device ready-state changes.

- For GSM-based devices, there is no general method to detect whether a device must have its service activated. Miniport drivers can implement their own proprietary method, specific to its carrier, to perform service detection and activation.

MB Service Activation

Service activation refers to the process of activating the MB service subscription so that the device can gain access to the provider's network. The MB Service is not equipped with service activation logic because the exact activation procedure must be performed by the miniport driver and/or third-party software because the actual activation process varies from cellular technologies and is usually customized for different provider networks. Service activation can be automatic, or manual, or a combination of both.

Miniport drivers should only need to perform service activation one time for each new subscription.

For more information about service detection and activation, see OID_WWAN_SERVICE_ACTIVATION.

MB Radio State

Article • 03/14/2023

Overview

This topic describes the operations that are used to set and read an MB device's radio power state(s). These states can be controlled through software (airplane mode) or hardware (if the appropriate switch is present). This topic explains how the radio power states are controlled, how to test radio power state functionality, and how to investigate radio power state issues.

Terminology

System Radio State - System Radio State is a system-wide state. It is the most evident indicator of airplane mode state. System Radio State is managed by the Radio Management Service (RmSvc).

Radio Manager - RmSvc iterates several RadioManagers (MediaManagers) in the system like WlanRadioManager, BlueTooth, and WwanRadioManager. WwanRadioManager(.lib) is hosted in RmSvc.dll and manages the wwan side of the radio logic. WwanRadioManager uses the WWAN Service (WwanSvc) RPC to:

1. Query and set cellular radio.
2. Control the flow before and after airplane mode.

Radio Instance - Each RadioManager can include multiple radio instances. For example, WwanRadioManager can have two radio instances if there are two cellular modems in the system. Each radio instance is an abstract object and should map to one hardware radio module. In most cases, each radio instance maps to one cellular modem.

Relevant Services and Drivers

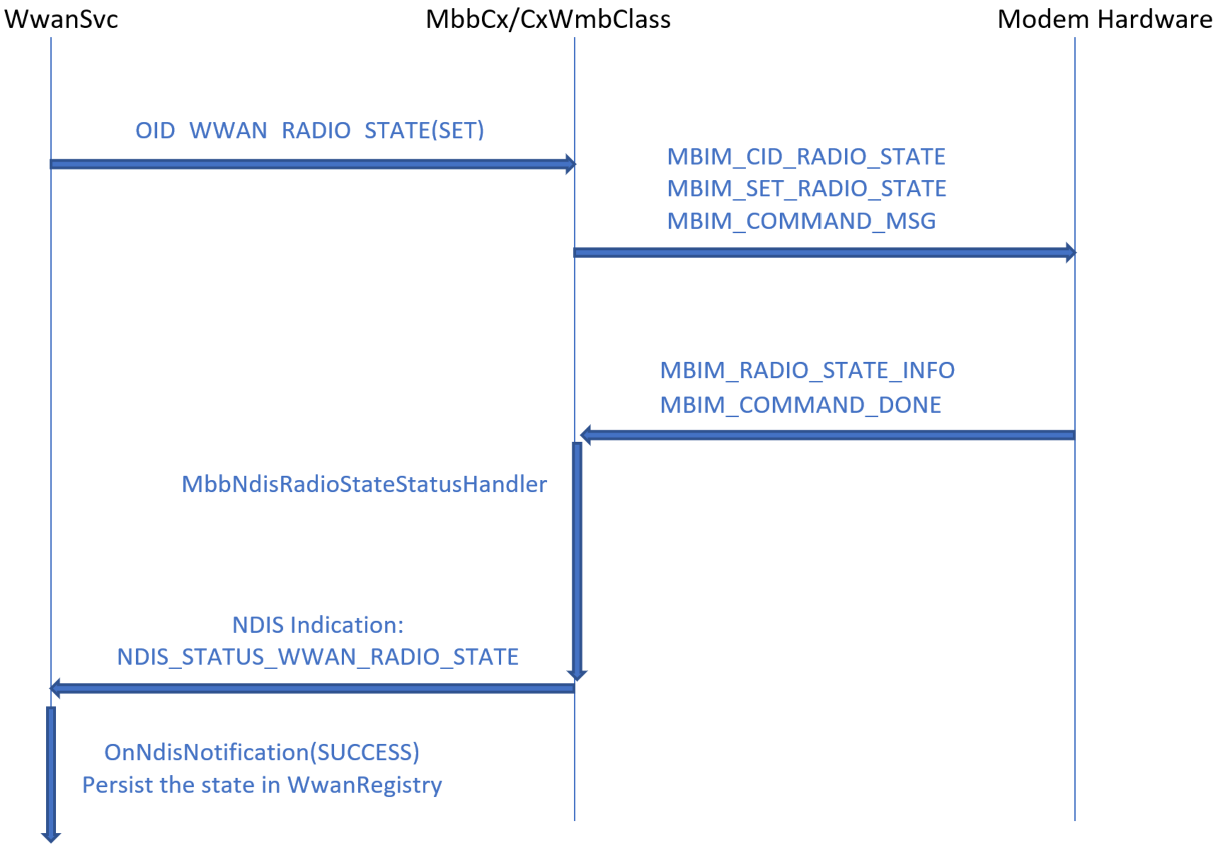
RmSvc.dll - Manages system-wide radio events like airplane mode. It also hosts all radio managers, including WwanRadioManager.

WwanSvc.dll - Cellular modems are managed by WwanSvc. Therefore, commands (OID/CID) are issued via WwanSvc. External requests from RmSvc or other components (UI) go via the WwanSvc RPC to query or set cellular radio state.

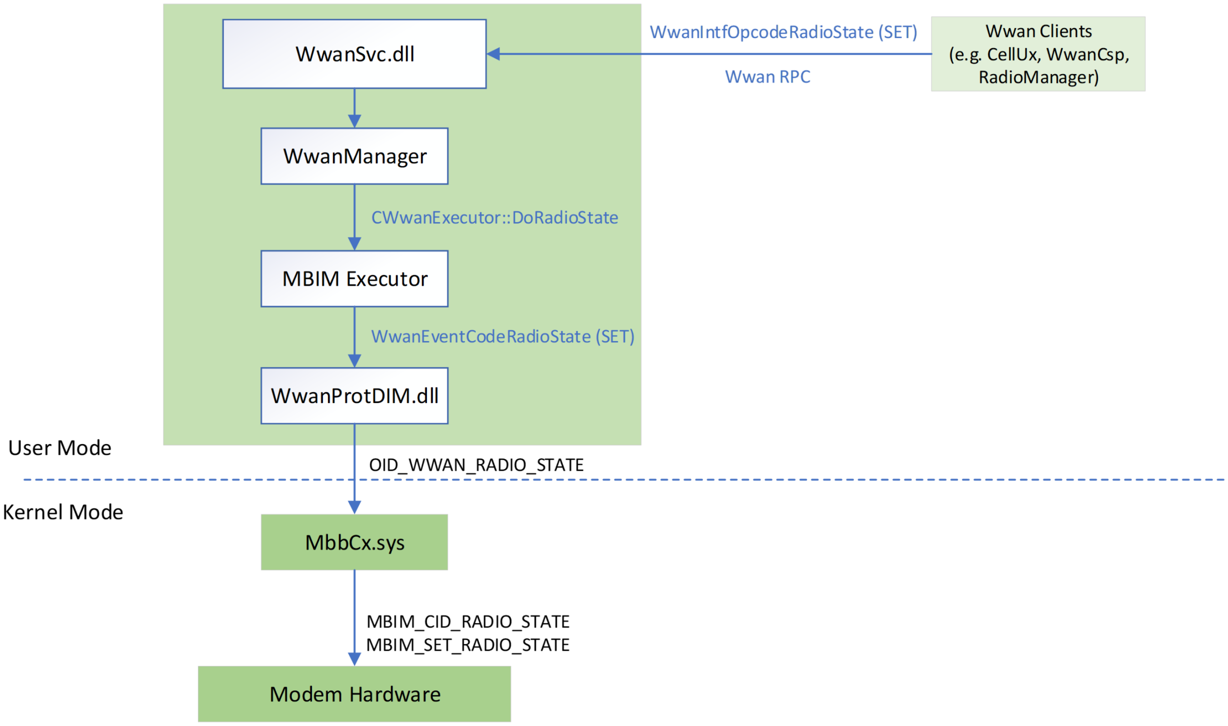
MbbCx.sys - The kernel-mode driver that manages device power state especially between D0 and Dx transition. On some system setups, the device is allowed to transition to Dx state and recover to D0 only when needed. MbbCx.sys manages the logic and control of radio state recovery before D0 and Dx.

Architecture/Flows

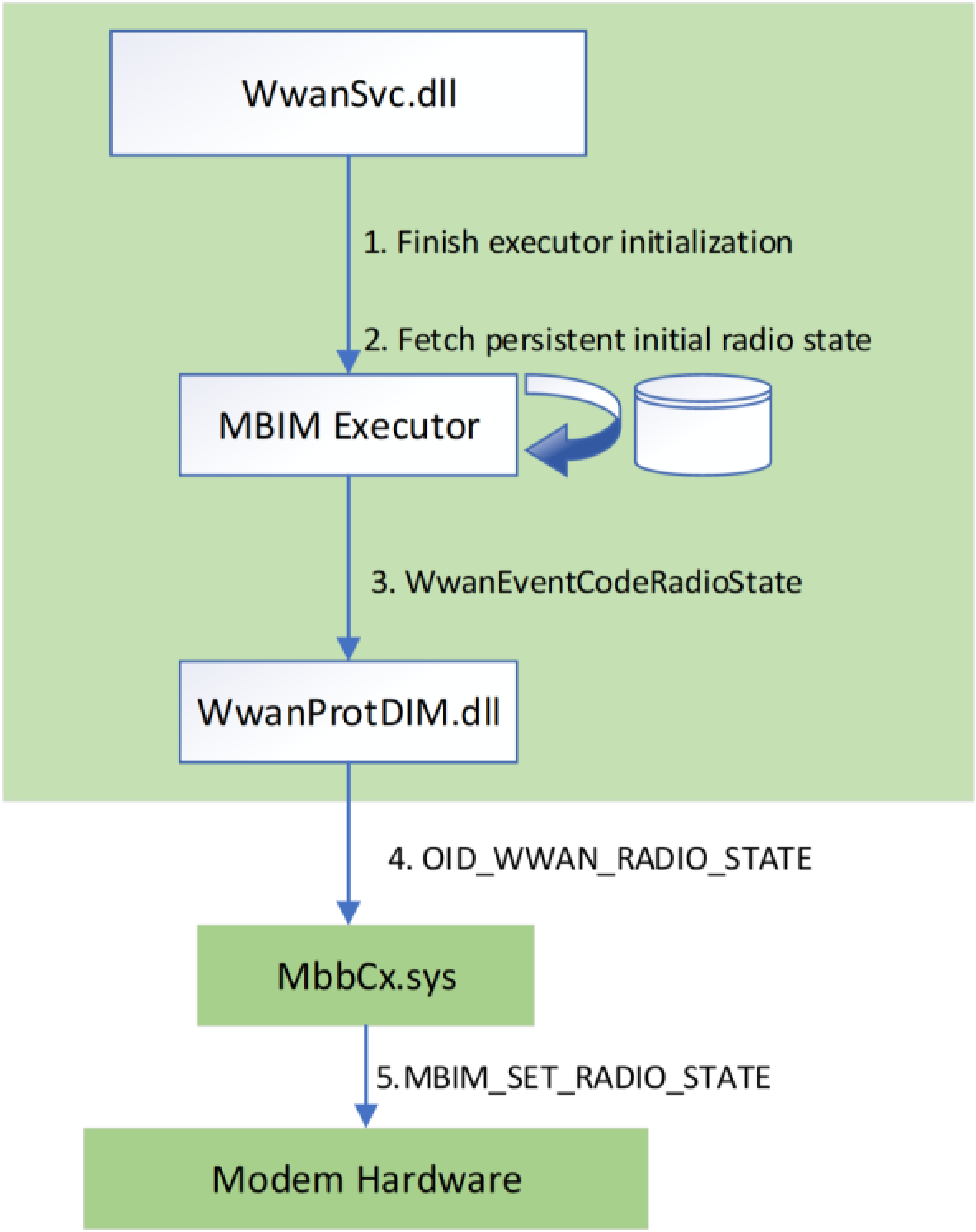
Radio Control from WwanSvc to Modem Hardware



SET Radio via WwanSvc API



Initial Radio State upon Device Arrival



MBIM_CID_RADIO_STATE

As seen in the above diagrams, the CID used in airplane mode operations is **MBIM_CID_RADIO_STATE**. This CID sets or returns information about a MB device's radio power state.

Query

The InformationBuffer on MBIM_COMMAND_MSG is not used. MBIM_RADIO_STATE_INFO is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Set

The InformationBuffer on MBIM_COMMAND_MSG contains MBIM_SET_RADIO_STATE. MBIM_RADIO_STATE_INFO is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Unsolicited Event

The Event InformationBuffer contains an MBIM_RADIO_STATE_INFO structure.

Parameters

	Set	Query	Notification
Command	MBIM_SET_RADIO_STATE	Empty	N/A
Response	MBIM_RADIO_STATE_INFO	MBIM_RADIO_STATE_INFO	MBIM_RADIO_STATE_INFO

Data Structures

Set

Offset	Size	Field	Type	Description
0	4	RadioState	MBIM_RADIO_SWITCH_STATE	Sets the software controlled radio state. See table below.

MBIM_RADIO_SWITCH_STATE

Types	Value
MBIMRadioOff	0
MBIMRadioOn	1

Query

The InformationBuffer will be **null** and InformationBufferLength will be **zero**.

Response

MBIM_RADIO_STATE_INFO

Offset	Size	Field	Type	Description
0	4	HwRadioState	MBIM_RADIO_SWITCH_STATE	The state of the W_DISABLE switch. If the device does not have a W_DISABLE switch, the function must return MBIMRadioOn in this field.
4	4	SwRadioState	MBIM_RADIO_SWITCH_STATE	Software configured radio state.

Notification

See the **MBIM_RADIO_STATE_INFO** table above.

Status Codes

This CID only uses Generic Status Codes.

Testing

Cellular Radio Tests

Function Name	Description
CellularRadioWinrtTest::VerifyCellularModemExistence	Assert winrt api can query cellular modem and radio state
CellularRadioWinrtTest::VerifyCellularRadioToggle	Assert winrt api can toggle radio state on each wwan adapter
CellularRadioRecoveryTest::VerifyCellularRadioRecoveryToOnAfterAPM	Assert cellular radio states are recovered to On when leaving Airplane Mode
CellularRadioRecoveryTest::VerifyCellularRadioRecoveryToOffAfterAPM	Assert cellular radio states remain off when leaving Airplane Mode

Function Name	Description
CellularRadioRecoveryTest::VerifyCellularRadioAcrossSvcRestart	Assert cellular radio states stay consistent across WwanSvc restarting
CellularRadioRecoveryTest::VerifyCellularRadioAcrossDevNodePnp	Assert cellular radio states stay consistent across device arrival/removal

CellularRadioTest.dll contains these tests.

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to the device Cellular modem driver and run these tests:

- **TestRadioStateSoftware**
- **TestRadioStateHardware**

Alternatively you can run the **TestRadioStateHardware** and **TestRadioStateSoftware** HLK testlist by **netsh-mbn** and **netsh-mbn-test-installation**.

```
netsh mbn test feature=radio testpath="C:\data\test\bin"
taefpath="C:\data\test\bin" param="AccessString=internet"
```

The two files showing the HLK test results should have been generated in the directory that the 'netsh mbn test' command was ran from: `TestRadioStateSoftware.htm` and `TestRadioStateHardware.htm`.

Logs can be collected and decoded using these instructions: MB Collecting Logs.

Analyzing Logs

Useful keywords/regexp for filtering traces

- OID_WWAN_RADIO_STATE
- CWwanRadioInstance::OnSysRadioChange

- Entering CUIRadioManager::_SetSysRadio
- Leaving CUIRadioManager::_SetSysRadio
- CWwanRadioInstance::_SetSoftwareRadioState
- [WwanRadioManager]
- PostD0Entry: previousPowerState
- CWwanRadioManager::OnSystemRadioStateChange(.)+sysradiostate
- RMAPI(.)+OnSystemRadioStateChange
- RMAPI(.)+OnSystemRadioStateChange
- Wwan-svc(.)+radio
- mbbcx(.)+radio

Investigation tips

- Identify whether this is a global (system-wide) or local (cellular-only) radio issue.
- Differentiate device power state (D0-Dx) from radio state. They are different concepts but highly-correlated.
- Ensure necessary ETW providers are included in the log.
- Narrow down the area by using the scenario. For example:
  - If it's related to airplane mode, focus on RmSvc and WwanRadioManager.
  - If it's related to D0<->Dx, hibernation, or sleep transitions, focus on MBBCx.
  - If it's related to UI displays or state out-of-sync, start with WwanSvc.

WinRT API

Windows.Devices.Radios

Windows.Devices.Radios is owned by the Radio Management Service which manages all radio managers and instances. For the WWAN side, the RadioKind is RadioKind::MobileBroadband.

- GetRadiosAsync( )
- SetStateAsync( )

Windows.Networking.NetworkOperators

Windows.Networking.NetworkOperators documentation page

The only useful utility under this namespace for radio management is MobileBroadbandDeviceInformation.CurrentRadioState.

See Also

OID_WWAN_RADIO_STATE

MB PIN Operations

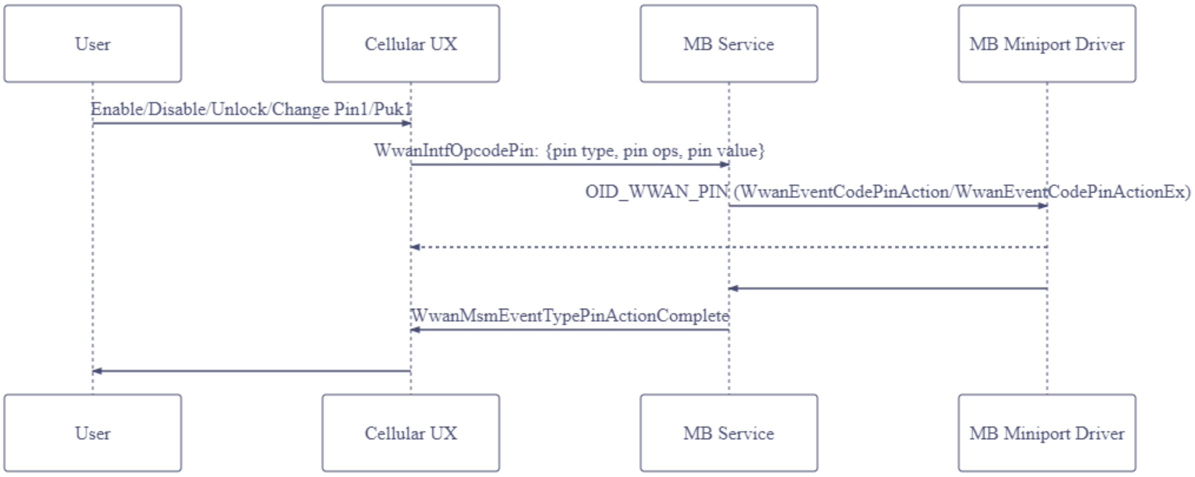
Article • 03/14/2023

Overview

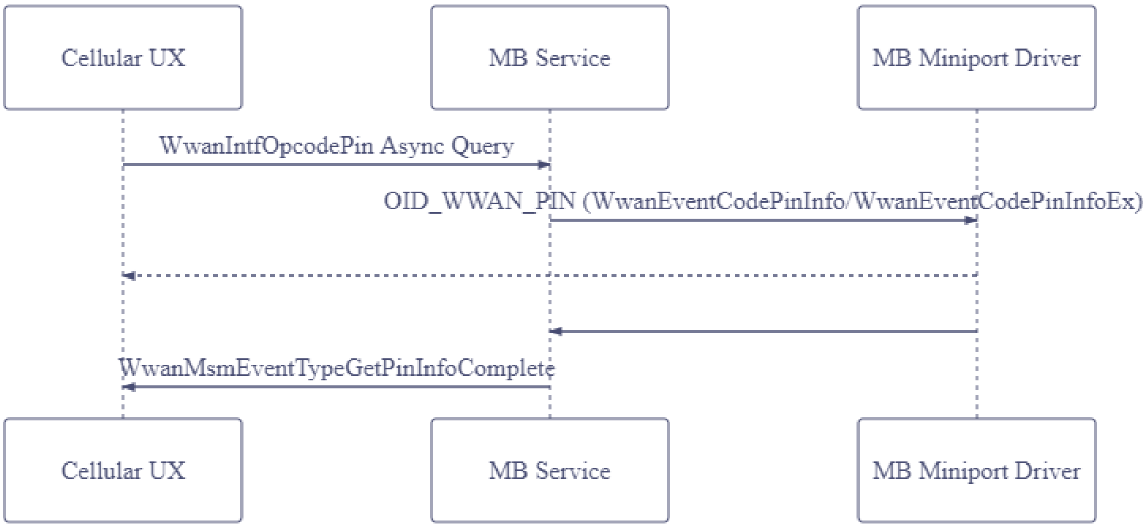
This topic describes the operations related to access control of subscription information stored either in the MB device memory or on the Subscriber Identity Module (SIM) card. This includes enabling, disabling, or changing the Personal Identification Number (PIN), as well as unlocking via PIN or Personal Unlocking Key (PUK).

Architecture/Flows

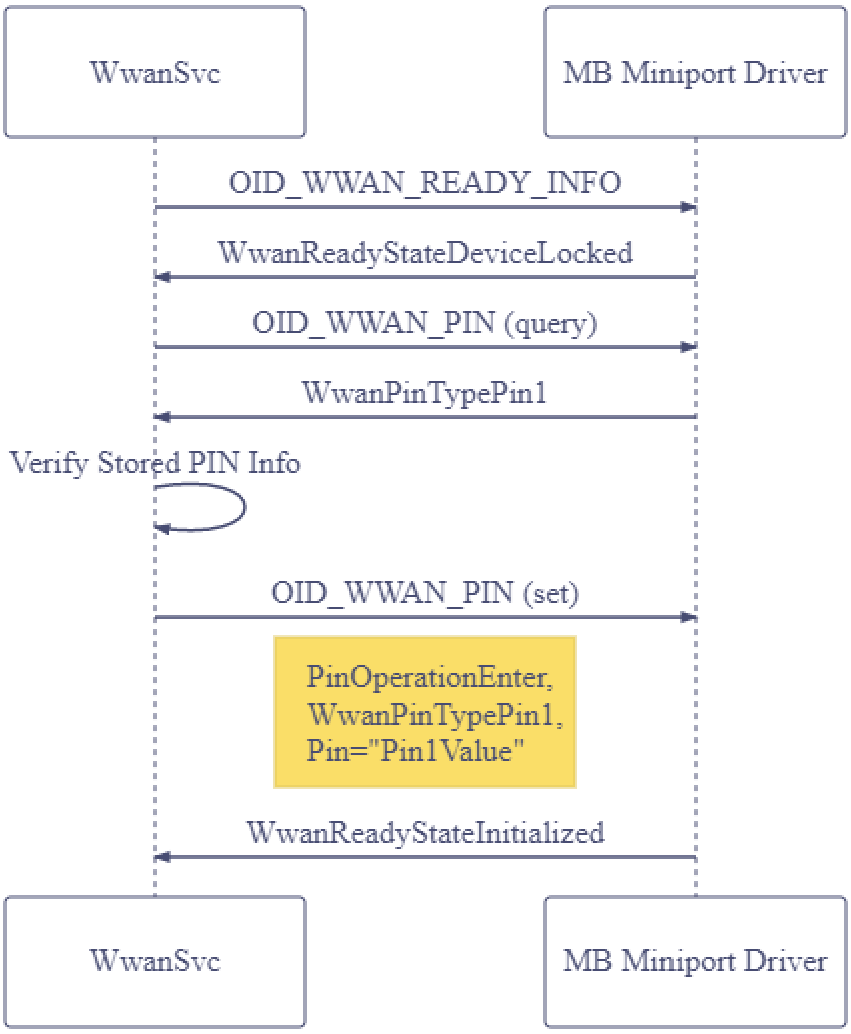
User actions to enable/disable/unlock/change PIN


User
Cellular UX
MB Service
MB Miniport Driver
Enable/Disable/Unlock/Change Pin1/Puk1
WwanIntfOpcodePin: {pin type, pin ops, pin value}
OID_WWAN_PIN (WwanEventCodePinAction/WwanEventCodePinActionEx)
WwanMsmEventTypePinActionComplete
User
Cellular UX
MB Service
MB Miniport Driver

Cellular UX query for PIN1/PUK1 state


Cellular UX
MB Service
MB Miniport Driver
WwanIntfOpcodePin Async Query
OID_WWAN_PIN (WwanEventCodePinInfo/WwanEventCodePinInfoEx)
WwanMsmEventTypeGetPinInfoComplete
Cellular UX
MB Service
MB Miniport Driver

Auto-unlock after resuming from hibernation



MBIM_CID_MS_PIN_EX

This CID is described here: MBIM_CID_MS_PIN_EX

MBIM_CID_PIN_LIST

Description

This command returns a list of all the different types of Personal Identification Numbers (PINs) that are supported by the MB device and additional details for each PIN type, such as the length of the PIN (minimum and maximum lengths), PIN format, and PIN-entry mode (enabled/disabled/not-available). This CID also specifies the current mode of each PIN supported by the function. Functions must report all the PINs they support. However, PIN1 for multi-mode devices must only be reported once.

A PIN reported as PIN1 must comply with PIN1 guidelines: For CDMA-based devices this is a PIN that provides power-on verification or identification functionality, and for GSM-based devices this is a Subscriber Identity Module (SIM) PIN.

Functions must be able to return this information when the ready-state changes to MBIMSubscriberReadyStateInitialized or when the ready-state is MBIMSubscriberReadyStateDeviceLocked (PIN locked). Functions should also return this information in other ready-states wherever possible.

Query only

InformationBuffer of the Query message is empty. InformationBuffer of MBIM_COMMAND_DONE contains an MBIM_PIN_LIST_INFO.

Parameters

	Set	Query	Notification
Command	N/A	Empty	N/A
Response	N/A	MBIM_PIN_LIST_INFO	N/A

Data Structures

MBIM_PIN_MODE

Types	Value
MBIMPinModeNotSupported	0
MBIMPinModeEnabled	1
MBIMPinModeDisabled	2

MBIM_PIN_FORMAT

Types	Value
MBIMPinFormatUnknown	0
MBIMPinFormatNumeric	1
MBIMPinFormatAlphaNumeric	2

MBIM_PIN_DESC

Offset	Size	Field	Type	Description
0	4	PinMode	MBIM_PIN_MODE	See above table MBIM_PIN_MODE. This shows if the lock is enabled or not. It does not show if the lock state is locked or unlocked.
4	4	PinFormat	MBIM_PIN_FORMAT	See above table MBIM_PIN_FORMAT.
8	4	PinLengthMin	UINT32	The minimum number of characters in the PIN. Devices should not specify a value that is greater than 16. Devices should specify 0xffffffff, if the PIN length is not available.
12	4	PinLengthMax	UINT32	The maximum number of characters in the PIN. Devices should not specify a value that is greater than 16. Devices should specify 0xffffffff, if the PIN length is not available.

Query

The InformationBuffer shall be **null** and InformationBufferLength shall be **zero**.

Response

The following structure shall be used in the InformationBuffer:

MBIM_PIN_LIST_INFO

Offset	Size	Field	Type	Description
0	16	PinDescPin1	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing PIN1. For GSMbased devices, this is a Subscriber Identity Module (SIM) PIN. For CDMA-based devices, power-on device lock is reported as PIN1.
16	16	PinDescPin2	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing PIN2. This is a SIM PIN2 that protects certain SIM functionality.

Offset	Size	Field	Type	Description
32	16	PinDescDeviceSimPin	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing the device-to-SIM-card PIN. This is a PIN that locks the device to a specific SIM.
48	16	PinDescDeviceFirstSimPin	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing device-to-very-first-SIM-card PIN. This is a PIN that locks the device to the very first inserted SIM.
64	16	PinDescNetworkPin	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing the network personalization PIN. This is a PIN that allows the device to be personalized to a network. For more information about this PIN type, see 3GPP specification 22.022.
80	16	PinDescNetworkSubsetPin	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing the network subset personalization PIN. This is a PIN that allows the device to be personalized to a subset of a network. For more information about this PIN type, see 3GPP specification 22.022.
96	16	PinDescServiceProviderPin	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing the Service Provider (SP) personalization PIN. This is a PIN that allows the device to be personalized to a service provider. For more information about this PIN type, see 3GPP specification 22.022.
112	16	PinDescCorporatePin	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing the corporate personalization PIN. This is a PIN that allows the device to be personalized to a specific company. For more information about this PIN type, see 3GPP specification 22.022.

Offset	Size	Field	Type	Description
128	16	PinDescSubsidyLock	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing the subsidy unlocks PIN. This is a PIN that allows the device to be restricted to operate on a specific network. For more information about this PIN type, see 3GPP specification 22.022.
144	16	PinDescCustom	MBIM_PIN_DESC	MBIM_PIN_DESC structure describing the custom PIN. This is a custom vendor-defined PIN type. It is not included in the above list.

Status Codes

Status Code	Description
MBIM_STATUS_PIN_REQUIRED	The PIN list operation failed because a PIN must be entered before this operation can proceed.

Testing

The following tests are run as part of the **TestPin** HLK test list:

Test Name	Description
PinListQueryRadioOn	This test attempts a pin list query with the radio on.
PinListQueryRadioOff	This test attempts a pin list query with the radio off.
NoPinSupport	This test verifies a device that does not support PIN1.
PinExSetEnableDisableWithValidPin	This test enables and disables PIN1 with a valid pin.
PinExSetDisableIncorrectPinWithValidLength	This test attempts to enable PIN1 with an incorrect pin with valid length.
PukEnableDisableThroughIncorrectPinExDisable	This test enables PUK1 by entering incorrect PIN1 multiple times and then disables PUK1.

Test Name	Description
PinExSetChangeWithBothInvalidAndValidPin	This test enables PIN1, immediately changes the PIN, and disables it.
RebootTestMachineToPutPinIntoLockState	This test reboots the device to make the modem enter lock state and prompt valid PIN entry.
PinExSetEnterWithValidPin	This test validates that the device is in lock state to request PIN code entry.

The **TestPowerStates** HLK test list also contains one relevant test -- **SimPinUnlockAfterHibernate**.

Log Analysis

Sample Logs:

Auto-unlock:

```
462678 [0]0F3C.1280::2020-05-05 13:03:46.378805100 [Microsoft-Windows-WWAN-
SVC-EVENTS][Request=0x53] Received PinInfo, status=WWAN_STATUS_SUCCESS ,
Type=WwanPinTypePin1  State=WwanPinStateEnter  Attempts=3, miniport=
{7971731f-33f9-4f1a-9718-087c12e64c5d}
462753 [7]0F3C.2A6C::2020-05-05 13:03:46.379718400 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::maybeUnlockPin:
Attempting auto-PIN-unlock. Interface: {{7971731f-33f9-4f1a-9718-
087c12e64c5d}}
462809 [7]0F3C.2A6C::2020-05-05 13:03:46.380157500 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Error] CWwanPinSM::maybeUnlockPin: Attempt
to auto-unlock PIN succeeded
```

Set Pin (WwanPinTypePin1):

```
546408 [3]0F3C.1240::2020/05/02-09:18:09.178460200 [Microsoft-Windows-WWAN-
SVC-EVENTS][Request=0x6C] Sent SET PinAction, Type=2(WwanPinTypePin1),
Operation=0(WwanPinOperationEnter), miniport={7971731f-33f9-4f1a-9718-
087c12e64c5d}, ErrorCode=3407873(WIN=The request will be completed later by
NDIS status indication.)
546425 [1]3DB0.12EC::2020/05/02-09:18:09.178564700
[Microsoft.Windows.CellCore.MBBSettingsUX]{"meta":
{"provider":"Microsoft.Windows.CellCore.MBBSettingsUX","event":"MBCategory::
_SetPinAction. WwanSetInterface succeeded","time":"2020-05-
02T16:18:09.1785647Z","cpu":1,"pid":15792,"tid":4844,"channel":11,"level":4}
}
546644 [2]0F3C.39E4::2020/05/02-09:18:09.426362600 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::processPinActionResponse:
SetPin rsp rcvd (result:0x0) PIN Info (state:1, type:3, attemptsRemaining:3)
IsPin1Blocked 0
546645 [2]0F3C.39E4::2020/05/02-09:18:09.426364800 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::maybeCapturePin: Capturing
PIN for PIN ENTER/ENABLE operation Interface: {{7971731f-33f9-4f1a-9718-
087c12e64c5d}}
546688 [7]3B64.2A80::2020/05/02-09:18:09.426727000 [MbaeApiLogging]
{"NotificationCode":"WwanMsmEventTypePinActionComplete:
Success","AdapterID":"{7971731f-33f9-4f1a-9718-
087c12e64c5d}","NotificationSize":24,"meta":
{"provider":"MbaeApiLogging","event":"CWwanTranslator::ProcessWwanNotificati
on","time":"2020-05-
02T16:18:09.4267270Z","cpu":7,"pid":15204,"tid":10880,"channel":11,"level":5
}}
546702 [0]3B64.242C::2020/05/02-09:18:09.426762000
[Microsoft.Windows.CellCore.MBBSettingsUX]{"meta":
{"provider":"Microsoft.Windows.CellCore.MBBSettingsUX","event":"MBMediaManag
er::ProcessWwanNotification WwanMsmEventTypePinActionComplete","time":"2020-
05-
02T16:18:09.4267620Z","cpu":0,"pid":15204,"tid":9260,"channel":11,"level":4}
}
546710 [7]0F3C.1208::2020/05/02-09:18:09.426809700 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] _PublishSebNotification: Event
Source=WwanNotificationSourceMsm, Code=WwanMsmEventTypePinActionComplete
547064 [2]3DB0.1194::2020/05/02-09:18:09.427921200 [MbaeApiLogging]
{"NotificationCode":"WwanMsmEventTypePinActionComplete:
Success","AdapterID":"{7971731f-33f9-4f1a-9718-
087c12e64c5d}","NotificationSize":24,"meta":
{"provider":"MbaeApiLogging","event":"CWwanTranslator::ProcessWwanNotificati
on","time":"2020-05-
02T16:18:09.4279212Z","cpu":2,"pid":15792,"tid":4500,"channel":11,"level":5}
}
547106 [2]3DB0.0B38::2020/05/02-09:18:09.428040100
[Microsoft.Windows.CellCore.MBBSettingsUX]{"meta":
{"provider":"Microsoft.Windows.CellCore.MBBSettingsUX","event":"MBMediaManag
er::ProcessWwanNotification WwanMsmEventTypePinActionComplete","time":"2020-
05-
02T16:18:09.4280401Z","cpu":2,"pid":15792,"tid":2872,"channel":11,"level":4}
}
```

Pin List:

```
465632 [4]0F3C.47F4::2020-05-05 13:03:46.395488200 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc: PIN1
(mode:1, format:1, pinlnmin:4, pinlnmax:8)
465633 [4]0F3C.47F4::2020-05-05 13:03:46.395489800 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc: PIN2
```

```
(mode:1, format:1, pinlnmin:4, pinlnmax:8)
465634 [4]0F3C.47F4::2020-05-05 13:03:46.395491400 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc: DEVSIMPIN
(mode:0, format:0, pinlnmin:0, pinlnmax:0)
465635 [4]0F3C.47F4::2020-05-05 13:03:46.395492800 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc:
DEVFIRSTSIMPIN (mode:0, format:0, pinlnmin:0, pinlnmax:0)
465636 [4]0F3C.47F4::2020-05-05 13:03:46.395494200 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc: NWPIN
(mode:0, format:0, pinlnmin:0, pinlnmax:0)
465637 [4]0F3C.47F4::2020-05-05 13:03:46.395495800 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc: NWSUBSETPIN
(mode:0, format:0, pinlnmin:0, pinlnmax:0)
465641 [5]0F3C.47F4::2020-05-05 13:03:46.395510100 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc:
SVCPROVIDERPIN (mode:0, format:0, pinlnmin:0, pinlnmax:0)
465643 [5]0F3C.47F4::2020-05-05 13:03:46.395513700 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc: CORPORATEPIN
(mode:0, format:0, pinlnmin:0, pinlnmax:0)
465644 [5]0F3C.47F4::2020-05-05 13:03:46.395515200 [Microsoft-Windows-WWAN-
SVC-EVENTS]WWAN Service event: [Info] CWwanPinSM::tracePinDesc: SUBSIDYLOCK
(mode:0, format:0, pinlnmin:0, pinlnmax:0)
```

WinRT API

MobileBroadbandPin Class

See Also

OID_WWAN_PIN_EX2

OID_WWAN_PIN_LIST

MB UICC application and file system access

For additional information about PIN operations, see OID_WWAN_PIN.

PIN Operations Log filter

Article • 12/15/2021

To make searching log files easier, below is a PIN-operation-specific filter file for the TextAnalysisTool.

To use this filter:

1. Copy and paste the lines below and save them into a text file named "WwanPin.tat."

2. Load the filter file into the TextAnalysisTool by clicking File > Load Filters.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<TextAnalysisTool.NET version="2015-06-23" showOnlyFilteredLines="True">
  <filters>
    <filter enabled="n" excluding="n" backColor="90ee90" type="matches_text"
case_sensitive="n" regex="n" text="maybeUnlockPin" />
    <filter enabled="n" excluding="n" foreColor="800000" type="matches_text"
case_sensitive="n" regex="n" text="Received PinInfo" />
    <filter enabled="n" excluding="n" backColor="ffb6c1" type="matches_text"
case_sensitive="n" regex="n" text="maybeCapturePin" />
    <filter enabled="n" excluding="n" foreColor="800080" type="matches_text"
case_sensitive="n" regex="n" text="[Microsoft-Windows-WWAN-SVC-EVENTS]" />
    <filter enabled="n" excluding="n" foreColor="00008b" type="matches_text"
case_sensitive="n" regex="n" text="PinAction" />
    <filter enabled="n" excluding="n" backColor="dda0dd" type="matches_text"
case_sensitive="n" regex="n" text="tracePinDesc" />
    <filter enabled="n" excluding="n" foreColor="008000" type="matches_text"
case_sensitive="n" regex="n" text="processPinInfoResponse" />
    <filter enabled="n" excluding="n" foreColor="0000ff" type="matches_text"
case_sensitive="n" regex="n" text="processPinActionResponse" />
    <filter enabled="n" excluding="n" foreColor="dc143c" type="matches_text"
case_sensitive="n" regex="n" text="processPinListResponse" />
    <filter enabled="n" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="NDIS_STATUS_WWAN_PIN_INFO" />
  </filters>
</TextAnalysisTool.NET>
```

MB Provider Operations

Article • 03/14/2023

This topic describes the operations related to *home*, *preferred*, *multicarrier*, and *visible* network providers.

For more information about provider operations, see OID_WWAN_HOME_PROVIDER, OID_WWAN_PREFERRED_PROVIDERS, OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS, and OID_WWAN_VISIBLE_PROVIDERS.

MB Registration Operations

Article • 03/14/2023

Miniport drivers use OID_WWAN_REGISTER_STATE to process both *query* and *set* registration operations. For example, these operations may be to register with a network provider, query the registration state of the device with a provider's network, and send notifications when registration state changes.

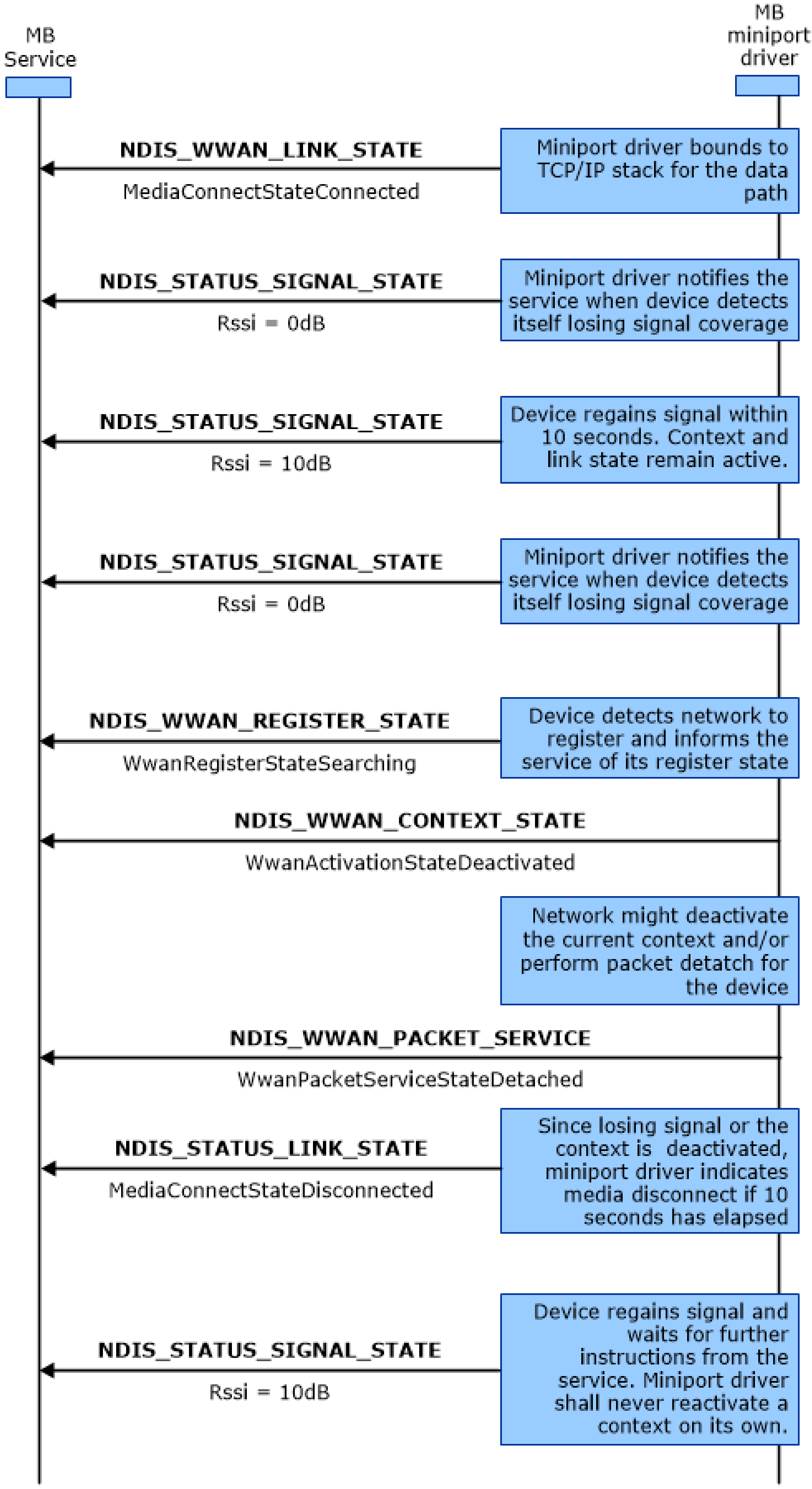
MB Packet Service Operations

Article • 03/14/2023

This topic describes the operations for losing and regaining packet data service, packet data service handoffs, and voice calls during packet data service connections.

Losing and Regaining Packet Data Service

The following diagram shows the process that miniport drivers should follow when they lose signal strength and packet service for various intervals. The labels in bold are OID identifiers or transactional flow control.The labels in regular text are the important flags within the OID structure.


MB Service
MB miniport driver
NDIS_WWAN_LINK_STATE
MediaConnectStateConnected
Miniport driver bounds to TCP/IP stack for the data path
NDIS_STATUS_SIGNAL_STATE
Rssi = 0dB
Miniport driver notifies the service when device detects itself losing signal coverage
NDIS_STATUS_SIGNAL_STATE
Rssi = 10dB
Device regains signal within 10 seconds. Context and link state remain active.
NDIS_STATUS_SIGNAL_STATE
Rssi = 0dB
Miniport driver notifies the service when device detects itself losing signal coverage
NDIS_WWAN_REGISTER_STATE
WwanRegisterStateSearching
Device detects network to register and informs the service of its register state
NDIS_WWAN_CONTEXT_STATE
WwanActivationStateDeactivated
Network might deactivate the current context and/or perform packet detatch for the device
NDIS_WWAN_PACKET_SERVICE
WwanPacketServiceStateDetached
NDIS_STATUS_LINK_STATE
MediaConnectStateDisconnected
Since losing signal or the context is deactivated, miniport driver indicates media disconnect if 10 seconds has elapsed
NDIS_STATUS_SIGNAL_STATE
Rssi = 10dB
Device regains signal and waits for further instructions from the service. Miniport driver shall never reactivate a context on its own.

To regain packet data service after it has been lost, use the following procedure:

1. The miniport driver sends NDIS_WWAN_LINK_STATE to the MB Service.

2. The miniport driver sends NDIS_WWAN_SIGNAL_STATE to the MB Service.

3. The miniport driver sends **NDIS_WWAN_SIGNAL_STATE** to the MB Service.

4. The miniport driver sends **NDIS_WWAN_SIGNAL_STATE** to the MB Service.

5. The miniport driver sends NDIS_WWAN_REGISTER_STATE to the MB Service.

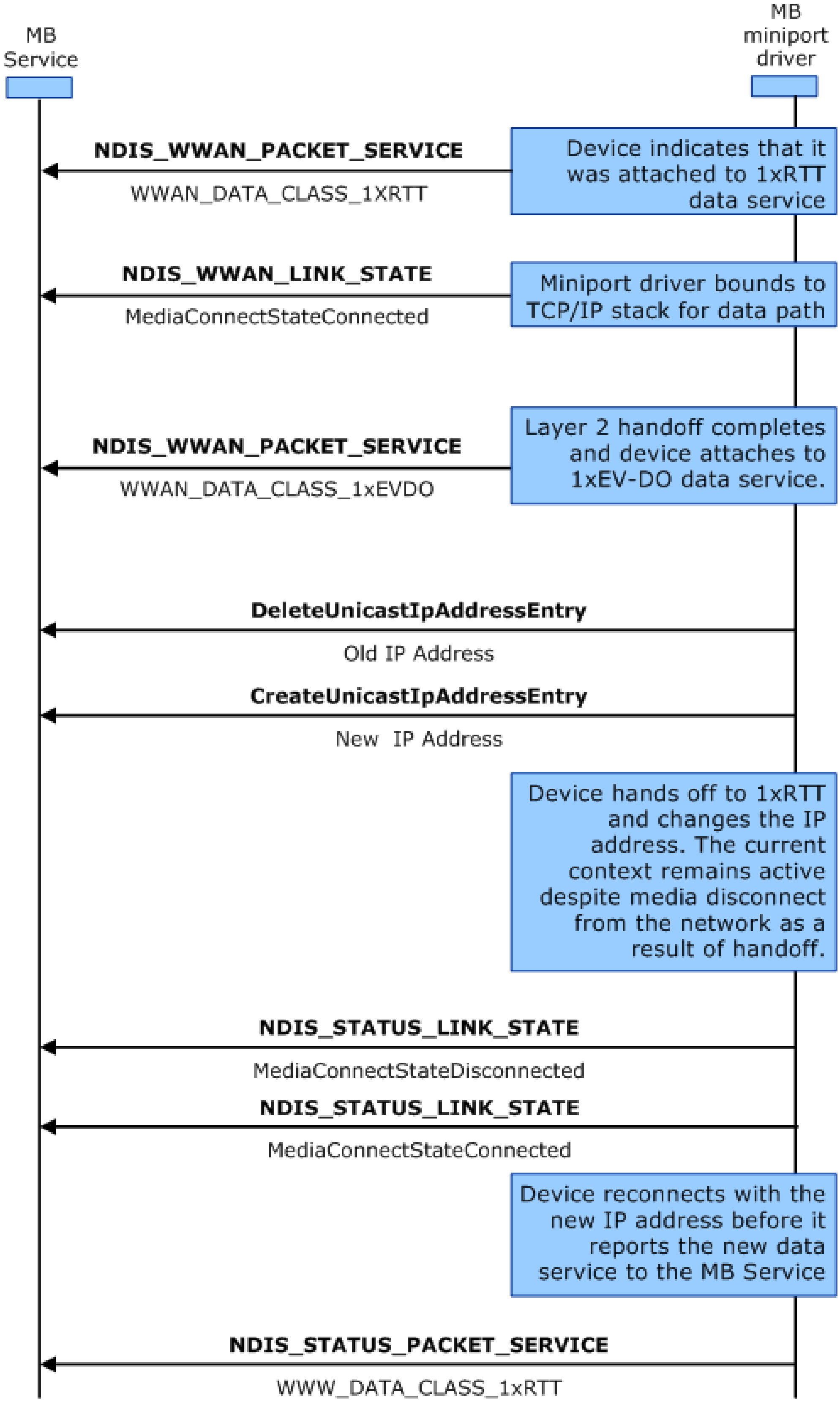
6. The miniport driver sends **NDIS_STATUS_WWAN_PACKET_SERVICE** to the MB Service.

7. The miniport driver sends **NDIS_STATUS_LINK_STATE** to the MB Service.

8. The miniport driver sends **NDIS_WWAN_SIGNAL_STATE** to the MB Service.

Packet Data Service Handoffs

The following diagram shows the steps that miniport drivers should follow when packet service moves between different GSM-based technologies, such as GPRS, EDGE, UMTS, HSDPA, or TD-SCDMA, or moves between different CDMA-based technologies, such as 1xRTT, EV-DO, or EV-DO RevA. The labels in bold are OID identifiers or transactional flow control. The labels in regular text are the important flags within the OID structure.


MB Service
MB miniport driver
NDIS_WWAN_PACKET_SERVICE
WWAN_DATA_CLASS_1XRTT
Device indicates that it was attached to 1xRTT data service
NDIS_WWAN_LINK_STATE
MediaConnectStateConnected
Miniport driver bounds to TCP/IP stack for data path
NDIS_WWAN_PACKET_SERVICE
WWAN_DATA_CLASS_1xEVDO
Layer 2 handoff completes and device attaches to 1xEV-DO data service.
DeleteUnicastIpAddressEntry
Old IP Address
CreateUnicastIpAddressEntry
New IP Address
Device hands off to 1xRTT and changes the IP address. The current context remains active despite media disconnect from the network as a result of handoff.
NDIS_STATUS_LINK_STATE
MediaConnectStateDisconnected
NDIS_STATUS_LINK_STATE
MediaConnectStateConnected
Device reconnects with the new IP address before it reports the new data service to the MB Service
NDIS_STATUS_PACKET_SERVICE
WWW_DATA_CLASS_1xRTT

Be aware that unless the IP address changes in the handoff process, the MB Service handles the handoff event transparently without disrupting the existing connection. However, miniport drivers must still notify the MB Service about media disconnect events if, and only if, the IP address changes.

Miniport drivers and the MB device they manage should be able to handle the layer-2 handoff between different air interfaces automatically, with minimal impact to the MB Service and other overlaying applications. The only possible impact is the change to the

IP address that might result from the technology handoff. In this case, miniport drivers should re-establish the MB connection before reporting the packet service change to the MB Service. Miniport drivers that do not implement DHCP functionality should use the IP Helper and associated functions. Miniport drivers that do implement DHCP functionality are not required to use the IP Helper functions.

To hand off packet data service, use the following procedure:

1. The miniport driver sends **NDIS_STATUS_WWAN_PACKET_SERVICE** to the MB Service.

2. The miniport driver sends NDIS_WWAN_LINK_STATE to the MB Service.

3. The miniport driver sends **NDIS_STATUS_WWAN_PACKET_SERVICE** to the MB Service.

4. The miniport driver calls the **DeleteUnicastIpAddressEntry** helper function with the old IP address

5. The miniport driver calls the **CreateUnicastIpAddressEntry** helper function with the new IP address

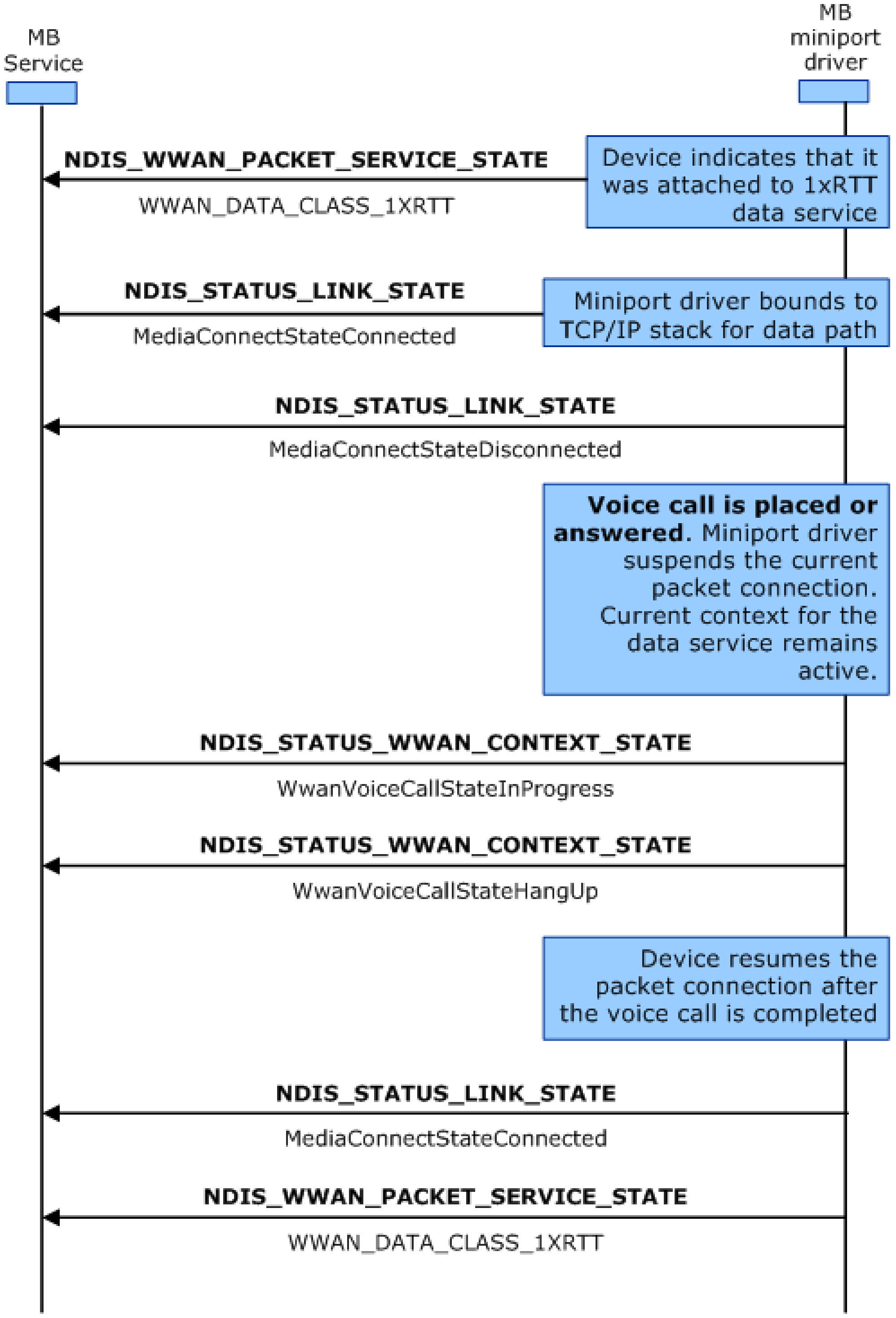
6. The miniport driver sends **NDIS_STATUS_LINK_STATE** to the MB Service.

7. The miniport driver sends **NDIS_STATUS_LINK_STATE** to the MB Service.

8. The miniport driver sends **NDIS_STATUS_WWAN_PACKET_SERVICE** to the MB Service.

Voice Calls during Packet Data Service Connections

The following diagram represents the process that miniport drivers should follow when a voice call is placed while packet data service is active. The diagram uses 1xRTT as an example, but the procedure applies to other air interfaces as well. The process outlined in the following graphic applies only to miniport drivers that return **WwanVoiceClassSeparateVoiceData** in the **WwanVoiceClass** member in response to an OID_WWAN_DEVICE_CAPS *query* request. The labels in bold represent OID identifiers or transactional flow control. The labels in regular text represent the important flags within the OID structure.


MB Service
MB miniport driver
NDIS_WWAN_PACKET_SERVICE_STATE
WWAN_DATA_CLASS_1XRTT
Device indicates that it was attached to 1xRTT data service
NDIS_STATUS_LINK_STATE
MediaConnectStateConnected
Miniport driver bounds to TCP/IP stack for data path
NDIS_STATUS_LINK_STATE
MediaConnectStateDisconnected
Voice call is placed or answered. Miniport driver suspends the current packet connection. Current context for the data service remains active.
NDIS_STATUS_WWAN_CONTEXT_STATE
WwanVoiceCallStateInProgress
NDIS_STATUS_WWAN_CONTEXT_STATE
WwanVoiceCallStateHangUp
Device resumes the packet connection after the voice call is completed
NDIS_STATUS_LINK_STATE
MediaConnectStateConnected
NDIS_WWAN_PACKET_SERVICE_STATE
WWAN_DATA_CLASS_1XRTT

The procedure assumes that accepting an incoming voice call will pre-empt any pre-existing packet connection. For miniport drivers that return **WwanVoiceClassSimultaneousVoiceData** in the **WwanVoiceClass** member in response to an OID_WWAN_DEVICE_CAPS *query* request, the current packet connection should not be affected.

Be aware that, by design, the MB Service does not support circuit voice nor does it prohibit the service. The process outlined in the graphic above applies only when the device can handle both data and circuit voice, but only one at a time. The process assumes that the voice call takes precedence over any potential pre-existing data connection. In this case, miniport drivers should suspend the data connection for the duration of the voice call. Afterwards, miniport drivers should resume the data service by re-establishing the MB connection automatically.

To handle voice calls during packet data service connections, use the following procedure:

1. For a successful Packet Data service connection, miniport drivers should send an NDIS_WWAN_PACKET_SERVICE_STATE notification to the MB service to indicate the current DataClass followed by an NDIS_STATUS_LINK_STATE notification to the MB service to indicate the media connect state as **MediaConnectStateConnected**.

2. When a voice call is placed or answered, miniport drivers should send an NDIS_STATUS_LINK_STATE notification to the MB service to indicate the media connect state as **MediaConnectStateDisconnected**.

3. Miniport drivers should then send an NDIS_STATUS_WWAN_CONTEXT_STATE notification to the MB service that indicates the *VoiceCall* state of the device as **WwanVoiceCallStateInProgress**.

4. On hangup, miniport drivers should send an NDIS_STATUS_WWAN_CONTEXT_STATE notification to the MB service that indicates the *VoiceCall* state of the device as **WwanVoiceCallStateHangup**.

5. The device resumes packet connection after the voice call is completed. Miniport drivers should send an NDIS_STATUS_LINK_STATE notification to the MB service to indicate the media connect state as **MediaConnectStateConnected**.

6. Miniport drivers should send an NDIS_WWAN_PACKET_SERVICE_STATE notification to the MB service that indicates the current DataClass.

See Also

For more information about packet service operations, see OID_WWAN_PACKET_SERVICE.

MB Signal Strength Operations

Article • 03/14/2023

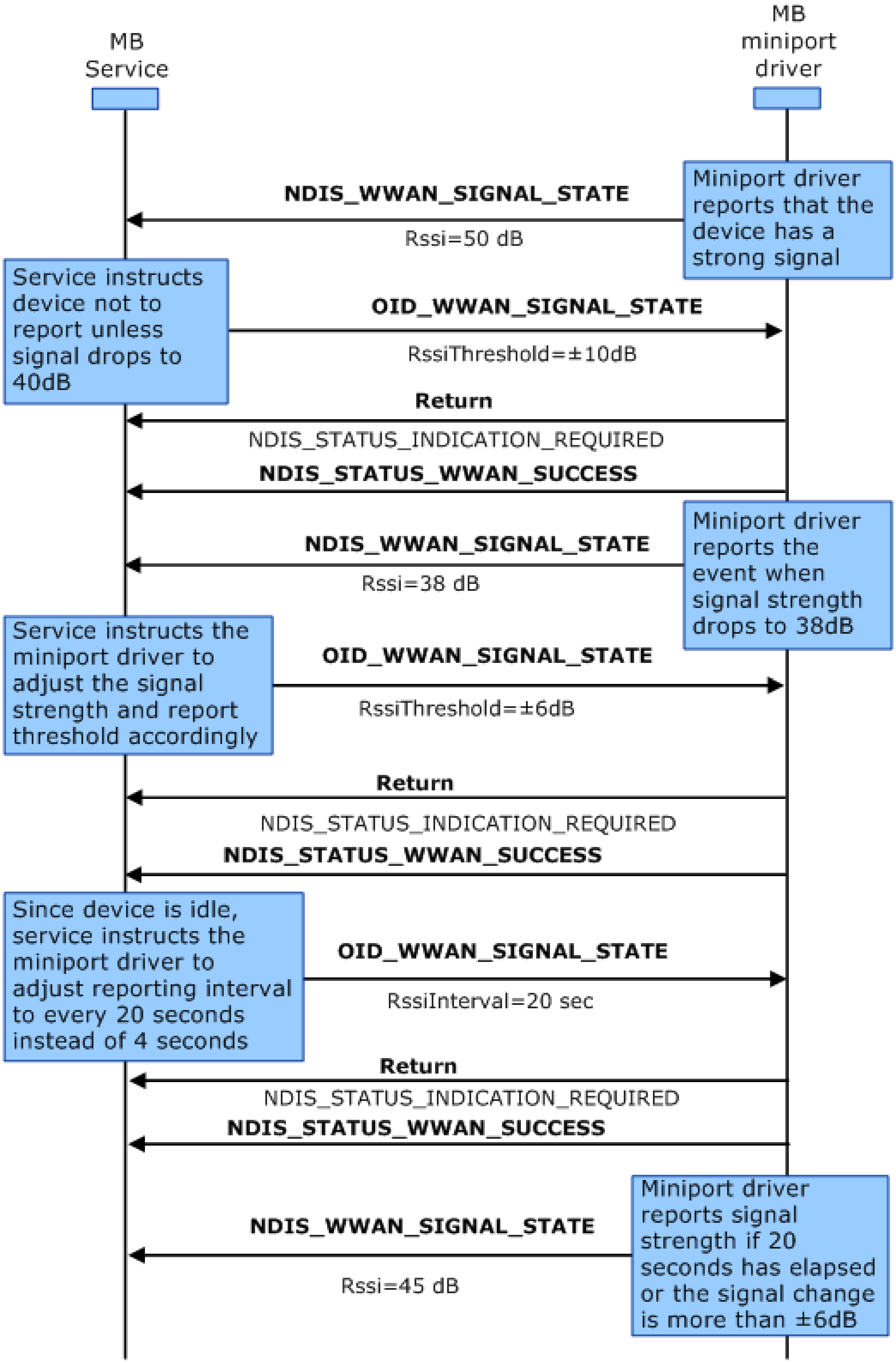
This topic describes the operations to report signal strength.

These operations require access to the network provider, but not to the Subscriber Identity Module (SIM card).

Be aware that in case of GSM-based devices, miniport drivers should send signal strength notifications only after the miniport driver has successfully registered with a network provider. For CDMA-based devices, miniport drivers can send signal strength notifications before the miniport driver has successfully registered with a network provider.

Signal Strength Indication Semantics

The following diagram shows the process that miniport drivers should follow to process signal strength indications. The MB Service adjusts the signal strength-reporting threshold and interval, based on the current device signal strength and how long the device has been idle. These actions are usually performed as part of the power management features provided by the MB Service. The labels in bold are OID identifiers or transactional flow control. The labels in regular text are the important flags within the OID structure.


MB Service
MB miniport driver
NDIS_WWAN_SIGNAL_STATE
Rssi=50 dB
Miniport driver reports that the device has a strong signal
Service instructs device not to report unless signal drops to 40dB
OID_WWAN_SIGNAL_STATE
RssiThreshold=±10dB
Return
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_SUCCESS
NDIS_WWAN_SIGNAL_STATE
Rssi=38 dB
Miniport driver reports the event when signal strength drops to 38dB
Service instructs the miniport driver to adjust the signal strength and report threshold accordingly
OID_WWAN_SIGNAL_STATE
RssiThreshold=±6dB
Return
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_SUCCESS
Since device is idle, service instructs the miniport driver to adjust reporting interval to every 20 seconds instead of 4 seconds
OID_WWAN_SIGNAL_STATE
RssiInterval=20 sec
Return
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_SUCCESS
NDIS_WWAN_SIGNAL_STATE
Rssi=45 dB
Miniport driver reports signal strength if 20 seconds has elapsed or the signal change is more than ±6dB

To update signal strength indications, use the following procedure:

1. The miniport driver sends **NDIS_WWAN_SIGNAL_STATE** to the MB Service.

2. The MB Service sends OID_WWAN_SIGNAL_STATE to the miniport driver. The miniport driver responds with a provisional acknowledgement (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

3. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.

4. The miniport driver sends **NDIS_WWAN_SIGNAL_STATE** to the MB Service.

5. The MB Service sends OID_WWAN_SIGNAL_STATE to the miniport driver. The miniport driver responds with a provisional acknowledgement (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

6. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.

7. The MB Service sends OID_WWAN_SIGNAL_STATE to the miniport driver. The miniport driver responds with a provisional acknowledgement (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

8. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.

9. The miniport driver sends **NDIS_WWAN_SIGNAL_STATE** to the MB Service.

MB Packet Context Management

Article • 03/14/2023

This topic describes the management of packet contexts, which are a specific set of network configuration parameters for setting up a *virtual circuit* or *flow* on top of the physical MB connection at layer 2. In GSM-based devices, this corresponds to the concept of a Packet Data Protocol (PDP). In CDMA-based devices, this corresponds to a network profile.

In most cases, the detailed settings of a packet context are either pre-provisioned by IHVs and/or network providers of the MB device, or provisioned through the network over-the-air (OTA) or using SMS. In either case, the end user is generally not required to provide most of the settings (for example, quality of service (QoS), security codes, mobile IP, and so on). However, the end user may need to provide the network access string, username, and password. It is these user configurable settings that constitute the content of a packet context from the perspective of the MB Service.

The MB driver model does not provide an explicit OID to set up or tear down the layer-2 connection for WWAN. Instead, activating the first packet context results in setting up the underlying layer-2 connection and deactivating the last packet context will effectively tear down the underlying layer-2 connection.

The MB driver model builds on these two constraints regarding the number of active packet contexts at any given time in the following manner:

1. Each packet context can be activated only one time.

2. Only a single packet context can be activated at any given time.

It is mandatory that any miniport driver that conforms to the MB driver model sets the **MaxActivatedContexts** member of the **WWAN_DEVICE_CAPS** structure to one, when responding to OID_WWAN_DEVICE_CAPS query requests. Even if a miniport driver sets this value to be greater than one, the MB Service ensures that, at most, only one packet context is activated at any given time.

Because each packet context can be activated no more than one time, a static packet context identifier can be used to identify the virtual circuit after being activated. The use of this static identifier is still valid as long as the first constraint still holds.

For more information about packet context management, see OID_WWAN_PROVISIONED_CONTEXTS and OID_WWAN_CONNECT.

Multiple PDP contexts

Article • 12/15/2021

Usage scenarios

UWP mobile broadband apps can take advantage of multiple Packet Data Protocol (PDP) contexts to activate a special PDP context and specify rules to route data traffic. These apps can create rules for specific destinations or for all data traffic.

When the mobile broadband app needs to exchange data with the network, it checks the available and connected networks. If the mobile broadband app has a special rule for any of these networks, it uses the Connection Manager API to open a special PDP context. If this connection is successful, the PDP context provides routing rules for this connection and transfers the data using networking APIs. The mobile broadband app should repeat this if it receives the NetworkStatusChanged event to see whether any connections have changed and whether it needs to open a PDP context for the new connection.

You can use multiple PDP contexts to enable premium services.

- Differentiated Billing – You can vary the data or billing restrictions by using multiple PDP contexts. For example, Contoso is a mobile operator that developed a data backup app for their customers. As a mobile operator, Contoso could create multiple PDP contexts and let premium subscribers use the app for free. All other subscribers are charged separately to use it.

- Rich Communication Services – A global initiative created by the GSM Association to provide rich communication services, such as an enhanced phonebook, enhanced messaging, and enriched calling. Rich Communication Services provide interoperability across mobile operators and offers new ways to use existing assets and capabilities to deliver high quality and innovative communication services.

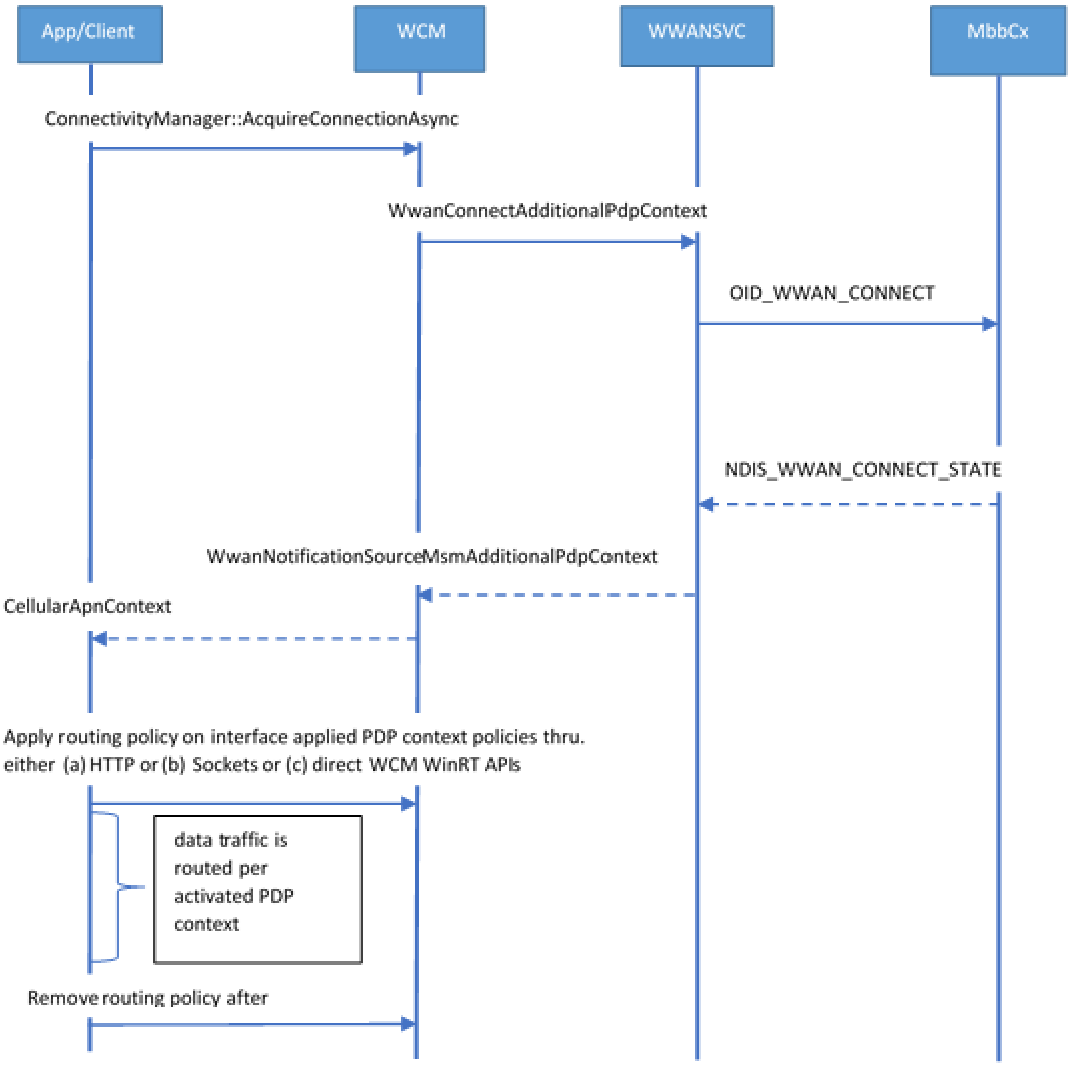
- Sponsored Connectivity – This allows users to a specific type of content without it going against their monthly data usage. The content provider makes an arrangement to reimburse the mobile operator by paying them directly, doing a revenue-sharing deal, or some other business arrangement.

- Personal Hotspot – Some mobile operators charge different rates when the connection is being used as a personal hotspot. You can use multiple PDP contexts to differentiate between the two.

For more information, see Developing apps using multiple PDP contexts.

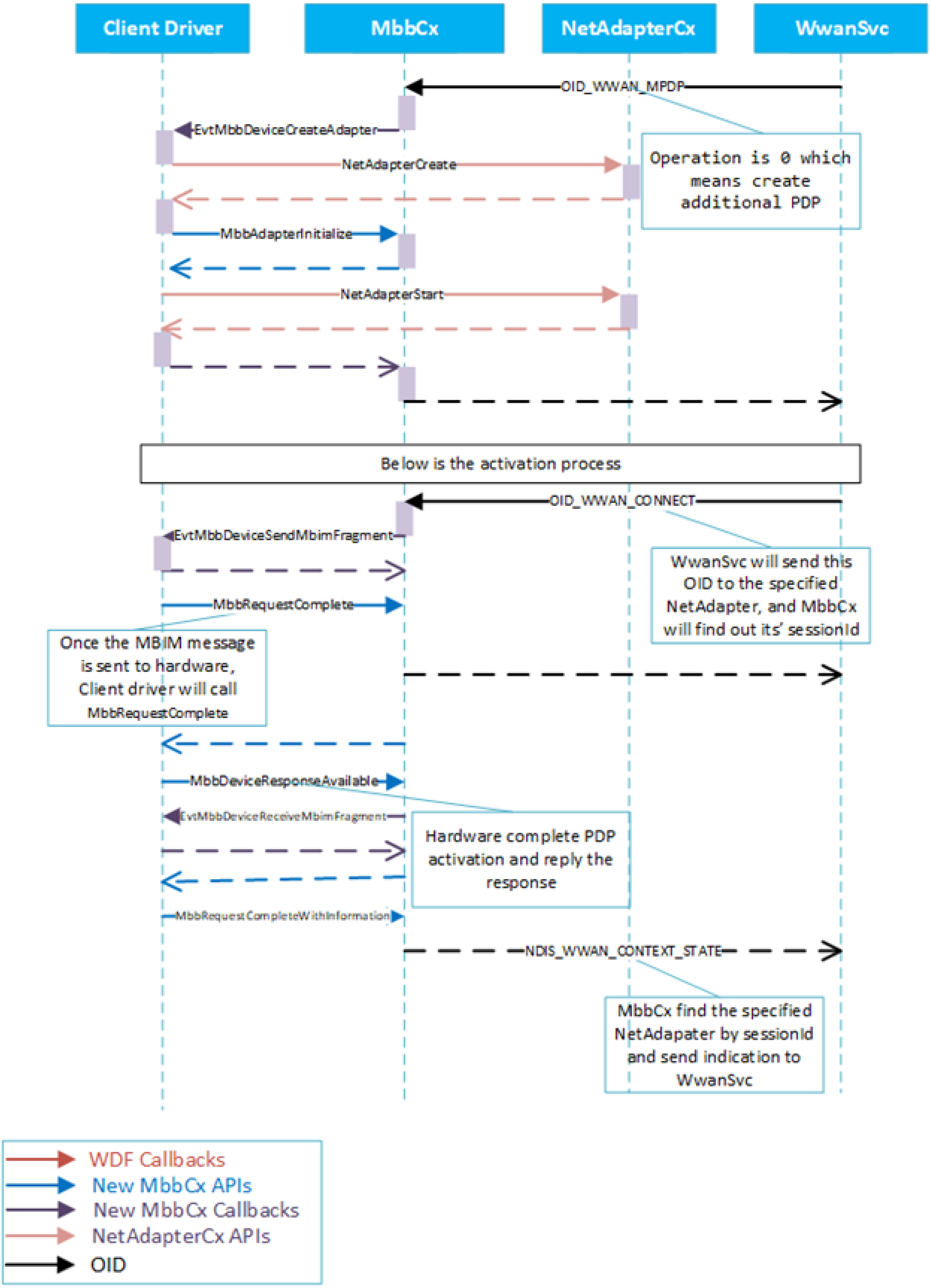
Primary Flow

App activates additional PDP contexts:


App/Client
WCM
WWANSVC
MbbCx
ConnectivityManager::AcquireConnectionAsync
WwanConnectAdditionalPdpContext
OID_WWAN_CONNECT
NDIS_WWAN_CONNECT_STATE
WwanNotificationSourceMsmAdditionalPdpContext
CellularApnContext
Apply routing policy on interface applied PDP context policies thru. either (a) HTTP or (b) Sockets or (c) direct WCM WinRT APIs
data traffic is routed per activated PDP context
Remove routing policy after

UWP app activate additional PDP context connection

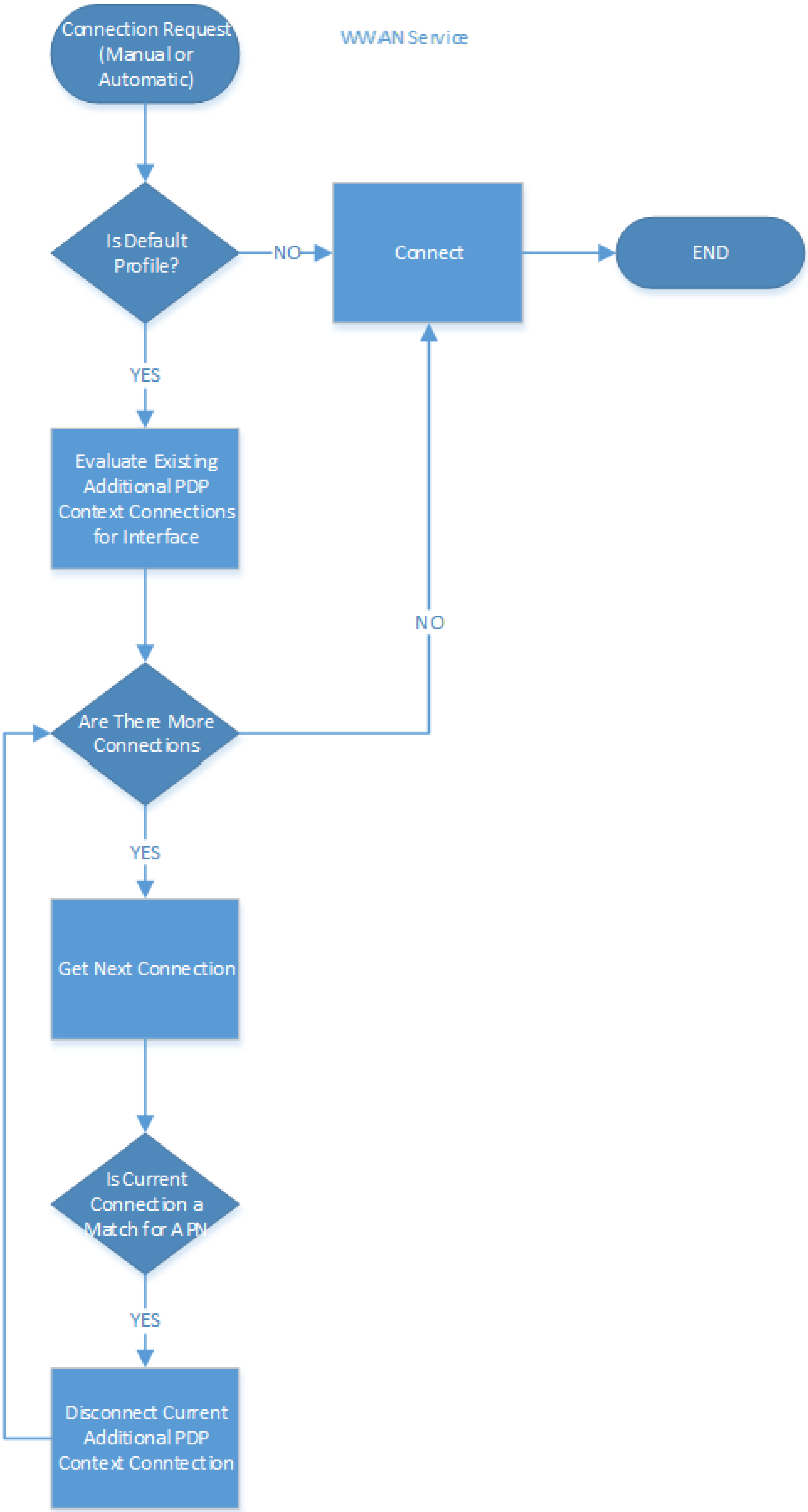
Additional NetAdapter Initialization


Client Driver
MbbCx
NetAdapterCx
WwanSvc
OID_WWAN_MPDP
EvtMbbDeviceCreateAdapter
Operation is 0 which means create additional PDP
NetAdapterCreate
MbbAdapterInitialize
NetAdapterStart
Below is the activation process
OID_WWAN_CONNECT
EvtMbbDeviceSendMbimFragment
WwanSvc will send this OID to the specified NetAdapter, and MbbCx will find out its' sessionId
MbbRequestComplete
Once the MBIM message is sent to hardware, Client driver will call MbbRequestComplete
MbbDeviceResponseAvailable
EvtMbbDeviceReceiveMbimFragment
Hardware complete PDP activation and reply the response
MbbRequestCompleteWithInformation
NDIS_WWAN_CONTEXT_STATE
MbbCx find the specified NetAdapater by sessionId and send indication to WwanSvc
WDF Callbacks
New MbbCx APIs
New MbbCx Callbacks
NetAdapterCx APIs
OID

Decision Logic in WwanSvc for Additional PDP Context Connections

1. Double check and update the "Is Default Profile" condition logic, as it is no longer applicable.

2. WCM should no longer use the *cost* property of the default profile.
3. If the new additional pdp context APN request coincides with the default internet APN, disconnect the current additional PDP context.


Connection Request (Manual or Automatic)
WWAN Service
Is Default Profile?
NO
Connect
END
YES
Evaluate Existing Additional PDP Context Connections for Interface
NO
Are There More Connections
YES
Get Next Connection
Is Current Connection a Match for APN
YES
Disconnect Current Additional PDP Context Connection

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to the device Cellular modem driver and run the test: Win6_4.MB.GSM.Data.TestMPDP.

MB Multiple PDP context Troubleshooting Guide

1. Logs can be collected and decoded using these instructions: MB Collecting Logs
2. Open the .txt file in TextAnalysisTool
3. Load the Bacis Connectivity filter

Sample log

```
e 04-01 12:39:12.798 P4912 T8420 Microsoft-Windows-WWAN-NDISUIO-EVENTS WWAN
NDISUIO Event: OID request sent to the driver
0                     Info Microsoft-Windows-WWAN-NDISUIO-EVENTS
e 04-01 12:39:12.798 P4912 T8420 Windows Mobile Broadband Class Driver Event
Provider [1] Miniport Request called Request=0xFFFFE3862EFF4A80,
OID=0xE010149, OID name=OID_WWAN_MPDP RequestId=0x10F, RequestHandle=0x0,
Type=1, InformationLength=32 0                    Info Windows Mobile
Broadband Class Driver Event Provider
w 04-01 12:39:12.798 P4912 T8420 mbbcx         [ReqMgr][ReqId=0x04ad]
Request created for OID_WWAN_MPDP [RequestContext=0xFFFFE386247597F0
OidRequest=0xFFFFE3862EFF4A80] SET=0x00000001(TRUE) MbbReqMgrCreateRequest
requestmanager_cpp702 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.798 P4912 T8420 mbbcx         [ReqFsm][ReqId=0x04ad]
Transition: MbbRequestStateReady -> MbbRequestStateDispatching
event=MbbRequestEventDispatch MbbReqMgrQueueEvent  requestmanager_cpp1002
TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.798 P4912 T8420 mbbcx         [ReqMgr][Timer]
MbbTimerTypeRequest already armed at 3fc53, not re-arming
MbbReqMgrTimerArm    requestmanager_cpp1269 TRACE_LEVEL_WARNING
w 04-01 12:39:12.798 P4912 T8420 mbbcx         [ReqMgr][ReqId=0x04ad],
IsPoweredRequest [0x00000001(TRUE)], IsSerialized[0x00000001(TRUE)],
IsQueueEmpty[0x00000001(TRUE)], DispatchRequest [0x00000000(FALSE)]
MbbReqFsmDispatching requestmanager_cpp1522 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.798 P4912 T8420 mbbcx
MbxDevice::QueuePoweredRequest: WDFREQUEST (0x00001C79D4B06DB8) is sent
MbxDevice::QueuePoweredRequest mbxdevice_cpp1339 TRACE_LEVEL_INFORMATION
e 04-01 12:39:12.798 P4912 T8420 Windows Mobile Broadband Class Driver Event
Provider [1] Miniport REQUEST exited with status=The operation that was
requested is pending completion., Request=0xFFFFE3862EFF4A80 0
Info Windows Mobile Broadband Class Driver Event Provider
```

```
w 04-01 12:39:12.808 P0004 T0376 mbbcx         EvtCxPreD0Entry:
previousPowerState: 3
EvtCxPreD0Entry      mbxdevice_cpp1583 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.808 P0004 T0376 cxwmbclass    EvtDeviceD0Entry:
previousPowerState: 3
EvtDeviceD0Entry     power_cpp19 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.808 P0004 T0376 usbbus        Entered Enabled=1
MbbBusSetNotificationState businit_c2606 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.808 P0004 T0376 usbbus        MbbUsbDeviceStartDataPipes:
Entered
MbbUsbDeviceStartDataPipes datapipe_c228 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.809 P0004 T0376 usbbus        MbbUsbDeviceStartDataPipes:
Exited
MbbUsbDeviceStartDataPipes datapipe_c286 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.809 P0004 T0376 mbbcx
MbxDevice::PostD0EntryPostHardwareEnabled: previousPowerState: 3
MbxDevice::PostD0EntryPostHardwareEnabled mbxdevice_cpp1933
TRACE_LEVEL_INFORMATION
e 04-01 12:39:12.810 P0004 T0376 Microsoft-Windows-NDIS Miniport {c2d9b876-
8b20-4cdd-a944-044fd39a97dc}, DeviceState[0x1]
Power                0 Microsoft-Windows-NDIS
e 04-01 12:39:12.812 P0004 T0376 Microsoft-Windows-NDIS Miniport {156ce913-
cc77-487d-8838-4811ce860b0e}, DeviceState[0x1]
Power                0 Microsoft-Windows-NDIS
e 04-01 12:39:12.813 P0004 T0376 Microsoft-Windows-NDIS Miniport {1e58668f-
811b-407d-b288-e1f57a432a24}, DeviceState[0x1]
Power                0 Microsoft-Windows-NDIS
e 04-01 12:39:12.815 P0004 T0376 Microsoft-Windows-NDIS Miniport {468c0f8c-
df7f-4619-85fd-c24ccebdeda3}, DeviceState[0x1]
Power                0 Microsoft-Windows-NDIS
w 04-01 12:39:12.815 P0004 T0376 mbbcx
MbxDevice::EnableWakeReasonReporting
MbxDevice::EnableWakeReasonReporting mbxdevice_cpp2099
TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0004 T0376 cxwmbclass    EvtDeviceDisarmWakeFromS0:
The device is disarmed for wake
EvtDeviceDisarmWakeFromS0 power_cpp130 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0004 T0376 mbbcx         MbxDevice::DisarmWake: Start
MbxDevice::DisarmWake mbxdevice_cpp1685 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0004 T0376 mbbcx         [ReqMgr][ReqId=0x04ae]
Internal Request created [RequestContext=0xFFFFE38624757650]
MbbReqMgrCreateRequest requestmanager_cpp713 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0004 T0376 mbbcx         [ReqFsm][ReqId=0x04ae]
Transition: MbbRequestStateReady -> MbbRequestStateDispatching
event=MbbRequestEventDispatch MbbReqMgrQueueEvent  requestmanager_cpp1002
TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0004 T0376 mbbcx         [ReqMgr][Timer]
MbbTimerTypeRequest already armed at 3fc53, not re-arming
MbbReqMgrTimerArm    requestmanager_cpp1269 TRACE_LEVEL_WARNING
w 04-01 12:39:12.815 P0004 T0376 mbbcx         [ReqMgr][ReqId=0x04ae],
IsPoweredRequest [0x00000000(FALSE)], IsSerialized[0x00000001(TRUE)],
IsQueueEmpty[0x00000001(TRUE)], DispatchRequest [0x00000001(TRUE)]
MbbReqFsmDispatching requestmanager_cpp1522 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0004 T0376 mbbcx         [ReqFsm][ReqId=0x04ae]
Transition: MbbRequestStateDispatching -> MbbRequestStateSendPending
```

```
event=MbbRequestEventStart MbbReqMgrQueueEvent  requestmanager_cpp1002
TRACE_LEVEL_INFORMATION
e 04-01 12:39:12.815 P0004 T0376 Windows Mobile Broadband Class Driver Event
Provider Sending command with the following parameters:
Caller Request Id: 0x0
Driver Request Id: 0
Service Id: {000004ae-cc33-a289-bbbc-4f8bb6b0133e}
Command Name: REDACTED-EMBEDDED-HEXREDACTED-EMBEDDED-HEXREDACTED-EMBEDDED-
HEXREDACTED-EMBEDDED-HEXBASIC_NOTIFY_DEVICE_SERVICE_UPDATES
Command Id: 19
Payload Length: 324
Payload:
0x0600000034000000640000009800000028000000C000000018000000D80000001C000000F4
00000034000000280100001C000000A289CC33BCBB8B4FB6B0133EC2AAE6DF14000000010000
0002000000030000000400000005000000060000000700000008000000090000000A0000000B
0000000C0000000D0000000F000000100000001300000014000000150000001600000017 0000
00533FBEEB14FE44679F9033A223E56C3F050000000100000002000000030000000400000005
000000E550A0C85E82479E82F710ABF4C3351F01000000010000001D2B5FF70AA148B2AA5250
F15767174E0200000001000000030000003D01DCC5FEF54D050D3ABEF7058E9AAF0800000001
0000000300000004000000050000000600000007000000080000000A00000068223D049F6C4E
0F822D28441FB72340020000000100000002000000 0                   Info Windows
Mobile Broadband Class Driver Event Provider
e 04-01 12:39:12.815 P0004 T0376 Windows Mobile Broadband Class Driver Event
Provider for OPN Sending command MessageType: 0x3, MessageLength: 372,
MessageTransactionId: 533, TotalFragments: 1, CurrentFragment: 0, ServiceId:
{a289cc33-bcbb-8b4f-b6b0-133ec2aae6df}, CommandId: 19, CommandType: 1,
InformationBufferLength: 324, InformationBuffer:
0x0600000034000000640000009800000028000000C000000018000000D80000001C000000F4
00000034000000280100001C000000A289CC33BCBB8B4FB6B0133EC2AAE6DF14000000010000
0002000000030000000400000005000000060000000700000008000000090000000A0000000B
0000000C0000000D0000000F00000010000000130000001400000015000000160000001700 00
00533FBEEB14FE44679F9033A223E56C3F050000000100000002000000030000000400000005
000000E550A0C85E82479E82F710ABF4C3351F01000000010000001D2B5FF70AA148B2AA5250
F15767174E0200000001000000030000003D01DCC5FEF54D050D3ABEF7058E9AAF0800000001
0000000300000004000000050000000600000007000000080000000A00000068223D049F6C4E
0F822D28441FB72340020000000100000002000000 0                   Info Windows
Mobile Broadband Class Driver Event Provider for OPN
w 04-01 12:39:12.815 P0004 T0376 usbbus        Sending 372 bytes on control
channel
MbbBusSendMessageFragment businit_c1472 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0004 T0376 usbbus        SetActivityIdForRequest
succeeded. Set request activityId = 207b1c4a-085c-0001-270f-83205c08d601
SetActivityIdForRequest businit_c1383 TRACE_LEVEL_INFORMATION
e 04-01 12:39:12.815 P0004 T0376 Windows Mobile Broadband Class Driver Event
Provider [1] Send encapsulted command MessageType=0x3, MessageLength=372,
TransactionId=533, TotalFrags=1, CurrentFrag=0, ServiceId={33cc89a2-bbbc-
4f8b-b6b0-133ec2aae6df}, CID=19, CommandType=1, InfoLength=324 0
Info Windows Mobile Broadband Class Driver Event Provider
w 04-01 12:39:12.815 P0004 T0376 mbbcx         [Util][ReqId=0x04ae]
[TID=0x00000215] Pending send Fragment 00/01
MbbUtilSendMessageFragments util_cpp1269 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0000 T0000 usbbus        CompletionRoutine() for
request 00001C79D4105668 status=STATUS_SUCCESS
SendCompletionRoutine businit_c1398 TRACE_LEVEL_INFORMATION
w 04-01 12:39:12.815 P0000 T0000 mbbcx         [Util][ReqId=0x04ae]
```

```
[TID=0x00000215] 01/01 fragment completed with status=STATUS_SUCCESS
MbbUtilSendMessageFragmentComplete util_cpp1401 TRACE_LEVEL_INFORMATION
e 04-01 12:39:12.815 P0000 T0000 Windows Mobile Broadband Class Driver Event
Provider Sending command completed with status STATUS_SUCCESS. Command was
sent with the following parameters:
```

MB DNS Updates

Article • 03/14/2023

This topic describes the operations to notify the MB Service about DNS address updates.

Miniport drivers should set the **NameServer** registry key to update Windows about DNS address changes. The following table describes the appropriate registry key, the expected value and an example string for IPv4 and IPv6 networks. If a miniport driver supports only IPv4 networks, it should set only the IPv4 registry key. Miniport drivers should set the appropriate registry key(s) before they notify Windows about media connect events by sending **NDIS_STATUS_LINK_STATE** notifications.

IPv4 / IPv6	Registry Key	Value	Example
IPv4	HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces\InterfaceGUID\NameServer	Space-separated DNS server IPv4 addresses	10.20.30.41 10.20.30.40
IPv6	HKLM\SYSTEM\CurrentControlSet\Services\Tcpip6\Parameters\Interfaces\InterfaceGUID\NameServer	Space-separated DNS server IPv6 addresses	2001:4898:7001:f000:1:2:3:4 2001:4898:7001:f000:1:2:3:5

These operations should be used only when the miniport driver specifies **EnableDhcp** to equal zero in its INF file. That is, the miniport driver does not implement DHCP.

For more information about processing IP address notifications, see Guidelines for MB Miniport driver IP Address Notifications.

MB SMS Operations

Article • 03/14/2023

This topic describes the operations to configure, read/receive, send, and delete messages using Short Message Service (SMS) capabilities of an MB device.

SMS support is mandatory. Miniport drivers must set the appropriate send and receive SMS capability flags that they support when processing OID_WWAN_DEVICE_CAPS query requests in the **WwanSmsCaps** member of the **WWAN_DEVICE_CAPS** structure. If miniport drivers do not support SMS, they should specify WWAN_SMS_CAPS_NONE and return WWAN_STATUS_SMS_UNKNOWN_ERROR for all SMS-related OIDs.

Miniport drivers should only process SMS operations after OID_WWAN_READY_INFO returns **WwanReadyStateInitialize** as the device ready-state. Miniport drivers should process some SMS operations, such as sending a SMS message, only after the device is registered on a provider network (though not necessarily data service registration).

The MB Service does not differentiate between different message stores available in the device. Therefore, miniport drivers must handle all message stores and project a single virtual message store accessed by means of a virtual index. For example, if the device has three message stores, the miniport driver must handle all of them collectively and present them as a single message store to the service.

The MB driver model supports the following SMS Operations:

- SMS configuration
- Read SMS
- Send SMS
- Delete SMS

We recommend miniport drivers support SMS configuration, read, send, and delete operations, as well as notifying the user of any new SMS message received by a device.

For more information about SMS operations, see OID_WWAN_SMS_CONFIGURATION, OID_WWAN_SMS_READ, OID_WWAN_SMS_SEND, OID_WWAN_SMS_DELETE, and OID_WWAN_SMS_STATUS.

Relevant Services and Drivers

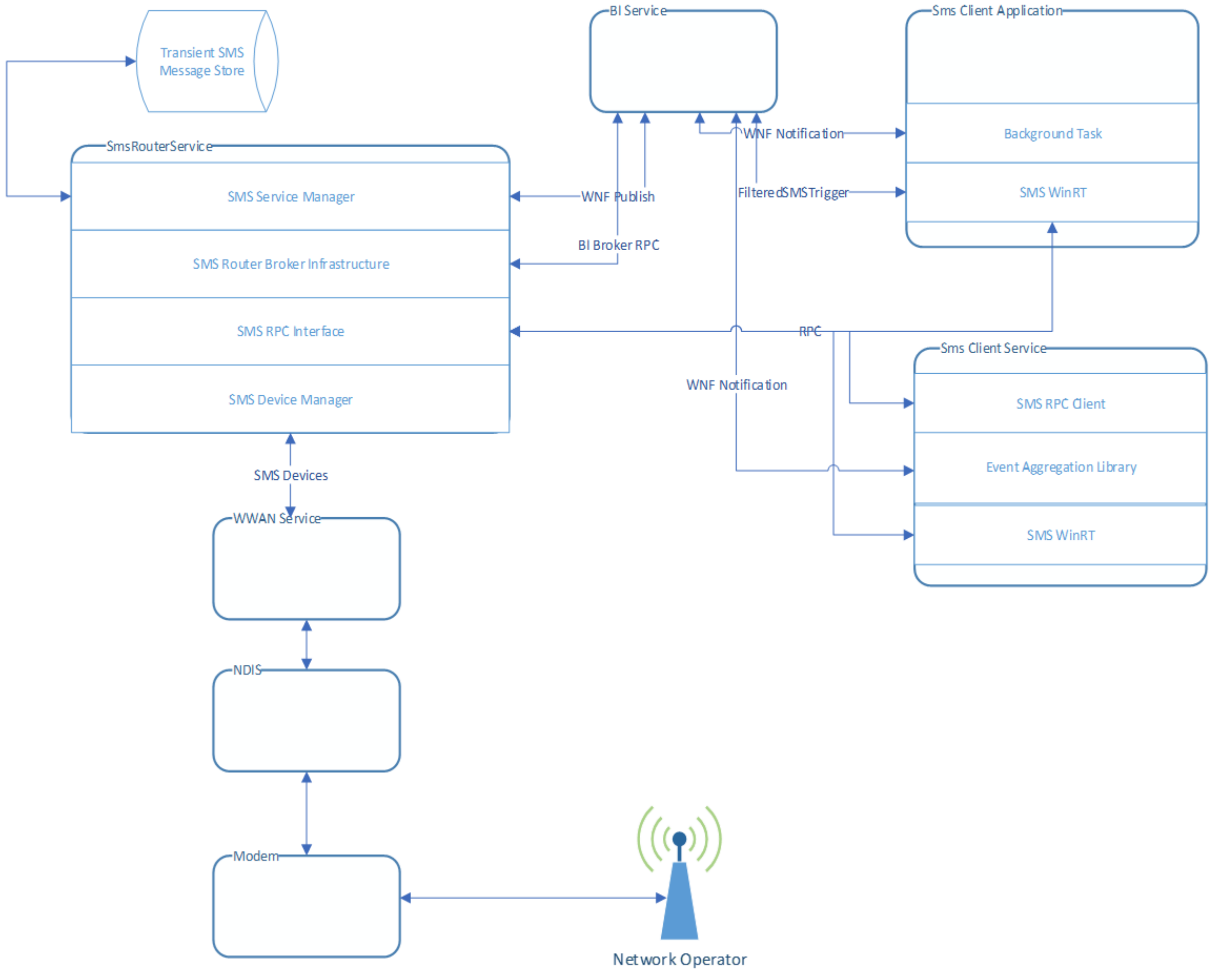
SmsRouterSvc.dll - The service that interacts with WwanSvc to handle sending and receiving images

MbSmsApi.dll - Implementation of WinRT SMS API

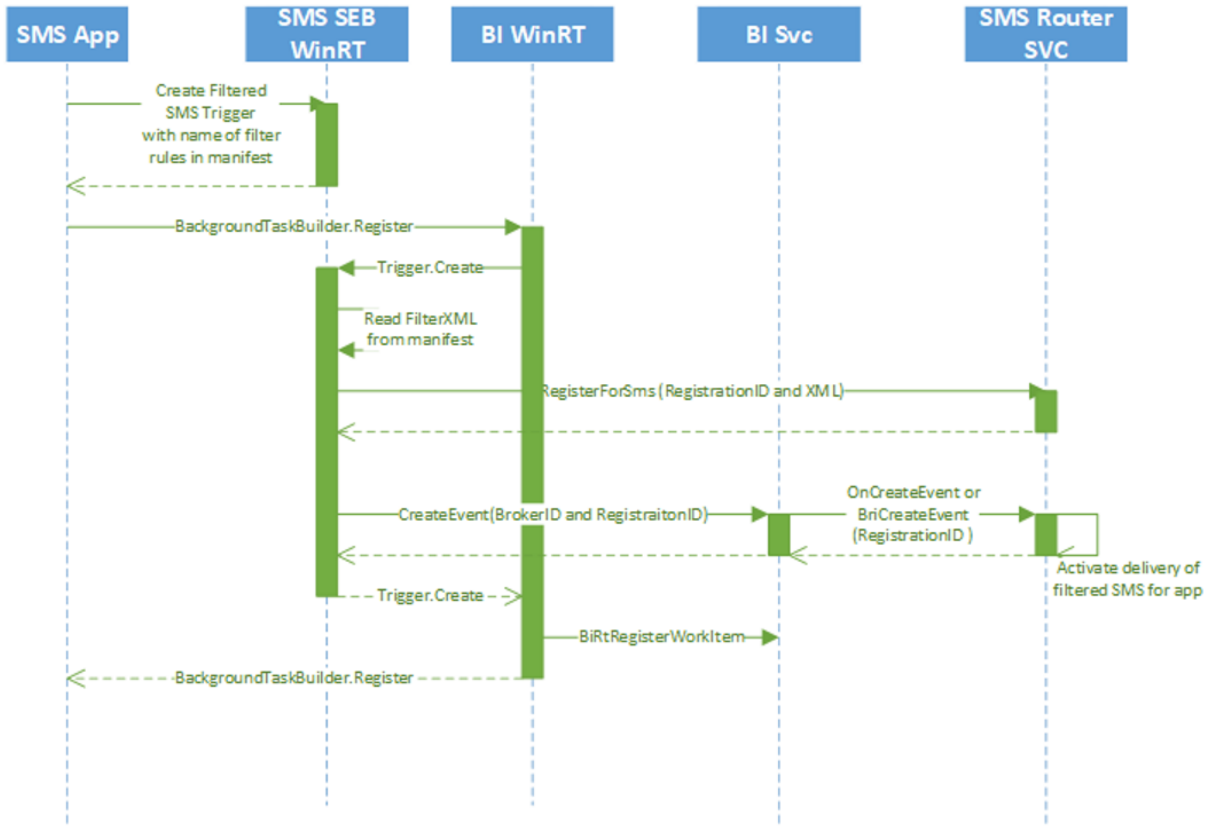
UT_SmsRouter.dll - Is onboarded to Real Device Testing

SMS Architecture/Flows

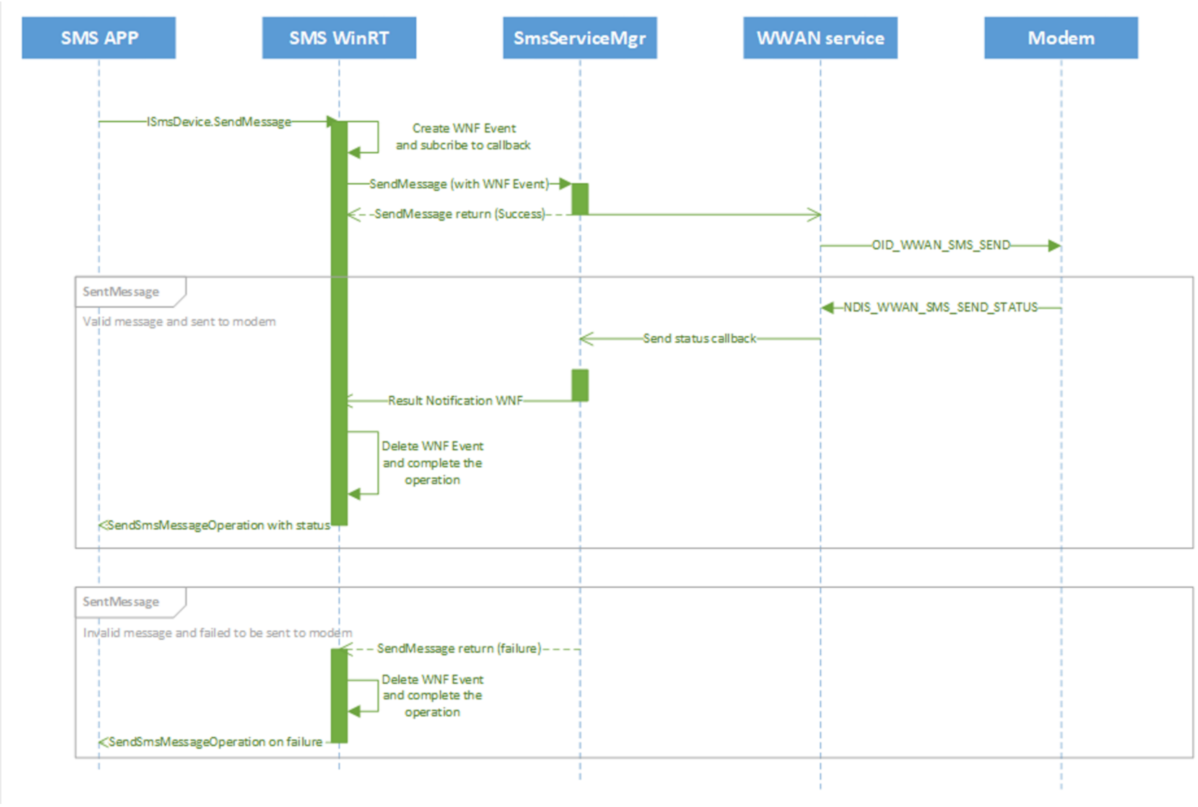
SMS Block Diagram



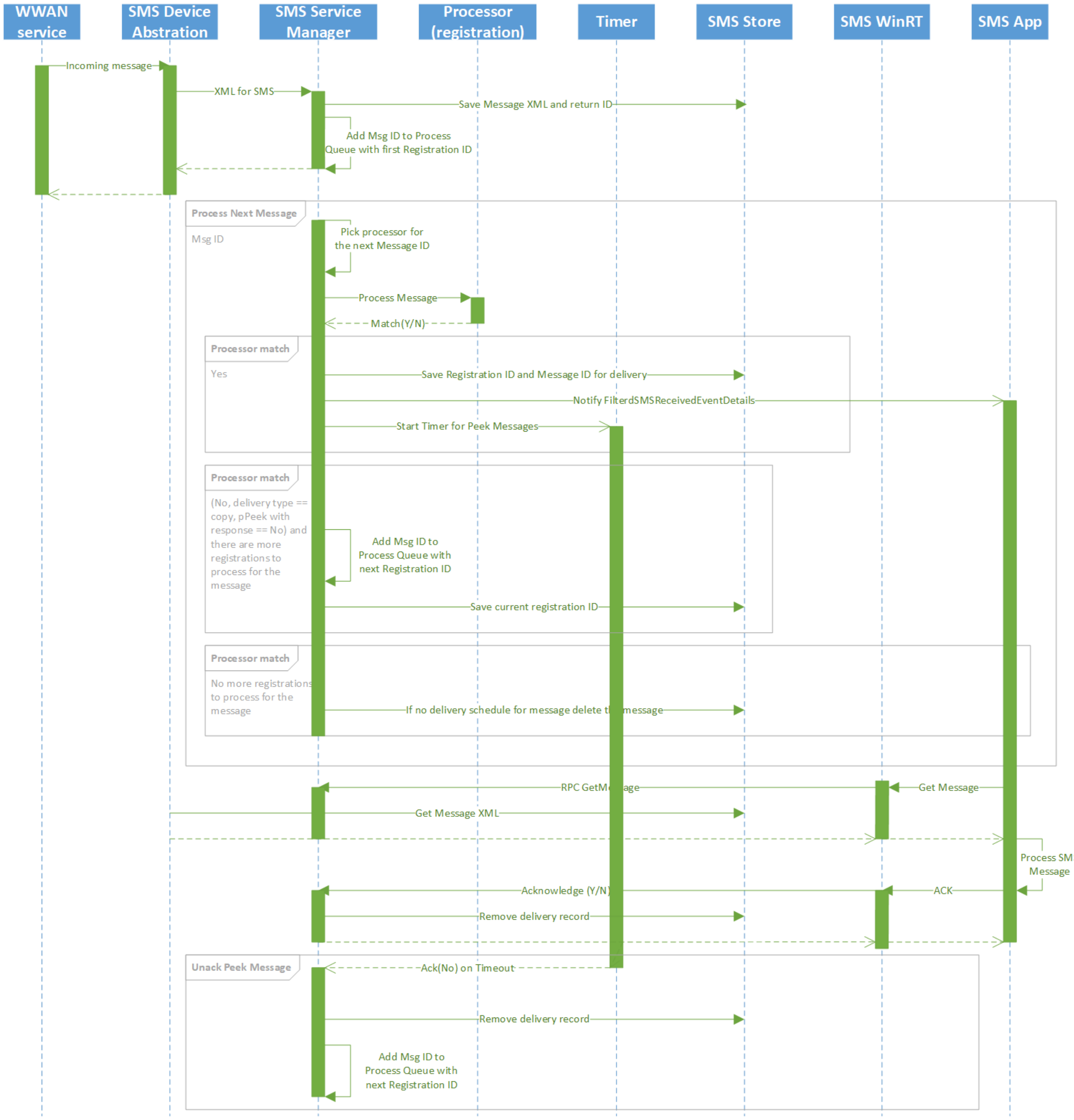
SMS App Registration


SMS App
SMS SEB WinRT
BI WinRT
BI Svc
SMS Router SVC
Create Filtered SMS Trigger with name of filter rules in manifest
BackgroundTaskBuilder.Register
Trigger.Create
Read FilterXML from manifest
RegisterForSms (RegistrationID and XML)
CreateEvent(BrokerID and RegistraitonID)
OnCreateEvent or BriCreateEvent (RegistrationID)
Activate delivery of filtered SMS for app
Trigger.Create
BiRtRegisterWorkItem
BackgroundTaskBuilder.Register

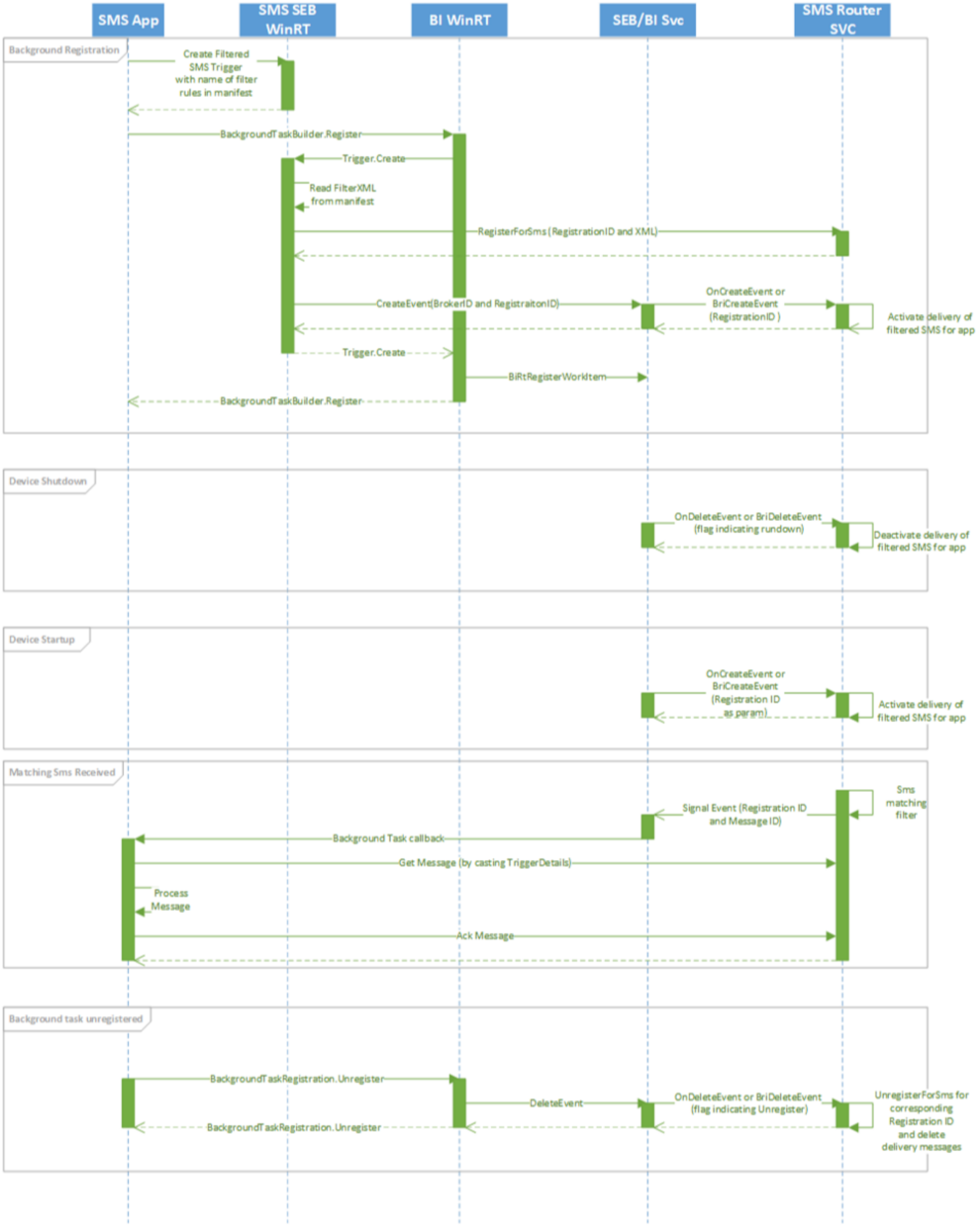
Send SMS


SMS APP
SMS WinRT
SmsServiceMgr
WWAN service
Modem
ISmsDevice.SendMessage
Create WNF Event and subcribe to callback
SendMessage (with WNF Event)
SendMessage return (Success)
OID_WWAN_SMS_SEND
SentMessage
Valid message and sent to modem
NDIS_WWAN_SMS_SEND_STATUS
Send status callback
Result Notification WNF
Delete WNF Event and complete the operation
SendSmsMessageOperation with status
SentMessage
Invalid message and failed to be sent to modem
SendMessage return (failure)
Delete WNF Event and complete the operation
SendSmsMessageOperation on failure

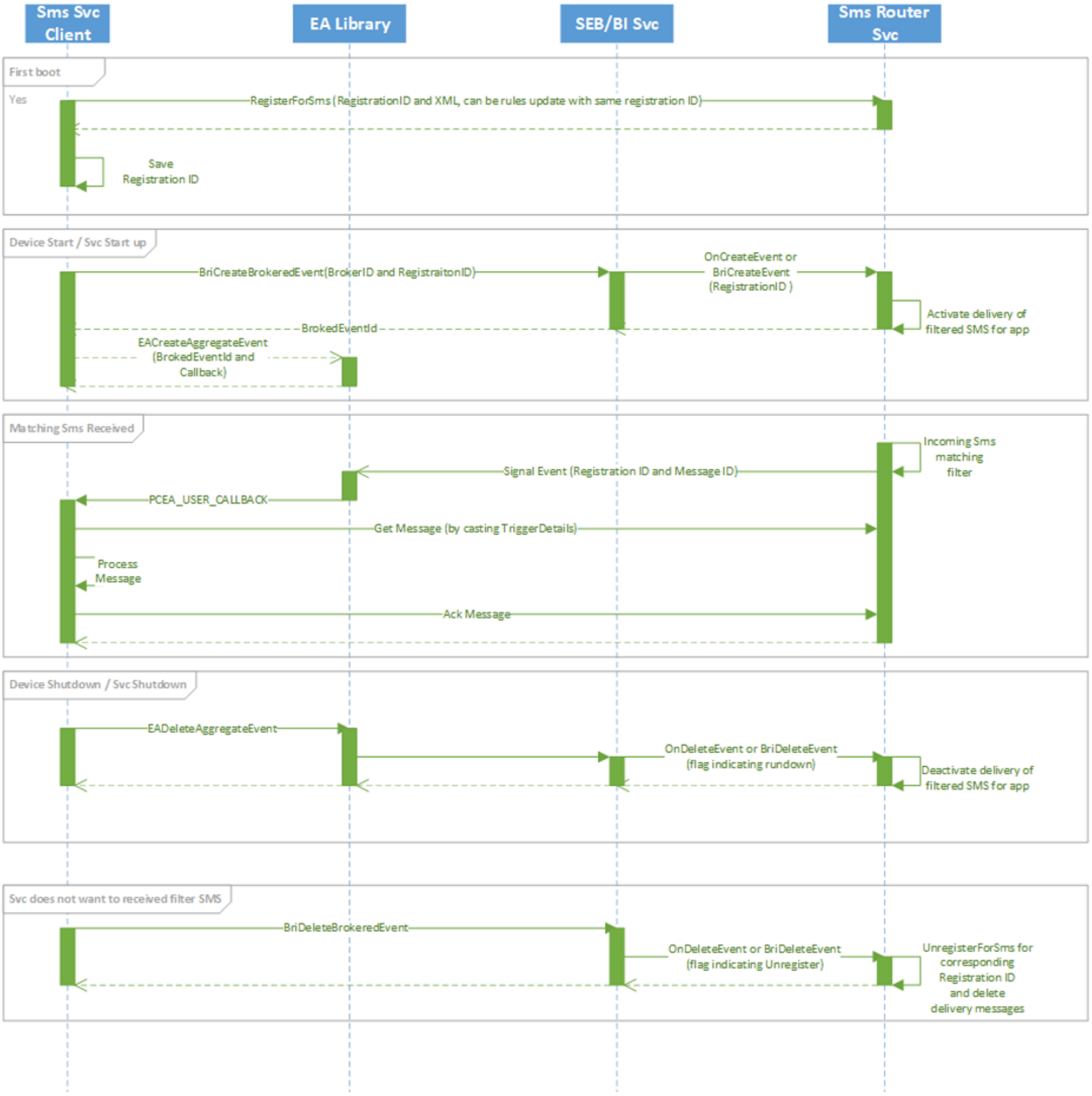
API Receive Message


WWAN service
SMS Device Abstraction
SMS Service Manager
Processor (registration)
Timer
SMS Store
SMS WinRT
SMS App
Incoming message
XML for SMS
Save Message XML and return ID
Add Msg ID to Process Queue with first Registration ID
Process Next Message
Msg ID
Pick processor for the next Message ID
Process Message
Match(Y/N)
Processor match
Yes
Save Registration ID and Message ID for delivery
Notify FilterdSMSReceivedEventDetails
Start Timer for Peek Messages
Processor match
(No, delivery type == copy, pPeek with response == No) and there are more registrations to process for the message
Add Msg ID to Process Queue with next Registration ID
Save current registration ID
Processor match
No more registrations to process for the message
If no delivery schedule for message delete the message
RPC GetMessage
Get Message
Get Message XML
Process SMS Message
Acknowledge (Y/N)
ACK
Remove delivery record
Unack Peek Message
Ack(No) on Timeout
Remove delivery record
Add Msg ID to Process Queue with next Registration ID

App Lifecycle


SMS App
SMS SEB WinRT
BI WinRT
SEB/BI Svc
SMS Router SVC
Background Registration
Create Filtered SMS Trigger with name of filter rules in manifest
BackgroundTaskBuilder.Register
Trigger.Create
Read FilterXML from manifest
RegisterForSms (RegistrationID and XML)
CreateEvent(BrokerID and RegistraitonID)
OnCreateEvent or BriCreateEvent (RegistrationID)
Activate delivery of filtered SMS for app
Trigger.Create
BiRtRegisterWorkItem
BackgroundTaskBuilder.Register
Device Shutdown
OnDeleteEvent or BriDeleteEvent (flag indicating rundown)
Deactivate delivery of filtered SMS for app
Device Startup
OnCreateEvent or BriCreateEvent (Registration ID as param)
Activate delivery of filtered SMS for app
Matching Sms Received
Sms matching filter
Signal Event (Registration ID and Message ID)
Background Task callback
Get Message (by casting TriggerDetails)
Process Message
Ack Message
Background task unregistered
BackgroundTaskRegistration.Unregister
DeleteEvent
OnDeleteEvent or BriDeleteEvent (flag indicating Unregister)
UnregisterForSms for corresponding Registration ID and delete delivery messages
BackgroundTaskRegistration.Unregister

Service Lifecycle


Sms Svc Client
EA Library
SEB/BI Svc
Sms Router Svc
First boot
Yes
RegisterForSms (RegistrationID and XML, can be rules update with same registration ID)
Save Registration ID
Device Start / Svc Start up
BriCreateBrokeredEvent(BrokerID and RegistraitonID)
OnCreateEvent or BriCreateEvent (RegistrationID)
Activate delivery of filtered SMS for app
BrokedEventId
EACreateAggregateEvent (BrokedEventId and Callback)
Matching Sms Received
Incoming Sms matching filter
Signal Event (Registration ID and Message ID)
PCEA_USER_CALLBACK
Get Message (by casting TriggerDetails)
Process Message
Ack Message
Device Shutdown / Svc Shutdown
EADeleteAggregateEvent
OnDeleteEvent or BriDeleteEvent (flag indicating rundown)
Deactivate delivery of filtered SMS for app
Svc does not want to received filter SMS
BriDeleteBrokeredEvent
OnDeleteEvent or BriDeleteEvent (flag indicating Unregister)
UnregisterForSms for corresponding Registration ID and delete delivery messages

Testing

Automated SMS Tests

The following tests are automated and onboarded to the RI-TP. They are run daily and should pass 100%.

- MobilebroadbandExperience\SmsApi
- MobilebroadbandExperience\SMSCDMA
- MobilebroadbandExperience\SMSDecodingTests
- MobilebroadbandExperience\SMSEncodingTests
- WWAN\SMS\Service\UnitTests

SmsApi Tests have different versions that run on desktop and onecoreuap. Desktop still uses *vnelib.dll* (C++ version) because the CDMA part of SMS is not ported to *vnelibrary.dll* (C# version). Therefore you will find two versions of functional test lists.

Hardware Lab Kit (HLK) Tests

These are all the currently available HLK tests related to MB-SMS:

- TestSms
  - CDMA, GSM
- TestSmsStoreFull
  - CDMA, GSM
- TestWake
  - CDMA: IncomingDataPacket, RegisterStateChange
  - GSM: IncomingDataPacket, RegisterStateChange
- TestSimBad
  - GSM
- TestDeviceCapsEx
  - CDMA
  - GSM
- TestSIMNotInserted
  - GSM

Running Tests

Via netsh, you can run the test lists and HLK tests. For more information on using the netsh tool see **netsh mbn** and **netsh mbn test installation**.

```
netsh mbn test feature=sms testpath="C:\data\test\bin"
taefpath="C:\data\test\bin" param="AccessString=internet"
```

Logs can be collected and decoded using these instructions: MB Collecting Logs.

Special Messages

Operator Messages

Operators can provision devices to handle particular messages earlier. This is no longer available, but the feature has not been completely removed yet. The code ProvisioningEngine processes the Operator Notifications. For more information see Operator Notifications and Operator Events.

Broadcast Messages

For more information on emergency alerts through SMS see SmsBroadcastMessage and SmsBroadcastType.

UWP Capabilities for SMS

Legacy SMS API

There are two legacy SMS APIs, *sms* and *smsSend*.

Latest SMS API

- *cellularMessaging*

For more information see UWP SMS.

Other Relevant Links

- Developing SMS apps

MB Vendor Specific Operations

Article • 03/14/2023

This topic describes the operations to interface vendor specific operations to the MB Service.

For more information about vendor specific operations, see OID_WWAN_VENDOR_SPECIFIC.

USSD Overview

Article • 12/15/2021

Unstructured Supplementary Service Data (USSD) is a communication protocol used by Global System for Mobile Communications (GSM) devices to communicate with mobile network operators (typically referred to as simply "MO").

To understand USSD it is helpful to compare it to its most closely-related sibling: short message service (SMS). USSD and SMS are both GSM standards, meaning they were introduced as features in the second generation of mobile devices. In contrast with SMS however, USSD is a session-based connection. While SMS is used for short session-less messaging, USSD is typically used for command and control of a mobile device. As it is a necessary to maintain a session, USSD does not support store-and-forward capability as SMS does. Both USSD and SMS messages are sent with 7-bit GSM-compliant characters, but USSD maxes out at 184 characters in contrast with 160 for SMS.

USSD messages may be sent from a mobile phone by opening the dialer and typing a code. Not all codes are supported by every phone or MO. In some cases, the phone software or operating system may prevent manually sending codes. One required code that must be implemented is *#06#. This code returns the International Mobile Equipment Identifier (IMEI) of the modem, but some phones will prevent you from dialing this directly. If you follow conventional means of locating the IMEI of the modem through your phone's settings, that number was retrieved using this code.

If the phone hardware can directly handle a code's command like in the IMEI example, no network session will be initiated. Other codes that require network communication will open a session and then send a message consisting of a command and any necessary parameters, if applicable. One example of this is a code which checks your current balance and plan status with the MO.

USSD in Windows is implemented as a WinRT API surface. The implementation classes of this interface serve as the state machine for USSD sessions, but ultimately rely on WWAN Service to do the heavy lifting. These APIs are implemented with a factory pattern.

Implementing USSD

A key thing to remember is that the public facing API is defined by the IDL. Implementation can be confusing because of this, especially if you are unfamiliar with WinRT. Part of the confusion comes from the seemingly ambiguous use of the word 'factory'. A factory can refer to either a class implementation of a static interface or a true factory that provides an activatable interface to a runtime class.

This topic reviews WinRT concepts and then describes the implementation based on these concepts. You may always refer to the IDL for further clarification.

Interfaces

Interfaces define the Application Binary Interface (ABI). They describe the functions that you can call on any class that implements the interface.

Runtime Classes

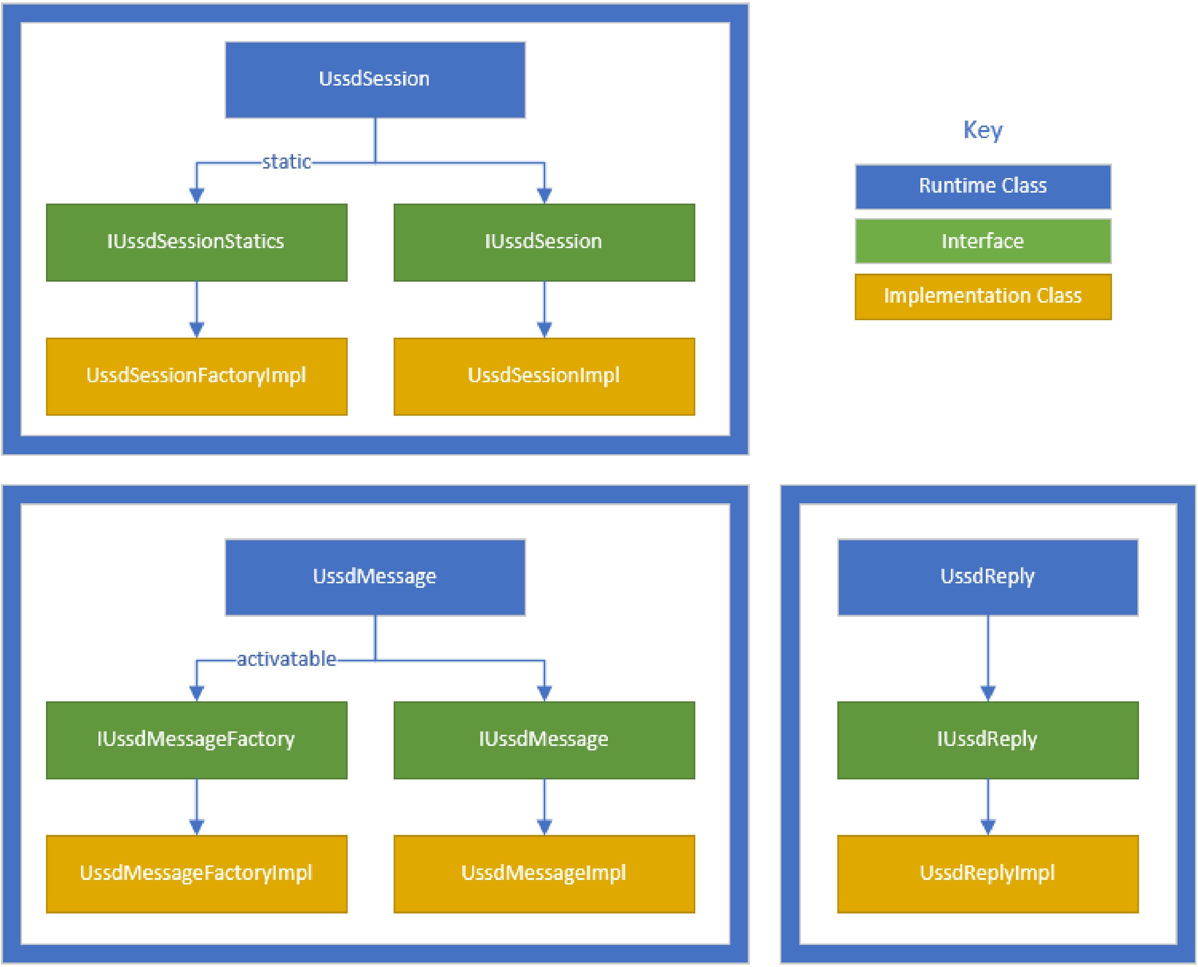
These are the actual classes. They represent, by name, what is ultimately exposed as class names to the ABI. Each runtime class may have zero or more interfaces (but must declare at least one default interface if it has one or more interfaces), zero or more static interfaces, and an activatable tag if necessary. Each of these interfaces are implemented in different files as different "Impl" classes yet they will appear to be a single, unified class to the ABI.

A typical interface appears as instance methods on an existing object.

A static interface appears to the client as static methods on the runtime class itself.

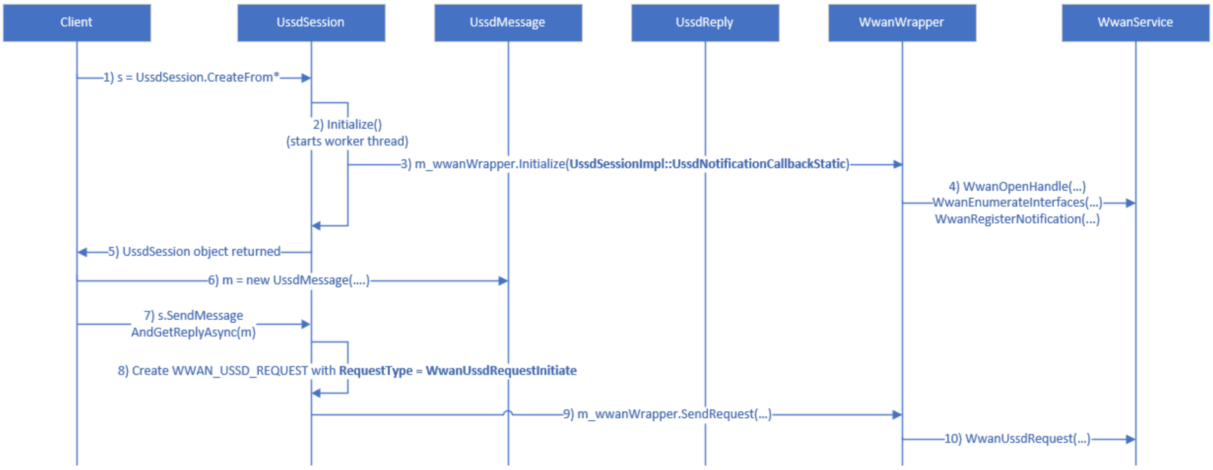
An activatable tag defines the factory interface that will produce an instance of a runtime class. This is completely obfuscated to the client, appearing as a constructor for that runtime class.

USSD Implementation



Flow: Open, Send, Receive, Close.

Open, Send


Client
UssdSession
UssdMessage
UssdReply
WwanWrapper
WwanService
1) s = UssdSession.CreateFrom*
2) Initialize()
(starts worker thread)
3) m_wwanWrapper.Initialize(UssdSessionImpl::UssdNotificationCallbackStatic)
4) WwanOpenHandle(...)
WwanEnumerateInterfaces(...)
WwanRegisterNotification(...)
5) UssdSession object returned
6) m = new UssdMessage(....)
7) s.SendMessage
AndGetReplyAsync(m)
8) Create WWAN_USSD_REQUEST with RequestType = WwanUssdRequestInitiate
9) m_wwanWrapper.SendRequest(...)
10) WwanUssdRequest(...)

1. The client uses one of the static functions UssdSession.CreateFromNetworkAccountId or UssdSession.CreateFromNetworkInterfaceId to create the UssdSession object.

2. Regardless of the API called, a network interface ID is required to initialize a UssdSession. In the case of *NetworkAccountID, steps are taken to retrieve the network interface ID from the Account ID. CreateInternal() is called to create a instance of UssdSession and invoke Initialize() on the newly-created instance. During the initialization steps, a worker thread is spun up and an event handle to trigger events for the thread is created. Steps 3 and 4 also take place during the instance's Initialize().

3. Initialize() is called on the WwanWrapper member object. This function accepts a static callback function as well as a context to allow the static function to map the callback to an object context.

4. WwanWrapper opens a handle to WwanService, enumerates interfaces, and subscribes to USSD notifications by providing a static callback function pointer and "this" as context.

5. The UssdSession object is returned to the client.

6. The client constructs a new UssdMessage by invoking the constructor with a message string. WinRT obfuscates the UssdMessageFactory in this process.

7. The client invokes SendMessageAndGetReplyAsync on the session object, passing the UssdMessage instance.

8. At this time SendMessageAndGetReplyAsync creates a special operation object called UssdSendMessageAndGetReplyOperation. From the its name, it appears that the object encapsulates the logic of a single message being sent down the stack (and waiting for reply), but this is not the case. WinRT requires a special out parameter for asynchronous operations, which we can see as the 2nd parameter on the definition for this function.

```
HRESULT SendMessageAndGetReplyAsync(
            [in] UssdMessage* message,
            [out, retval]
Windows.Foundation.IAsyncOperation<UssdReply>** asyncInfo);
```

   It is the IUssdSendMessageAndGetReplyOperation, a named interface through typedef, that satisfies this parameter by promising that this operation will inevitably return a UssdReply. This interface is not defined in the IDL, but is implemented by the UssdSendMessageAndGetReplyOperationImpl class. Note that the header for this class has a special extension:

```
class UssdSendMessageAndGetReplyOperationImpl :
    public Microsoft::WRL::RuntimeClass<

Windows::Networking::NetworkOperators::IUssdSendMessageAndGetReplyOpera
tion,

Windows::Internal::AsyncBaseFTM<IUssdSendMessageAndGetReplyCompletedHan
dler, Microsoft::WRL::SingleResult>>
```

The UssdSendMessageAndGetReplyOperation object allows WinRT to obfuscate the complexities of this asynchronous operation and all of the compartmentalization and memory proxying that goes along with it. For more information, see SendMessageAndGetReplyAsync.

For now, understand that the asynchronous operation described above simply calls back into the UssdSession object where the logic for this operation is actually contained. We can visualize for simplicity that the UssdSession itself encapsulates the work here. We can now assert that, despite the asynchronous nature, only one UssdMessage may be sent at a time.

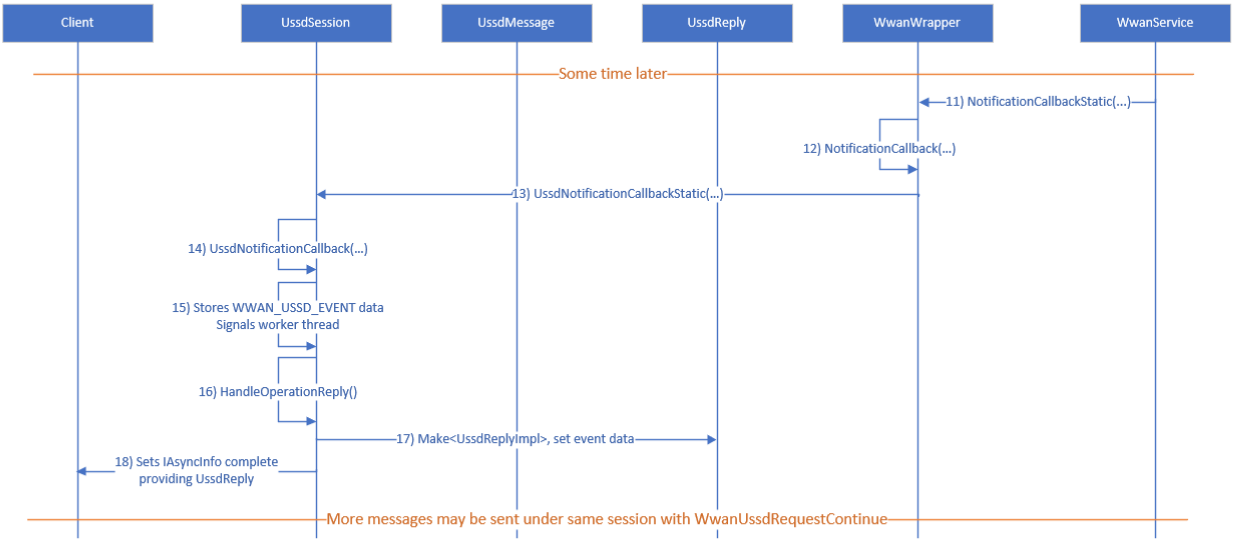
What the SendMessageAndGetReplyAsync function actually does:

- The UssdSession object changes to a busy state, stores the content of the UssdMessage, and fires off the asynchronous action.
- OnOperationStart() is the entry point for the asynchronous logic. Assume for this scenario that there is no active session. This function creates a WWAN_USSD_REQUEST object with RequestType=WwanUssdRequestInitiate.
- Steps 9 and 10 occur as actions taken by this function.

9. m_wwanWrapper.SendRequest is invoked to handle the work of passing the message to WwanService.

10. WwanWrapper uses the WwanService handle to invoke WwanService APIs to carry out the action.

Receive

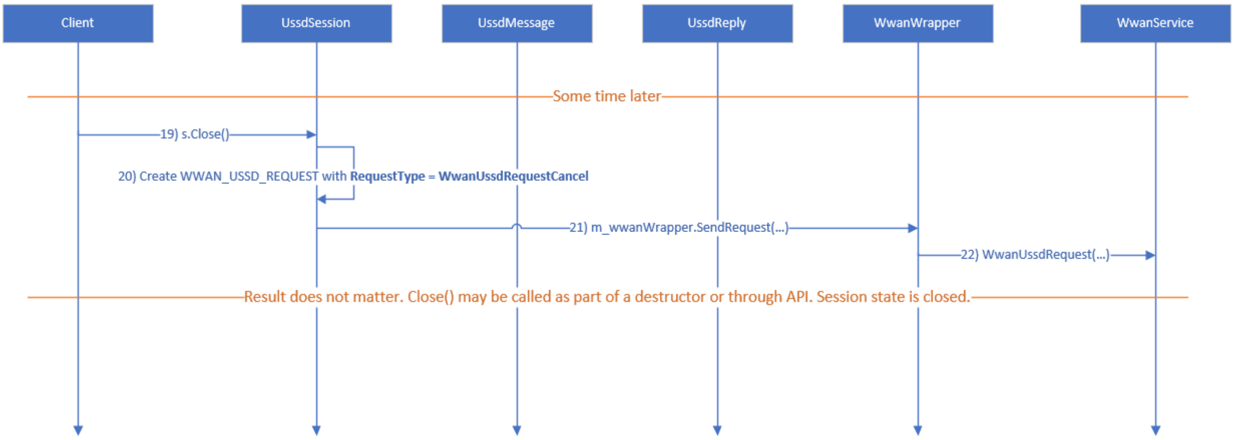

Client
UssdSession
UssdMessage
UssdReply
WwanWrapper
WwanService
Some time later
11) NotificationCallbackStatic(...)
12) NotificationCallback(...)
13) UssdNotificationCallbackStatic(...)
14) UssdNotificationCallback(...)
15) Stores WWAN_USSD_EVENT data
Signals worker thread
16) HandleOperationReply()
17) Make<UssdReplyImpl>, set event data
18) Sets IAsyncInfo complete
providing UssdReply
More messages may be sent under same session with WwanUssdRequestContinue

After step 10, we are left in a state where a request was sent to WwanService to initialize a new USSD session and send a USSD message under that session. After some time, the reply will be available.

11. WwanService will invoke the static callback function provided in step 4 with the context that was also attached.
12. The context will be used to retrieve the WwanWrapper instance and invoke NotificationCallback().
13. WwanWrapper will follow the same pattern as step 11, invoking a static callback to UssdSession, providing the context stored in step 3.
14. Similar to step 12, the context is used to invoke the callback on an instance of UssdSession.
15. The UssdSession stores the WWAN_USSD_EVENT (under a lock) and notifies the worker thread to handle the event.
16. HandleOperationReply() takes the existing UssdSendMessageAndGetReplyOperationImpl object and passes the event data to its internal handler.
17. The operation will construct and UssdReply and invoke FireCompletion() to mark the async action as finished.
18. WinRT obfuscates the completion of the asynchronous action to the client. (Either they have awaited the action or have callback logic.)

More messages can be sent under the same session. If the session was maintained, the future RequestType will be WwanUssdRequestContinue.

Close


Client
UssdSession
UssdMessage
UssdReply
WwanWrapper
WwanService
Some time later
19) s.Close()
20) Create WWAN_USSD_REQUEST with RequestType = WwanUssdRequestCancel
21) m_wwanWrapper.SendRequest(...)
22) WwanUssdRequest(...)
Result does not matter. Close() may be called as part of a destructor or through API. Session state is closed.

After step 18, the client has received the reply to their UssdMessage. They may or may not have continued to use the active UssdSession to send additional messages. We will assume that at some point in the future, the client will manually invoke Close() on the UssdSession. If the client does not explicitly invoke Close(), it will be called during the destructor of UssdSession.

19. Client invokes Close() on the UssdSession instance.
20. A WWAN_USSD_REQUEST is created with RequestType=WwanUssdRequestCancel.
21. The request is sent to m_wwanWrapper as in step 9.
22. The request is sent to WwanService as in step 10.

The result of this request is unimportant. For all intents and purposes, the session is closed. Even in the extreme edge case where the message is somehow never delivered, a new USSD session will always override an existing session.

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to the device Cellular modem driver and run the test: Win6_4.MB.GSM.Data.TestUssd.

MB USSD Troubleshooting Guide

- Collect and decod the logs using the instructions in MB Collecting Logs.
- Keywords for filtering
    1. OID_WWAN_USSD
    2. NDIS_STATUS_WWAN_USSD
    3. WWAN_USSD_REQUEST
    4. WWAN_USSD_EVENT

See Also

MB USSD Operations

MB USSD Operations

Article • 03/14/2023

This topic describes the operations to send and receive messages using the Unstructured Supplementary Service Data (USSD) capabilities of an MB device.

USSD support is optional and when supported is only available on GSM networks. Miniport drivers that support USSD must set the WWAN_CTRL_CAPS_USSD capability flag as part of the **WwanControlCaps** member of the **WWAN_DEVICE_CAPS** structure when processing OID_WWAN_DEVICE_CAPS requests. If miniport drivers do not support USSD, they must not set this flag and should return WWAN_STATUS_NO_DEVICE_SUPPORT for all USSD-related OIDs.

The MB driver model supports the following USSD operations: Device initiated operations:

- Sending a USSD message on a newly created USSD session
- Sending a USSD message on a newly created USSD session
- Sending a USSD message on an existing USSD session
- Terminating the USSD session

For more information on device initiated operations, see OID_WWAN_USSD.

Network initiated operations:

- Receiving a USSD message on a newly created USSD session
- Receiving a USSD message on an existing USSD session
- Termination the USSD session

For more information on network initiated operations, see **NDIS_STATUS_WWAN_USSD**.

The USSD protocol only allows a single USSD session at any time. For device initiated operations, the **RequestType** member of the **WWAN_USSD_REQUEST** structure indicates the purpose of the request OID:

- **WwanUssdRequestInitiate** is used to create a new USSD session and send the provided USSD string to the network. If a USSD session already exists, the driver must fail the request with an event of type **WwanUssdEventOtherLocalClient**. A USSD string must be present. For example, the length must be between 1 and 160 bytes.

- **WwanUssdRequestContinue** is used to send a USSD string on an existing session. A USSD string must be present. For example, the length must be between 1 and 160 bytes.

- **WwanUssdRequestCancel** is used to terminate the existing session. The driver must respond with an event of type **WwanUssdEventTerminated**, even if no session existed (which may happen during a concurrent release of the session from the network and the local client). The content of the USSD string must be ignored for this request; the string length is set to zero to indicate that there is no USSD string.

For network initiated operations, the **EventType** member of the WWAN_USSD_EVENT structure indicates the high level purpose of the indication:

- The event **WwanUssdEventNoActionRequired** is used for network initiated USSD notifications, or when no further information is needed after a mobile initiated operation. The event **WwanUssdEventActionRequired** is used for network initiated USSD requests, or when further information is needed after a mobile initiated operation. Both events require a non-empty USSD string to be present. The **SessionState** member is used to indicate if the USSD string is the first message of a USSD session; it must be set to **WwanUssdSessionStateNew** for the first message of a network initiated USSD session and to **WwanUssdSessionStateExisting** in all other cases.

- The event **WwanUssdEventActionRequired** also indicates that the session is still open. All other events indicate that the session has been closed.

- The events **WwanUssdEventNoActionRequired** and **WwanUssdEventActionRequired** are the only events that contain a USSD string. All other events must set the USSD string length to 0 to indicate that the string is absent. The value of the **SessionState** member is ignored if no string is present.

- The event **WwanUssdEventTerminated** is used to indicate that the USSD session has been terminated.

- The event **WwanUssdEventOtherLocalClient** is used to indicate that a new USSD session cannot be established because there is already a session opened. This includes sessions that are invisible to the MB stack such as a USSD session termination in the SIM.

- The event **WwanUssdEventOperationNotSupported** is used to indicate that the previous request is not supported by the driver or device.

- The event **WwanUssdEventNetworkTimeOut** is used to indicate that the session was closed due to a session timeout either by the network or locally. The driver or device is responsible for timing out an inactive USSD session after an implementation specific timeout.

MB Device Services

Article • 03/14/2023

Windows 7 introduced a NDIS (Network Device Interface Specification) based driver model for supporting Mobile Broadband (MB) devices. Windows 8 expanded the model to implement a standardized hardware interface for USB-based Mobile Broadband devices. This hardware interface specification is referred as the Mobile Broadband Interface Model (MBIM).

Windows 8 provides an updated class driver that works with devices conforming to the MBIM specification. This model is referred to as the MB Class Driver. However, no class driver can support all of the functionality exposed by an MB device. In order to allow IHV partners to continue to innovate, the MB Class Driver provide mechanisms, such as the **IMbnDeviceService interface** to allow IHVs to extend the behavior of the class driver functionality.

Note Functionality to extend MB devices services is accomplished via a user-mode application, not a kernel-mode driver extension.

While the class driver introduced in Windows 7 featured limited MB device feature support, the MB Class Driver in Windows 8 added native support for some additional features such as USSD, EAP-SIM/AKA and USB selective suspend, and offers an extensible device representation and control mechanisms. The Mobile broadband WinRT API overview provides some additional information about extending device services.

The MB Class Driver in Windows 8 enables vertical solution providers to use the Mobile Broadband API Interfaces to create enhanced user experiences that are outside of those provided by Windows. The extension mechanism is a way to augment, but not to replace, the functionality supported in the MB Class Driver itself. For example, an IHV can provide vendor-specific software that performs firmware updates on the device. Or, an IHV can provide vendor-specific software that provides value-add services such as SIM toolkit (STK) or Phonebook. The AppContainer mobile broadband pin, connection and management sample demonstrates Win32/COM Mobile Broadband APIs within the AppContainer to access and manage mobile broadband features.

In addition to providing a mechanism to extend the MB Class Driver' functionality, Windows also provides mechanisms to enable IHVs to deploy and install their value-add software through Windows Update (WU).

For more information see:

- The "MBIM Service and CID Extensibility" section of the Mobile Broadband Interface Model (MBIM) specification

MB Multimode Multicarrier

Article • 03/14/2023

This topic describes the support for MB multimode multicarrier that has been integrated into mobile broadband for Windows 8. Windows 8 introduces native support that allows users of carrier-unlocked mobile broadband devices that support carrier switching (most mobile broadband users outside the US) to select and connect to any supported carrier from within the Windows UI.

These features are targeted at IHVs to enable them to implement support for this scenario. They supplement the USB NCM (MBIM) spec (current version 1.0) that use the CIDs and device behavior required for supporting the MB multimode multicarrier scenario.

For more information see the following topics:

- MB Provider Operations
- OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS
- **NDIS_STATUS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS**
- **NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS**

MB Multi-SIM operations

Article • 03/14/2023

Desktop Multi-Modem Multi-Executor Support

Traditionally, non-phone Windows devices have not been configured for multi-SIM modems because they have fewer physical space restraints than phones. This allows them to truly harness multiple active radios at the same time instead of having one modem with multiple SIM cards like a phone does; however, due to the rise of eSIM and scenarios in the enterprise , the demand for multi-SIM-per-modem support on non-phone devices has increased.

Most typical multi-SIM phone devices have dual SIM slots but are limited to one primary SIM card supporting data while the other only supports voice features. Such a limitation does not exist in the non-phone PC model as all SIM cards are used for data connection.

While the framework defined in this specification can theoretically support an unbounded number of modems and SIM cards, Windows 10, version 1703 and later supports only the dual-SIM/single-active (DSSA) scenario end to end.

NDIS Modem Interface Specification

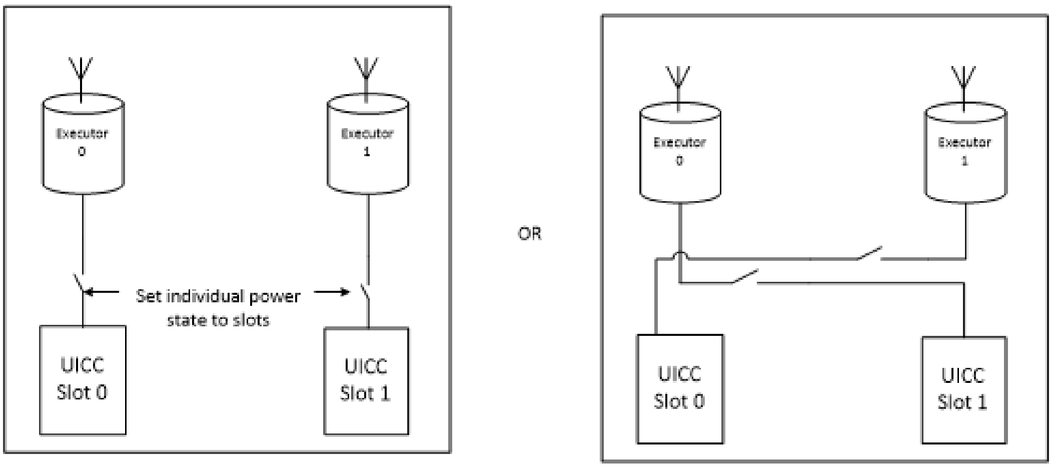
Existing Interface and Feature Gaps

It is possible to support dual-SIM/dual-active functionality with multiple independent modems, where each modem is a separate device and operates completely independently. However, this is outside this documentation’s scope, which instead focuses on a WWAN miniport modem that is capable of presenting multiple and simultaneous cellular stacks to the host. This section defines the various objects and establishes the terminology used in all MB documentation related to multi-SIM functionality.

Advancements in hardware have resulted in devices that can maintain simultaneous registrations with multiple cellular networks. In such devices, there are assumed to be “multiple instances of the cellular stack” running in parallel which are each able to maintain registration, monitor signal strengths, perform handovers and listen for incoming pages. Each instance of this “cellular stack” will be referred to as an *executor* for the rest of this document. For example, in a device capable of maintaining registrations with two networks simultaneously the modem hardware is considered to have two executors.

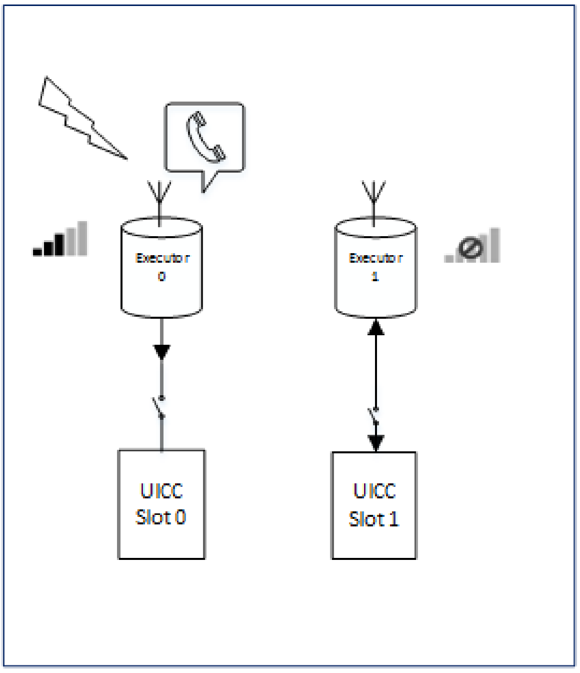
The executor is a logical representation of the hardware and may in fact be one single hardware transceiver being multiplexed. Exact hardware specifics are regarded as vendor implementation details and are out of scope for this specification. For an NDIS miniport driver, executors are exposed as multiple instances of a WWAN miniport adapter. For an MBIM modem, executors are represented by multiple MBIM functions on an enumerated composite device.

The following two images illustrate the logical view of a dual SIM modem. Each shows a possible combination of executor and UICC.



The cellular stack inside an executor is considered mostly self-contained except in the case of a Dual Standby modem implementation where the executor conducting traffic (voice and/or data) may prevent the other from maintaining registration.

The following diagram illustrates the logical view of a dual standby modem. Traffic on Executor 0, a phone call, causes Executor 1 to lose registration.



The Windows Desktop modem interface model in NDIS 6.7 does not accommodate such an architecture because it is based upon several implicit assumptions:

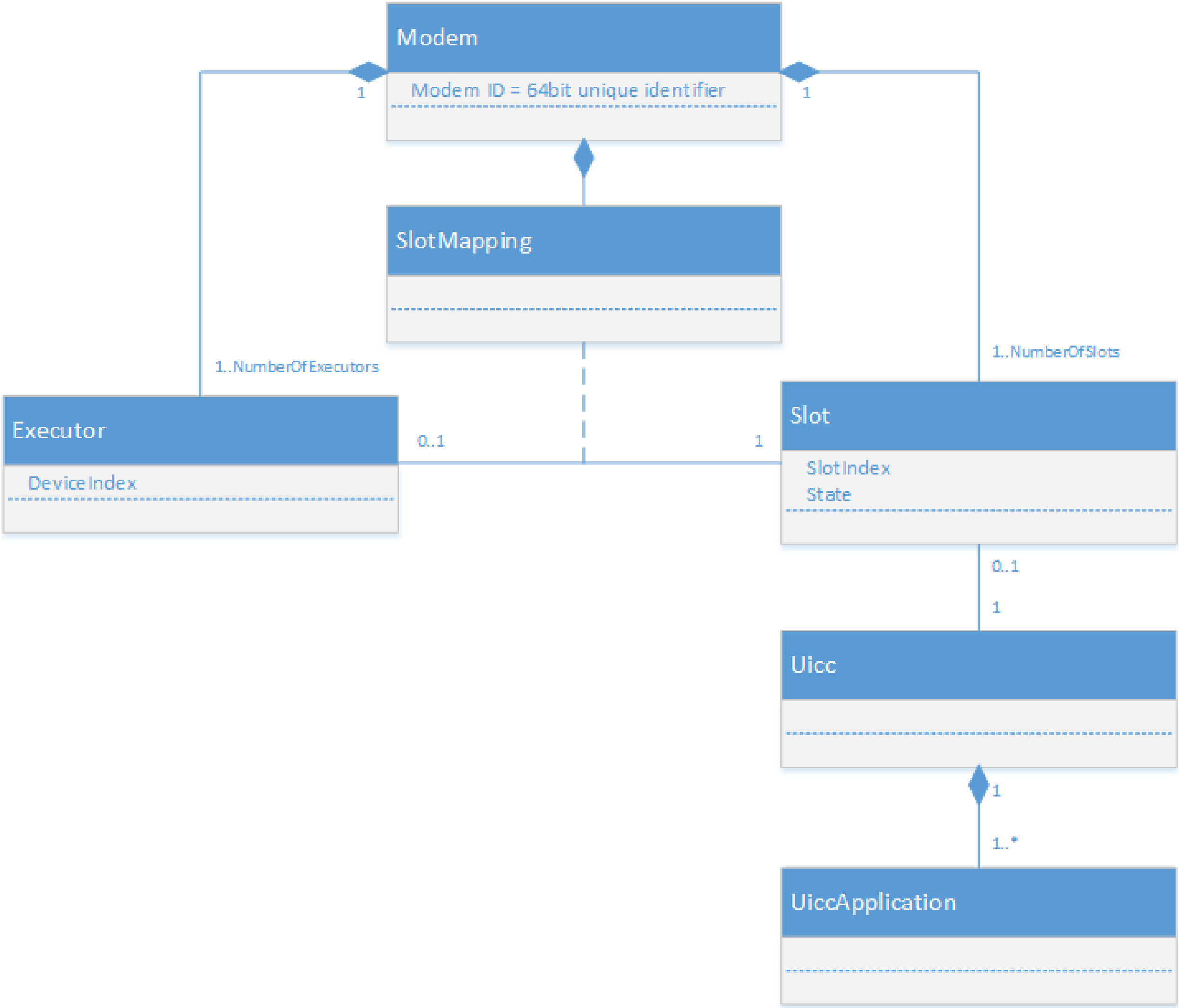
- The model assumes that there is a single executor within the modem.
- The model assumes that there is a single UICC card directly associated with the modem hardware.
- The UICC is treated as if it were a single-application SIM card.

By contrast, the Microsoft Radio Interface Layer (RIL) interface on Windows Mobile explicitly exposes the multiplicity of these assumptions. The Mobile Broadband interface in Windows Mobile exposes the ability to register independently through separate miniports and assumes that some basic configuration of the device has already been accomplished through the RIL interface. To provide equivalent functionality, Windows Desktop must provide mechanisms to discover the number of executors and slots, to access executors independently, to define the mapping between executors and slots, and to define the applications within the mapped UICC card that each executor will use.

For more information about cellular architecture and the differences between Windows 10 Mobile and Desktop, please see Cellular architecture and implementation.

Major Objects and Operations

The following figure shows an abstract model of a modem.



Each modem is identified by a globally unique identifier (GUID) and contains a set of one or more executors, each of which is capable of independent registration on a cellular network. Each executor has an associated executor index, an integer, beginning with 0 for the first executor. In addition, the modem exposes one or more slots that may contain UICC cards. It is assumed is that the number of slots is greater than or equal to the number of executors. Each slot has an associated index, also beginning with 0, and a current state related to the power state of the slot and availability state of a card in the slot (if any).

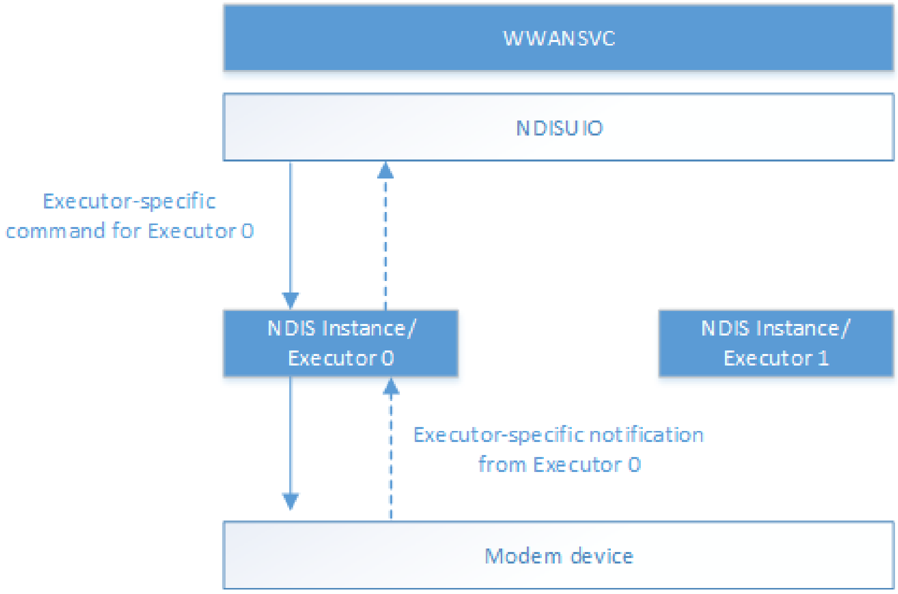
To maintain compatibility with existing modems, each executor operates with information provided by a UICC card in a single slot. The association between executors and slots is defined by slot mapping, which maps each executor to exactly one slot.

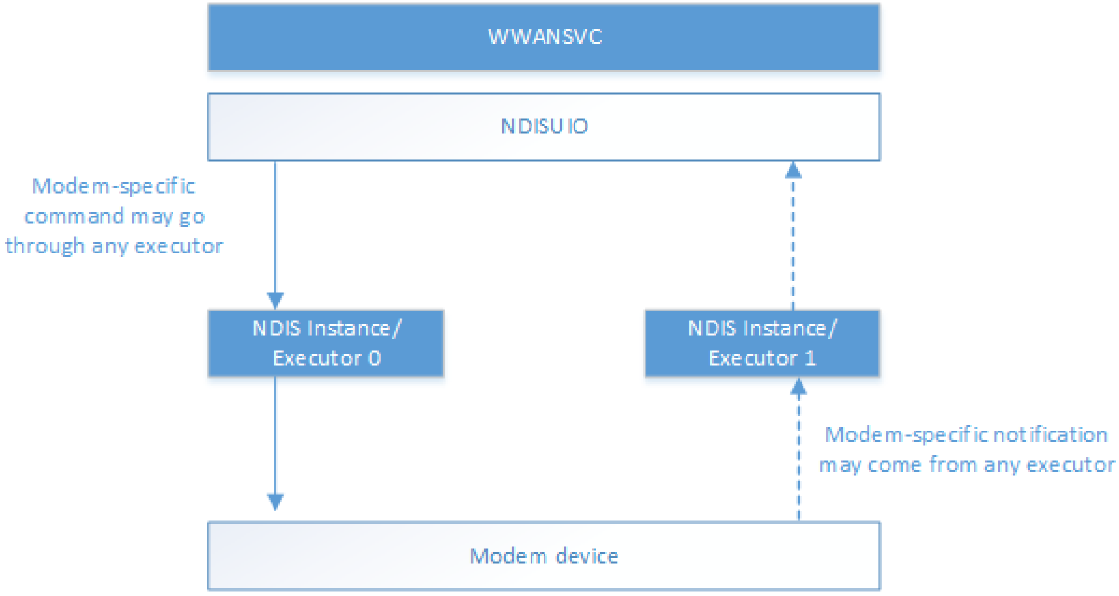
A slot may contain a UICC card; each card contains one or more UICC applications such as a USIM, CSIM, ISIM, or possibly other telephony and non-telephony applications such as PKCS#15 or Global Platform applications for an NFC secure element. The addressing and use of these individual UICC applications is a topic for future specification and out of scope of this documentation.

The Windows Desktop NDIS interface to the modem is characterized by the exchange of OIDs and NDIS notifications. In most cases these OIDs are directed to individual executors; however, a few commands and notifications are scoped to the modem.

For non-Windows Mobile operating systems, a multi-executor modem appears as one device with multiple physical WWAN miniport instances. Each physical miniport instance represents an executor that can maintain registration as an NDIS instance. Additional virtual instances may be created at runtime to manage context-specific packet data and device service sessions. Executor-specific commands and notifications are exchanged through the WWAN miniport NDIS physical instance representing that executor. Modem-specific commands (in other words, those that are not executor-specific) and their corresponding notifications may be sent to or come from any physical miniport instance.

The following two diagrams show the difference in executor-specific commands and notifications (the first diagram), where commands and notifications go through and come from the same executor, and modem-specific commands and notifications (the second diagram), where commands may go through any executor and come from any executor.


WWANSVC
NDISUIO
Executor-specific command for Executor 0
NDIS Instance/ Executor 0
NDIS Instance/ Executor 1
Executor-specific notification from Executor 0
Modem device


WWANSVC
NDISUIO
Modem-specific command may go through any executor
NDIS Instance/ Executor 0
NDIS Instance/ Executor 1
Modem-specific notification may come from any executor
Modem device

All OID set or query requests issued to a miniport instance are executed against the modem and executor with which the miniport instance is associated. Likewise, all unsolicited notifications and unsolicited Device Service events sent from a miniport instance are applicable to the modem and the executor with which the miniport instance is associated. For example, an unsolicited NDIS_STATUS_WWAN_REGISTER_STATE or NDIS_STATUS_WWAN_PACKET_SERVICE notification from a miniport indicates the registration (or packet service state) of the associated modem and the executor only and is unrelated to the state of other modem(s) or other executor(s).

When there are multiple modems and/or multiple executors in a device, the physical miniport adapter associated with that modem and executor combination issues non-context-specific unsolicited notifications related to a particular modem and executor combination.

In the same way, if a device has multiple modems and/or multiple executors, the physical miniport adapter instance associated with a particular modem and executor combination can receive non-context-specific OID query requests related to that modem and executor. The adapter receiving such a query request processes it according to the OID definition. If so chosen by miniport driver, this query request can be processed concurrently with any other in-process OID set or query requests in any instance of adapters associated with that modem and executor. All instances of a miniport adapter associated with a same modem and executor report the same state information for that cellular modem and executor (such as radio power state, registration state, packet service state, etc.).

For a device which has multiple modems and/or multiple executors, the physical miniport adapter instance associated with a modem and executor combination can receive non-context-specific OID set requests. The miniport driver shall keep track of the progress of such a request. If one such set request is in progress in any adapter and has not completed yet, a second such set request attempt (to any adapter instance associated with the same modem and executor) shall be queued and processed after the previous requests have completed.

The Windows 10 desktop WMBCLASS driver follows the specification outlined in the previous paragraph to handle this set request race condition, but if the race condition occurs at the modem layer the modem should follow the same guidance to queue up conflicting device-wide commands on the MBIM function if it is still processing another function that is linked to the same underlying device.

OIDs for Set and Query Requests

To query the number of devices (executors) and slots in the modem, as well as the number of executors that may be active concurrently, the host uses OID_WWAN_SYS_CAPS.

To query the capability of an executor, the host uses OID_WWAN_DEVICE_CAPS_EX.

To define the slot that is bound to each executor or query the current mapping, the host uses OID_WWAN_DEVICE_SLOT_MAPPINGS.

To query the status of a particular slot on the modem, the host uses OID_WWAN_SLOT_INFO_STATUS.

Per-device and Per-executor Commands

With the addition of the executor concept to non-Windows Mobile devices in Windows 10, version 1703 and later, OIDs are now split into two categories: per-device OIDs and per-executor OIDs. The table below explains which OIDs fall into which category.

Per-device or Per-executor	OID name
Per-device	OID_WWAN_DRIVER_CAPS
	OID_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS
	OID_WWAN_ENUMERATE_DEVICE_SERVICES
	OID_WWAN_PRESHUTDOWN
	OID_WWAN_VENDOR_SPECIFIC
	OID_WWAN_SYS_CAPS
	OID_WWAN_DEVICE_SLOT_MAPPINGS
Per-executor	OID_WWAN_AUTH_CHALLENGE
	OID_WWAN_CONNECT
	OID_WWAN_DEVICE_CAPS
	OID_WWAN_DEVICE_CAPS_EX
	OID_WWAN_DEVICE_SERVICE_COMMAND
	OID_WWAN_DEVICE_SERVICE_SESSION

Per-device or Per-executor	OID name
	OID_WWAN_DEVICE_SERVICE_SESSION_WRITE
	OID_WWAN_DEVICE_SERVICES
	OID_WWAN_HOME_PROVIDER
	OID_WWAN_NETWORK_IDLE_HINT
	OID_WWAN_PACKET_SERVICE
	OID_WWAN_PIN
	OID_WWAN_PIN_EX
	OID_WWAN_PIN_LIST
	OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS
	OID_WWAN_PREFERRED_PROVIDERS
	OID_WWAN_PROVISIONED_CONTEXTS
	OID_WWAN_RADIO_STATE
	OID_WWAN_READY_INFO
	OID_WWAN_REGISTER_STATE
	OID_WWAN_SERVICE_ACTIVATION
	OID_WWAN_SIGNAL_STATE
	OID_WWAN_SMS_CONFIGURATION
	OID_WWAN_SMS_DELETE
	OID_WWAN_SMS_READ
	OID_WWAN_SMS_SEND
	OID_WWAN_SMS_STATUS
	OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS
	OID_WWAN_USSD
	OID_WWAN_VISIBLE_PROVIDERS
	OID_WWAN_SLOT_INFO_STATUS

ⓘ Note

OID_WWAN_RADIO_STATE has been updated for Windows 10, version 1703 as well. See OID_WWAN_RADIO_STATE for more information.

MBIM Interface Update for Multi-SIM Operations

For non-Windows Mobile operating systems, a multi-executor modem appears as one USB composite device with multiple MBIM functions. Each MBIM function represents an executor that can maintain registration. Executor-specific commands and notifications are exchanged through the MBIM function representing that executor, while modem-specific commands (in other words, those that are not executor-specific) and their corresponding notifications may be sent to or come from any MBIM function that belongs to the same underlying USB composite device.

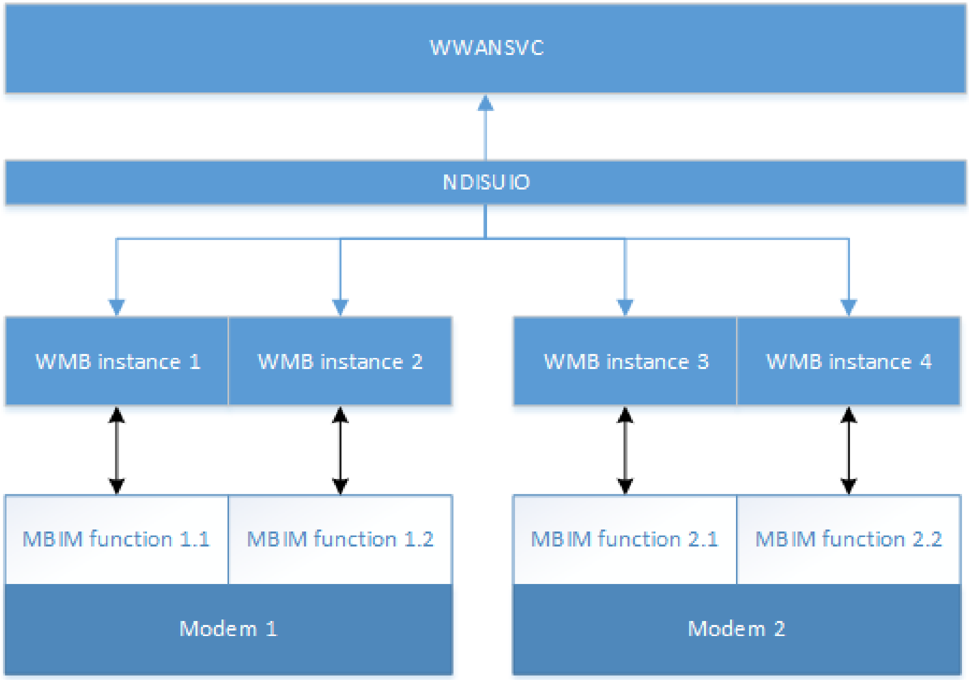
All CID set or query requests issued to a MBIM function are executed against the modem and executor with which the miniport instance is associated; likewise, all unsolicited notifications sent from a MBIM function are applicable to the modem and the executor with which the MBIM function is associated. In the same way, all unsolicited Device Service events sent from a miniport instance are applicable to the modem and the executor with which the MBIM function is associated. For example, an unsolicited MBIM_CID_REGISTER_STATE or MBIM_CID_PACKET_SERVICE notification from a MBIM function indicates the registration or packet service state of the associated modem/executor only and is unrelated to the state of other modem(s) or other executor(s).

When there are multiple modems and/or multiple executors in a device, non-context-specific unsolicited notifications related to a particular modem and executor combination shall be issued from the MBIM function associated with the aforementioned modem and executor.

In a device with multiple modems and/or multiple executors, non-context-specific CID query requests related to a particular modem and executor may be issued to the MBIM function associated with that modem and executor combination. The function receiving such a query request shall process it according to the CID definition. If so chosen by the modem firmware, such a query request may be processed concurrently with any other CID set or query requests being processed by any MBIM functions associated with that modem and executor. All MBIM functions associated with the same modem shall report the same state information for that cellular modem in addition to the executor that they represent.

When there are multiple modems and/or multiple executors in a device, non-executor-specific CID set requests may be issued to the MBIM function associated with that modem and executor. The modem shall keep track of the progress of such requests as a whole. If one such set request is in progress in any adapter and has not completed yet, a second such set request attempt (to any adapter instance associated with the same modem and executor) shall be queued and processed after the previous requests have been completed.

The following diagram illustrates the information flow between the WWANSVC and MBIM functions in two different modems.



This section contains the detailed modem-wide and per-executor CID descriptions for the defined device services. The definitions reference back to existing public MBIM1.0 specification. An MBIM-compliant device implements and reports the following device service when queried by CID_MBIM_DEVICE_SERVICES. The existing well-known services are defined in section 10.1 of the USB NCM MBIM 1.0 specification. Microsoft extends this to define the following service.

Service Name = **Basic Connect Extensions**

UUID = **UUID_BASIC_CONNECT_EXTENSIONS**

UUID Value = **3d01dcc5-fef5-4d05-0d3abef7058e9aaf**

The following CIDs are defined for **UUID_MS_BasicConnect**:

CID	Command Code	Minimum OS Version
MBIM_CID_MS_SYS_CAPS	5	Windows 10, version 1703
MBIM_CID_MS_DEVICE_CAPS_V2	6	Windows 10, version 1703
MBIM_CID_MS_DEVICE_SLOT_MAPPINGS	7	Windows 10, version 1703
MBIM_CID_MS_SLOT_INFO_STATUS	8	Windows 10, version 1703

All offsets in the following CID sections are calculated from the beginning of the InformationBuffer MBIM_COMMAND_MSG.

MBIM_CID_MS_SYS_CAPS

Description

This CID retrieves information about the modem. This can be sent on any of the MB instances exposed as a USB function.

Query

The InformationBuffer on MBIM_COMMAND_MSG contains the response data as MBIM_MS_SYS_CAPS_INFO.

Set

Not applicable.

Unsolicited Event

Not applicable.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	Not applicable	Not applicable
Response	Not applicable	MBIM_MS_SYS_CAPS_INFO	Not applicable

Data Structures

Query

The InformationBuffer shall be null and InformationBufferLength shall be zero.

Set

Not applicable.

Response

The following MBIM_SYS_CAPS_INFO structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	NumberOfExecutors	UINT32	Number of MBB instances reported by this modem
4	4	NumberOfSlots	UINT32	Number of physical UICC slots available on this modem

Offset	Size	Field	Type	Description
8	4	Concurrency	UINT32	Number of MBB instances that may be active concurrently
12	8	ModemId	UINT64	Unique 64-bit identifier for each modem

The *NumberOfExecutors* field denotes the number of *executors* that are supported by the modem in its current configuration. This directly maps to the number of 'sub-phone' stacks the modem supports.

The *NumberofSlots* field denotes the number of slots that are physically present on the modem. Each slot reported must be capable of receiving a UICC card (the slots themselves can be a heterogeneous mix if needed – mini SIM, micro SIM, nano SIM or any standard as defined by ETSI). The number of slots must be equal to or greater than the number of executors supported. The 'greater than' provision allows use of non-telephony UICC such as for security, NFC, etc.

The *Concurrency* field denotes the number of executors (MBB instances) that may be active at the same time. It range must be *1 ≤ Concurrency ≤ NumberOfExecutors*. For example, a dual-standby modem would have a Concurrency of 1 while a dual-active modem would have a concurrency of 2

The *ModemId* field denotes the unique 64-bit identifier for a given modem hardware. An IHV may implement its own logic to generate a unique 64-bit value for each modem; for instance, hashing one of the IMEI numbers, randomly generating 64-bit numbers, etc. Once the 64-bit ID is generated, it should persist across reboots and SIM card removals/insertions.

Status Codes

This CID uses Generic Status Codes (see Use of Status Codes in Section 9.4.5 of the public USB MBIM standard).

MBIM_CID_MS_DEVICE_CAPS_V2

Description

This CID retrieves the capability information related to an executor. Since this CID is an extension of MBIM_CID_DEVICE_CAPS, only the changes from MBIM_CID_DEVICE_CAPS as stated in Section 10.5.1 of the public USB MBIM standard are presented here.

This CID continues to be query-only and will return a MBIM_MS_DEVICE_CAPS_INFO_V2 structure in response to MBIM_COMMAND_MSG with the MBIM service MSUUID_BASIC_CONNECT and CID MBIM_CID_MS_DEVICE_CAPS_V2.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	Not applicable	Not applicable
Response	Not applicable	MBIM_MS_DEVICE_CAPS_INFO_V2	Not applicable

Data Structures

Query

The same as Section 10.5.1.4 of the public USB MBIM standard.

Set

Not applicable.

Response

The following MBIM_DEVICE_CAPS_INFO_V2 structure shall be used in the InformationBuffer. Compared with the MBIM_CID_DEVICE_CAPS structure defined in section 10.5.1 of the public USB MBIM standard, the following structure has a new field called *DeviceIndex*. Unless stated here, the field descriptions in Table 10-14 of the public USB MBIM standard apply here.

Offset	Size	Field	Type	Description
0	4	DeviceType	MBIM_DEVICE_TYPE	
4	4	CellularClass	MBIM_CELLULAR_CLASS	
8	4	VoiceClass	MBIM_VOICE_CLASS	
12	4	SimClass	MBIM_SIM_CLASS	For MBIM modems which support this CID, SimClass will always be reported as MBIMSimClassSimRemovable.
16	4	DataClass	MBIM_DATA_CLASS	
20	4	SmsCaps	MBIM_SMS_CAPS	
24	4	ControlCaps	MBIM_CTRL_CAPS	
28	4	MaxSessions	UINT32	
32	4	CustomDataClassOffset	OFFSET	
36	4	CustomDataClassSize	SIZE(0..22)	

Offset	Size	Field	Type	Description
40	4	DeviceIdOffset	OFFSET	
44	4	DeviceIdSize	SIZE(0..26)	
48	4	FirmwareInfoOffset	OFFSET	
52	4	FirmwareInfoSize	SIZE(0..60)	
56	4	HardwareInfoOffset	OFFSET	
60	4	HardwareInfoSize	SIZE(0..60)	
64	4	ExecutorIndex	UINT32	The executor index. It ranges from *0* to *n-1* where *n* is the number of MBB instances contained in the MBIM modem. Its value is always constant and independent of the enumeration order.
68		DataBuffer	DATABUFFER	The data buffer containing the *CustomDataClass*, *DeviceId*, *FirmwareInfo*, and *HardwareInfo* members.

Status Codes

This CID uses Generic Status Codes (see Use of Status Codes in Section 9.4.5 of the public USB MBIM standard).

MBIM_CID_MS_DEVICE_SLOT_MAPPINGS

Description

This CID sets or returns the device-slot mappings (in other words the executor-slot mappings).

Query

The InformationBuffer on MBIM_COMMAND_MSG is not used. MBIM_MS_DEVICE_SLOT_MAPPING_INFO is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Set

The InformationBuffer of MBIM_COMMAND_MSG contains MBIM_MS_DEVICE_SLOT_MAPPING_INFO. MBIM_MS_DEVICE_SLOT_MAPPING_INFO is returned in the InformationBuffer of MBIM_COMMAND_DONE. Regardless of whether the Set CID succeeds or fails, the MBIM_MS_DEVICE_SLOT_MAPPING_INFO contained in the response represents the current device-slot mappings.

Unsolicited Events

Not applicable.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_DEVICE_SLOT_MAPPING_INFO	Not applicable	Not applicable
Response	MBIM_MS_DEVICE_SLOT_MAPPING_INFO	MBIM_MS_DEVICE_SLOT_MAPPING_INFO	Not applicable

Data Structures

Query

The InformationBuffer shall be null and InformationBufferLength shall be zero.

Set

The following MBIM_MS_DEVICE_SLOT_MAPPING_INFO structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	MapCount (MC)	UINT32	Number of mappings, which is always equal to the number of devices/executors.
4	8 * MC	SlotMapList	OL_PAIR_LIST	The *i-th* pair of this list, where (0 <= i <= (MC-1)) records the index of the slot which is currently mapped to the *i-th* device/executor. The first element in the pair is a 4-byte field with the Offset into the DataBuffer, calculated from the beginning (offset 0) of this MBIM_MS_DEVICE_SLOT_MAPPINGS_INFO structure, to an UINT32. The second element of the pair is a 4-byte size of the record element. Since the type of the slot index is UINT32, the second element in the pair is always 4.
4 + (8 * MC)	4 * MC	DataBuffer	DATABUFFER	The data buffer that contains *SlotMapList*. Since the size of the slot is 4 bytes and MC is equal to the number of slot indices, the total size of DataBuffer is 4 * MC.

Response

The MBIM_MS_DEVICE_SLOT_MAPPING_INFO used in Set is also used in the InformationBuffer for Response.

Status Codes

Status Code	Description
MBIM_STATUS_BUSY	The operation failed because the device is busy. In the absence of any explicit information from the function to clear this condition, the host can use subsequent actions by the function (e.g., notifications or command completions) as a hint to retry the failed operation.
MBIM_STATUS_FAILURE	The operation failed (a generic failure).
MBIM_STATUS_VOICE_CALL_IN_PROGRESS	The operation failed because a voice call is in progress.
MBIM_STATUS_INVALID_PARAMETERS	The operation failed because of invalid parameters (e.g. slot numbers out of range or duplicated values in the mapping).

MBIM_CID_MS_SLOT_INFO_STATUS

Description

This CID retrieves a high-level aggregated status of a specified UICC slot and the card within it (if any). It may also be used to deliver an unsolicited notification when the status of one of the slots changes.

Query

The InformationBuffer of MBIM_COMMAND_MSG contains an MBIM_MS_SLOT_INFO_REQ structure. The InformationBuffer of the MBIM_COMMAND_DONE message contains an MBIM_MS_SLOT_INFO structure.

Set

Not applicable.

Unsolicited Events

The Event InformationBuffer contains an MBIM_MS_SLOT_INFO structure. The function sends this event in the event that the composite slot/card state changes.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	MBIM_MS_SLOT_INFO_REQ	Not applicable
Response	Not applicable	MBIM_MS_SLOT_INFO	MBIM_MS_SLOT_INFO

Data Structures

Query

The following MBIM_MS_SLOT_INFO_REQ structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	SlotIndex	UINT32	The index of the slot to be queried.

Set

Not applicable.

Response

The following MBIM_MS_SLOT_INFO structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	SlotIndex	UINT32	The index of the slot.
4	4	State	MBIM_MS_UICC_SLOT_STATE	The state of the slot and card (if applicable).

The following MBIM_MS_UICCSLOT_STATE structure describes the possible states of the slot.

States	Value	Description
UICCSlotStateUnknown	0	The modem is still in the process of initializing so the SIM slot state is not deterministic.
UICCSlotStateOffEmpty	1	The UICC slot is powered off and no card is present. An implementation that is unable to determine the presence of a card in a slot that is powered off reports its state as UICCSlotStateOff.
UICCSlotStateOff	2	The UICC slot is powered off.
UICCSlotStateEmpty	3	The UICC slot is empty (there is no card in it).

States	Value	Description
UICCSlotStateNotReady	4	The UICC slot is occupied and powered on but the card within it is not yet ready.
UICCSlotStateActive	5	The UICC slot is occupied and the card within it is ready.
UICCSlotStateError	6	The UICC slot is occupied and powered on but the card is in an error state and cannot be used until it is next reset.
UICCSlotStateActiveEsim	7	The card in the slot is an eSIM with an active profile and is ready to accept commands.
UICCSlotStateActiveEsimNoProfiles	8	The card in the slot is an eSIM with no profiles (or no active profiles) and is ready to accept commands.

MBIM_MS_UICCSLOT_STATE transition guidance for multi-sim devices

Conforming to the correct UICC slot state transitions ensures that the OS handles all changes properly and displays the correct toast notifications to the user.

For the *SIM inserted* toast notification, the OS expects the embedded slot (SIM2/Slot 1) to be selected and the following state transition to occur upon the insertion of a SIM in the physical slot (SIM1/Slot 0).

Possible values of Slot 0 before SIM insertion	Possible values of Slot 0 after SIM insertion
UICCSlotStateEmpty	UICCSlotStateActive
UICCSlotStateOffEmpty	• UICCSlotStateActiveEsim • UICCSlotStateActiveEsimNoProfile

For the *SIM removed* toast notification, the OS expects the physical slot (SIM1/Slot 0) to be selected with a SIM inserted and the following state transition to occur upon the removal of the SIM from the physical slot (SIM1/Slot 0).

Possible values of Slot 0 before SIM removal	Possible values of Slot 0 after SIM removal
UICCSlotStateActive	UICCSlotStateEmpty
• UICCSlotStateActiveEsim • UICCSlotStateActiveEsimNoProfile	UICCSlotStateOffEmpty

Status Codes

This CID uses Generic Status Codes (see Use of Status Codes in Section 9.4.5 of the public USB MBIM standard).

Non-NDIS Mapping of Per-executor and Per-modem MBIM CIDs

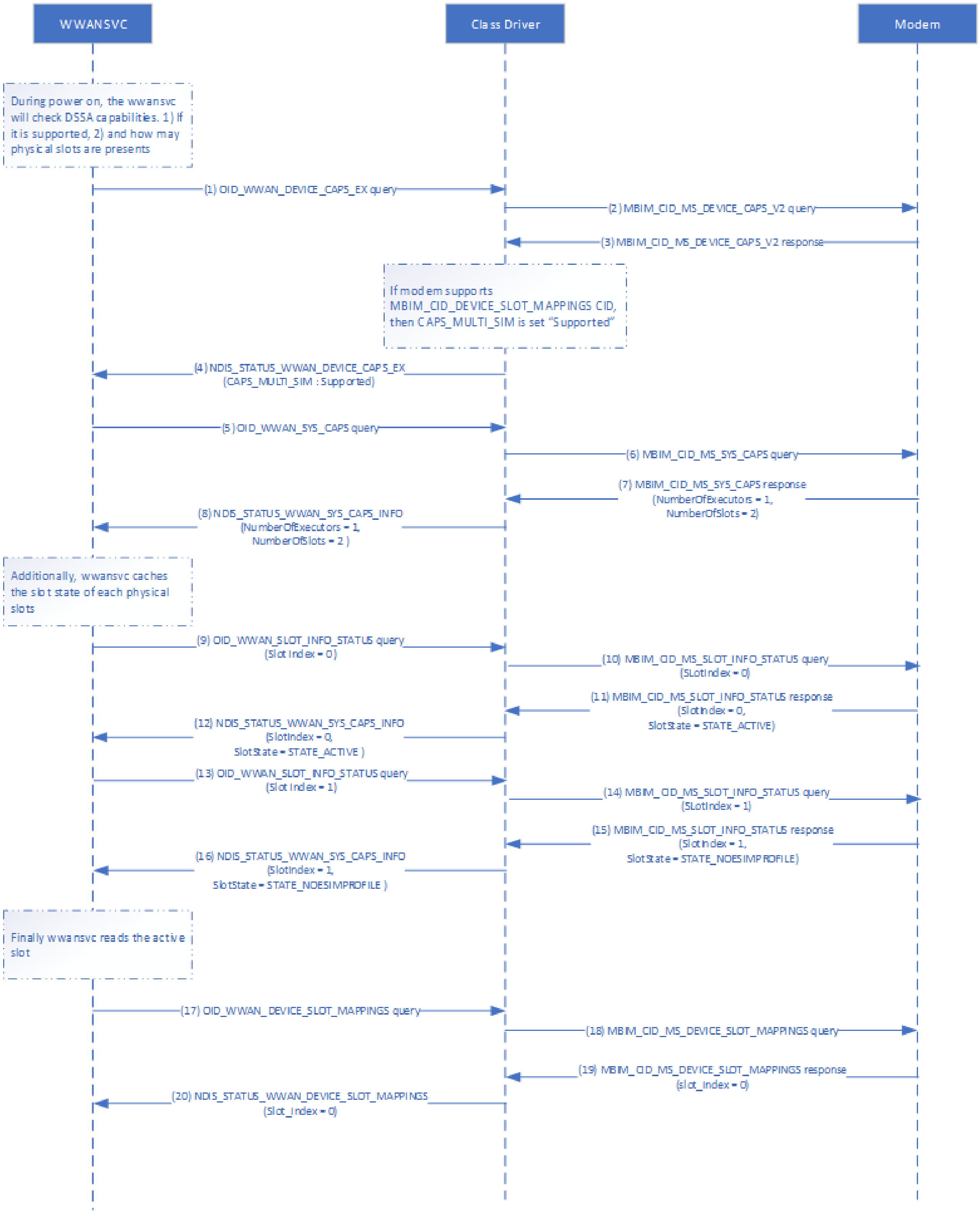
Most of the MBIM CIDs map or relate to NDIS OIDs, but there are a few commands that are used by the Windows WMB class driver that do not have an NDIS counterpart. This section provides clarity on whether those commands are per-modem or per-executor.

Per-device or Per-executor	CID Name
Per-device	CID_MBIM_MSEMERGENCYMODE
	CID_MBIM_MSHOSTSHUTDOWN
Per-executor	CID_MBIM_MSIPADDRESSINFO
	CID_MBIM_MSNETWORKIDLEHINT
	CID_MBIM_MULTICARRIER_CURRENT_CID_LIST

Dual SIM Single Active

Dual SIM single active (DSSA) is the only form of multi-SIM operation that is fully supported in Windows 10. DSSA allows for two SIM cards to be used with the modem, with the restriction that only one SIM can be active at any given time.

Architecture/Flow


WWANSVC
Class Driver
Modem
During power on, the wwansvc will check DSSA capabilities. 1) If it is supported, 2) and how may physical slots are presents
(1) OID_WWAN_DEVICE_CAPS_EX query
(2) MBIM_CID_MS_DEVICE_CAPS_V2 query
(3) MBIM_CID_MS_DEVICE_CAPS_V2 response
If modem supports MBIM_CID_DEVICE_SLOT_MAPPINGS CID, then CAPS_MULTI_SIM is set "Supported"
(4) NDIS_STATUS_WWAN_DEVICE_CAPS_EX (CAPS_MULTI_SIM : Supported)
(5) OID_WWAN_SYS_CAPS query
(6) MBIM_CID_MS_SYS_CAPS query
(7) MBIM_CID_MS_SYS_CAPS response (NumberOfExecutors = 1, NumberOfSlots = 2)
(8) NDIS_STATUS_WWAN_SYS_CAPS_INFO (NumberOfExecutors = 1, NumberOfSlots = 2)
Additionally, wwansvc caches the slot state of each physical slots
(9) OID_WWAN_SLOT_INFO_STATUS query (Slot Index = 0)
(10) MBIM_CID_MS_SLOT_INFO_STATUS query (SLotIndex = 0)
(11) MBIM_CID_MS_SLOT_INFO_STATUS response (SlotIndex = 0, SlotState = STATE_ACTIVE)
(12) NDIS_STATUS_WWAN_SYS_CAPS_INFO (SlotIndex = 0, SlotState = STATE_ACTIVE)
(13) OID_WWAN_SLOT_INFO_STATUS query (Slot Index = 1)
(14) MBIM_CID_MS_SLOT_INFO_STATUS query (SLotIndex = 1)
(15) MBIM_CID_MS_SLOT_INFO_STATUS response (SlotIndex = 1, SlotState = STATE_NOESIMPROFILE)
(16) NDIS_STATUS_WWAN_SYS_CAPS_INFO (SlotIndex = 1, SlotState = STATE_NOESIMPROFILE)
Finally wwansvc reads the active slot
(17) OID_WWAN_DEVICE_SLOT_MAPPINGS query
(18) MBIM_CID_MS_DEVICE_SLOT_MAPPINGS query
(19) MBIM_CID_MS_DEVICE_SLOT_MAPPINGS response (slot_index = 0)
(20) NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPINGS (Slot_Index = 0)

Slot Switch Behavior

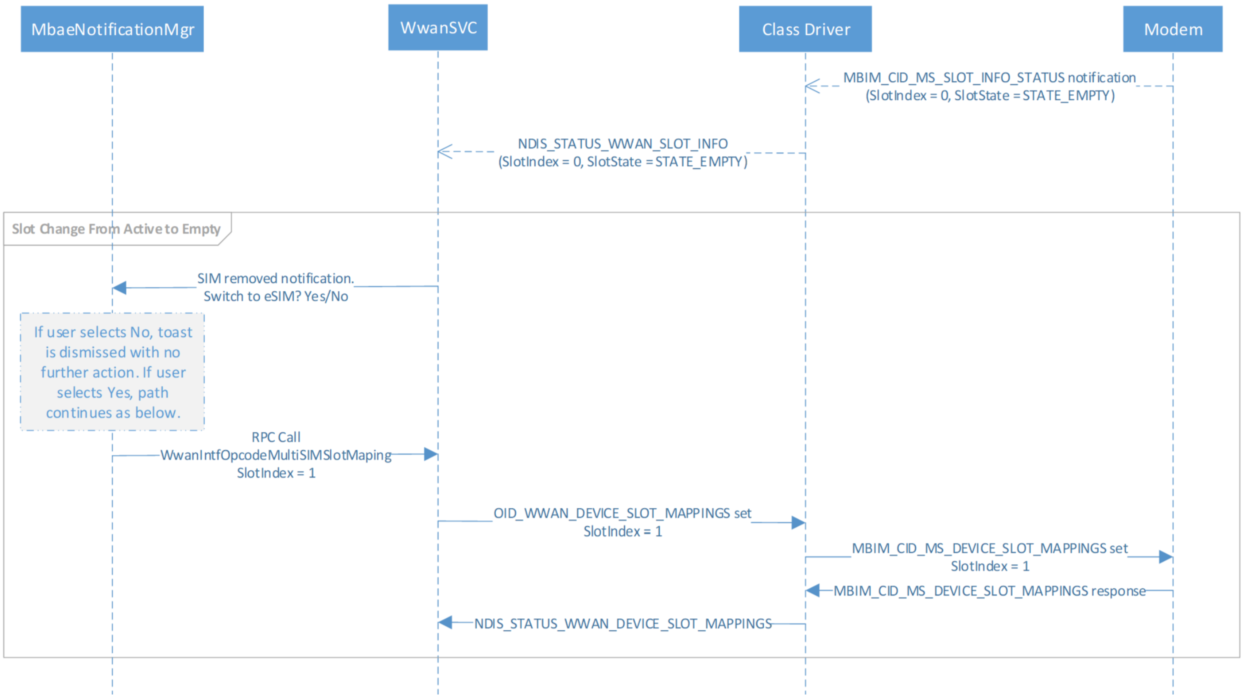
If DSSA is supported on the device, there are some scenarios where slot switch is performed either automatically or prompted by the user via notification toasts.

Out-of-Box Experience (OOBE)

- During OOBE, WwanSvc may perform a slot remap based on the state of the physical slot. If the physical slot is empty, then the embedded slot is selected. If the physical slot has a SIM, the physical slot is selected.

SIM Removal

- If the SIM is removed from the physical slot and the physical slot is the currently selected slot, a toast is displayed asking the user if they want to switch to the embedded slot.
- If the user selects "Yes" then the slot is switched.

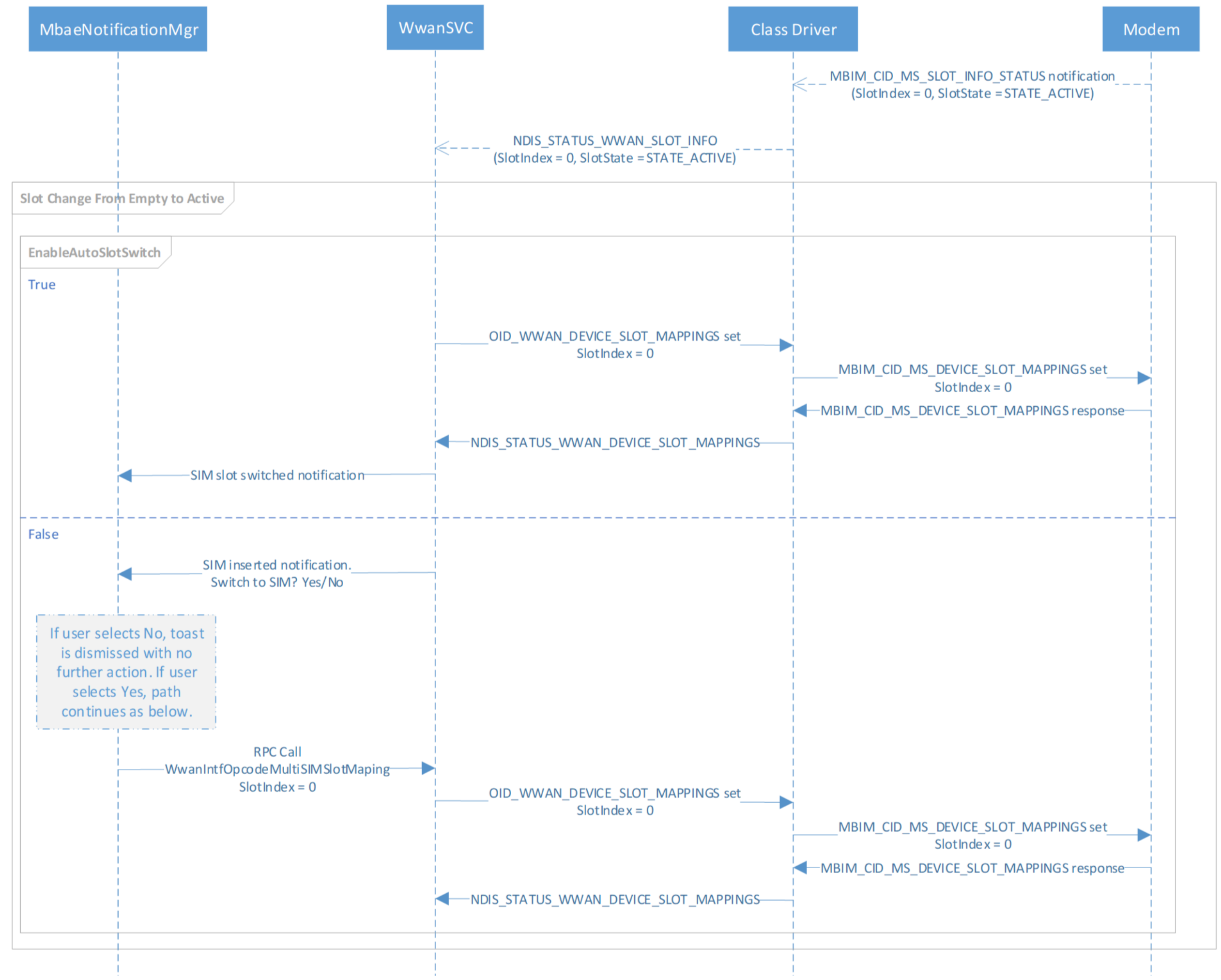

MbaeNotificationMgr
WwanSVC
Class Driver
Modem
MBIM_CID_MS_SLOT_INFO_STATUS notification
(SlotIndex = 0, SlotState = STATE_EMPTY)
NDIS_STATUS_WWAN_SLOT_INFO
(SlotIndex = 0, SlotState = STATE_EMPTY)
Slot Change From Active to Empty
SIM removed notification.
Switch to eSIM? Yes/No
If user selects No, toast is dismissed with no further action. If user selects Yes, path continues as below.
RPC Call
WwanIntfOpcodeMultiSIMSlotMaping
SlotIndex = 1
OID_WWAN_DEVICE_SLOT_MAPPINGS set
SlotIndex = 1
MBIM_CID_MS_DEVICE_SLOT_MAPPINGS set
SlotIndex = 1
MBIM_CID_MS_DEVICE_SLOT_MAPPINGS response
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPINGS

SIM Insert

- If auto-switch is enabled via regkey:
  - If the SIM is inserted in the physical slot while the selected slot is embedded, the slot is automatically switched to the physical slot and a toast is displayed informing the user about the switch.
  - The toast has a button that opens the settings page.
- If auto-switch is disabled via regkey
  - If the SIM is inserted in the physical slot while the selected slot is embedded, a toast is displayed asking if the user wants switch to the physical slot.
  - If user selects "Yes" then the slot is switched.

Use this registry key to configure auto-switch. It does not exist by default.

Location: HKLM\Software\Microsoft\Cellular\MVSettings\DeviceSpecific\CellUX
Key: EnableAutoSlotSwitch
Type: REG_DWORD
Value: 1 | 0 (default, disabled)


MbaeNotificationMgr
WwanSVC
Class Driver
Modem
MBIM_CID_MS_SLOT_INFO_STATUS notification (SlotIndex = 0, SlotState = STATE_ACTIVE)
NDIS_STATUS_WWAN_SLOT_INFO (SlotIndex = 0, SlotState = STATE_ACTIVE)
Slot Change From Empty to Active
EnableAutoSlotSwitch
True
OID_WWAN_DEVICE_SLOT_MAPPINGS set SlotIndex = 0
MBIM_CID_MS_DEVICE_SLOT_MAPPINGS set SlotIndex = 0
MBIM_CID_MS_DEVICE_SLOT_MAPPINGS response
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPINGS
SIM slot switched notification
False
SIM inserted notification. Switch to SIM? Yes/No
If user selects No, toast is dismissed with no further action. If user selects Yes, path continues as below.
RPC Call WwanIntfOpcodeMultiSIMSlotMaping SlotIndex = 0
OID_WWAN_DEVICE_SLOT_MAPPINGS set SlotIndex = 0
MBIM_CID_MS_DEVICE_SLOT_MAPPINGS set SlotIndex = 0
MBIM_CID_MS_DEVICE_SLOT_MAPPINGS response
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPINGS

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to the device Cellular modem driver and run the test: Win6_4.MB.GSM.Data.TestSlot. This test contains the following four tests:

Test Name	Description
QuerySlotMapping	This test verifies the test can successfully query devcie slot mapping.
SetSlotMapping	This test verifies the test can successfully set device slot mapping.
QuerySlotInfo	This test verifies the test can successfully query device slot information.
ValidateSlotInfoState	This test validates UICC Slot state against ReadyInfoState.

Alternatively, you can run the **TestSlot** HLK testlist by **netsh-mbn** and **netsh-mbn-test-installation**.

```
netsh mbn test feature=dssa testpath="C:\data\test\bin" taefpath="C:\data\test\bin"
param="AccessString=internet"
```

This file showing the HLK test results should have been generated in the directory that the 'netsh mbn test' command was ran from: `TestSlot.htm`.

Log Analysis

1. Logs can be collected and decoded using these instructions: MB Collecting Logs
2. Open the .txt file in the TextAnalysisTool
3. Load the DSSA filter

Here is an example log for querying and setting slot mappings:

```
  1619 [5]6C6C.0824::01/09/2020-10:57:17.118 [WwanDimCommon]QUERY
OID_WWAN_DEVICE_CAPS_EX (e01012e), RequestId 11, Status 340001
  1673 [5]6C6C.0824::01/09/2020-10:57:17.118 [WwanDimCommon]QUERY OID_WWAN_SYS_CAPS
(e01012d), RequestId 21, Status 340001
  2488 [5]6C6C.2738::01/09/2020-10:57:17.120 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_DEVICE_CAPS_EX (0x4004103f)
  2520 [5]6C6C.2738::01/09/2020-10:57:17.120 [WwanDimCommon]
SSERVICE_CAPS_MULTI_SIM     : Supported
  2669 [2]6C6C.2738::01/09/2020-10:57:17.121 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_SYS_CAPS_INFO (0x4004102c)
  2679 [2]6C6C.2738::01/09/2020-10:57:17.121 [WwanDimCommon]    NumberOfExecutors
0x1
  2680 [2]6C6C.2738::01/09/2020-10:57:17.121 [WwanDimCommon]    NumberOfSlots 0x2
  3497 [5]6C6C.0824::01/09/2020-10:57:17.125 [WwanDimCommon]QUERY
OID_WWAN_SLOT_INFO_STATUS (e010130), RequestId 42, Status 340001
  3502 [5]6C6C.0824::01/09/2020-10:57:17.125 [WwanDimCommon]    Slot Index    : 0
  3531 [5]6C6C.0824::01/09/2020-10:57:17.126 [WwanDimCommon]QUERY
OID_WWAN_SLOT_INFO_STATUS (e010130), RequestId 32, Status 340001
  3536 [5]6C6C.0824::01/09/2020-10:57:17.126 [WwanDimCommon]    Slot Index    : 1
  6356 [4]6C6C.2738::01/09/2020-10:57:17.133 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
  6890 [4]6C6C.2738::01/09/2020-10:57:17.134 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
  6912 [4]6C6C.2738::01/09/2020-10:57:17.134 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
  6926 [4]6C6C.2738::01/09/2020-10:57:17.134 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_SLOT_INFO (0x4004102e)
  6934 [4]6C6C.2738::01/09/2020-10:57:17.134 [WwanDimCommon]    SlotIndex     : 0x0
  6935 [4]6C6C.2738::01/09/2020-10:57:17.134 [WwanDimCommon]    SlotState     :
WwanUiccSlotStateActive (0x5)
  6955 [4]6C6C.2738::01/09/2020-10:57:17.134 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
  7060 [7]6C6C.2738::01/09/2020-10:57:17.135 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
  7100 [6]6C6C.2738::01/09/2020-10:57:17.135 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_SLOT_INFO (0x4004102e)
  7108 [6]6C6C.2738::01/09/2020-10:57:17.135 [WwanDimCommon]    SlotIndex     : 0x1
  7109 [6]6C6C.2738::01/09/2020-10:57:17.135 [WwanDimCommon]    SlotState     :
WwanUiccSlotStateActiveEsimNoProfile (0x8)
  7140 [6]6C6C.2738::01/09/2020-10:57:17.135 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
  7177 [6]6C6C.2738::01/09/2020-10:57:17.135 [WwanDimCommon]    ReadyState     :
```

```
WwanReadyStateInitialized (0x1)
  8424 [4]6C6C.2738::01/09/2020-10:57:17.137 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
 10616 [6]6C6C.2738::01/09/2020-10:57:17.145 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
 12731 [4]6C6C.2738::01/09/2020-10:57:17.149 [WwanDimCommon]QUERY
OID_WWAN_SYS_SLOTMAPPINGS (e01012f), RequestId 1e1, Status 340001
 12991 [2]6C6C.2738::01/09/2020-10:57:17.150 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO (0x4004102d)
 13003 [2]6C6C.2738::01/09/2020-10:57:17.150 [WwanDimCommon]        Executor Index
0 is mapped to Uicc Slot Index 0
123489 [4]6C6C.2738::01/09/2020-10:57:24.048 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
128251 [2]6C6C.2738::01/09/2020-10:57:24.064 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
128317 [2]6C6C.2738::01/09/2020-10:57:24.064 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
128407 [7]6C6C.2738::01/09/2020-10:57:24.064 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
128445 [7]6C6C.2738::01/09/2020-10:57:24.065 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
129265 [5]6C6C.2738::01/09/2020-10:57:24.067 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
129292 [5]6C6C.2738::01/09/2020-10:57:24.067 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
130122 [7]6C6C.2738::01/09/2020-10:57:24.069 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
155583 [2]6C6C.2738::01/09/2020-10:57:26.637 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
159010 [7]6C6C.2738::01/09/2020-10:57:26.644 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
159034 [7]6C6C.2738::01/09/2020-10:57:26.644 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
161963 [7]6C6C.2738::01/09/2020-10:57:26.655 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
161986 [7]6C6C.2738::01/09/2020-10:57:26.655 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
162110 [2]6C6C.2738::01/09/2020-10:57:26.655 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
162355 [4]6C6C.2738::01/09/2020-10:57:26.656 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
162381 [6]6C6C.2738::01/09/2020-10:57:26.656 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
162441 [4]6C6C.2738::01/09/2020-10:57:26.656 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
194294 [6]6C6C.2738::01/09/2020-10:57:28.722 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
200029 [0]6C6C.2738::01/09/2020-10:57:28.738 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
200131 [4]6C6C.2738::01/09/2020-10:57:28.738 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
200354 [7]6C6C.2738::01/09/2020-10:57:28.739 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
200671 [6]6C6C.2738::01/09/2020-10:57:28.739 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
200729 [7]6C6C.2738::01/09/2020-10:57:28.739 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
200864 [1]6C6C.2738::01/09/2020-10:57:28.740 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
```

```
201464 [0]6C6C.2738::01/09/2020-10:57:28.741 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
265128 [1]6C6C.2218::01/09/2020-10:57:32.150 [WwanDimCommon]SET
OID_WWAN_SYS_SLOTMAPPINGS (e01012f), RequestId a6, Len 10, Status 340001
265133 [1]6C6C.2218::01/09/2020-10:57:32.150 [WwanDimCommon]
SlotMapListHeader.ElementType    : 0xe
265134 [1]6C6C.2218::01/09/2020-10:57:32.150 [WwanDimCommon]
SlotMapListHeader.ElementCount    : 0x1
265135 [1]6C6C.2218::01/09/2020-10:57:32.150 [WwanDimCommon]    Executor Index 0 is
mapped to Uicc Slot Index 1
265523 [6]6C6C.2738::01/09/2020-10:57:32.152 [WwanDimCommon]    ReadyState     :
WwanReadyStateOff (0x0)
270760 [5]6C6C.2738::01/09/2020-10:57:32.171 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO (0x4004102d)
270770 [5]6C6C.2738::01/09/2020-10:57:32.171 [WwanDimCommon]        Executor Index
0 is mapped to Uicc Slot Index 1
270799 [5]6C6C.2738::01/09/2020-10:57:32.171 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_SLOT_INFO (0x4004102e)
270807 [5]6C6C.2738::01/09/2020-10:57:32.171 [WwanDimCommon]    SlotIndex     : 0x0
270808 [5]6C6C.2738::01/09/2020-10:57:32.171 [WwanDimCommon]    SlotState     :
WwanUiccSlotStateEmpty (0x3)
270827 [5]6C6C.2738::01/09/2020-10:57:32.171 [WwanDimCommon]    ReadyState     :
WwanReadyStateFailure (0x4)
271044 [5]6C6C.2738::01/09/2020-10:57:32.172 [WwanDimCommon]    ReadyState     :
WwanReadyStateFailure (0x4)
271089 [5]6C6C.2738::01/09/2020-10:57:32.172 [WwanDimCommon]    ReadyState     :
WwanReadyStateFailure (0x4)
271130 [5]6C6C.2738::01/09/2020-10:57:32.172 [WwanDimCommon]    ReadyState     :
WwanReadyStateSimNotInserted (0x2)
274729 [7]6C6C.2738::01/09/2020-10:57:32.188 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
283027 [6]6C6C.2738::01/09/2020-10:57:32.211 [WwanDimCommon]    ReadyState     :
WwanReadyStateSimNotInserted (0x2)
323130 [5]6C6C.2738::01/09/2020-10:57:32.352 [WwanDimCommon]    ReadyState     :
WwanReadyStateNoEsimProfile (0x7)
403200 [0]6C6C.2738::01/09/2020-10:57:33.748 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_SLOT_INFO (0x4004102e)
403208 [0]6C6C.2738::01/09/2020-10:57:33.748 [WwanDimCommon]    SlotIndex     : 0x0
403209 [0]6C6C.2738::01/09/2020-10:57:33.748 [WwanDimCommon]    SlotState     :
WwanUiccSlotStateActive (0x5)
407008 [5]6C6C.33A8::01/09/2020-10:57:40.355 [WwanDimCommon]SET
OID_WWAN_SYS_SLOTMAPPINGS (e01012f), RequestId 18f, Len 10, Status 340001
407015 [5]6C6C.33A8::01/09/2020-10:57:40.355 [WwanDimCommon]
SlotMapListHeader.ElementType    : 0xe
407017 [5]6C6C.33A8::01/09/2020-10:57:40.355 [WwanDimCommon]
SlotMapListHeader.ElementCount    : 0x1
407018 [5]6C6C.33A8::01/09/2020-10:57:40.355 [WwanDimCommon]    Executor Index 0 is
mapped to Uicc Slot Index 0
407079 [4]6C6C.2738::01/09/2020-10:57:40.355 [WwanDimCommon]    ReadyState     :
WwanReadyStateOff (0x0)
409570 [2]6C6C.2738::01/09/2020-10:57:40.371 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO (0x4004102d)
409580 [2]6C6C.2738::01/09/2020-10:57:40.371 [WwanDimCommon]        Executor Index
0 is mapped to Uicc Slot Index 0
409591 [5]6C6C.2738::01/09/2020-10:57:40.371 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_SLOT_INFO (0x4004102e)
409600 [5]6C6C.2738::01/09/2020-10:57:40.371 [WwanDimCommon]    SlotIndex     : 0x1
409601 [5]6C6C.2738::01/09/2020-10:57:40.371 [WwanDimCommon]    SlotState     :
WwanUiccSlotStateEmpty (0x3)
```

```
411302 [7]6C6C.2738::01/09/2020-10:57:40.385 [WwanDimCommon]    ReadyState     :
WwanReadyStateSimNotInserted (0x2)
416851 [4]6C6C.2738::01/09/2020-10:57:40.510 [WwanDimCommon]    StatusCode    :
NDIS_STATUS_WWAN_SLOT_INFO (0x4004102e)
416859 [4]6C6C.2738::01/09/2020-10:57:40.510 [WwanDimCommon]    SlotIndex     : 0x1
416860 [4]6C6C.2738::01/09/2020-10:57:40.510 [WwanDimCommon]    SlotState     :
WwanUiccSlotStateActiveEsimNoProfile (0x8)
418613 [0]6C6C.2738::01/09/2020-10:57:42.632 [WwanDimCommon]    ReadyState     :
WwanReadyStateOff (0x0)
434410 [4]6C6C.2738::01/09/2020-10:57:44.558 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
443914 [7]6C6C.2738::01/09/2020-10:57:44.593 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
529138 [4]6C6C.2738::01/09/2020-10:57:45.270 [WwanDimCommon]    ReadyState     :
WwanReadyStateInitialized (0x1)
```

DSSA Log Filter

Article • 12/15/2021

To make searching log files easier, below is a DSSA filter file for the TextAnalysisTool.

To use the DSSA log filter:

1. Copy and paste the lines below and save them into a text file named "esimdownload.tat."

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<TextAnalysisTool.NET version="2016-06-16" showOnlyFilteredLines="True">
  <filters>
    <filter enabled="y" excluding="n" description="" backColor="add8e6"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]QUERY
OID_WWAN_DEVICE_CAPS_EX" />
    <filter enabled="y" excluding="n" description="" backColor="ffb6c1"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
StatusCode  : NDIS_STATUS_WWAN_DEVICE_SLOT" />
    <filter enabled="y" excluding="n" description="" backColor="f0e68c"
type="matches_text" case_sensitive="n" regex="n" text="SET
OID_WWAN_SYS_SLOTMAPPINGS" />
    <filter enabled="y" excluding="n" description="" backColor="90ee90"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]QUERY
OID_WWAN_SYS_CAPS" />
    <filter enabled="y" excluding="n" description="" backColor="ffb6c1"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
Executor Index" />
    <filter enabled="y" excluding="n" description="" backColor="add8e6"
type="matches_text" case_sensitive="n" regex="n" text="CAPS_MULTI_SIM" />
    <filter enabled="y" excluding="n" description="" backColor="f08080"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
StatusCode  : NDIS_STATUS_WWAN_SLOT_INFO" />
    <filter enabled="y" excluding="n" description="" backColor="f08080"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
SlotState" />
    <filter enabled="y" excluding="n" description="" backColor="90ee90"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
StatusCode  : NDIS_STATUS_WWAN_SYS_CAPS_INFO" />
    <filter enabled="y" excluding="n" description="" backColor="90ee90"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
NumberOfExecutors" />
    <filter enabled="y" excluding="n" description="" backColor="90ee90"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
NumberOfSlots" />
    <filter enabled="n" excluding="n" description="" backColor="afeeee"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
ReadyState" />
    <filter enabled="y" excluding="n" description="" backColor="add8e6"
```

```
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
StatusCode  : NDIS_STATUS_WWAN_DEVICE_CAPS_EX" />
    <filter enabled="y" excluding="n" description="" backColor="f0e68c"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
SlotMapListHeader" />
    <filter enabled="y" excluding="n" description="" backColor="f0e68c"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
Executor Index" />
    <filter enabled="y" excluding="n" description="" backColor="f08080"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]
SlotIndex" />
    <filter enabled="y" excluding="n" description="" backColor="dda0dd"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]QUERY
OID_WWAN_SYS_SLOTMAPPINGS" />
    <filter enabled="y" excluding="n" description="" backColor="dcdcdc"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]QUERY
OID_WWAN_SLOT_INFO_STATUS" />
    <filter enabled="y" excluding="n" description="" backColor="d3d3d3"
type="matches_text" case_sensitive="n" regex="n" text="[WwanDimCommon]  Slot
Index" />
  </filters>
</TextAnalysisTool.NET>
```

2. Load the filter file into the TextAnalysisTool by clicking File > Load Filters.

MB Provisioned Context Operations

Article • 03/14/2023

Provisioning is vital for cellular-connectable devices because each mobile operator has different APN configurations for its network. APN configurations can generally be split into two categories:

1. APN configurations that are known to the OS because there are applications or clients above the OS that requires those connections.
2. APN configurations that are not made known to the OS because they are internally consumed by the modem for connections that are not leveraged by the OS and its clients.

Ideally, the modem should only store the APN configurations the OS does not have to know. However, IHV and OEM partners have traditionally provided the Internet and Purchase APNs, configurations known to the OS, in the modem as well. Before Windows 10, version 1703's release, Windows only read the Internet and Purchase APN configurations from the modem to establish Internet connections. Starting in Windows 10, version 1703, there might be additional cases in which the modem's APN configuration would have to be managed by Windows, especially if there are clients in the OS such as user settings or OMA-DM that want to change cellular configuration. This in turn could also affect the modem's APN configuration. For example, there might be an IMS stack in the modem that is using the IMS APN for SMS over IMS. Typically, those connections are not exposed to the OS, but under certain scenarios the IMS APN configuration may have to be changed. This change could be done through the OS. In order to support this, starting in Windows 10, version 1703 the OS can configure different types of APNs into the modem.

The USB forum's MBIM 1.0 and Microsoft NDIS each have an existing CID and OID respectively to allow the OS to set and query the APN configurations in the modem. For MBIM 1.0 it does this through MBIM_CID_PROVISIONED_CONTEXT while for NDIS it does this through OID_WWAN_PROVISIONED_CONTEXTS. However, the existing CID and OID were not designed with clear guidance on how the modem is expected to behave in various situations such as a power cycle or SIM swap. Devices that want to support OS configuring and updating of modem-provisioned contexts going forward will have to implement the newer version of the CID and OID in Windows 10, version 1703. To ensure backward compatibility, for IHVs/OEMs that want to support new hardware on OS versions older than 1703, they will have to continue to support the existing MBIM_CID_PROVISIONED_CONTEXT and OID_WWAN_PROVISIONED_CONTEXTS. Starting from Windows 10, version 1703, if the device supports the new version of the CID and OID then the OS will only use the newer version of the command to query and set APN context configuration in the modem.

MB Interface Update for Provisioned Context Operations

While MBIM has a command for retrieving and replacing contexts stored in the modem, it does not have a field to "disable" or "enable" a profile. Therefore, the existing MBIM_CID_PROVISIONED_CONTEXT must be updated for Windows 10, version 1703 to include this capability. Because MBIM does not have a versioning mechanism, a new MSFT proprietary CID is defined as MBIM_CID_MS_PROVISIONED_CONTEXT_V2.

Service Name = **Basic Connect Extensions**

UUID = **UUID_BASIC_CONNECT_EXTENSIONS**

UUID Value = **3d01dcc5-fef5-4d05-0d3abef7058e9aaf**

CID	Command Code	Minimum OS Version
MBIM_CID_MS_PROVISIONED_CONTEXT_V2	1	Windows 10, version 1703

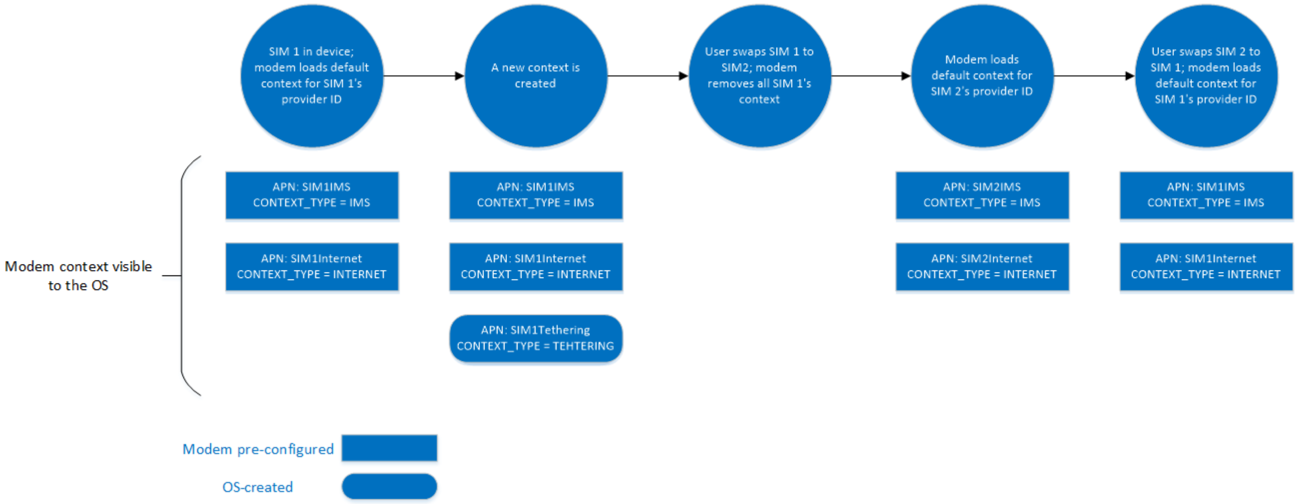
MBIM_CID_MS_PROVISIONED_CONTEXT_V2

Description

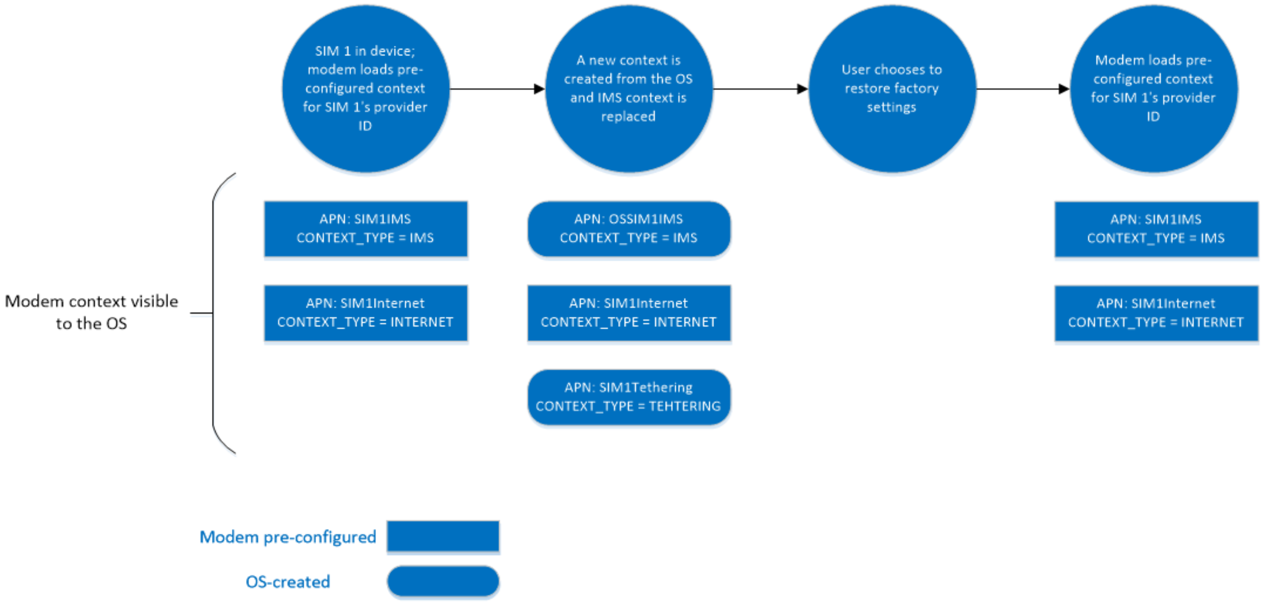
Although MBIM 1.0 has defined MBIM_CID_PROVISIONED_CONTEXT for the OS and its upper clients to manage provisioned contexts in the modem, Windows traditionally only queried the context in the modem but did not set it from the OS. Starting in Windows 10, version 1703, there is increasing need for the OS to be able to configure the contexts in the modem. For example, if there is an IMS stack in the modem that is opaque to the OS, the OS should be able to specify the IMS APN that the modem should use. Because each modem IHV can have its own proprietary way to store contexts in the modem, it is impossible for the OS to manage profiles on the ContextId level the way MBIM_CID_PROVISIONED_CONTEXT might suggest. Instead, from the OS's perspective it is more important to prescribe which context to use for each context type. Returning to the IMS example, regardless of how many existing provisioned contexts are in the modem, if the OS sets a context that has MBIM_CONTEXT_TYPE = IMS then all IMS traffic initiated by the modem should only be attempted on that context.

MBIM 1.0 specifies that MBIM_CID_PROVISIONED_CONTEXT can only call Query on contexts that match the Provider ID (MCC/MNC pair) of the inserted SIM card. For Set requests, MBIM_CID_PROVISIONED_CONTEXT can specify the Provider ID of the context that it wants to be stored. MBIM_CID_MS_PROVISIONED_CONTEXT_V2 specifies a similar but different behavior from MBIM 1.0. For each Query, the OS continues to expect the modem to only return contexts that match the Provider ID of the inserted SIM card. For Set, the command will no longer enable the OS to Set contexts that do not match the current Provider ID in the SIM card. It is expected that the Set request is to create a context for the current Provider ID of the presented SIM card. As an example, the user swaps from SIM 1 to SIM 2, then back to SIM 1. It is expected that during first SIM swap, the modem should resolve all its contexts before loading the context for SIM 2. When the user swaps back to SIM 1, SIM 1's factory default configuration should be restored. It is not expected for the modem to save runtime configuration across SIM swaps.

The following diagram illustrates a sample flow for when a user swaps from one SIM to another, then back to the first one.


SIM 1 in device; modem loads default context for SIM 1's provider ID
A new context is created
User swaps SIM 1 to SIM2; modem removes all SIM 1's context
Modem loads default context for SIM 2's provider ID
User swaps SIM 2 to SIM 1; modem loads default context for SIM 1's provider ID
Modem context visible to the OS
APN: SIM1IMS CONTEXT_TYPE = IMS
APN: SIM1Internet CONTEXT_TYPE = INTERNET
APN: SIM1IMS CONTEXT_TYPE = IMS
APN: SIM1Internet CONTEXT_TYPE = INTERNET
APN: SIM1Tethering CONTEXT_TYPE = TEHTERING
APN: SIM2IMS CONTEXT_TYPE = IMS
APN: SIM2Internet CONTEXT_TYPE = INTERNET
APN: SIM1IMS CONTEXT_TYPE = IMS
APN: SIM1Internet CONTEXT_TYPE = INTERNET
Modem pre-configured
OS-created

OEMs and IHVs that have preconfigured the modem should keep the original factory configuration in case the OS or user wants to restore the context settings in the modem to the original settings. Only the original factory contexts for the currently inserted SIM's Provider ID should be restored. The original factory setting preconfigured contexts should never be overwritten by the OS's configuration. The following diagram is an example flow for when a user chooses to restore factory settings:


SIM 1 in device; modem loads pre-configured context for SIM 1's provider ID
A new context is created from the OS and IMS context is replaced
User chooses to restore factory settings
Modem loads pre-configured context for SIM 1's provider ID
Modem context visible to the OS
APN: SIM1IMS CONTEXT_TYPE = IMS
APN: SIM1Internet CONTEXT_TYPE = INTERNET
APN: OSSIM1IMS CONTEXT_TYPE = IMS
APN: SIM1Internet CONTEXT_TYPE = INTERNET
APN: SIM1Tethering CONTEXT_TYPE = TEHTERING
APN: SIM1IMS CONTEXT_TYPE = IMS
APN: SIM1Internet CONTEXT_TYPE = INTERNET
Modem pre-configured
OS-created

It is expected for the modem to fail Query or Set requests when the SIM is missing, locked, or the Provider ID is inaccessible. The modem should have only one context per CONTEXT_TYPE per Provider ID. If the IHV or OEM decides to preconfigure modem contexts in the modem, it is important for it to make sure that the contexts are configured correctly for each provider for which it chooses to do so. In the case that the inserted SIM card has no IHV preconfigured contexts, the modem should not have any contexts without the OS's configuration. IHVs and OEMs have to make sure MBIM_MS_CONTEXT_SOURCE =

MbimMsContextSourceModemProvisioned so that the OS will use the modem's context for connection, if it exists, and not overwrite it from Windows's APN database.

How the modem maps handle the context and present it back through the existing MBIM_CID_PROVISIONED_CONTEXT is up to each IHV and is out of scope of this documentation.

The new MBIM_CID_MS_PROVISONED_CONTEXT_V2 command is almost identical to MBIM 1.0's existing MBIM_CID_PROVISIONED_CONTEXT command, but with several additions. The first provides the OS with the ability to enable or disable the context associated with a context type in the modem. When the context is disabled in the modem, the modem is expected not to use the stored context for any connection with the network, even those not made aware to the OS. If the OS requests a connection matching a disabled context in the modem, the modem should fail the request immediately without signaling to the network. The matching process should match all fields in the MBIM_MS_CONTEXT_V2 structure.

The MBIM_CONTEXT_IP_TYPE structure from MBIM 1.0 is only used for MBIM_CID_CONNECT. In MBIM_CID_MS_PROVISIONED_CONTEXT_V2, Microsoft has added the IP type as one of the parameters for each context. The modem should report MBIMContextIPTypeDefault if it is not configured for the given context.

In Windows 10, version 1703, with new hardware that supports MBIM_CID_MS_PROVISIONED_CONTEXT_V2, the legacy MBIM_CID_PROVISIONED_CONTEXT will not be used from first party components. If there are other legacy client/OS components that send down MBIM_CID_PROVISIONED_CONTEXT, the modem is expected to return results as it did in versions of Windows earlier than Windows 10, version 1703.

Query

MBIM_MS_PROVISIONED_CONTEXTS_INFO is returned from both Query and Set complete messages in the InformationBuffer.

For Query, the InformationBuffer is null.

Set

For Set, the InformationBuffer contains an MBIM_MS_SET_PROVISIONED_CONTEXT_V2 structure. In the Set operation, because each modem IHV can have proprietary ways of managing context storage, the OS no longer specifies the ContextId field and expects the modem to map the contexts to the appropriate slot. When the OS Sets contexts, it expects the modem to use it for all connections that match the MBIM_CONTEXT_TYPE of the given context. If the MBIM_CONTEXT_TYPE is not recognized by the modem, it should still store it even though it may not connect with it.

Unsolicited Event

The Event InformationBuffer contains an MBIM_MS_PROVISIONED_CONTEXTS_INFO_V2 structure. In some cases, the list of provisioned contexts is updated by the network either Over-The-Air (OTA) or by Short Message Service (SMS) that does not go over the MBIM_CID_MS_PROVISIONED_CONTEXT_V2 command from the OS. The function must update the list of provisioned contexts and tag MBIM_MS_CONTEXT_SOURCE = MbimMsContextSourceOperatorProvisioned accordingly. After that, functions must notify the host about updates using this event with the updated list.

Parameters

Operation	Set	Query	Notification
Command	MBIM_SET_MS_PROVISIONED_CONTEXT_V2	Not applicable	Not applicable
Response	MBIM_MS_PROVISIONED_CONTEXT_INFO_V2	MBIM_MS_PROVISIONED_CONTEXT_INFO_V2	MBIM_MS_PROVISIONED_CONTEXT_INFO_V2

Data Structures

Query

The InformationBuffer shall be NULL and InformationBufferLength shall be zero.

Set

The following MBIM_SET_MS_PROVISIONED_CONTEXT_V2 data structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	Operation	MBIM_MS_CONTEXT_OPERATIONS	Specifies the type of operation for which the SET command is used. If set to MbimMsContextOperationDelete then the context for the specified MBIM_CONTEXT_TYPES should be deleted and all other fields in MBIM_SET_MS_PROVISIONED_CONTEXT_V2 should be ignored. If set to MbimMsContextOperationRestoreFactory then all OS-created or modified contexts should be removed, the default factory preconfigured contexts should be loaded, and all other fields in MBIM_SET_MS_PROVISIONED_CONTEXT_V2 should be ignored.
4	16	ContextType	MBIM_CONTEXT_TYPES	Specifies the type of context being represented; for example, Internet connectivity, VPN (a connection to a corporate network), or Voice-over-IP (VOIP). For more information, see the MBIM_CONTEXT_TYPES table.
20	4	IPType	MBIM_CONTEXT_IP_TYPES	Specifies the type of context being represented; for example, Internet connectivity, VPN (a connection to a corporate network), or Voice-over-IP (VOIP). For more information, see the MBIM_CONTEXT_IP_TYPES table.
24	4	Enable	MBIM_MS_CONTEXT_ENABLE	Specifies whether the context could be used by the modem. If it is set to MbimMsContextDisabled, then any OS connection request that matches the context should be failed without signaling to the network. For more information, see the MBIM_MS_CONTEXT_ENABLE table.
28	4	Roaming	MBIM_MS_CONTEXT_ROAMING_CONTROL	Specifies whether roaming is allowed or not for this context. For more information, see the MBIM_MS_CONTEXT_ROAMING_CONTROL table.
32	4	MediaType	MBIM_MS_CONTEXT_MEDIA_TYPE	Specifies what type of media transport the context is used for. For more information, see the MBIM_MS_CONTEXT_MEDIA_TYPE table.
36	4	Source	MBIM_MS_CONTEXT_SOURCE	Specifies the creation source of the context. For more information, see the MBIM_MS_CONTEXT_SOURCE table.
40	4	AccessStringOffset	OFFSET	Offset in the data buffer to a string, AccessString, to access the network. For GSM-based networks, this would be an Access Point Name (APN) string such as "data.thephone-company.com". For CDMA-based networks, this might be a special dial code such as "#777" or a Network Access Identifier (NAI) such as "foo@thephone-company.com". This member can be NULL to request that the network assign the default APN. Note: Not all networks support this NULL APN convention, so a connect failure caused by an invalid APN is a possible outcome. The size of the string should not exceed 100 characters.
44	4	AccessStringSize	SIZE(0..200)	Size used for AccessString.
48	4	UserNameOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, UserName, that represents the username to authenticate. This member can be NULL.
52	4	UserNameSize	SIZE(0..510)	Size used for UserName .
56	4	PasswordOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, Password, that represents the username's password. This member can be NULL.
60	4	PasswordSize	SIZE(0..510)	Size used for Password.
64	4	Compression	MBIM_COMPRESSION	Specifies the compression to be used in the data connection for header and data. This member applies only to GSM-based devices. The Host sets this member to MBIMCompressionNone for CDMA-based devices. For more information, see the MBIM_COMPRESSION table.

Offset	Size	Field	Type	Description
68	4	AuthProtocol	MBIM_AUTH_PROTOCOL	Authentication type to use for the PDP activation. For more information, see the MBIM_AUTH_PROTOCOL table.
72	4	DataBuffer	DATABUFFER	The data buffer that contains AccessString, UserName, and Password.

The following data structures are used in the preceding table.

MBIM_MS_CONTEXT_ROAMING_CONTROL specifies the per-context roaming policy. The OS can specify whether the given context can be enabled during roaming or not. The modem should not self-activate the context without OS intervention if the roaming state does not satisfy the specified conditions. In cases in which the modem does not support partners, then all partner configurations should be treated as equivalent to home.

Type	Value	Description
MbimMsContextRoamingControlHomeOnly	0	Indicates whether the context is only allowed to be used in the home network or not.
MbimMsContextRoamingControlPartnerOnly	1	Indicates whether the context is only allowed to be used in partner roaming networks or not.
MbimMsContextRoamingControlNonPartnerOnly	2	Indicates whether the context is only allowed to be used in non-partner roaming networks or not.
MbimMsContextRoamingControlHomeAndPartner	3	Indicates whether the context is allowed to be used in home and partner roaming networks.
MbimMsContextRoamingControlHomeAndNonPartner	4	Indicates whether the context is allowed to be used in home and non-partner roaming networks.
MbimMsContextRoamingControlPartnerAndNonPartner	5	Indicates whether the context is allowed to be used in partner and non-partner roaming networks.
MbimMsContextRoamingControlAllowAll	6	Indicates whether the context is allowed to be used in any roaming condition.

MBIM_MS_CONTEXT_MEDIA_TYPE has been added to be able to specify whether the context is used for cellular or iWLAN when Wi-Fi offload becomes supported in future platforms. For example, if a context is set as cellular and the modem is currently Wi-Fi offloading then it should not initiate a connection by using that context.

Type	Value	Description
MbimMsContextMediaTypeCellularOnly	0	Indicates whether the context is only allowed to be used when registered over cellular.
MbimMsContextMediaTypeWifiOnly	1	Indicates whether the context is only allowed to be used when registered over iWLAN (Wi-Fi offload).
MbimMsContextMediaTypeAll	2	Indicates whether the context is allowed to be used when registered either through Cellular or Wi-Fi.

MBIM_MS_CONTEXT_ENABLE specifies whether a context is enabled or disabled.

Type	Value	Description
MbimMsContextDisabled	0	The provisioned context is disabled. The modem should not enable activation on this context from the OS and itself.
MbimMsContextEnabled	1	The provisioned context is enabled. The context can be enabled if other conditions are met; for example, if roaming is disallowed then the context should not be enabled during roaming.

MBIM_MS_CONTEXT_SOURCE has been added to give the OS visibility on how the modem context was created. This helps the OS to behave correctly after various situations such as factory reset, so it can know what should persist and what should be returned to default state based on various operator requirements.

Type	Value	Description
MbimMsContextSourceAdmin	0	The context was created by an Enterprise IT admin from the OS.

Type	Value	Description
MbimMsContextSourceUser	1	The context was created by the user through OS settings.
MbimMsContextSourceOperator	2	The context was created by the operator through OMA-DM or other channels.
MbimMsContextSourceModem	3	The context was created by the IHV or OEM that was included with the modem firmware.
MbimMsContextSourceDevice	4	The context was created by the OS APN database.

MBIM_MS_CONTEXT_OPERATIONS specifies the operations the OS can perform to configure contexts in the modem.

Type	Value	Description
MbimMsContextOperationDefault	0	Default operation including adding or replacing an existing context in the modem.
MbimMsContextOperationDelete	1	Delete operation requires the modem to delete an existing context in the modem.
MbimMsContextOperationRestoreFactory	2	Restore factory preconfigured context for the Provider ID of currently inserted SIM. All contexts replaced or created by OS should be removed and replaced. If there is no default preconfigured OS context for the current inserted SIM Provider ID, then the provisioned context in the modem should be removed.

The original MBIM_CONTEXT_TYPES from MBIM 1.0 is still valid. Microsoft is adding additional context types as more types of contexts were introduced since MBIM 1.0 was defined. The following table defines the new types being introduced. IHVs and OEMs may define other proprietary context types with other unique UUID values that will not be recognizable by the OS for its own purposes.

Type	Value	Description
MBIMMsContextTypeAdmin	5f7e4c2e-e80b-40a9-a239-f0abcfd11f4b	The context is used for administrative purposes such as device management.
MBIMMSContextTypeApp	74d88a3d-dfbd-4799-9a8c-7310a37bb2ee	The context is used for certain applications allowlisted by mobile operators.
MBIMMsContextTypeXcap	50d378a7-baa5-4a50-b872-3fe5bb463411	The context is used for XCAP provisioning on IMS services.
MBIMMsContextTypeTethering	5e4e0601-48dc-4e2b-acb8-08b4016bbaac	The context is used for Mobile Hotspot tethering.
MBIMMsContextTypeEmergencyCalling	5f41adb8-204e-4d31-9da8-b3c970e360f2	The context is used for IMS emergency calling.

Response

The following MBIM_MS_PROVISIONED_CONTEXT_INFO_V2 structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	ElementCount (EC)	UINT32	Count of MBIM_MS_CONTEXT_V2 structures that follow in the DataBuffer.
4	8 * EC	MsProvisionedContextV2RefList	OL_PAIR_LIST	The first element of the pair is a 4 byte Offset in bytes, calculated from the beginning (offset 0) of this MBIM_MS_PROVISIONED_CONTEXTS_INFO_V2 structure, to an MBIM_MS_CONTEXT_V2 structure (for more information, see the MBIM_MS_CONTEXT_V2 table). The second element of the pair is a 4-byte size of a pointer to the corresponding MBIM_MS_CONTEXT_V2 structure.
4 + 8 * EC		DataBuffer	DATABUFFER	Array of MBIM_MS_CONTEXT_V2 structuers.

MBIM_MS_CONTEXT_V2, used in the preceding table, provides information about a given context.

Offset	Size	Field	Type	Description
0	4	ContextId	UINT32	A unique ID for this context.

Offset	Size	Field	Type	Description
4	16	ContextType	MBIM_CONTEXT_TYPES	Specifies the type of context being represented; for example, Internet connectivity, VPN (a connection to a corporate network), or Voice-over-IP (VOIP). Devices should specify MBIMContextTypeNone for empty or un-provisioned contexts. For more information, see the MBIM_CONTEXT_TYPES table.
20	4	IPType	MBIM_CONTEXT_IP_TYPES	For more information, see the MBIM_CONTEXT_IP_TYPES table.
24	4	Enable	MBIM_MS_CONTEXT_ENABLE	Specifies whether the context could be used by the modem. If it is set to MbimMsContextDisabled, then any OS connection request that matches the context should be failed without signaling to the network. For more information, see the MBIM_MS_CONTEXT_ENABLE table.
28	4	Roaming	MBIM_MS_CONTEXT_ROAMING_CONTROL	Specifies whether roaming is allowed or not for this context. For more information, see the MBIM_MS_CONTEXT_ROAMING_CONTROL table.
32	4	MediaType	MBIM_MS_CONTEXT_MEDIA_TYPE	Specifies what type of media transport the context is used for. For more information, see the MBIM_MS_CONTEXT_MEDIA_TYPE table.
36	4	Source	MBIM_MS_CONTEXT_SOURCE	Specifies the creation source of the context. For more information, see the MBIM_MS_CONTEXT_SOURCE table.
40	4	AccessStringOffset	OFFSET	Offset in data buffer to a string, AccessString, to access the network. For GSM-based networks, this would be an Access Point Name (APN) string such as "data.thephone-company.com". For CDMA-based networks, this might be a special dial code such as "#777" or a Network Access Identifier (NAI) such as "foo@thephone-company.com". This member can be NULL, to request that the network assign the default APN. Note: Not all networks support this NULL APN convention, so a connect failure caused by an invalid APN is a possible outcome. The size of the string should not exceed 100 characters.
44	4	AccessStringSize	SIZE(0..200)	Size used for AccessString.
48	4	UserNameOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, UserName, that represents the username to authenticate. This member can be NULL.
52	4	UserNameSize	SIZE(0..510)	Sized used for UserName.
56	4	PasswordOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, Password, that represents the username's password. This member can be NULL.
60	4	PasswordSize	SIZE(0..510)	Size used for Password.
64	4	Compression	MBIM_COMPRESSION	Specifies the compression to be used in the data connection for header and data. This member applies only to GSM-based devices. The Host sets this member to MBIMCompressionNone for CDMA-based devices. For more information, see the MBIM_COMPRESSION table.
68	4	AuthProtocol	MBIM_AUTH_PROTOCOL	Authentication type to use for the PDP activation. For more information, see the MBIM_AUTH_PROTOCOL table.
72		DataBuffer	DATABUFFER	The data buffer that contains AccessString, UserName, and Password.

Notification

For more information, see the MBIM_MS_PROVISIONED_CONTEXT_V2 table.

Status Codes

For Query and Set operations:

Status Code	Description
MBIM_STATUS_READ_FAILURE	The operation failed because the device was unable to retrieve provisioned contexts.
MBIM_STATUS_NO_DEVICE_SUPPORT	The operation failed because the device does not support the operation.

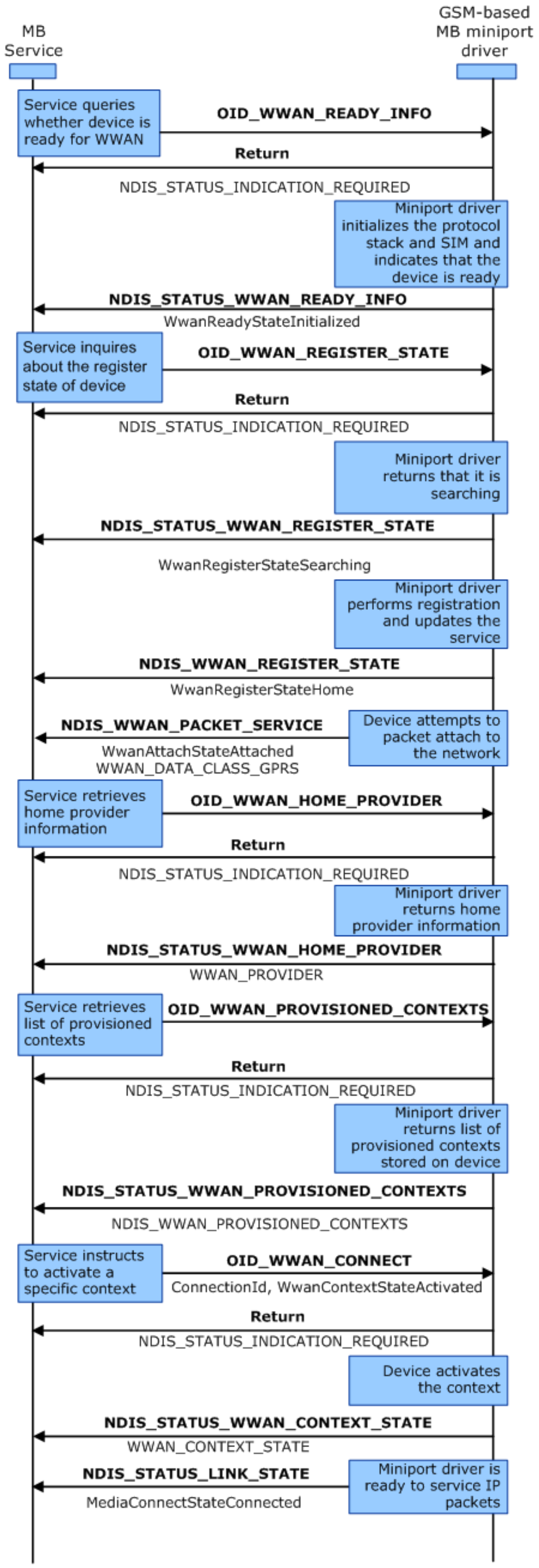
For Set operations only:

Status Code	Description
MBIM_STATUS_INVALID_PARAMETERS	The operation failed because of invalid parameters.
MBIM_STATUS_WRITE_FAILURE	The operation failed because the update request was unsuccessful.

Initialization of devices with a provisioned context

Initialization of a non-SIM-locked GPRS device with a provisioned context

The following diagram represents the optimal user experience for GSM-based MB devices. The out-of-box experience requires no user configuration. It is assumed that the device is configured to automatically select the network to register with. The labels in bold represent OID identifiers or transactional flow control. The labels in regular text represent the important flags within the OID structure.


MB Service
GSM-based MB miniport driver
Service queries whether device is ready for WWAN
OID_WWAN_READY_INFO
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver initializes the protocol stack and SIM and indicates that the device is ready
NDIS_STATUS_WWAN_READY_INFO
WwanReadyStateInitialized
Service inquires about the register state of device
OID_WWAN_REGISTER_STATE
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver returns that it is searching
NDIS_STATUS_WWAN_REGISTER_STATE
WwanRegisterStateSearching
Miniport driver performs registration and updates the service
NDIS_WWAN_REGISTER_STATE
WwanRegisterStateHome
NDIS_WWAN_PACKET_SERVICE
WwanAttachStateAttached
WWAN_DATA_CLASS_GPRS
Device attempts to packet attach to the network
Service retrieves home provider information
OID_WWAN_HOME_PROVIDER
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver returns home provider information
NDIS_STATUS_WWAN_HOME_PROVIDER
WWAN_PROVIDER
Service retrieves list of provisioned contexts
OID_WWAN_PROVISIONED_CONTEXTS
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver returns list of provisioned contexts stored on device
NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS
NDIS_WWAN_PROVISIONED_CONTEXTS
Service instructs to activate a specific context
OID_WWAN_CONNECT
ConnectionId, WwanContextStateActivated
Return
NDIS_STATUS_INDICATION_REQUIRED
Device activates the context
NDIS_STATUS_WWAN_CONTEXT_STATE
WWAN_CONTEXT_STATE
NDIS_STATUS_LINK_STATE
MediaConnectStateConnected
Miniport driver is ready to service IP packets

To initialize a non-SIM-locked GSM-based device, implement the following steps:

1. The MB Service sends an asynchronous (non-blocking) OID_WWAN_READY_INFO query request to the miniport driver to identify the ready state of the device. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

2. The miniport driver sends an NDIS_STATUS_WWAN_READY_INFO notification to the MB Service that indicates to the MB Service that the state of the MB device is **WwanReadyStateInitialized**.

3. The MB Service sends an asynchronous (non-blocking) OID_WWAN_REGISTER_STATE query request to the miniport driver to identify the registration state of the device. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

4. The miniport driver sends an NDIS_STATUS_WWAN_REGISTER_STATE notification to the MB Service that indicates that the registration mode of the device is **WwanRegistraterModeAutomatic** and its current registration state is **WwanRegisterStateSearching**.

5. Later, when the device is registered to a network provider, the miniport driver sends an unsolicited NDIS_STATUS_WWAN_REGISTER_STATE notification to the MB Service that indicates that the current registration state of the device is **WwanRegisterStateHome**.

6. The device attempts to attach the packet service. When the packet service state changes to attached, the miniport driver sends an unsolicited NDIS_STATUS_WWAN_PACKET_SERVICE notification to the MB Service that indicates that the packet service is attached and current data class is **WWAN_DATA_CLASS_GPRS**.

7. The MB Service sends an asynchronous (non-blocking) OID_WWAN_HOME_PROVIDER query request to the miniport driver to retrieve home provider information. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that is has received the request, and it will send a notification with the requested information in the future.

8. The miniport driver sends an NDIS_STATUS_WWAN_HOME_PROVIDER notification to the MB Service that indicates the home provider details.

9. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PROVISIONED_CONTEXTS query request to the miniport driver to retrieve the list of provisioned contexts. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

10. The miniport driver sends an NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS notification to the MB Service that contains a list of WWAN_CONTEXT structures.

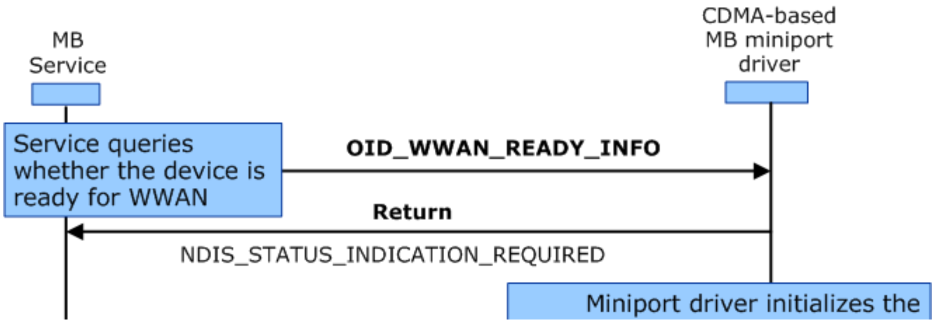
11. The MB Service sends an asynchronous (non-blocking) OID_WWAN_CONNECT set request to the miniport driver to activate the Packet Data Protocol (PDP) context. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.

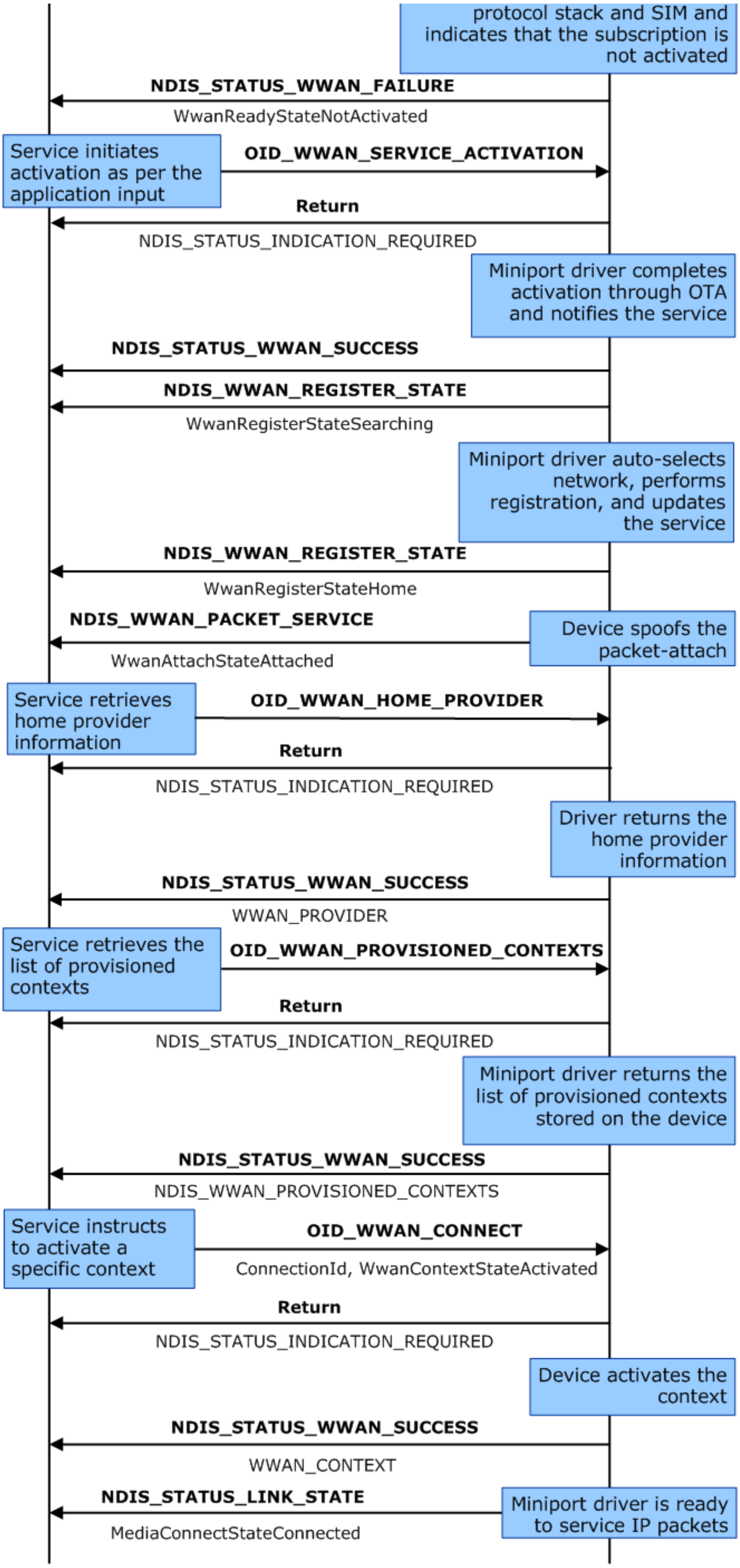
12. The miniport driver sends an NDIS_STATUS_WWAN_CONTEXT_STATE notification to the MB Service that indicates that the PDP context is activated.

13. The miniport driver sends an NDIS_STATUS_LINK_STATE notification to indicate that the media connect state is **MediaConnectStateConnected**.

Initialization of a CDMA Packet Device with a Provisioned Context

The following diagram illustrates the optimal user experience for CDMA-based devices. The out-of-box experience does not require user configuration. This scenario assumes that the CDMA-based account has not been activated. Unlike GSM-based devices, a CDMA-based device automatically starts registration with the network after activation is complete. The labels in bold are OID identifiers or transactional flow control. The labels in regular text are the important flags within the OID structure.


MB Service
CDMA-based MB miniport driver
Service queries whether the device is ready for WWAN
OID_WWAN_READY_INFO
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver initializes the



protocol stack and SIM and indicates that the subscription is not activated
NDIS_STATUS_WWAN_FAILURE
WwanReadyStateNotActivated
Service initiates activation as per the application input
OID_WWAN_SERVICE_ACTIVATION
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver completes activation through OTA and notifies the service
NDIS_STATUS_WWAN_SUCCESS
NDIS_WWAN_REGISTER_STATE
WwanRegisterStateSearching
Miniport driver auto-selects network, performs registration, and updates the service
NDIS_WWAN_REGISTER_STATE
WwanRegisterStateHome
NDIS_WWAN_PACKET_SERVICE
Device spoofs the packet-attach
WwanAttachStateAttached
Service retrieves home provider information
OID_WWAN_HOME_PROVIDER
Return
NDIS_STATUS_INDICATION_REQUIRED
Driver returns the home provider information
NDIS_STATUS_WWAN_SUCCESS
WWAN_PROVIDER
Service retrieves the list of provisioned contexts
OID_WWAN_PROVISIONED_CONTEXTS
Return
NDIS_STATUS_INDICATION_REQUIRED
Miniport driver returns the list of provisioned contexts stored on the device
NDIS_STATUS_WWAN_SUCCESS
NDIS_WWAN_PROVISIONED_CONTEXTS
Service instructs to activate a specific context
OID_WWAN_CONNECT
ConnectionId, WwanContextStateActivated
Return
NDIS_STATUS_INDICATION_REQUIRED
Device activates the context
NDIS_STATUS_WWAN_SUCCESS
WWAN_CONTEXT
NDIS_STATUS_LINK_STATE
Miniport driver is ready to service IP packets
MediaConnectStateConnected

To initialize a CDMA-based packet device with a provisioned context, implement the following steps:

1. The MB Service sends an asynchronous (non-blocking) OID_WWAN_READY_INFO to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.
2. The miniport driver sends NDIS_STATUS_WWAN_FAILURE to the MB Service.
3. The MB Service sends an asynchronous (non-blocking) OID_WWAN_SERVICE_ACTIVATION to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.
4. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.
5. The miniport driver sends **NDIS_STATUS_WWAN_REGISTER_STATE** to the MB Service.
6. The miniport driver sends **NDIS_STATUS_WWAN_REGISTER_STATE** to the MB Service.
7. The miniport driver sends **NDIS_STATUS_WWAN_PACKET_SERVICE** to the MB Service.
8. The MB Service sends an asynchronous (non-blocking) OID_WWAN_HOME_PROVIDER to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.
9. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.
10. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PROVISIONED_CONTEXTS to the miniport driver. The miniport driver responds with a provisional acknowledgement (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and that it will send a notification with the requested information in the future.
11. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.
12. The MB Service sends an asynchronous (non-blocking) OID_WWAN_PROVISIONED_CONTEXTS to the miniport driver. The miniport driver responds with a provisional acknowledgment (NDIS_STATUS_INDICATION_REQUIRED) that it has received the request, and it will send a notification with the requested information in the future.
13. The miniport driver sends NDIS_STATUS_WWAN_SUCCESS to the MB Service.
14. The miniport driver sends **NDIS_STATUS_LINK_STATE** to the MB Service.

See Also

MB Device Readiness

MB Network Blacklist Operations

Article • 03/14/2023

ⓘ Important

Bias-free communication

Microsoft supports a diverse and inclusive environment. This article contains references to terminology that the Microsoft **style guide for bias-free communication** recognizes as exclusionary. The word or phrase is used in this article for consistency because it currently appears in the software. When the software is updated to remove the language, this article will be updated to be in alignment.

A device could be required to not register to a network under various scenarios, such as when a specific SIM card is inserted or if a device does not want to register to a specific network. To address these situations, Windows 10, version 1703 is adding modem interfaces to enable the OS to configure blacklists for SIM cards and network providers.

At any time, the OS can configure the MCC/MNC pair in the modem to specify the SIM or network to which the device is not allowed to register. The interface is flexible enough to allow two different lists, one for SIM providers, and another for network providers. If the device did not attempt registration because a particular SIM or network provider was blacklisted, the modem must report the registration status as denied.

MB Interface Update for Network Blacklist Operations

A new MBIM command has been created to enable the OS to query and set the MCC and MNC pair with which the modem should not attempt registration when a matching SIM cards or network provider is present on the device. For this command, a new MSFT proprietary CID has been defined as MBIM_CID_MS_NETWORK_BLACKLIST.

Service Name = **Basic Connect Extensions**

UUID = **UUID_BASIC_CONNECT_EXTENSIONS**

UUID Value = **3d01dcc5-fef5-4d05-0d3abef7058e9aaf**

CID	Command Code	Minimum OS Version
MBIM_CID_MS_NETWORK_BLACKLIST	2	Windows 10, version 1703

MBIM_CID_MS_NETWORK_BLACKLIST

Description

Enterprises, users or mobile operators may specify the SIM cards and networks on which they do not want the modem to register. This command is used for the OS to be able to query and set the blacklists on the modem. There are two blacklists:

1. A SIM card blacklist – SIM cards whose provider is a member of the blacklist should not be allowed to register on any network.
2. A network provider blacklist – networks on the blacklist should not be allowed to register regardless of what SIM card is present on the device.

The modem has to maintain both blacklists per modem and persist across SIM swaps and power cycles. Both blacklists can be accessed with Query or Set at all times, regardless of the SIM state.

For the Set command it is expected to overwrite the existing blacklists in the modem with the Set command's payload.

Query

MBIM_MS_NETWORK_BLACKLIST_INFO is returned from completed Query and Set messages in the InformationBuffer. For Query, the InformationBuffer is NULL.

Set

For Set, the InformationBuffer contains an MBIM_MS_NETWORK_BLACKLIST_INFO. In the Set operation, a list of MNC/MCC combinations should be provided to the modem. When the SIM card's IMSI matches the MNC and MCC value specified, the modem should deregister from the network and should not try to reregister until a new SIM card that does not match the MNC/MCC is inserted.

Unsolicited Event

An Unsolicited Event is expected if any of the blacklist states have changed from actuated to not actuated, or vice versa; for example, if a SIM is inserted whose provider matches the SIM provider blacklist.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_NETWORK_BLACKLIST_INFO	Not applicable	Not applicable
Response	MBIM_MS_NETWORK_BLACKLIST_INFO	MBIM_MS_NETWORK_BLACKLIST_INFO	MBIM_MS_NETWORK_BLACKLIST_INFO

Data Structures

Query

The InformationBuffer shall be NULL and InformationBufferLength shall be zero.

Set

The following MBIM_MS_NETWORK_BLACKLIST_INFO structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	BlacklistState	MBIM_MS_NETWORK_BLACKLIST_STATE	Indicates whether any of the blacklist conditions are met that result in the modem not registering to the network. For more information, see the MBIM_MS_NETWORK_BLACKLIST_STATE table.
4	4	ElementCount (EC)	UINT32	Count of MBIM_MS_NETWORK_BLACKLIST_PROVIDER structures that follow in the DataBuffer.

Offset	Size	Field	Type	Description
8	8 * EC	BlacklistProviderRefList	OL_PAIR_LIST	The first element of the pair is a 4 byte offset, calculated from the beginning (offset 0) of this MBIM_MS_NETWORK_BLACKLIST_INFO structure, to a MBIM_MS_NETWORK_BLACKLIST_PROVIDER structure. For more information, see the MBIM_MS_NETWORK_BLACKLIST_PROVIDER table. The second element of the pair is a 4-byte size of a pointer to the corresponding MBIM_MS_NETWORK_BLACKLIST_PROVIDER structure.
8 + (8 * EC)		DataBuffer	DATABUFFER	Array of MBIM_MS_NETWORK_BLACKLIST_PROVIDER structures.

The following data structures are used in the preceding table.

MBIM_MS_NETWORK_BLACKLIST_STATE describes the possible states of the two different blacklists.

Type	Mask	Description
MbimMsNetworkBlacklistStateNotActuated	0h	Both blacklist conditions are not met.
MbimMsNetworkBlacklistSIMProviderActuated	1h	Inserted SIM is blacklisted as its Provider ID matches the blacklist for SIM Provider ID.
MbimMsNetworkBlacklistNetworkProviderActuated	2h	Available networks are blacklisted since their Provider IDs are all in the blacklist for network Provider ID.

MBIM_MS_NETWORK_BLACKLIST_PROVIDER specifies the provider of the blacklist.

Offset	Size	Field	Type	Description
0	4	MCC	UINT32	As specified by 3GPP, MCC is part of IMSI and specifies the country of the provider.
4	4	MNC	UINT32	As specified by 3GPP, MNC is part of IMSI and specifies the network of the provider.
8	4	NetworkBlacklistType	MBIM_MS_NETWORK_BLACKLIST_TYPE	Specifies for which type of blacklist the MCC/MNC pair is used. For more information, see the MBIM_MS_NETWORK_BLACKLIST_TYPE table.

MBIM_MS_NETWORK_BLACKLIST_TYPE is used by the preceding data structure. It specifies which of the two blacklists will be used.

Type	Value	Description
MbimMsNetworkBlacklistTypeSIM	0	The MCC/MNC pair are used for SIM provider blacklist.
MbimMsNetworkBlacklistTypeNetwork	1	The MCC/MNC pair are used for network provider blacklist.

Response

For more information, see the MBIM_MS_NETWORK_BLACKLIST_INFO table.

Status Codes

For Query and Set operations:

Status Code	Description
MBIM_STATUS_READ_FAILURE	The operation failed because the device was unable to retrieve provisioned contexts.
MBIM_STATUS_NO_DEVICE_SUPPORT	The operation failed because the device does not support the operation.

For Set operations only:

Status Code	Description
MBIM_STATUS_INVALID_PARAMETERS	The operation failed because of invalid parameters.
MBIM_STATUS_WRITE_FAILURE	The operation failed because the update request was unsuccessful.

MB LTE Attach Operations

Article • 03/14/2023

LTE Attach APN Configuration for MBIM Modems

Traditionally, LTE attach has been considered part of registration and Windows has not directly been involved in LTE attach procedures. However, unlike typical circuit switch network registrations, LTE is a packet switch-only network and requires a default EPS bearer to be enabled for the device to maintain registration on the LTE network.

To establish a default EPS bearer with the network the device must request a PDP context activation during the LTE attach procedure, which requires Access Point Name (APN) specification. Per the 3GPP standard, there are four scenarios where a device can specify APN when it is trying LTE attach:

1. The device specifies a specific LTE attach APN.
2. The device specifies a specific LTE attach APN but the network decides to let the device attach on another APN instead during roaming.
3. The device does not specify a LTE attach APN and lets network assign one back to the device.
4. The device registered from a 2G/3G network to LTE and there was already at minimum one active PDP context. The network uses it as the LTE attach APN.

Today, all LTE attach APN information is provided by IHVs and OEMs directly in the modem for each provider for which it has configuration. However, it is not a fully scalable model for IHVs and OEMs to have all possible LTE attach APN settings for all operators around the globe. Starting in Windows 10, version 1703, new interfaces are defined for both NDIS OIDs and MBIM Microsoft proprietary CIDs to support LTE attach APN configuration from the OS.

Starting in Windows 10, version 1703, if the underlying hardware supports LTE attach APN configuration from the OS then the user will be able to configure the LTE attach APN from Settings. Hardware that has default LTE attach APN configurations must also make its configuration available by the OS.

This feature is supported by adding in two new OIDs and CIDs. For IHV partners that implement MBIM, only the CID version has to be supported.

MB Interface Update for LTE Attach Operations

Two new MBIM CIDs have been created to allow for LTE attach APN configuration and for the OS to retrieve the latest LTE attach status of the device. If IHV partners decide to support OS default LTE attach APN management then both commands must be supported.

Service Name = **Basic Connect Extensions**

UUID = **UUID_BASIC_CONNECT_EXTENSIONS**

UUID Value = **3d01dcc5-fef5-4d05-0d3abef7058e9aaf**

CID	Command Code	Minimum OS Version
MBIM_CID_MS_LTE_ATTACH_CONFIG	3	Windows 10, version 1703
MBIM_CID_MS_LTE_ATTACH_STATUS	4	Windows 10, version 1703

MBIM_CID_MS_LTE_ATTACH_CONFIG

Description

LTE attach contexts can be different , depending on how the network interacts with the device during runtime. For the rest of this documentation, LTE attach context will be referred to as the current PDP context that is being used for LTE attach and default LTE attach context will be referred to as what is configured on the device performing LTE attach with when there is no other existing enabled PDP context. MBIM_CID_MS_LTE_ATTACH_CONFIG enables the OS to Query and Set the default LTE attach context of the inserted SIM's provider (MCC/MNC pair).

Although the LTE attach APN could be technically considered as a context, it differs from all other contexts stored in the modem. For all other contexts activation happens after registration and, based on various conditions, the OS can decide which context is the best fit for connection. However, the LTE attach context is enabled as part of device registration on the LTE network. The OS is unable to retrieve any network-related status before the completion of registration; because of this limitation, the OS must be able to configure LTE attach context for all different roaming conditions of the device to make sure the device can register on the LTE network regardless of what the roaming status is.

LTE attach context activation with the network does not require an OS-explicit connection request as the OS is not aware of any modem self-initiated context activation. Default LTE attach context falls into this category. When the OS issues a MBIM_CID_CONNECT request to enable a PDP context and the given PDP context matches all the following, the modem should complete the CID activation request with success without bringing up a new over-the-air bearer with the network:

1. There is an existing enabled PDP context that is initiated by the modem and not made available to the OS.
2. The PDP context matches the specified APN in the CID request.
3. The IP type of the enabled PDP context is compatible with the requested IP type in the CID.

This is important as the OS is not aware of all the PDP contexts that were initiated by the modem. This will reduce network noise and load. Otherwise, the modem should bring up a new over-the-air bearer matching OS APN specification as per a normal context activation request. The IP type compatibility is specified here:

IP type of the enabled PDP context within the modem	Compatible with requested IP type(s)	Incompatible with requested IP type
IPv4	Default; IPv4; IPv4v6; IPv4 and v6	IPv6
IPv6	Default; IPv6; IPv4v6; IPv4 and v6	IPv4
IPv4v6	Default; IPv4; IPv6; IPv4v6; IPv4 and v6	None

ⓘ **Note**

The modem should not bring up a second PDP context if only one of the IP type is enabled over the air. For example, if IPv4 is enabled and the host requests IPv4 and IPv6 then the modem should complete the activation request without bringing up an IPv6 bearer.

When the OS issues a MBIM_CID_CONNECT request to deactivate a PDP context then the modem should check the following:

1. Whether the device is LTE attached and the context to be deactivated is the only enabled PDP context to maintain LTE registration
2. Whether the context to be deactivated is also used by the modem internally for any services that are not exposed to the OS

If either of these are true, then the modem should complete the CID deactivation request but continue to maintain the over-the-air bearer with the network. Otherwise the modem should deactivate the context as per normal deactivation requests.

All default LTE attach APN configuration provided by the OS is per-provider and matches to the inserted SIM card's home Provider ID (MCC/MNC pair). The modem should only provide configured LTE attach context for the current inserted SIM's Provider ID when queried. The modem should always return three default LTE attach contexts that matches the inserted SIM's Provider ID, one for each roaming condition (home/partner/non-partner).

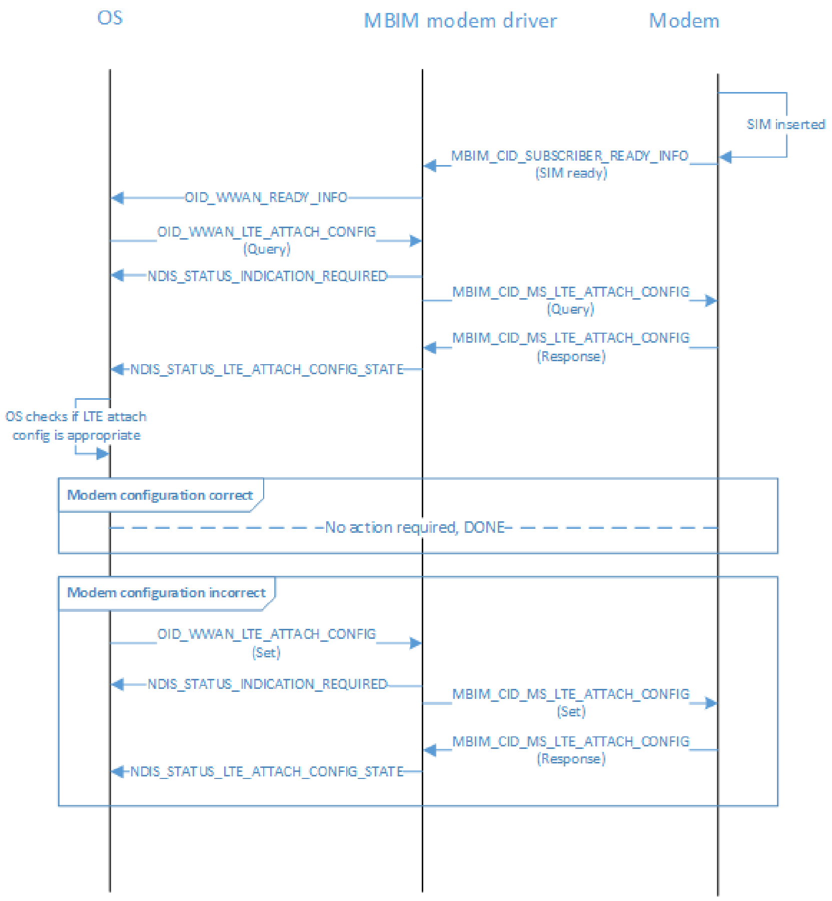
It is expected that across SIM swaps, the modem should clear its default LTE attach context before applying the configuration for the next SIM card. If the newly inserted SIM card has no default LTE attach context configuration, then the device should return NULL empty strings for the APN of the LTE attach context for all roaming conditions while keeping the context enabled. If the context is disabled, it is expected for the device to not attach on LTE because there is no usable configuration for LTE attach. When the user swaps back to a SIM card that was previously configured on the device, the modem should restore its factory default LTE attach configuration for the SIM card. It is not expected for run time configuration to persist across SIM swaps. At any time, there should only be one default LTE attach APN in the modem per roaming condition (home/partner/non-partner).

The OS will always set all three default LTE attach contexts when a Set command is issued, one for each roaming condition. If the list provided by the OS does not have exactly three then the Set command should be rejected. If one of the provided default LTE attach contexts is configured by the OS where the roaming condition matches the current registration status, then the modem should detach from the network and re-perform LTE attach with the newly specified LTE attach context. Otherwise, the device is expected to use the specified default LTE attach context the next time when roaming conditions match. If the device-specified default LTE attach context fails to register on LTE network, then the device should fall back to 3G/2G as appropriate. When the modem cannot differentiate between partner and non-partner networks, the modem should use the non-partner default LTE attach context for all roaming scenarios. If the OS configures default LTE attach context as IP type = default, then it is expected for the modem to assign the most appropriate IP type for LTE attach context. However, the OS expects the modem to still return partner roaming conditions and IP type of LTE attach context that reflects the configuration accurately.

IHVs and OEMs can preconfigure LTE attach context as the default configuration in the modem, but those contexts must be tagged as MBIM_MS_CONTEXT_SOURCE = MbimMsContextSourceModemProvisioned.

Per the 3GPP standard, the default LTE attach context can be split into two categories: UE-initiated and network-initiated. If the device is configured with a NULL empty access string, the device is expected not to provide any LTE attach context to the network and wait for the network to assign one back to the device. Just as prescribed by MBIM 1.0, if the LTE attach context's IP type is configured to be default then the modem should select the best IP type based on its internal algorithm.

The following diagram illustrates an example flow of LTE attach configuration.


OS
MBIM modem driver
Modem
SIM inserted
MBIM_CID_SUBSCRIBER_READY_INFO
(SIM ready)
OID_WWAN_READY_INFO
OID_WWAN_LTE_ATTACH_CONFIG
(Query)
NDIS_STATUS_INDICATION_REQUIRED
MBIM_CID_MS_LTE_ATTACH_CONFIG
(Query)
MBIM_CID_MS_LTE_ATTACH_CONFIG
(Response)
NDIS_STATUS_LTE_ATTACH_CONFIG_STATE
OS checks if LTE attach config is appropriate
Modem configuration correct
No action required, DONE
Modem configuration incorrect
OID_WWAN_LTE_ATTACH_CONFIG
(Set)
NDIS_STATUS_INDICATION_REQUIRED
MBIM_CID_MS_LTE_ATTACH_CONFIG
(Set)
MBIM_CID_MS_LTE_ATTACH_CONFIG
(Response)
NDIS_STATUS_LTE_ATTACH_CONFIG_STATE

Query

MBIM_MS_LTE_ATTACH_CONFIG_INFO is returned from completed Query and Set messages in the InformationBuffer. For Query, the InformationBuffer is NULL.

Set

For Set, the InformationBuffer contains an MBIM_MS_SET_LTE_ATTACH_CONFIG.

Unsolicited Events

The Event InformationBuffer contains an MBIM_MS_LTE_ATTACH_CONFIG_INFO structure. In some cases, the default LTE attach context is updated by the network either Over-The-Air (OTA) or by Short Message Service (SMS) that does not go over the

MBIM_CID_MS_LTE_ATTACH_CONFIG command from the OS. The function must update default LTE attach contexts and tag MBIM_MS_CONTEXT_SOURCE = MbimMsContextSourceOperatorProvisioned accordingly. After that, functions must notify the Host about updates that use this event with the updated list.

Parameters

Operation	Set	Query	Notification
Command	MBIM_SET_MS_LTE_ATTACH_CONFIG	Not applicable	Not applicable
Response	MBIM_MS_LTE_ATTACH_CONFIG_INFO	MBIM_MS_LTE_ATTACH_CONFIG_INFO	MBIM_MS_LTE_ATTACH_CONFIG_INFO

Data Structures

Query

The InformationBuffer shall be NULL and InformationBufferLength shall be zero.

Set

The following MBIM_MS_SET_LTE_ATTACH_CONFIG structure shall be used in the InformationBuffer. The Set command is only valid if the list contains an element count of three, one for each roaming condition (home/partner/non-partner).

Offset	Size	Field	Type	Description
0	4	Operation	MBIM_MS_LTE_CONTEXT_OPERATIONS	Specifies the type of operation for which the Set command is used. If set to MbimMsLteAttachContextOperationRestoreFactory then all other fields should be ignored. OS-created or -modified default LTE attach contexts should be removed and the default factory preconfigured default LTE attach contexts should be loaded. If the modem does not have a default configuration, then all roaming condition default LTE attach contexts should be set to an empty APN string and IP type = default.
4	4	ElementCount (EC)	UINT32	Count of MBIM_MS_LTE_ATTACH_CONTEXT structures that follow in the DataBuffer. This component is currently specified to three, one for each roaming condition (home/partner/non-partner).
8	8 * EC	MsLteAttachContextRefList	OL_PAIR_LIST	The first element of the pair is a 4-byte offset, calculated from the beginning (offset 0) of this MBIM_MS_LTE_ATTACH_CONFIG_INFO structure, to an MBIM_MS_LTE_ATTACH_CONTEXT structure (For more information, see the MBIM_MS_LTE_ATTACH_CONTEXT table). The second element of the pair is a 4-byte size of a pointer to the corresponding MBIM_MS_LTE_ATTACH_CONTEXT structure.
8 + (8 * EC)		DataBuffer	DATABUFFER	Array of MBIM_MS_LTE_ATTACH_CONTEXT structures.

The following structures are used in the preceding table.

MBIM_MS_LTE_ATTACH_CONTEXT_OPERATIONS describes the types of operations that can be used in the Set command.

Type	Value	Description
MbimMsLteAttachContextOperationDefault	0	Default operation for overwriting existing default LTE attach contexts in the modem. The OS will always replace all three default LTE attach context for roaming conditions.

Type	Value	Description
MbimMsLteAttachContextOperationRestoreFactory	1	Restore factory preconfigured default LTE attach context for the Provider ID of currently inserted SIM. All default LTE attach contexts replaced or created by the OS should be removed and replaced. If there is no default preconfigured default LTE attach context for the current inserted SIM Provider ID with one or more roaming conditions, then the default LTE attach should return an empty APN string and IP type = default.

MBIM_MS_LTE_ATTACH_CONTEXT specifies the context to be used for LTE attach configuration.

Offset	Size	Field	Type	Description
0	4	IPType	MBIM_CONTEXT_IP_TYPE	For more information, see the MBIM_CONTEXT_IP_TYPE table.
4	4	Roaming	MBIM_MS_LTE_ATTACH_CONTEXT_ROAMING_CONTROL	Indicates which roaming condition applies to this default LTE attach context. For more information, see the MBIM_MS_LTE_ATTACH_CONTEXT_ROAMING_CONTROL table.
8	4	Source	MBIM_MS_CONTEXT_SOURCE	Specifies the creation source of the context. For more information, see the MBIM_MS_CONTEXT_SOURCE table.
12	4	AccessStringOffset	OFFSET	Offset in data buffer to a string, AccessString, to access the network. For GSM-based networks, this would be an Access Point Name (APN) string such as "data.thephone-company.com". The size of the string should not exceed 100 characters. If the AccessString is empty, then the device expects the network to assign an access string back to the device. IP type still has to be specified in this case.
16	4	AccessStringSize	SIZE(0..200)	Size used for AccessString. This value should be 0 if the device expects the network to assign an access string back to the device for LTE attach.
20	4	UserNameOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, UserName, that represents the username to authenticate. This member can be NULL.
24	4	UserNameSize	SIZE(0..510)	Size used for UserName.
28	4	PasswordOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, Password, that represents the username's password. This member can be NULL.
32	4	PasswordSize	SIZE(0..510)	Size used for Password.
36	4	Compression	MBIM_COMPRESSION	Specifies the compression to be used in the data connection for header and data. This member applies only to GSM-based devices. The Host sets this member to MBIMCompressionNone for CDMA-based devices. For more information, see the MBIM_COMPRESSION table.
40	4	AuthProtocol	MBIM_AUTH_PROTOCOL	Authentication type to use for the PDP activation. For more information, see the MBIM_AUTH_PROTOCOL table.
44		DataBuffer	DATABUFFER	The data buffer that contains AccessString, UserName, and Password.

MBIM_MS_LTE_ATTACH_CONTEXT_ROAMING_CONTROL indicates which roaming condition applies to this default LTE attach context.

Type	Value	Description

Type	Value	Description
MbimMsLteAttachContextRoamingControlHome	0	Indicates whether the default LTE attach context is allowed to be used on home network or not.
MbimMsLteAttachContextRoamingControlPartner	1	Indicates whether the context is allowed to be used on partner roaming networks or not.
MbimMsLteAttachContextRoamingControlNonPartner	2	Indicates whether the context is allowed to be used on non-partner roaming networks or not.

MBIM_MS_CONTEXT_SOURCE specifies the creation source of the context.

Type	Value	Description
MbimMsContextSourceAdmin	0	The context was created by an Enterprise IT admin from the OS.
MbimMsContextSourceUser	1	The context was created by user through the OS settings.
MbimMsContextSourceOperator	2	The context was created by the operator through OMA-DM or other channels.
MbimMsContextSourceModem	3	The context was created by the IHV or OEM.
MbimMsContextSourceDevice	4	The context was created by the OS APN database.

Response

The following MBIM_MS_LTE_ATTACH_CONFIG_INFO structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	ElementCount (EC)	UINT32	Count of MBIM_MS_LTE_ATTACH_CONTEXT structures that follow in the DataBuffer. This component is currently specified to three, one for each roaming condition (home/partner/non-partner).
4	8 * EC	MsLteAttachContextRefList	OL_PAIR_LIST	The first element of the pair is a 4-byte offset, calculated from the beginning (offset 0) of this MBIM_MS_LTE_ATTACH_CONFIG_INFO structure, to an MBIM_MS_LTE_ATTACH_CONTEXT structure (For more information, see the MBIM_MS_LTE_ATTACH_CONTEXT table). The second element of the pair is a 4-byte size of a pointer to the corresponding MBIM_MS_LTE_ATTACH_CONTEXT structure.
4 + (8 * EC)		DataBuffer	DATABUFFER	Array of MBIM_MS_LTE_ATTACH_CONTEXT structures.

Notification

For more information, see the MBIM_MS_LTE_ATTACH_CONFIG_INFO table.

Status Codes

For Query and Set operations:

Status Code	Description
MBIM_STATUS_READ_FAILURE	The operation failed because the device was unable to retrieve provisioned contexts.
MBIM_STATUS_NO_DEVICE_SUPPORT	The operation failed because the device does not support the operation.

For Set operations only:

Status Code	Description
MBIM_STATUS_INVALID_PARAMETERS	The operation failed because of invalid parameters.
MBIM_STATUS_WRITE_FAILURE	The operation failed because the update request was unsuccessful.

MBIM_CID_MS_LTE_ATTACH_STATUS

Description

Per 3GPP requirement, although a device can specify the default LTE attach context to be used when LTE attaching to the network without any enabled PDP context, there might be situations where the device will LTE-attach on a PDP context that differs from the default LTE attach context configured on the device. The following is a list of all possible scenarios:

1. The UE specifies a specific LTE attach APN.
2. The UE specifies a specific LTE attach APN but the network decides to let the device attach on another APN instead during roaming.
3. The UE does not specify a LTE attach APN and lets network assign one back to the device.
4. The UE registered from 2G/3G network to LTE and there was already at minimum one active PDP context. The network uses it as the LTE attach APN.

When the device default LTE attaches, it should send a notification of MBIM_CID_MS_LTE_ATTACH_STATUS to the OS to provide details of the PDP context on the latest LTE attachment. Default LTE attach occurs when one of the following scenarios is fulfilled:

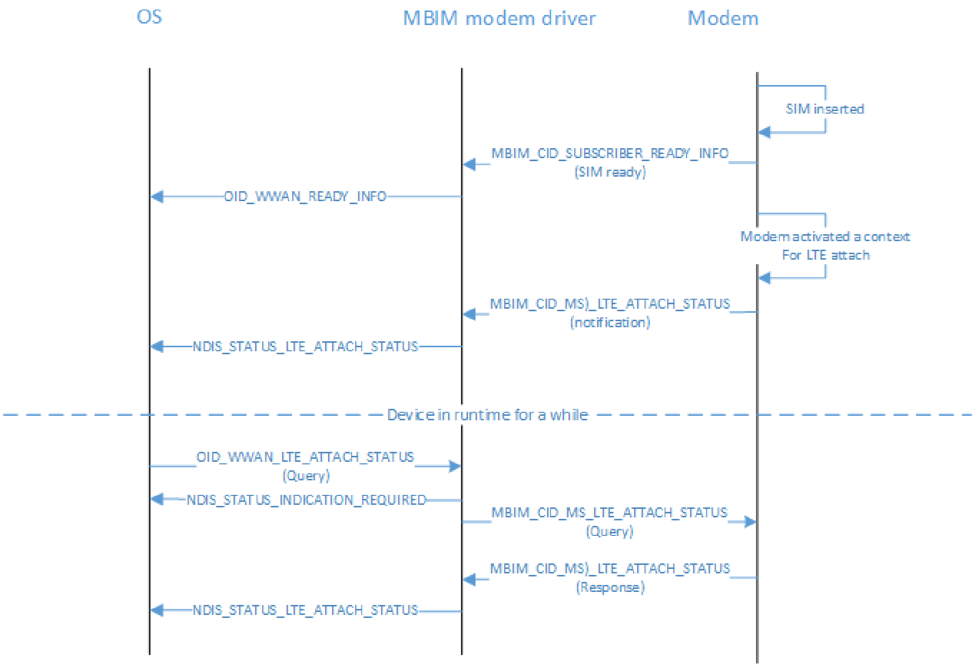
1. Device initially attaches to the LTE network.
2. Device hands up from 2G/3G to LTE without any prior enabled PDP context.

The LTE attach context returned from MBIM_CID_LTE_ATTACH_STATUS could be one of the following:

1. Default LTE attach context stored in the modem.
2. Default LTE attach context that was assigned back from the network.

During runtime, the OS should also be able to query what the last used attach information was for default LTE attach. The modem is expected to return the last known default LTE attach context. If the device was handed off from LTE to 2G/3G network, it is expected for the modem to return the context that was used for the previous LTE attach. Every time that the device deregisters from the network, it is expected for the APN to become empty.

The below diagram illustrates an example message flow for LTE attach status.


OS
MBIM modem driver
Modem
SIM inserted
MBIM_CID_SUBSCRIBER_READY_INFO
(SIM ready)
OID_WWAN_READY_INFO
Modem activated a context
For LTE attach
MBIM_CID_MS)_LTE_ATTACH_STATUS
(notification)
NDIS_STATUS_LTE_ATTACH_STATUS
Device in runtime for a while
OID_WWAN_LTE_ATTACH_STATUS
(Query)
NDIS_STATUS_INDICATION_REQUIRED
MBIM_CID_MS_LTE_ATTACH_STATUS
(Query)
MBIM_CID_MS)_LTE_ATTACH_STATUS
(Response)
NDIS_STATUS_LTE_ATTACH_STATUS

Query

MBIM_MS_LTE_ATTACH_STATUS is returned from Query complete messages in the InformationBuffer. For Query, the InformationBuffer is NULL.

Set

Set operations are not supported.

Unsolicited Events

The Event InformationBuffer contains an MBIM_MS_LTE_ATTACH_STATUS structure.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	Not applicable	Not applicable
Response	Not applicable	MBIM_MS_LTE_ATTACH_STATUS	MBIM_MS_LTE_ATTACH_STATUS

Data Structures

Query

The InformationBuffer shall be NULL and InformationBufferLength shall be zero.

Set

Set operations are not supported.

Response

The following MBIM_MS_LTE_ATTACH_STATUS structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	LteAttachState	MBIM_MS_LTE_ATTACH_STATE	Indicates whether the device is currently attached to a LTE network or not. For more information, see the MBIM_MS_LTE_ATTACH_STATE table.
4	4	IPType	MBIM_CONTEXT_IP_TYPES	For more information, see the MBIM_CONTEXT_IP_TYPE table.
8	4	AccessStringOffset	OFFSET	Offset in data buffer to a string, AccessString, to access the network. For GSM-based networks, this would be an Access Point Name (APN) string such as "data.thephone-company.com". For CDMA-based networks, this might be a special dial code such as "#777" or a Network Access Identifier (NAI) such as "foo@thephone-company.com". This member can be NULL to request that the network assign the default APN. Note: Not all networks support this NULL APN convention. Therefore, a connect failure caused by an invalid APN is a possible outcome. The size of the string should not exceed 100 characters.
12	4	AccessStringSize	SIZE(0..200)	Size in bytes used for AccessString.
16	4	UserNameOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, UserName, that represents the username to authenticate. This member can be NULL.
20	4	UserNameSize	SIZE(0..510)	Size in bytes used for UserName.
24	4	PasswordOffset	OFFSET	Offset in bytes, calculated from the beginning of this structure, to a string, Password, that represents the username's password. This member can be NULL.
28	4	PasswordSize	SIZE(0..510)	Size in bytes used for Password.
32	4	Compression	MBIM_COMPRESSION	Specifies the compression to be used in the data connection for header and data. This member applies only to GSM-based devices. The Host sets this member to MBIMCompressionNone for CDMA-based devices. For more information, see the MBIM_COMPRESSION table.

Offset	Size	Field	Type	Description
36	4	AuthProtocol	MBIM_AUTH_PROTOCOL	Authentication type to use for the PDP activation. For more information, see the MBIM_AUTH_PROTOCOL table.
40	4		DataBuffer	DATABUFFER

The following data structure is used in the preceding table.

MBIM_MS_LTE_ATTACH_STATE indicates whether the device is currently attached to a LTE network or not.

Type	Value	Description
MbimMsLteAttachStateDetached	0	Indicates the device is not attached to LTE network.
MbimMsLteAttachStateAttached	1	Indicates the device is attached to LTE network.

Notification

For more information, see the MBIM_MS_LTE_ATTACH_STATUS table.

Status Codes

For Query and Set operations:

Status Code	Description
MBIM_STATUS_READ_FAILURE	The operation failed because the device was unable to retrieve provisioned contexts.
MBIM_STATUS_NO_DEVICE_SUPPORT	The operation failed because the device does not support the operation.

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to device Cellular modem driver and run the test: Win6_4.MB.GSM.Data.TestLteAttach.

Alternatively, run the **TestLteAttach** HLK testlist by **netsh-mbn** and **netsh-mbn-test-installation**.

```
netsh mbn test feature=lte testpath="C:\\data\\test\\bin" taefpath="C:\\data\\test\\bin"
```

This file showing the HLK test results should have been generated in the directory that the 'netsh mbn test' command was ran from: `TestLteAttach.htm`.

Manual Tests

- Requirement: A sim with the correct APN setting and one more APN information for manual use.

1. Open Settings->Network & Internet -> Cellular
2. Click ***Advanced options***

Using Cellular Settings:

3. There should at least be an apn which is the setting from the sim information. You can get the APN's detailed information by clicking the APN and clicking the "view" button.

Using Manual Settings:

3. Follow the "Add an APN" section in Cellular settings to set the APN manually.
4. Attach the APN and check the attached status.

MB LTE Attach Troubleshooting Guide

1. Get all the Attach APN profiles under %ProgramData%\Microsoft\WwanSvc\DMProfiles
2. Understand which particular profile will be applied based on creation type priorities
3. Investigate the logs to check why the LTE Attach APN was wrongly configured
4. Collect and decode the logs using the instructions in Collecting Logs
5. Open the .txt file generated in the TextAnalysisTool
6. Load the LTE Attach filter

Sample log of LTE Attach

```
10409 [0]0370.0434::2020-03-06 01:16:13.118424000 [WwanDimCommon] ReadyState  : WwanReadyStateInitialized (0x1)
14137 [0]0370.0684::2020-03-06 01:16:13.146883200 [WwanProfileManager]INFO: SaveModemConfiguredLteAttachConfig: 
added modem configured LTE attach profile
14362 [0]0370.0684::2020-03-06 01:16:13.149255900 [WwanProfileManager]INFO: SaveModemConfiguredLteAttachConfig: 
added modem configured LTE attach profile
14476 [1]0370.0434::2020-03-06 01:16:13.149677900 [WwanDimCommon] ReadyState  : WwanReadyStateInitialized (0x1)
14503 [0]0370.0684::2020-03-06 01:16:13.151412000 [WwanProfileManager]INFO: SaveModemConfiguredLteAttachConfig: 
added modem configured LTE attach profile
14962 [0]0370.0684::2020-03-06 01:16:13.156860700 [Microsoft-Windows-WWAN-SVC-EVENTS]WWAN Service event: [Info] 
CWwanDataExecutor::OnLteAttachProfileUpdate: WwanPmGetLteAttachProfileInEffect() didn't find anything, using 
Network Assigned.
14963 [0]0370.0684::2020-03-06 01:16:13.156862600 [Microsoft-Windows-WWAN-SVC-EVENTS]WWAN Service event: [Info] 
CWwanDataExecutor::OnLteAttachProfileUpdate: LTEAttachConfig has same config as modem has, skip
```

LTE Attach Log Filter

Article • 12/15/2021

To load a TextAnalysisTool filter for the LTE Attach operation:

1. Copy and paste the lines below and save them into a text file named "lte-attach.tat."
2. Load the filter file into the TextAnalysisTool by clicking File > Load Filters.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<TextAnalysisTool.NET version="2014-04-22" showOnlyFilteredLines="True">
  <filters>
    <filter enabled="y" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="SaveModemConfiguredLteAttachConfig" />
    <filter enabled="y" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="[WwanProtDim] StatusCode :
NDIS_STATUS_WWAN_LTE_ATTACH_CONFIG (0x4004103d)" />
    <filter enabled="y" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="WwanPmSetDMConfigProfile" />
    <filter enabled="y" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="CWwanDataExecutor::OnLteAttachProfileUpdate" />
    <filter enabled="y" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="ReadyState  : WwanReadyStateInitialized" />
    <filter enabled="y" excluding="n" color="ff0000" type="matches_text"
case_sensitive="n" regex="n" text="WwanPmGetLteAttachProfileInEffect" />
    <filter enabled="y" excluding="n" color="0000ff" type="matches_text"
case_sensitive="n" regex="n" text="IsSameLTEAttachAPN" />
    <filter enabled="y" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="[WwanProtDim] ReadyState" />
    <filter enabled="n" excluding="n" type="matches_text" case_sensitive="n"
regex="n" text="LTEAttachConfig" />
  </filters>
</TextAnalysisTool.NET>
```

MB Protocol Configuration Options (PCO) operations

Article • 03/14/2023

Overview

The purpose of Protocol Configuration Options (PCO) is to transfer the external network protocol options associated with a packet data protocol (PDP) context activation. Windows NDIS definitions for PCO values have typically been generic in order to receive full PCO values from the modem and network in the future. However starting with Windows 10 version 1709 some modems are only able to pass up operator specific PCO elements to the OS. This topic defines the behavior of the current operator specific-only PCO implementation.

Flows

There are three scenarios where the PCO value will be passed to the host:

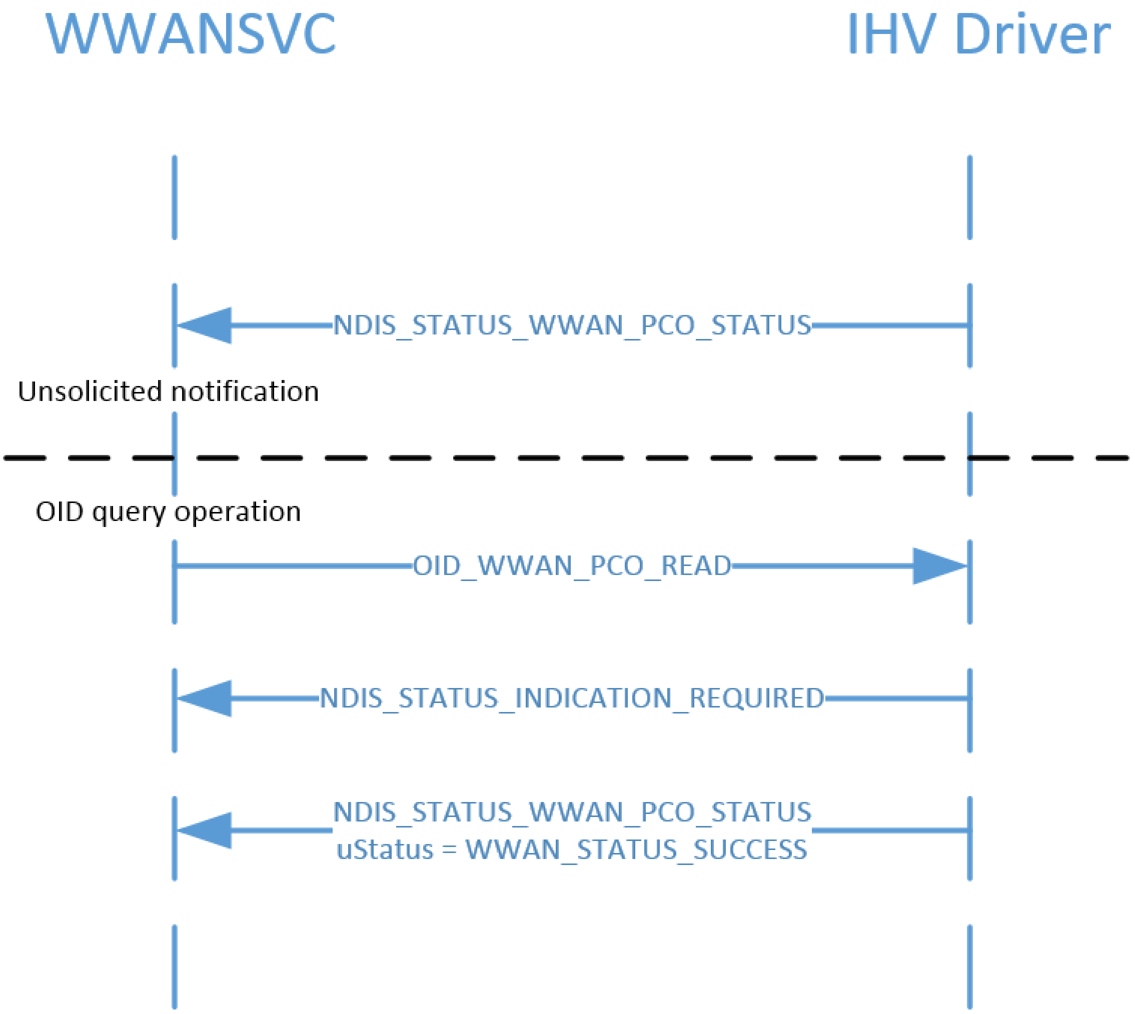
- When a new PCO value has arrived on an activated connection
- When an app or service queries for the latest PCO value from the modem
- When a connection is bridged or activated for the first time and a PCO value already exists in the modem

For the first scenario, the modem should send an NDIS_STATUS_WWAN_PCO_STATUS notification to the OS indicating a new PCO value change whenever a new PCO value is received from the network, with the appropriate NDIS port number to represent the corresponding PDN. To avoid draining the battery unnecessarily, the modem should avoid noisy notifications, as described in Modem behavior with Selective Suspend and Connected Standby.

For the second scenario, when an app or service queries for PCO value from the modem on an activated PDN connection, the host will send the modem an OID_WWAN_PCO query request to read the latest cached PCO value in the modem.

For the third scenario, when a connection is activated or bridged on the host, the modem should send an **NDIS_STATUS_WWAN_PCO_STATUS** notification when a PCO value already exists in the modem for the activated or bridged connection the host requested. The notification should be passed up from the corresponding NDIS port number of the PDN.

The following figure shows the scenario flow:


WWANSVC
IHV Driver
NDIS_STATUS_WWAN_PCO_STATUS
Unsolicited notification
OID query operation
OID_WWAN_PCO_READ
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_PCO_STATUS
uStatus = WWAN_STATUS_SUCCESS

Modem behavior with Selective Suspend and Connected Standby

When Selective Suspend is enabled, the modem can notify the OS whenever it receives a PCO data structure from the network. However, the modem should avoid unnecessary device wakeup. Otherwise, noisy PCO notifications from the network will wake the device up frequently and drain the battery unnecessarily.

When Connected Standby is enabled, the modem shouldn’t notify the OS when it receives PCO data structures from the network because it will not only wake up the device, but it will also wake up the OS, which is not necessary. Instead, the modem should cache all the latest PCO elements from the data structure and notify the OS once the OS exits Connected Standby. For an MBIM modem, it should cache all PCO data structures and only send PCO notifications to the OS after the host has subscribed to it. This will be done using the MBIM_CID_DEVICE_SERVICE_SUBSCRIBE_LIST CID when system power has returned to full power after coming out of Connected Standby.

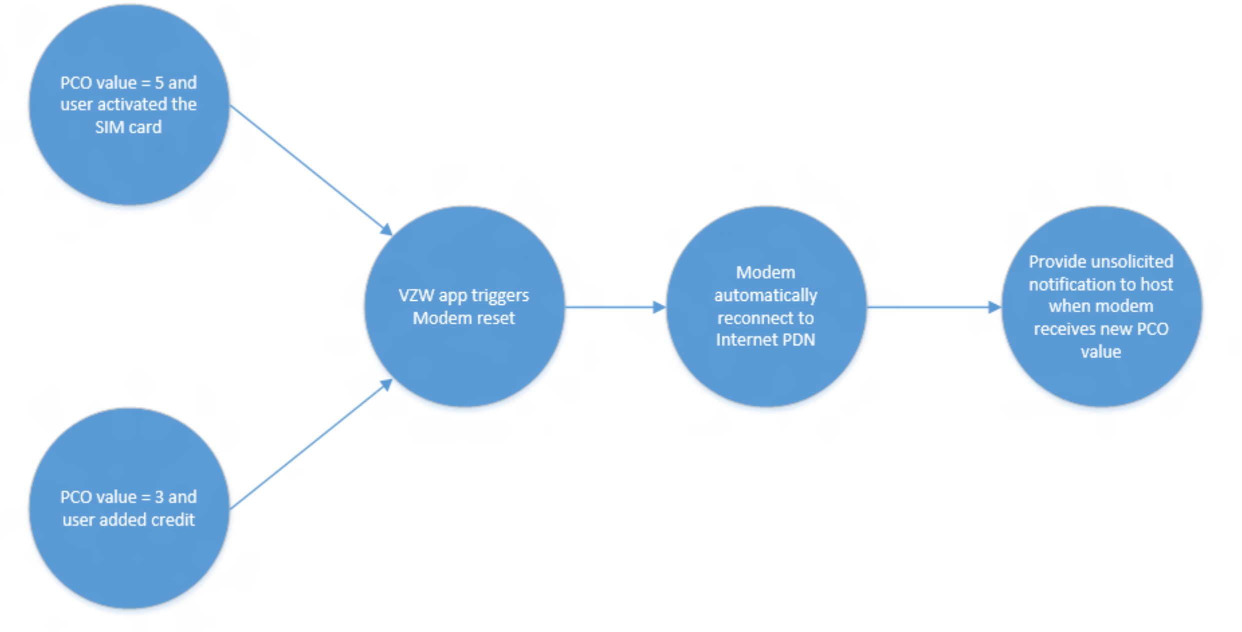
Resetting the modem based on PCO values

Based on PCO values received from the network, the modem will be reset in the following scenarios:

- The user completed self-activation after receiving PCO = 5 from the network. A new PCO value (3, 0 or anything Mobile Operator App can recognize) will be sent to the OS and the OS will pass it to Mobile Operator App.
- The user added more credit to their account after receiving PCO = 3. A new PCO value (0, or anything Mobile Operator App can recognize) will be sent to the OS and the OS will pass it to Mobile Operator App.

The host is not aware of the modem being reset, so the activated connections from the host will not be deactivated and the modem should automatically re-establish connection with those PDN after resetting. Upon establishing connection and receiving a new incoming PCO value from the network, the modem will provide an unsolicited NDIS_STATUS_WWAN_PCO_STATUS notification to the host.

The following diagram illustrates the modem's reset flow when one of these scenarios occurs, with Verizon Wireless as the example MO:


PCO value = 5 and user activated the SIM card
PCO value = 3 and user added credit
VZW app triggers Modem reset
Modem automatically reconnect to Internet PDN
Provide unsolicited notification to host when modem receives new PCO value

NDIS interface to the modem

For querying the status and payload of a PCO value the modem received from the operator network, see OID_WWAN_PCO. **OID_WWAN_PCO** uses the **NDIS_WWAN_PCO_STATUS** structure, which in turn contains a **WWAN_PCO_VALUE** structure representing the PCO information payload from the network.

For the status notification sent by a modem miniport driver to inform the OS of the current PCO state in the modem, see NDIS_STATUS_WWAN_PCO_STATUS.

MB CID to the modem

Service = **MBB_UUID_BASIC_CONNECT_EXT_CONSTANT**

Service UUID = **3d01dcc5-fef5-4d05-0d3a-bef7058e9aaf**

The following CIDs are defined for PCO:

CID	Command code	Minimum OS Version
MBIM_CID_PCO	9	Windows 10, version 1709

MBIM_CID_PCO

This command is used to query the PCO data cached in modem from the mobile operator network.

Query

The InformationBuffer contains an **MBIM_PCO_VALUE** in which the only relevant field is *SessionId*. *SessionId* is reserved for future use and will always be 0 in Windows 10, version 1709. The *SessionId* in a query indicates which IP data stream's PCO value is to be returned by the function.

Set

Not applicable.

Unsolicited Event

Unsolicited events contain an MBIM_PCO_VALUE and are sent when a new PCO value has arrived on an activated connection.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	MBIM_PCO_VALUE	Not applicable

Operation	Set	Query	Notification
Response	Not applicable	MBIM_PCO_VALUE	MBIM_PCO_VALUE

Data Structures

MBIM_PCO_TYPE

Type	Value	Description
MBIMPcoTypeComplete	0	Specifies that the complete PCO structure will be passed up as received from the network and the header realistically reflects the protocol in octet 3 of the PCO structure, defined in the 3GPP TS24.008 spec.
MBIMPcoTypePartial	1	Specifies that the modem will only be passing up a subset of PCO structures that it received from the network. The header matches the PCO structure defined in the 3GPP TS24.008 spec, but the “Configuration protocol” of octet 3 may not be valid.

MBIM-PCO-TYPE

Offset	Size	Field	Type	Description
0	4	SessionId	UINT32	The SessionId in a query indicates which IP data stream’s PCO value is to be returned by the function.
4	4	PcoDataSize	UINT32	The length of PcoData, from 0 to 256. This value will be 0 in a query.
8	4	PcoDataType	UINT32	The PCO data type. For more info, see MBIM_PCO_TYPE.
12		PcoDataBuffer	DATABUFFER	The PCO structure from the 3GPP TS24.008 spec.

Status Codes

This CID only uses Generic Status Codes.

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to the device Cellular modem driver and run the test: TestPco.

WinRT API

PCO

PCO Background Trigger

See Also

NDIS_STATUS_WWAN_PCO_STATUS

NDIS_WWAN_PCO_STATUS

WWAN_PCO_VALUE

OID_WWAN_PCO

MB low level UICC access

Article • 03/14/2023

Overview

The Mobile Broadband Interface Model Revision 1.0, or MBIM1, defines an OEM- and IHV-agnostic interface between a host device and a cellular data modem.

An MBIM1 function includes a UICC smart card and provides access to some of its data and internal state. However, the smart card may have additional capabilities beyond those that are defined by the MBIM interface. These additional capabilities include support for a secure element for mobile payment solutions based upon near-field communication, or for remote provisioning of an entire UICC profile.

In a mobile broadband-enabled Windows device, the MBIM interface is used in addition to the Radio Interface Layer (RIL) interface. One of the features the RIL provides is an interface for low-level access to the UICC. This topic describes a set of Microsoft extensions to MBIM that describe this additional functionality at the MBIM interface.

The Microsoft extensions comprise a set of device service commands (both Set and Query) and notifications. These extensions do not include any new uses of device service streams.

MBIM service and CID values

Service name	UUID	UUID value
Microsoft Low-Level UICC Access	UUID_MS_UICC_LOW_LEVEL	C2F6588E-F037-4BC9-8665-F4D44BD09367

The following table specifies the command code for each CID, as well as whether the CID supports Set, Query, or Event (notification) requests. See each CID's individual section within this topic for more info about its parameters, data structures, and notifications.

CID	Command code	Set	Query	Notify
MBIM_CID_MS_UICC_ATR	1	N	Y	N
MBIM_CID_MS_UICC_OPEN_CHANNEL	2	Y	N	N
MBIM_CID_MS_UICC_CLOSE_CHANNEL	3	Y	N	N
MBIM_CID_MS_UICC_APDU)	4	Y	N	N
MBIM_CID_MS_UICC_TERMINAL_CAPABILITY	5	Y	Y	N
MBIM_CID_MS_UICC_RESET	6	Y	Y	N

Status codes

MBIM status codes are defined in Section 9.4.5 of the MBIM standard. In addition, the following additional failure status codes are defined:

Status Code	Value (hex)	Description
MBIM_STATUS_MS_NO_LOGICAL_CHANNELS	87430001	The logical channel open was not successful because no logical channels are available on the UICC (either it does not support them or all of them are in use).
MBIM_STATUS_MS_SELECT_FAILED	87430002	The logical channel open was not successful because SELECT failed.
MBIM_STATUS_MS_INVALID_LOGICAL_CHANNEL	87430003	The logical channel number is invalid (it was not opened by MBIM_CID_MS_UICC_OPEN_CHANNEL).

MBIM_SUBSCRIBER_READY_STATE

Type	Value	Description
MBIMSubscriberReadyStateNoEsimProfile	7	The card is ready but does not have any enabled profiles.

UICC responses and status

The UICC may implement either a character-based or record-based interface, or both. Although the specific mechanism is different, the result is that the UICC responds to each command with two status bytes (named SW1 and SW2) and a response (which may be empty). A normal success status is indicated by 90 00. However, if the UICC supports the card application toolkit and the UICC wishes to send a proactive command to the terminal, a successful return will be indicated by a status of 91 XX (where XX varies). The MBIM function, or terminal, is responsible for handling this proactive command just as it would handle a proactive command received during any other UICC operation (sending a FETCH to the UICC, handling the proactive command, or sending it to the host with MBIM_CID_STK_PAC). When the MBIM host sends either MBIM_CID_MS_UICC_OPEN_CHANNEL or MBIM_CID_MS_UICC_APDU it should consider both 90 00 and 91 XX as a normal status.

Commands must be able to return responses that are larger than 256 bytes. This mechanism is described in Section 5.1.3 of the ISO/IEC 7816-4:2013 standard. In this case, the card will return SW1 SW2 status words of 61 XX, rather than 90 00, where XX is either the number of remaining bytes or 00 if there are 256 bytes or more remaining. The modem must issue a GET RESPONSE with the same class byte repeatedly until all the data has been received. This will be indicated by the final status words 90 00. The sequence must be uninterrupted within a specific logical channel. Additional APDUs should be handled at the modem and should be transparent to the host. If handled in the host, there is no guarantee that some other APDU may asynchronously reference the card during the sequence of APDUs.

Comparison to IHVRIL

Sections 5.2.3.3.10 through 5.2.3.3.14 of the IHVRIL specification define a similar interface upon which this specification is based. Some differences include:

- The RIL interface does not provide a way to specify secure messaging. The MBIM command to exchange APDUs specifies this as an explicit parameter.

- The RIL interface does not clearly define the interpretation of the class byte within the APDU. The MBIM specification states that the class byte sent from the host must be present but is not used (and instead the MBIM function constructs this byte).
- The RIL interface uses a separate function to close all UICC channels in a group, whereas the MBIM interface accomplishes this with variant arguments to a single CID.
- The relationship between MBIM error status and UICC status (SW1 SW2) is more clearly defined than the relationship between RIL errors and UICC status.
- The MBIM interface distinguishes failure to allocate a new logical channel from failure to SELECT a specified application.
- The MBIM interface permits sending the modem terminal capability objects to send to the card.

MBIM_CID_MS_UICC_ATR

The Answer to Reset (ATR) is the first string of bytes sent by the UICC after a reset has been performed. It describes the capabilities of the card, such as the number of logical channels that it supports. The MBIM function must save the ATR when it is received from the UICC. Subsequently, the host may use the MBIM_CID_MS_UICC_ATR command to retrieve the ATR.

Parameters

Type	Set	Query	Notification
Command	Not applicable	Empty	Not applicable
Response	Not applicable	MBIM_MS_ATR_INFO	Not applicable

Query

The InformationBuffer of a Query message is empty.

Set

Not applicable.

Response

The InformationBuffer of MBIM_COMMAND_DONE contains the following MBIM_MS_ATR_INFO structure describing the answer to reset for the UICC attached to this function.

MBIM_MS_ATR_INFO

Offset	Size	Field	Type	Description
0	4	AtrSize	SIZE(0..33)	The length of **AtrData**.
4	4	AtrOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a byte array called **AtrData** that contains the ATR data.

Offset	Size	Field	Type	Description
8	AtrSize	DataBuffer	DATABUFFER	The **AtrData** byte array.

Unsolicited events

Not applicable.

Status codes

The following status codes are applicable.

Status code	Description
MBIM_STATUS_SUCCESS	Basic MBIM status as defined for all commands.
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_NO_DEVICE_SUPPORT	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_NOT_INITIALIZED	Unable to perform the UICC operation because the UICC is not yet fully initialized.

MBIM_CID_MS_UICC_OPEN_CHANNEL

The host uses the MBIM_CID_MS_UICC_OPEN_CHANNEL command to request that the function open a new logical channel on the UICC card and select a specified UICC application (specified by its application ID).

The function implements this MBIM command using a sequence of UICC commands:

1. The function sends a MANAGE CHANNEL command to the UICC, as described by section 11.1.17 of the ETSI TS 102 221 technical specification, to create a new logical channel. If this command fails, the function returns the MBIM_STATUS_MS_NO_LOGICAL_CHANNELS status with SW1 SW2 and takes no further action.
2. If the MANAGE CHANNEL command succeeds, the UICC reports the channel number of the new logical channel to the function. The function sends a SELECT [by name] command where P1 = 04, as described by section 11.1.1 of the ETSI TS 102 221 technical specification. If this operation fails, the function sends a MANAGE CHANNEL command to the UICC to close the logical channel and returns the MBIM_STATUS_MS_SELECT_FAILED status with SW1 SW2 from the SELECT.
3. If the SELECT command succeeds, the function records the logical channel number and the channel group specified by the host for future reference. It will then return the logical channel number, SW1 SW2 from the SELECT, and the response from the SELECT to the host.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_SET_UICC_OPEN_CHANNEL	Not applicable	Not applicable
Response	MBIM_MS_UICC_OPEN_CHANNEL_INFO	Not applicable	Not applicable

Query

Not applicable.

Set

The InformationBuffer of MBIM_COMMAND_MSG contains the following MBIM_MS_SET_UICC_OPEN_CHANNEL structure.

MBIM_MS_SET_UICC_OPEN_CHANNEL

Offset	Size	Field	Type	Description
0	4	AppIdSize	SIZE(0..32)	The size of the application ID (AppId).
4	4	AppIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a byte array called **AppId** that defines the AppId to be SELECTed.
8	4	SelectP2Arg	UINT32(0..255)	The *P2* argument to the SELECT command.
12	4	ChannelGroup	UINT32	A tag value that identifies the channel group for this channel.
16	AppIdSize	DataBuffer	DATABUFFER	The **AppId** byte array.

Response

The InformationBuffer of MBIM_COMMAND_DONE contains the following MBIM_MS_UICC_OPEN_CHANNEL_INFO structure.

MBIM_MS_UICC_OPEN_CHANNEL_INFO

Offset	Size	Field	Type	Description
0	4	Status	BYTE[2]	SW1 and SW2, in that byte order. For more info, see the notes following this table.
4	4	Channel	UINT32(0..19)	The logical channel identifier. If this member is 0, then the operation failed.
8	4	ResponseLength	SIZE(0..256)	The response length in bytes.

Offset	Size	Field	Type	Description
12	4	ResponseOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a byte array called **Response** that contains the response from the SELECT.
16	-	DataBuffer	DATABUFFER	The **Response** byte array data.

If the command returns MBIM_STATUS_MS_NO_LOGICAL_CHANNELS, the **Status** field shall contain the UICC status words SW1 and SW2 from the MANAGE CHANNEL command and the remaining fields will be zero. If the command returns MBIM_STATUS_MS_SELECT_FAILED, the **Status** field shall contain the UICC status words SW1 and SW2 from the SELECT command and the remaining fields will be zero. For any other status, the InformationBuffer shall be empty.

Unsolicited events

Not applicable.

Status codes

The following status codes are applicable:

Status code	Description
MBIM_STATUS_SUCCESS	Basic MBIM status as defined for all commands.
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_NO_DEVICE_SUPPORT	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_NOT_INITIALIZED	Unable to perform the UICC operation because the UICC is not yet fully initialized.
MBIM_STATUS_MS_NO_LOGICAL_CHANNELS	The logical channel open failed because no logical channels are available on the UICC (either it does not support them or all of them are in use).
MBIM_STATUS_MS_SELECT_FAILED	The logical channel open was not successful because SELECT failed.

MBIM_CID_MS_UICC_CLOSE_CHANNEL

The host sends MBIM_CID_MS_UICC_CLOSE_CHANNEL to the function to close a logical channel on the UICC. The host may specify a channel number or may specify a channel group.

If the host specifies a channel number, the function should check whether this channel was opened by a previous MBIM_CID_MS_UICC_OPEN_CHANNEL. If so, it should send a MANAGE CHANNEL command to the UICC to close the channel, return a status of MBIM_STATUS_SUCCESS, and return the SW1 SW2 from the MANAGE CHANNEL. If not, it should take no action and return the MBIM_STATUS_MS_INVALID_LOGICAL_CHANNEL failure status.

If the host specifies a channel group, the function determines which (if any) logical channels were opened with that channel group and sends a MANAGE CHANNEL command to the UICC for each such channel. It returns a status of MBIM_STATUS_SUCCESS with the SW1 SW2 of the last MANAGE CHANNEL. If no channels were closed it shall return 90 00.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_SET_UICC_CLOSE_CHANNEL	Not applicable	Not applicable
Response	MBIM_MS_UICC_CLOSE_CHANNEL_INFO	Not applicable	Not applicable

Query

Not applicable.

Set

The InformationBuffer of MBIM_COMMAND_MSG contains the following MBIM_MS_SET_UICC_CLOSE_CHANNEL structure.

MBIM_MS_SET_UICC_CLOSE_CHANNEL

Offset	Size	Field	Type	Description
0	4	Channel	UINT32(0..19)	If nonzero, specifies the channel to be closed. If zero, specifies that the channel(s) associated with **ChannelGroup** are to be closed.
4	4	ChannelGroup	UINT32	If **Channel** is zero, this specifies a tag value and all channels with this tag are closed. If **Channel** is nonzero, this field is ignored.

Response

The InformationBuffer of MBIM_COMMAND_DONE contains the following MBIM_MS_UICC_CLOSE_CHANNEL_INFO structure.

MBIM_MS_UICC_CLOSE_CHANNEL_INFO

Offset	Size	Field	Type	Description
0	4	Status	BYTE[2]	SW1 and SW2 of the last MANAGE CHANNEL executed by the function on behalf of this command.

Unsolicited events

Not applicable.

Status codes

Status code	Description
MBIM_STATUS_SUCCESS	Basic MBIM status as defined for all commands.
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_NO_DEVICE_SUPPORT	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_NOT_INITIALIZED	Unable to perform the UICC operation because the UICC is not yet fully initialized.
MBIM_STATUS_MS_INVALID_LOGICAL_CHANNEL	The logical channel number is not valid (in other words, it was not opened with MBIM_CID_MS_UICC_OPEN_CHANNEL).

MBIM_CID_MS_UICC_APDU

The host uses MBIM_CID_MS_UICC_APDU to send a command APDU to a specified logical channel on the UICC and receive the response. The MBIM function should ensure that the logical channel was previously opened with MBIM_CID_MS_UICC_OPEN_CHANNEL and fail with status MBIM_STATUS_MS_INVALID_LOGICAL_CHANNEL if it was not.

The host must send a complete APDU to the function. The APDU may be sent with a class byte value defined in the first interindustry definition in section 4 of the ISO/IEC 7816-4:2013 standard, or in the extended definition in Section 10.1.1 of the ETSI TS 102 221 technical specification. The APDU may be sent without secure messaging or with secure messaging. The command header not authenticated. The host specifies the type of class byte, logical channel number, and secure messaging along with the APDU.

The first byte of the command APDU is the class byte, coded as defined by section 4 of the ISO/IEC 7816-4:2013 standard or section 10.1.1 of the ETSI TS 102 221 technical specification. The host may send 0X, 4X, 6X, 8X, CX, or EX class bytes. However, the function does not pass this byte directly to the UICC. Instead, before sending the APDU to the UICC the function will replace the first byte from the host with a new class byte (encoded as defined by section 4 of the ISO/IEC 7816-4:2013 standard or section 10.1.1 of the ETSI TS 102 221 technical specification) based upon the Type, Channel, and SecureMessaging values specified by the host:

Byte class	Description

Byte class	Description
0X	7816-4 interindustry, 1 <= channel <= 3, encodes security in low nibble if relevant
4X	7816-4 interindustry, 4 <= channel <= 19, no secure messaging
6X	7816-4 interindustry, 4 <= channel <= 19, secure (header not authenticated)
8X	102 221 extended, 1<= channel <= 3, encodes security in low nibble if relevant
CX	102 221 extended, 4 <= channel <= 19, no secure messaging
EX	102 221 extended, 4 <= channel <= 19, secure (header not authenticated)

The function shall return the status, SW1 SW2, and response from the UICC to the host.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_SET_UICC_APDU	Not applicable	Not applicable
Response	MBIM_MS_UICC_APDU_INFO	Not applicable	Not applicable

Query

Not applicable.

Set

The InformationBuffer of MBIM_COMMAND_MSG contains the following MBIM_MS_SET_UICC_APDU structure.

MBIM_MS_SET_UICC_APDU

Offset	Size	Field	Type	Description
0	4	Channel	UINT32(1..19)	Specifies the channel on which the APDU will be sent.
4	4	SecureMessaging	MBIM_MS_UICC_SECURE_MESSAGING	Specifies whether the APDU is exchanged using secure messaging.
8	4	Type	MBIM_MS_UICC_CLASS_BYTE_TYPE	Specifies the type of class byte definition.
12	4	CommandSize	UINT32(0..261)	The **Command** length in bytes.
16	4	CommandOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a byte array called **Command** that contains the APDU.
20	-	DataBuffer	DATABUFFER	The **Command** byte array.

The MBIM_MS_SET_UICC_APDU structure uses the following MBIM_MS_UICC_SECURE_MESSAGING and MBIM_MS_UICC_CLASS_BYTE_TYPE data structures.

MBIM_MS_UICC_SECURE_MESSAGING

Type	Value	Description
MBIMMsUiccSecureMessagingNone	0	No secure messaging.
MBIMMsUiccSecureMessagingNoHdrAuth	1	Secure messaging, command header not authenticated.

MBIM_MS_UICC_CLASS_BYTE_TYPE

Type	Value	Description
MBIMMsUiccInterindustry	0	Defined according to first interindustry definition in ISO 7816-4.
MBIMMsUiccExtended	1	Defined according to the extended definition in ETSI 102 221.

Response

The InformationBuffer of MBIM_COMMAND_DONE contains the following MBIM_MS_UICC_APDU_INFO structure.

MBIM_MS_UICC_APDU_INFO

Offset	Size	Field	Type	Description
0	4	Status	BYTE[2]	The SW1 and SW2 status words resulting from the command.
4	4	ResponseLength	SIZE	The Response length in bytes.
8	4	ResponseOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a byte array called **Response** that contains the response from the SELECT.
12	-	DataBuffer	DATABUFFER	The **Response** byte array.

Unsolicited events

Not applicable.

Status codes

The following status codes are applicable:

Status code	Description
MBIM_STATUS_SUCCESS	Basic MBIM status as defined for all commands.
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.

Status code	Description
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_NO_DEVICE_SUPPORT	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_NOT_INITIALIZED	Unable to perform the UICC operation because the UICC is not yet fully initialized.
MBIM_STATUS_MS_INVALID_LOGICAL_CHANNEL	The logical channel number is not valid (in other words, it was not opened with MBIM_CID_MS_UICC_OPEN_CHANNEL).

If the function can send the APDU to the UICC, it returns MBIM_STATUS_SUCCESS along with the SW1 SW2 status words and the response from the UICC (if any). The host must examine the status (SW1 SW2) to determine whether the APDU command succeeded on the UICC or the reason that it failed.

MBIM_CID_MS_UICC_TERMINAL_CAPABILITY

The host sends MBIM_CID_MS_UICC_TERMINAL_CAPABILITY to inform the modem about the capabilities of the host. The TERMINAL CAPABILITY APDU, specified in Section 11.1.19 of the ETSI TS 102 221 technical specification, must be sent to the card before the first application is selected (if it is supported). Therefore, the host cannot directly send the TERMINAL CAPABILITY APDU but rather sends the MBIM_CID_MS_UICC_TERMINAL_CAPABILITY command containing one or more terminal capability objects which would be stored persistently by the modem. On the next card insertion or reset, after the ATR, the modem would SELECT the MF and check whether TERMINAL CAPABILITY is supported. If so, the modem would send the TERMINAL CAPABILITY APDU with the information specified by the MBIM_CID_MS_UICC_TERMINAL_CAPABILITY command as well as any modem-generated information.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_SET_UICC_TERMINAL_CAPABILITY	Empty	Not applicable
Response	Not applicable	MBIM_MS_TERMINAL_CAPABILITY_INFO	Not applicable

Query

The InformationBuffer shall be null and InformationBufferLength shall be zero.

Set

The InformationBuffer of MBIM_COMMAND_MSG contains the following MBIM_MS_SET_UICC_TERMINAL_CAPABILITY structure.

MBIM_MS_SET_UICC_TERMINAL_CAPABILITY

Offset	Size	Field	Type	Description
0	4	ElementCount	UINT32	The element count of terminal capability objects.
4	8*EC	CapabilityList OL_PAIR_LIST	An offset-length pair list for each terminal capability object TLV.	
4+8*EC	-	DataBuffer	DATABUFFER	A byte array of the actual terminal capability object TLVs.

Response

Responses will contain the exact SET command with the last sent terminal capability objects to the modem. Therefore, MBIM_MS_TERMINAL_CAPABILITY_INFO is identical to MBIM_MS_SET_UICC_TERMINAL_CAPABILITY.

MBIM_MS_TERMINAL_CAPABILITY_INFO

Offset	Size	Field	Type	Description
0	4	ElementCount	UINT32	The element count of terminal capability objects.
4	8*EC	CapabilityList OL_PAIR_LIST	An offset-length pair list for each terminal capability object TLV.	
4+8*EC	-	DataBuffer	DATABUFFER	A byte array of the actual terminal capability object TLVs.

Unsolicited events

Not applicable.

Status codes

Status code	Description
MBIM_STATUS_SUCCESS	Basic MBIM status as defined for all commands.
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_NO_DEVICE_SUPPORT	Basic MBIM status as defined for all commands.

Status code	Description
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_NOT_INITIALIZED	Unable to perform the UICC operation because the UICC is not yet fully initialized.
MBIM_STATUS_MS_INVALID_LOGICAL_CHANNEL	The logical channel number is not valid (in other words, it was not opened with MBIM_CID_MS_UICC_OPEN_CHANNEL).

MBIM_CID_MS_UICC_RESET

The host sends MBIM_CID_MS_UICC_RESET to the MBIM function to reset the UICC or to query the passthrough state of the function.

When the host requests that the function reset the UICC, it specifies a passthrough action.

If the host specifies the *MBIMMsUICCPassThroughEnable* passthrough action, the function resets the UICC and, upon UICC power up, treats the UICC as if it were in a passthrough mode that enables communication between the host and UICC (even if the UICC has no Telecom UICC file system). The function does not send any APDUs to the card and does not interfere at any time with the communication between the host and the UICC.

If the host specifies the *MBIMMsUICCPassThroughDisable* passthrough action, the function resets the UICC and, upon UICC power up, treats the UICC as a regular Telecom UICC and expects a Telecom UICC file system to be present on the UICC.

When the host queries the function to determine the passthrough status, if the function responds with the *MBIMMsUICCPassThroughEnabled* status, it means that passthrough mode is enabled. If the function responds with the *MBIMMsUICCPassThroughDisabled* status, it means that passthrough mode is disabled.

Parameters

Type	Set	Query	Notification
Command	MBIM_MS_SET_UICC_RESET	Empty	Not applicable
Response	MBIM_MS_UICC_RESET_INFO	MBIM_MS_UICC_RESET_INFO	Not applicable

Query

The InformationBuffer shall be null and *InformationBufferLength* shall be zero.

Set

MBIM_SET_MS_UICC_RESET

The MBIM_SET_MS_UICC_RESET structure contains the passthrough action specified by the host.

Offset	Size	Field	Type	Description
0	4	PassThroughAction	MBIM_MS_UICC_PASSTHROUGH_ACTION	For more info, see MBIM_MS_UICC_PASSTHROUGH_ACTION.

MBIM_MS_UICC_PASSTHROUGH_ACTION

The MBIM_MS_UICC_PASSTHROUGH_ACTION enumeration defines the types of passthrough actions the host can specify to the MBIM function.

Types	Value
MBIMMsUiccPassThroughDisable	0
MBIMMsUiccPassThroughEnable	1

Response

MBIM_MS_UICC_RESET_INFO

The MBIM_MS_UICC_RESET_INFO structure contains the passthrough status of the MBIM function.

Offset	Size	Field	Type	Description
0	4	PassThroughStatus	MBIM_MS_UICC_PASSTHROUGH_STATUS	For more info, see MBIM_MS_UICC_PASSTHROUGH_STATUS.

MBIM_MS_UICC_PASSTHROUGH_STATUS

The MBIM_MS_UICC_PASSTHROUGH_STATUS enumeration defines the types of passthrough status the MBIM function specifies to the host.

Types	Value
MBIMMsUiccPassThroughDisabled	0
MBIMMsUiccPassThroughEnabled	1

Unsolicited events

Not applicable.

Status codes

Status code	Description

Status code	Description
MBIM_STATUS_SUCCESS	Basic MBIM status as defined for all commands.
MBIM_STATUS_BUSY	The device is busy.
MBIM_STATUS_FAILURE	The operation failed.
MBIM_STATUS_NO_DEVICE_SUPPORT	The device does not support this operation.

OID_WWAN_UICC_RESET

The NDIS equivalent for MBIM_CID_MS_UICC_RESET is OID_WWAN_UICC_RESET.

MB UICC application and file system access

Article • 03/14/2023

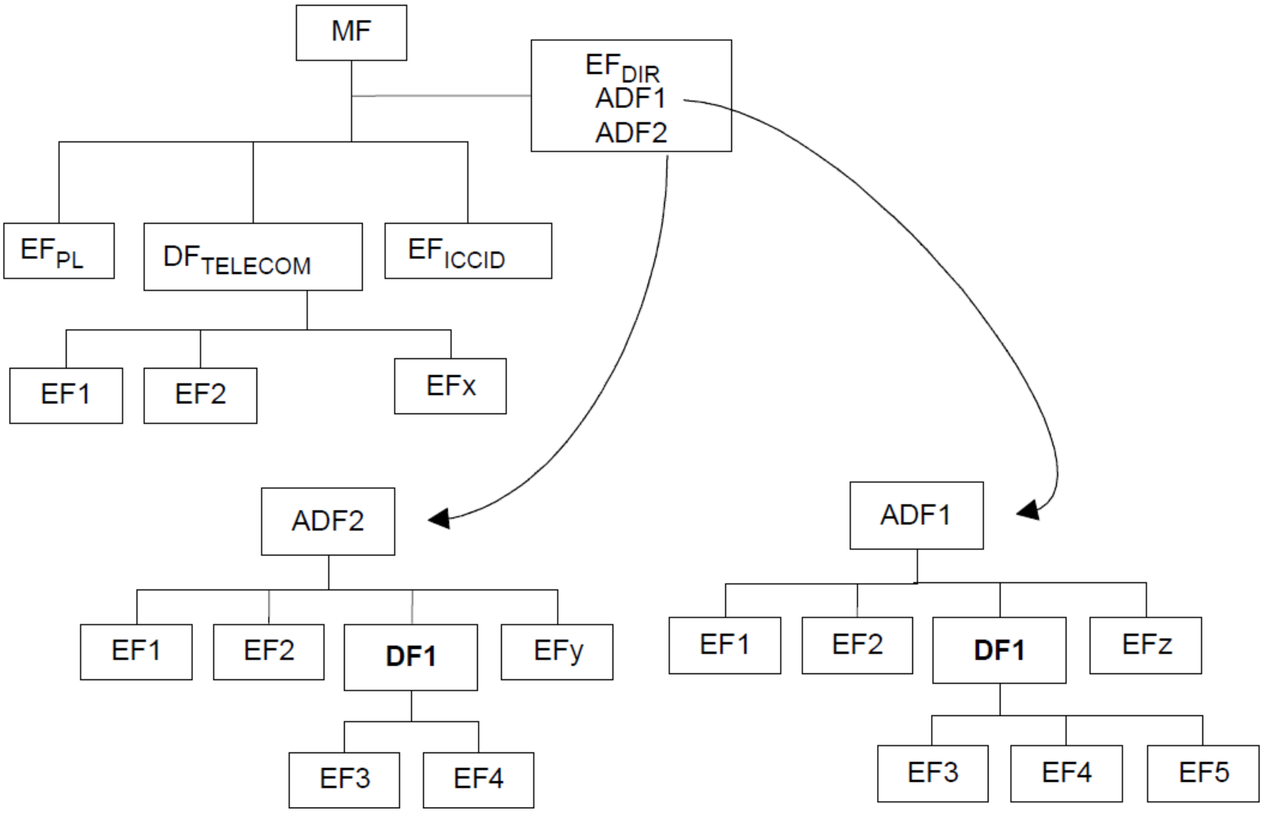
Overview

This topic specifies an extension to the Mobile Broadband Interface Model (MBIM) interface to permit accessing UICC smart card application and file systems. This extension to MBIM exposes logical access to the UICC's ETSI TS 102 221 technical specification -compliant applications and filesystems, and is supported in Windows 10, version 1903 and later.

UICC access and security

The UICC provides a file system and supports a set of applications that can run concurrently. These include the USIM for UMTS, CSIM for CDMA, and ISIM for IMS. The SIM is a legacy portion of the UICC that can be modeled as one of these applications (for GSM).

The following diagram from Section 8.1 of the ETSI TS 102 221 technical specification shows an example card application structure.



The UICC file system can be regarded as a forest of directory trees. The legacy SIM tree is rooted at a Master File (MF) and contains up to two levels of subdirectories (Dedicated Files, or DFs) containing Elemental Files (EFs) that hold various types of information. The SIM defines DFs under the MF, one of which, DFTelecom, contains information common to multiple access types such as the common phone book. Additional applications are effectively implemented as separate trees, each rooted in its own Application Directory File (ADF). Each ADF is identified by an application identifier that can be up to 128 bits long. A file under the card root (EFDir under the MF in the diagram) contains the application names and corresponding identifiers. Within a tree (the MF or an ADF), DFs and EFs might be identified by a path of file IDs, where a file ID is a 16-bit integer.

NDIS interface extensions

The following OIDs have been defined to support UICC application and file system access.

- OID_WWAN_UICC_APP_LIST
- OID_WWAN_UICC_FILE_STATUS
- OID_WWAN_UICC_ACCESS_BINARY
- OID_WWAN_UICC_ACCESS_RECORD
- OID_WWAN_PIN_EX2

MBIM service and CID values

Service name	UUID	UUID value
Microsoft Low-Level UICC Access	UUID_MS_UICC_LOW_LEVEL	C2F6588E-F037-4BC9-8665-F4D44BD09367
Microsoft Basic IP Connectivity Extensions	UUID_BASIC_CONNECT_EXTENSIONS	3D01DCC5-FEF5-4D05-9D3A-BEF7058E9AAF

The following table specifies the UUID and command code for each CID, as well as whether the CID supports Set, Query, or Event (notification) requests. See each CID's individual Section within this topic for more info about its parameters, data structures, and notifications.

CID	UUID	Command code	Set	Query	Notify
MBIM_CID_MS_UICC_APP_LIST	UUID_MS_UICC_LOW_LEVEL	7	N	Y	N
MBIM_CID_MS_UICC_FILE_STATUS	UUID_MS_UICC_LOW_LEVEL	8	N	Y	N
MBIM_CID_MS_UICC_ACCESS_BINARY	UUID_MS_UICC_LOW_LEVEL	9	Y	Y	N
MBIM_CID_MS_UICC_ACCESS_RECORD	UUID_MS_UICC_LOW_LEVEL	10	Y	Y	N
MBIM_CID_MS_PIN_EX	UUID_BASIC_CONNECT_EXTENSIONS	14	Y	Y	N

MBIM_CID_MS_UICC_APP_LIST

This CID retrieves a list of applications in a UICC and information about them. When the UICC in the modem is fully initialized and ready to register with the mobile operator, a UICC application must be selected for registration and a query with this CID should return the selected application in the **ActiveAppIndex** field in the MBIM_UICC_APP_LIST structure used in response.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	Empty	Not applicable
Response	Not applicable	MBIM_UICC_APP_LIST	Not applicable

Query

The InformationBuffer of MBIM_COMMAND_MSG is empty.

Set

Not applicable.

Response

The InformationBuffer in MBIM_COMMAND_DONE contains the following MBIM_UICC_APP_LIST structure.

MBIM_UICC_APP_LIST (version 1)

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of the structure that follows. This field must be set to **1** for version 1 of this structure.
4	4	AppCount	UINT32	The number of UICC application **MBIM_UICC_APP_INFO** structures being returned in this response.
8	4	ActiveAppIndex	UINT32(0..NumApp - 1)	The index of the application selected by the modem for registration with the mobile network. This field must be between **0** and the **AppCount - 1**. It indexes to the array of applications returned by this response. If no application is selected for registration, this field contains **0xFFFFFFFF**.
12	4	AppListSize	UINT32	The size of the app list data, in bytes.
	8*AppCount	AppList	OL_PAIR_LIST	First element of the pair is a 4-byte field with the Offset of an app info in the DataBuffer. Second element of the pair is a 4-byte field with the size of the app info.
	AppListSize	DataBuffer	DATABUFFER	An array of **AppCount** * **MBIM_UICC_APP_INFO** structures.

MBIM_UICC_APP_INFO

Offset	Size	Field	Type	Description
0	4	AppType	MBIM_UICC_APP_TYPE	The type of the UICC application.
4	4	AppIdOffset	OFFSET	Offset for the application ID in the databuffer. Only the first AppIdSize bytes are meaningful. If the application ID is longer than MBIM_MAXLENGTH_APPID bytes, then AppIdSize specifies the actual length but only the first MBIM_MAXLENGTH_APPID bytes are in this field. This field is valid only when AppType is not MBIMUiccAppTypeMf, MBIMUiccAppTypeMfSIM, or MBIMUiccAppTypeMfRUIM.
8	4	AppIdSize	SIZE (0..16)	The size of the application ID, in bytes, as defined in Section 8.3 of the ETSI TS 102 221 technical specification. AppIdSize may contain a number greater than 16, but in this case only the first 16 (MBIM_MAXLENGTH_APPID) bytes are in the databuffer. This field is set to zero for the MBIMUiccAppTypeMf, MBIMUiccAppTypeMfSIM, or MBIMUiccAppTypeMfRUIM app types.
12		AppNameOffset	OFFSET	Offset for the application name in the databuffer. A UTF-8 string specifying the name of the application. The length of this field is specified by AppNameLength. If the length is greater than or equal to MBIM_MAXLENGTH_APPNAME bytes, this field contains the first MBIM_MAXLENGTH_APPNAME - 1 bytes of the name. The string is always null-terminated.
16	4	AppNameLength	SIZE (0..256)	The length, in bytes, of the application name. AppNameLength may contain a number equal to or greater than 256, but in these cases only the first 255 (MBIM_MAXLENGTH_APPNAME - 1) bytes are in the databuffer.
20	4	NumPinKeyRefs	SIZE (0..8)	The number of application PIN key references. In other words, the number of elements of PinKeyRef that are valid. Applications on a virtual R-UIM have no PIN key references.

Offset	Size	Field	Type	Description
24	4	KeyRefOffset	OFFSET	Offset of the PinKeyRef in the DataBuffer. The PinKeyRef is a byte array specifying the application's PIN key references for different levels of verification (keys for PIN1, PIN2, and possibly a universal PIN), as defined in Table 9.3 and Section 9.4.2 of the ETSI TS 102 221 technical specification. In the case of a single-verification card, or an MBB driver and/or modem that does not support different application keys for different applications, the first byte of the PinKeyRef field must be 0x01 (PIN1) and the second byte must be 0x81 (PIN2), as described in Section 9.5.1 of ETSI TS 102 221.
28	4	KeyRefSize	SIZE (0..8)	The size of PinKeyRef.
32		DataBuffer	DATABUFFER	The data buffer containing AppId, AppName, and PinKeyRef.of a single-verification card, or an MBB driver and/or modem that does not support different application keys for different applications, this field must be **0x01**.

MBIM_UICC_APP_TYPE

Type	Value	Description
MBIMUiccAppTypeUnknown	0	Unknown type.
MBIMUiccAppTypeMf	1	Legacy SIM directories rooted at the MF.
MBIMUiccAppTypeMfSIM	2	Legacy SIM directories rooted at the DF_GSM.
MBIMUiccAppTypeMfRUIM	3	Legacy SIM directories rooted at the DF_CDMA.
MBIMUiccAppTypeUSIM	4	USIM application.
MBIMUiccAppTypeCSIM	5	CSIM applicaton.
MBIMUiccAppTypeISIM	6	ISIM application.

Constants

The following constants are defined for MBIM_CID_MS_UICC_APP_INFO.

`const int MBIM_MAXLENGTH_APPID = 32`

`const int MBIM_MAXLENGTH_APPNAME = 256`

`const int MBIM_MAXNUM_PINREF = 8`

Unsolicited Events

Not applicable.

Status Codes

The following status codes are applicable:

Status code	Description
MBIM_STATUS_SUCCESS	Basic MBIM status as defined for all commands.
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_NO_DEVICE_SUPPORT	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_NOT_INITIALIZED	Unable to perform the UICC operation because the UICC is not yet fully initialized.

MBIM_CID_MS_UICC_FILE_STATUS

This CID retrieves information about a specified UICC file.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	MBIM_UICC_FILE_PATH	Not applicable
Response	Not applicable	MBIM_UICC_FILE_STATUS	Not applicable

Query

The InformationBuffer of MBIM_COMMAND_MSG contains the target EF as an MBIM_UICC_FILE_PATH structure.

MBIM_UICC_FILE_PATH (version 1)

Offset	Size	Field	Type	Description

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of the structure that follows. This field must be **1** for version 1 of this structure.
4	4	AppIdOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the application ID.
8	4	AppIdSize	SIZE (0..16)	The size of the application ID, in bytes, as defined in Section 8.3 of the ETSI TS 102 221 technical specification. For 2G cards, this field must be set to zero (0).
12	4	FilePathOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the file path. The file path is an array of 16-bit file IDs. The first ID must be either **0x7FFF** or **0x3F00**. If the first ID is **0x7FFF**, then the path is relative to the ADF of the application desginated by **AppId**. Otherwise, it is an absolute path starting from the MF.
16	4	FilePathSize	SIZE (0..8)	The size of the file path, in bytes.
20		DataBuffer	DATABUFFER	The data buffer containing AppId and FilePath.

Set

Not applicable.

Response

The following MBIM_UICC_FILE_STATUS structure is used in the InformationBuffer.

MBIM_UICC_FILE_STATUS (version 1)

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of the structure that follows. This field must be **1** for version 1 of this structure.
4	4	StatusWord1	UINT32(0..256)	A return parameter specific to the UICC command.
8	4	StatusWord2	UINT32(0..256)	A return parameter specific to the UICC command.
12	4	FileAccessibility	MBIM_UICC_FILE_ACCESSIBILITY	The UICC file accessibility.
16	4	FileType	MBIM_UICC_FILE_TYPE	The UICC file type.
20	4	FileStructure	MBIM_UICC_FILE_STRUCTURE	The UICC file structure.

Offset	Size	Field	Type	Description
24	4	ItemCount	UINT32	The number of items in the UICC file. For transparent and TLV files, this is set to **1**.
28	4	Size	UINT32	The size of each item, in bytes. For transparent or TLV files, this is the size of the entire EF. For record-based files, this represents the total number of records.
32	16	FileLockStatus	MBIM_PIN_TYPE_EX[4]	An array of type MBIM_PIN_TYPE_EX that describes the access condition for each operation (READ, UPDATE, ACTIVATE, and DEACTIVATE in that order) on that file.

MBIM_UICC_FILE_ACCESSIBILITY

The MBIM_UICC_FILE_ACCESSIBILITY enumeration is used in the preceding MBIM_UICC_FILE_STATUS structure.

Type	Value	Description
MBIMUiccFileAccessibilityUnknown	0	File shareability unknown.
MBIMUiccFileAccessibilityNotShareable	1	Not shareable file.
MBIMUiccFileAccessibilityShareable	2	Shareable file.

MBIM_UICC_FILE_TYPE

The MBIM_UICC_FILE_TYPE enumeration is used in the preceding MBIM_UICC_FILE_STATUS structure.

Type	Value	Description
MBIMUiccFileTypeUnknown	0	File type unknown.
MBIMUiccFileTypeWorkingEf	1	Working EF.
MBIMUiccFileTypeInternalEf	2	Internal EF.
MBIMUiccFileTypeDfOrAdf	3	Dedicated file, a directory that is the parent of other nodes. This may be a DF or ADF.

MBIM_UICC_FILE_STRUCTURE

The MBIM_UICC_FILE_STRUCTURE enumeration is used in the preceding MBIM_UICC_FILE_STATUS structure.

Type	Value	Description
MBIMUiccFileStructureUnknown	0	An unknown file structure.
MBIMUiccFileStructureTransparent	1	A single record of variable length.
MBIMUiccFileStructureCyclic	2	A cyclic set of records, each of the same length.
MBIMUiccFileStructureLinear	3	A linear set of records, each of the same length.
MBIMUiccFileStructureBerTLV	4	A set of data values accessible by tag.

MBIM_PIN_TYPE_EX

The MBIM_PIN_TYPE_EX enumeration is used in the preceding MBIM_UICC_FILE_STATUS structure.

Type	Value	Description
MBIMPinTypeNone	0	No PIN is pending to be entered.
MBIMPinTypeCustom	1	The PIN type is a custom type and is none of the other PIN types listed in this enumeration.
MBIMPinTypePin1	2	The PIN1 key.
MBIMPinTypePin2	3	The PIN2 key.
MBIMPinTypeDeviceSimPin	4	The device to SIM key.
MBIMPinTypeDeviceFirstSimPin	5	The device to very first SIM key.
MBIMPinTypeNetworkPin	6	The network personalization key.
MBIMPinTypeNetworkSubsetPin	7	The network subset personalization key.
MBIMPinTypeServiceProviderPin	8	The service provider (SP) personalization key.
MBIMPinTypeCorporatePin	9	The corporate personalization key.
MBIMPinTypeSubsidyLock	10	The subsidy unlock key.
MBIMPinTypePuk1	11	The Personal Identification Number 1 Unlock Key (PUK1).
MBIMPinTypePuk2	12	The Personal Identification Number 2 Unlock Key (PUK2).
MBIMPinTypeDeviceFirstSimPuk	13	The device to very first SIM PIN unlock key.
MBIMPinTypeNetworkPuk	14	The network personalization unlock key.
MBIMPinTypeNetworkSubsetPuk	15	The network subset personalization unlock key.
MBIMPinTypeServiceProviderPuk	16	The service provider (SP) personalization unlock key.

Type	Value	Description
MBIMPinTypeCorporatePuk	17	The corporate personalization unlock key.
MBIMPinTypeNev	18	The NEV key.
MBIMPinTypeAdm	19	The administrative key.

Unsolicited Events

Not applicable.

Status Codes

The following status codes are applicable:

Status code	Description
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_SHAREABILITY_CONDITION_ERROR	The file cannot be selected because it is not shareable and is currently being accessed by another application. The status word returned by the SIM is 6985.

MBIM_CID_MS_UICC_ACCESS_BINARY

This CID sends a specific command to access a UICC binary file, with structure type **MBIMUiccFileStructureTransparent** or **MBIMUiccFileStructureBerTLV**.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	MBIM_UICC_ACCESS_BINARY	Not applicable
Response	Not applicable	MBIM_UICC_RESPONSE	Not applicable

Query

Reads a binary file. The InformationBuffer for MBIM_COMMAND_MSG contains an MBIM_UICC_ACCESS_BINARY structure. An MBIM_UICC_RESPONSE structure is returned in the InformationBuffer of MBIM_COMMAND_DONE.

MBIM_UICC_ACCESS_BINARY (version 1)

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of the structure that follows. This field must be set to **1** for version 1 of this structure.
4	4	AppIdOffset	OFFSET	The offset, in bytes, from the beginning of this structure to the buffer containing the application ID.
8	4	AppIdSize	SIZE (0..16)	The size of the application ID, in bytes, as defined in Section 8.3 of the ETSI TS 102 221 technical specification. For 2G cards, this field must be set to zero (0).
12	4	FilePathOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the file path. The file path is an array of 16-bit file IDs. The first ID must be either **0x7FFF** or **0x3F00**. If the first ID is **0x7FFF**, then the path is relative to the ADF of the application desginated by **AppId**. Otherwise, it is an absolute path starting from the MF.
16	4	FilePathSize	SIZE	The size of the file path, in bytes.
20	4	FileOffset	UINT32	The offset to be used when reading from the file. This field can be bigger than 256, and it combines both offset high and offset low as defined in the ETSI TS 102 221 technical specification.
24	4	NumberOfBytes	UINT32	The number of bytes to be read. For example, a client driver could use this function to read a transparent (binary) file that is larger than 256 bytes, although the maximum amount that can be read or written in a single UICC operation is 256 bytes per the ETSI TS 102 221 technical specification. It is the function's responsibility to split this into multiple APDUs and send back the result in a single response.
28	4	LocalPinOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the password. This is the local PIN (PIN2) and is used in case the operation requires local PIN validation.
32	4	LocalPinSize	SIZE (0..16)	The size of the password, in bytes.
36	4	BinaryDataOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the command-specific data. Binary data is only used for SET operations.

Offset	Size	Field	Type	Description
40	4	BinaryDataSize	SIZE (0..32768)	The size of the data, in bytes.
44		DataBuffer	DATABUFFER	The data buffer containing AppId, FilePath, LocalPin, and BinaryData.

Set

Not applicable.

Response

The following MBIM_UICC_RESPONSE structure is used in the InformationBuffer.

MBIM_UICC_RESPONSE (version 1)

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of the structurethat follows. This field must be **1** for version 1 of this structure.
4	4	StatusWord1	UINT32(0..256)	A return parameter specific to the UICC command.
8	4	StatusWord2	UINT32(0..256)	A return parameter specific to the UICC command.
12	4	ResponseDataOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the response data. The response data is only used for QUERY operations.
16	4	ResponseDataSize	SIZE (0..32768)	The size of the data, in bytes.
20		DataBuffer	DATABUFFER	The data buffer containing ResponseData.

Unsolicited Events

Not applicable.

Status Codes

The following status codes are applicable:

Status code	Description
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.

Status code	Description
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_SHAREABILITY_CONDITION_ERROR	The file cannot be selected because it is not shareable and is currently being accessed by another application. The status word returned by the SIM is 6985.
MBIM_STATUS_PIN_FAILURE	The operation failed due to a PIN error.

MBIM_CID_MS_UICC_ACCESS_RECORD

This CID sends a specific command to access a UICC linear fixed or cyclic file, with structure type of **MBIMUiccFileStructureCyclic** or **MBIMUIccFileStructureLinear**.

Parameters

Operation	Set	Query	Notification
Command	Not applicable	MBIM_UICC_ACCESS_RECORD	Not applicable
Response	Not applicable	MBIM_UICC_RESPONSE	Not applicable

Query

Reads contents of a record. The InformationBuffer for MBIM_COMMAND_MSG contains the following MBIM_UICC_ACCESS_RECORD structure. MBIM_UICC_RESPONSE is returned in the InformationBuffer of MBIM_COMMAND_DONE.

MBIM_UICC_ACCESS_RECORD (version 1)

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of the structure that follows. This field must be set to **1** for version 1 of this structure.
4	4	AppIdOffset	OFFSET	The offset, in bytes, from the beginning of this structure to the buffer containing the application ID.
8	4	AppIdSize	SIZE (0..16)	The size of the application ID, in bytes, as defined in Section 8.3 of the ETSI TS 102 221 technical specification. For 2G cards, this field must be set to zero (0).

Offset	Size	Field	Type	Description
12	4	FilePathOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the file path. The file path is an array of 16-bit file IDs. The first ID must be either **0x7FFF** or **0x3F00**. If the first ID is **0x7FFF**, then the path is relative to the ADF of the application desginated by **AppId**. Otherwise, it is an absolute path starting from the MF.
16	4	FilePathSize	SIZE	The size of the file path, in bytes.
20	4	RecordNumber	UINT32(0..256)	The record number. This represents the absolute record index at all times. Relative record access is not supported because the modem can perform multiple accesses on a file (NEXT, PREVIOUS).
24	4	LocalPinOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the password. The lock password is a null-terminated UTF-8 string of decimal digits.
28	4	LocalPinSize	SIZE (0..16)	The size of the password, in bytes.
32	4	RecordDataOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the command-specific data. Record data is only used for SET operations.
36	4	RecordDataSize	SIZE (0..256)	The size of the data, in bytes.
40		DataBuffer	DATABUFFER	The data buffer containing AppId, FilePath, LocalPin, and RecordData.

Set

Not applicable.

Response

An MBIM_UICC_RESPONSE structure is used in the InformationBuffer.

Unsolicited Events

Not applicable.

Status Codes

The following status codes are applicable:

Status code	Description
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_SHAREABILITY_CONDITION_ERROR	The file cannot be selected because it is not shareable and is currently being accessed by another application. The status word returned by the SIM is 6985.
MBIM_STATUS_PIN_FAILURE	The operation failed due to a PIN error.

MBIM_CID_MS_PIN_EX

This CID is used to perform all PIN security operations as defined in Section 9 of the ETSI TS 102 221 technical specification. The CID is similar to MBIM_CID_MS_PIN, but extends it to support multi-app UICC cards. Only single-verification-capable UICCs are supported. Multi-verification-capable UICCs that support more than one application PIN are not supported. One application PIN (PIN1) is assigned to all ADFs/DFs and files on the UICC. However, each application can specify a local PIN (PIN2) as a level 2 user verification requirement, resulting in the need for additional validation for every access command. This scenario is what MBIM_CID_MS_PIN_EX supports.

Just like MBIM_CID_MS_PIN, with MBIM_CID_MS_PIN_EX the device only reports one PIN at a time. If multiple PINs are enabled and reporting multiple PINs is also enabled, then functions must report PIN1 first. For example, if subsidy lock reporting is enabled and the SIM's PIN1 is enabled, then the subsidy lock PIN should be reported in a subsequent query request only after PIN1 has been successfully verified. An empty PIN is permitted together with MBIMPinOperationEnter. An empty PIN is specified by setting the PinSize to zero. In this case, a SET command is similar to a QUERY and returns the state of the PIN referenced. This is fully aligned to the behavior of the VERIFY command as specified in Section 11.1.9 of the ETSI TS 102 221 technical specification.

Parameters

Operation	Set	Query	Notification
Command	MBIM_SET_PIN_EX	MBIM_PIN_APP	Not applicable
Response	MBIM_PIN_INFO_EX	MBIM_PIN_INFO_EX	Not applicable

Query

The following MBIM_PIN_APP structure is used in the InformationBuffer.

MBIM_PIN_APP (version 1)

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of the structure that follows. This field must be set to **1** for version 1 of this structure.
4	4	AppIdOffset	OFFSET	The offset, in bytes, from the beginning of this structure to the buffer containing the application ID.
8	4	AppIdSize	SIZE (0..16)	The size of the application ID, in bytes, as defined in Section 8.3 of the ETSI TS 102 221 technical specification. For 2G cards, this field must be set to zero (0).
12		DataBuffer	DATABUFFER	The AppId as defined in the ETSI TS 102 221 technical specification.

Set

The following MBIM_SET_PIN_EX structure is used in the InformationBuffer.

MBIM_SET_PIN_EX

Offset	Size	Field	Type	Description
0	4	PinType	MBIM_PIN_TYPE_EX	The PIN type. See the MBIM_PIN_TYPE_EX table in this topic.
4	4	PinOperation	MBIM_PIN_OPERATION	The PIN operation. See MBIM 1.0.
8	4	PinOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to a string PIN that represents the PIN value with which to perform the action, or the PIN value required to enable or disable PIN settings. This field applies for all values of **PinOperation**.
12	4	PinSize	SIZE (0..32)	The size, in bytes, used for the PIN.
16	4	NewPinOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the **NewPin** string that represents the new PIN value to set when **PinOperation** is MBIMPinOperationChange or MBIMPinOperationEnter, for PinTypeMBIMPinTypePuk1 or PinTypeMBIMPinTypePuk2.
20	4	NewPinSize	SIZE (0..32)	The size, in bytes, used for the NewPin.

Offset	Size	Field	Type	Description
24	4	AppIdOffset	OFFSET	The offset, in bytes, calculated from the beginning of this structure to the buffer containing the application ID.
28	4	AppIdSize	SIZE (0..16)	The size of the application ID, in bytes, as defined in Section 8.3 of the ETSI TS 102 221 technical specification. For 2G cards, this field must be set to zero (0).
32		DataBuffer	DATABUFFER	The data buffer containing the Pin, NewPin, and AppId.

Response

The following MBIM_PIN_INFO_EX structure is used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	PinType	MBIM_PIN_TYPE_EX	The PIN type. See the MBIM_PIN_TYPE_EX table in this topic.
4	4	PinState	MBIM_PIN_STATE	The PIN state. See MBIM 1.0.
8	4	RemainingAttempts	UINT32	The number of remaining attempts for any PIN-related operations such as enter, enable, or disable. Devices that do not support this information must set this member to 0xFFFFFFFF.

Unsolicited Events

Not applicable.

Status Codes

The following status codes are applicable:

Status code	Description
MBIM_STATUS_BUSY	Basic MBIM status as defined for all commands.
MBIM_STATUS_FAILURE	Basic MBIM status as defined for all commands.
MBIM_STATUS_SIM_NOT_INSERTED	Unable to perform the UICC operation because the UICC is missing.
MBIM_STATUS_BAD_SIM	Unable to perform the UICC operation because the UICC is in an error state.
MBIM_STATUS_PIN_DISABLED	The operation failed because the PIN is disabled.

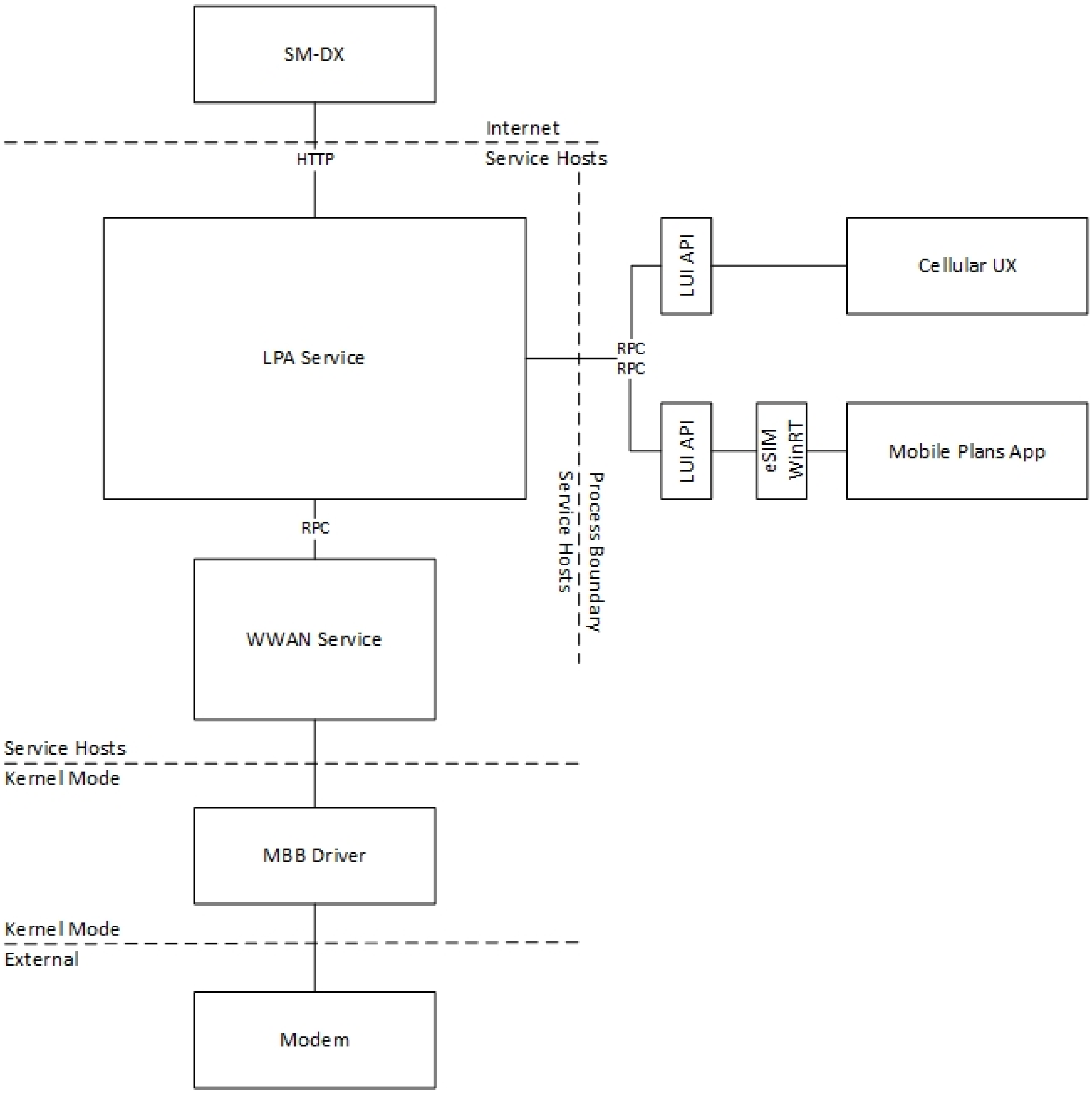
Status code	Description
MBIM_STATUS_PIN_REQUIRED	The operation failed because a PIN must be entered to proceed.
MBIM_STATUS_NO_DEVICE_SUPPORT	The operation failed because a SET on a corresponding PIN type is not supported by the device.

MB eSIM Operations

Article • 03/14/2023

eSIM Architecture

The Local Profile Assitant (LPA) component of the Windows operating system is the LPA Service while the low level UICC access is exposed through the WWAN Service. The LPA Service handles profile discovery, downloading profiles, and profile management.

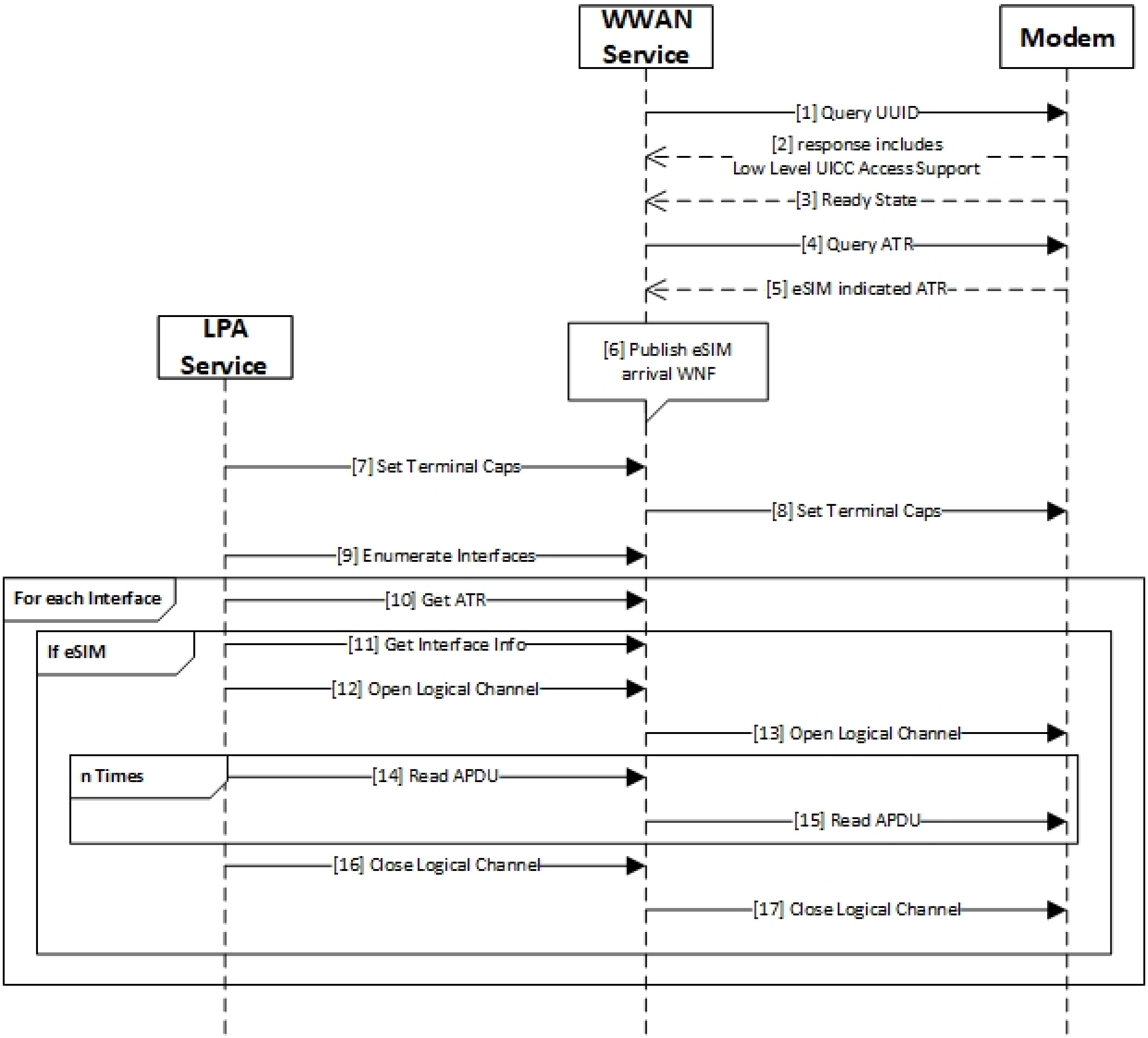


MB Interface Update for eSIM Operations

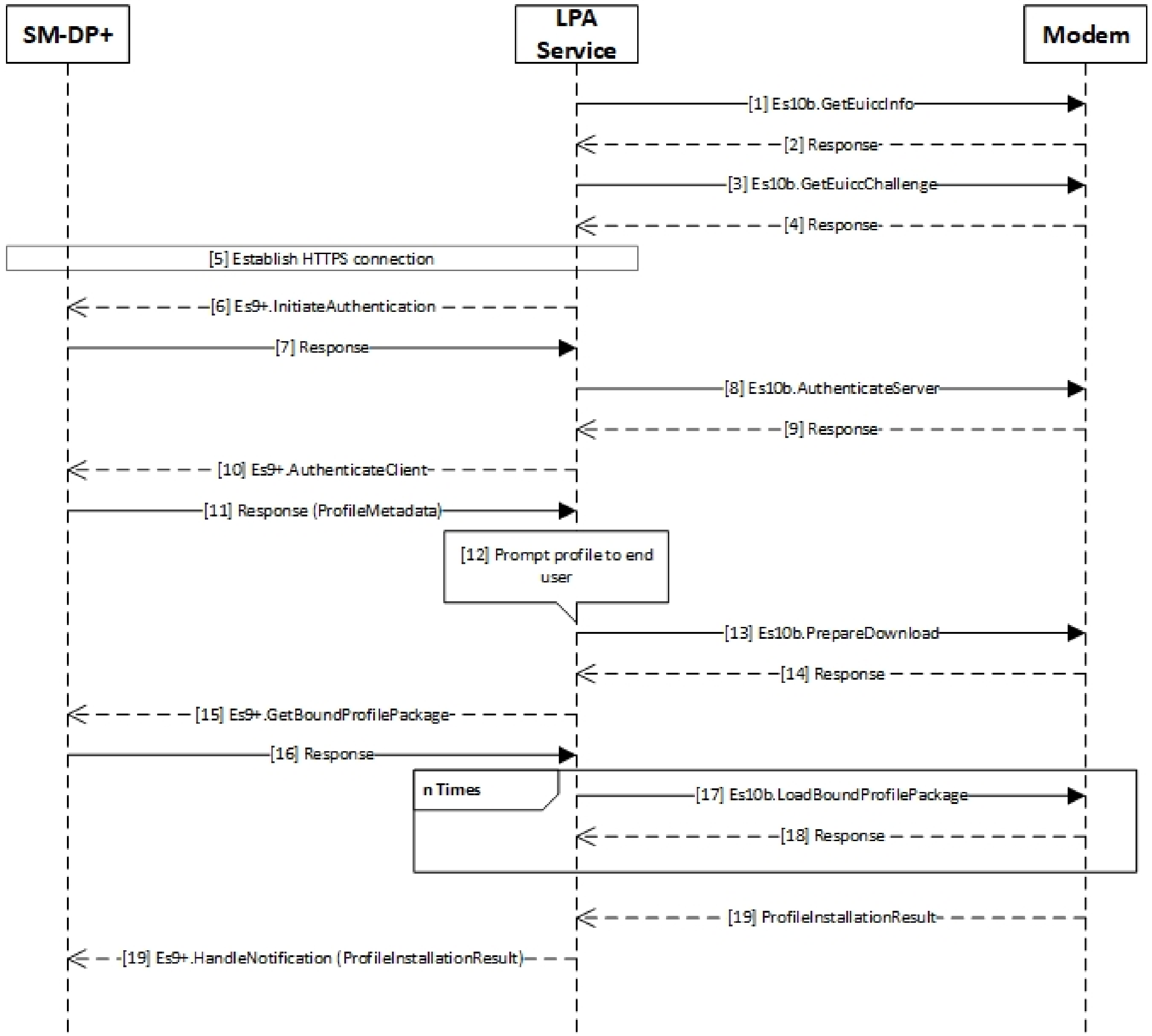
The modem needs to support the following MB Low Level UICC access CIDs for eSIM functionality.

```
MBIM_CID_MS_UICC_ATR
MBIM_CID_MS_UICC_OPEN_CHANNEL
MBIM_CID_MS_UICC_CLOSE_CHANNEL
MBIM_CID_MS_UICC_APDU
MBIM_CID_MS_UICC_TERMINAL_CAPABILITY
MBIM_CID_MS_UICC_RESET
```

eSIM Service Initialization



eSIM Profile Download and Install

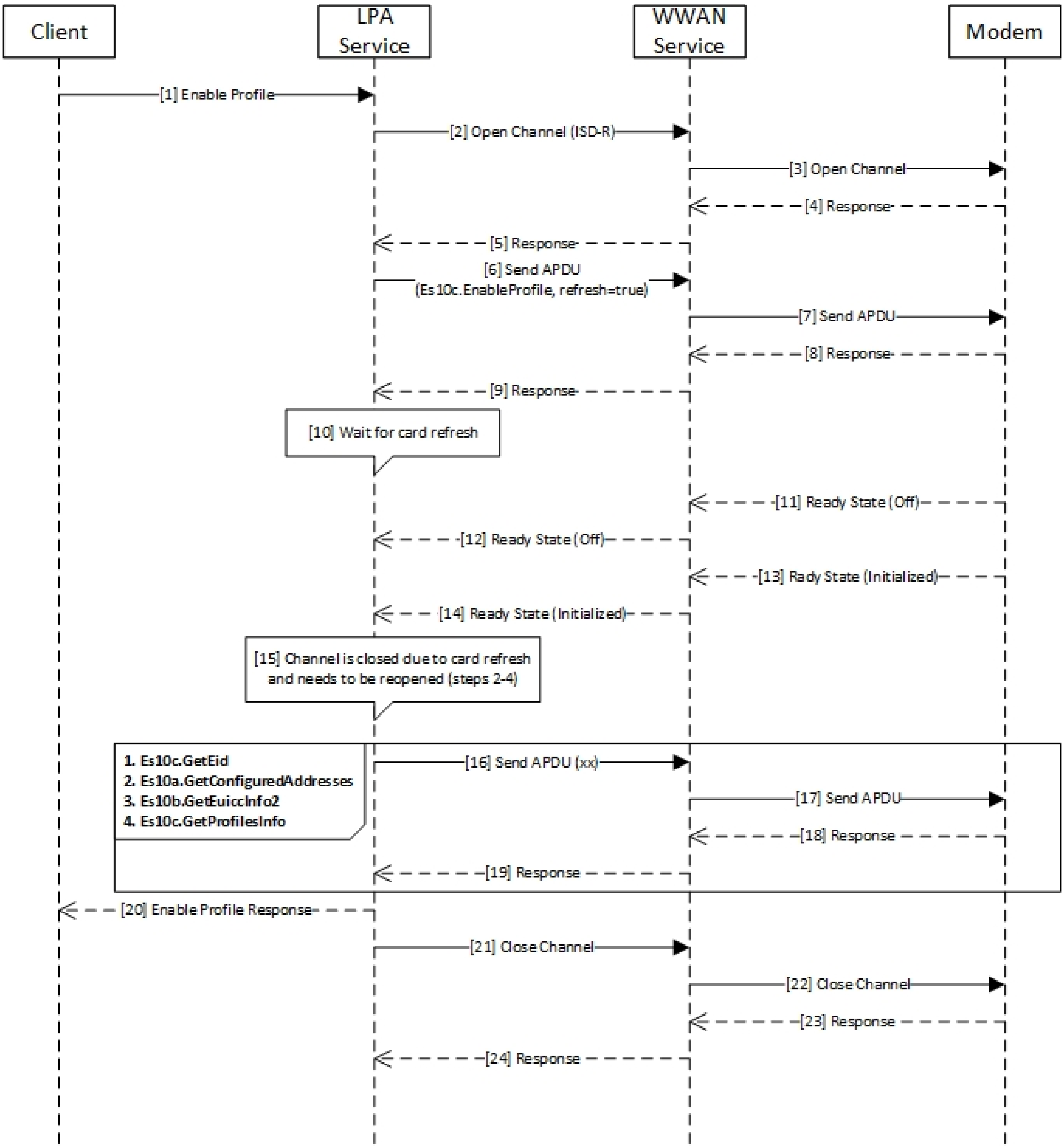

SM-DP+
LPA Service
Modem
[1] Es10b.GetEuiccInfo
[2] Response
[3] Es10b.GetEuiccChallenge
[4] Response
[5] Establish HTTPS connection
[6] Es9+.InitiateAuthentication
[7] Response
[8] Es10b.AuthenticateServer
[9] Response
[10] Es9+.AuthenticateClient
[11] Response (ProfileMetadata)
[12] Prompt profile to end user
[13] Es10b.PrepareDownload
[14] Response
[15] Es9+.GetBoundProfilePackage
[16] Response
n Times
[17] Es10b.LoadBoundProfilePackage
[18] Response
[19] ProfileInstallationResult
[19] Es9+.HandleNotification (ProfileInstallationResult)

eSIM Profile Operations

eSIM Profile Operations include:

```
Enable Profile
Disable Profile
Delete Profile
Wipe eSIM
Update NickName
```

Below is a sample flow for the Enable Profile operation. The other Profile Operations follow a similar flow except that MBIM_CID_MS_UICC_APDU will contain the Es10c command for the respective operation.



Card Refresh

The eSIM Profile Operations expect that card refresh will be performed followed by the ready state change according to the MB eSIM MBIM ready state guidance.

Hardware Lab Kit (HLK) Tests

The following tests in the HLK can be used to verify eSIM functionality:

TestLowLevelUiccAccess

TestReadyInfo

TestResetPassthrough

Via netsh, we can run the **TestLowLevelUiccAccess** HLK testlist. For more information on using the netsh tool, see **netsh-mbn** and **netsh-mbn-test-installation**.

```
netsh mbn test feature=esim testpath="C:\data\test\bin"
taefpath="C:\data\test\bin" param="AccessString=internet"
```

The file showing the HLK test results should have been generated in the directory that the 'netsh mbn test' command was ran from: `TestLowLevelUiccAccess.htm`.

Manual Tests

Test eSIM profile management

1. Required resources: An eSIM and an activation code or a QR code for the profile download. The profile needs a confirmation code to download.
2. Open Settings->Network & Internet -> Cellular.
3. Click ***Manage eSIM profiles***.
4. Click ***Add a new profile***.
5. Choose ***Let me enter an activation code I have from my mobile operator*** and click ***Next***.
6. Scan the QR code or type in the activation code.
7. Wait until the confirmation page shows up. Fill in the confirmation code and click ***Next***.
8. Dismiss the dialog and then enable the profile.
9. Change the name of the profile. The new name should be displayed for the profile.
10. Delete the profile.
11. Install the profile again (repeating steps 4-7), but this time with the wrong confirmation code. For example, when scanning a QR code the code hash displays for a second before contacting the server. Very quickly click anywhere inside of it and hit the back key to delete a single character before it talks to the server.
12. You should see an error message and eventually land at the confirmation page again. Type in the correct confirmation code this time and click ***Next***.
13. Dismiss the dialog and delete the profile.

Test autoconnect after reboot

1. Make sure Ethernet is unplugged and Wi-Fi is toggled off. With a known good eSIM profile present on the physical eSIM in the device, browse to Settings -> Network & Internet -> Cellular -> Manage eSIM profiles -> eSIM profiles, select the profile, and click ***Use***. Verify that browsing the internet works normally.
2. Reboot the machine, login, and browse to Settings -> Network & Internet -> Cellular -> Manage eSIM profiles -> eSIM profiles. The profile should show as Active, systray should show Cellular connection bars, and browsing the internet should work normally.
3. Back in Manage eSIM profiles, select the profile, and click ***Stop using***. The profile should disconnect data.

MB eSIM Troubleshooting Guide

Follow this guide to debug eSIM issues.

Profile Download and Install Failures

1. Collect and decode the logs using the instructions in MB Collecting Logs
2. Open the .txt file generated in TextAnalysisTool
3. Load the eSIM Download and Install Filter

Here is sample success log for Profile Download and Install:

```
 37 24515    None    2020-03-04T08:54:48.6922406    0.0000922    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::OnLuiRpcRegisterForLpaNotifications,ulTransactionId=1
 47 25637    None    2020-03-04T08:54:48.7058023    0.0000081    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::OnLuiRpcRegisterForEsimNotifications,ulTransactionId=2
 48 25638    None    2020-03-04T08:54:48.7058116    0.0000093    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::OnLuiRpcRegisterForAllProfileNotifications,ulTransactionId=3
 87 42955    None    2020-03-04T08:54:50.8459033    0.0000357    8820
3524    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::OnLuiRpcRegisterForLpaNotifications,ulTransactionId=1
 96 43009    None    2020-03-04T08:54:50.8470189    0.0000401    8820
3524    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
```

```
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::OnLuiRpcRegisterForEsimNotifications,ulTransactionId=2
104 43039    None    2020-03-04T08:54:50.8473061    0.0000092    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::OnLuiRpcRegisterForAllProfileNotifications,ulTransactionId=3
110 43856    None    2020-03-04T08:55:10.1453397    19.2978242    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    RpcDownloadProfile:
Component=LuiApiServer,esimId=89033023********************06,hr=0,Location
=LuiApiProcessActivationCode,luicontext=2,strActivation=1$*****************
*************,ulTransactionId=4
113 43861    None    2020-03-04T08:55:10.1459912    0.0000161    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnInitiateCommonMutualAuthenticat
ion,fsmevent=3,Location=LPA::Lpd::DownloadInstance::InitiateCommonMutualAuth
entication,task=2,taskId=6
115 43863    None    2020-03-04T08:55:10.1461554    0.0000022    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=ProcessActivation,fsmevent=1,Loca
tion=LPA::Lpd::DownloadInstance::ProcessActivationCode,task=3,taskId=6
130 44745    None    2020-03-04T08:55:10.1901431    0.0000464    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bGetEuiccChallenge,fsmevent
=4,Location=LPA::Lpd::DownloadInstance::OnEs10bGetChallengeComplete,task=2,t
askId=6
134 45589    None    2020-03-04T08:55:10.2226549    0.0000204    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bGetUiccInfo1,fsmevent=5,Lo
cation=LPA::Lpd::DownloadInstance::OnEs10bGetUiccInfo1Complete,task=2,taskId
=6
146 46509    None    2020-03-04T08:55:11.7773024    0.0002176    8820
3524    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs9OrEs11InitiateAuthentication
,fsmevent=6,Location=LPA::Lpd::DownloadInstance::OnEs9OrEs11InitiateAuthenti
cation,task=2,taskId=6
153 50421    None    2020-03-04T08:55:12.9320399    0.0008653    8820
3524    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bAuthenticateServer,fsmeven
t=7,Location=LPA::Lpd::DownloadInstance::OnEs10bAuthenticateServerComplete,t
ask=2,taskId=6
164 51032    None    2020-03-04T08:55:13.3763368    0.0001906    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs9AuthenticateClient,fsmevent=
8,Location=LPA::Lpd::DownloadInstance::OnEs9AuthenticateClient,task=2,taskId
=6
176 51183    None    2020-03-04T08:55:14.9352658    0.0000603    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnInstallProfile,fsmevent=11,Loca
tion=LPA::Lpd::DownloadInstance::DownloadAndInstallProfile,task=4,taskId=6
190 54213    None    2020-03-04T08:55:18.1904783    0.0000651    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bPrepareDownload,fsmevent=1
4,Location=LPA::Lpd::DownloadInstance::OnEs10bPrepareDownloadComplete,task=4
```

```
,taskId=11
197 54784    None    2020-03-04T08:55:19.5200686    0.0013163    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs9BoundProfileDownloaded,fsmev
ent=15,Location=LPA::Lpd::DownloadInstance::OnEs9BoundProfileDownloaded,task
=4,taskId=11
407 257461    None    2020-03-04T08:55:36.3645723    0.0001640    8820
7812    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bLoadBoundProfilePackage,fs
mevent=16,Location=LPA::Lpd::DownloadInstance::OnEs10bLoadBoundProfilePackag
eComplete,task=4,taskId=11
412 258044    None    2020-03-04T08:55:36.6932923    0.3206194    8820
7812    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs9HandleNotification,fsmevent=
17,Location=LPA::Lpd::DownloadInstance::OnEs9HandleNotification,task=4,taskI
d=11
416 258628    None    2020-03-04T08:55:37.6234007    0.0000315    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bRemoveNotificationFromList
Complete,fsmevent=18,Location=LPA::Lpd::DownloadInstance::OnEs10bRemoveNotif
icationFromListComplete,task=4,taskId=11
426 258790    None    2020-03-04T08:55:37.6239355    0.0000168    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::CompleteLuiRpcOperation,ulTransactionId=4
449 261202    None    2020-03-04T08:55:37.6883598    0.0000121    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::CompleteLuiRpcOperation,ulTransactionId=5
509 326884    None    2020-03-04T08:55:45.5722501    0.0000155    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::CompleteLuiRpcOperation,ulTransactionId=6
522 329627    None    2020-03-04T08:55:45.8306288    0.0000257    8820
1044    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bGetUiccInfo1,fsmevent=5,Lo
cation=LPA::Lpd::DownloadInstance::OnEs10bGetUiccInfo1Complete,task=2,taskId
=27
524 330152    None    2020-03-04T08:55:46.6963292    0.8655163    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs9HandleNotification,fsmevent=
17,Location=LPA::Lpd::DownloadInstance::OnEs9HandleNotification,task=4,taskI
d=27
528 330865    None    2020-03-04T08:55:46.7211677    0.0000375    8820
10356    Microsoft_Windows_Cellcore_LPA_Service    DownloadSequenceEvent:
Component=LpaServiceLpd,error=0,eventlabel=OnEs10bRemoveNotificationFromList
Complete,fsmevent=18,Location=LPA::Lpd::DownloadInstance::OnEs10bRemoveNotif
icationFromListComplete,task=4,taskId=27
```

Profile Operation Failures

1. Collect and decode the logs using the instructions in MB Collecting Logs
2. Open the .txt file generated in TextAnalysisTool
3. Load the eSIM Profile Operations Filter

Here is sample success log for the Enable Profile Operation:

```
 2 39    None    2020-03-04T09:06:12.6782819        11720    2720
Microsoft_Windows_Cellcore_LPA_Service    RpcEnableProfile:
Component=LuiApiServer,esimId=89033023**********************06,hr=0,Location
=LuiApiEnableProfile,luicontext=2,profileId=8935401************6,ulTransacti
onId=5
12 209    None    2020-03-04T09:06:12.6937921    0.0152614    11720    2720
Microsoft_Windows_Cellcore_LPA_Service    OpenChannel:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::OpenChannelOperation::Execute
17 1003    None    2020-03-04T09:06:12.7248172    0.0309320    11720    2720
Microsoft_Windows_Cellcore_LPA_Service    ChannelOpened:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::OnChannelOperationOpenChannelComplete
,ulChannel=1
18 1343    None    2020-03-04T09:06:12.7271141    0.0022969    11720    2720
Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,20,191,49,17,160,12,90,10,152,83,4,1,0,0,0,0,85,101,129,1,255
,0],vCommandApdu.data().Count=26
19 2067    None    2020-03-04T09:06:12.7489842    0.0218701    11720    2720
Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=[191,49,3,128,1,0],pApduCompleteInfo-
>ApduInfo.Response.Count=6
20 2445    None    2020-03-04T09:06:12.7505212    0.0015370    11720    2720
Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,16,191,45,13,92,11,90,79,159,112,144,145,146,147,148,149,153,
0],vCommandApdu.data().Count=22
23 4341    None    2020-03-04T09:06:12.7651848    0.0000718    11720    2720
Microsoft_Windows_Cellcore_LPA_Service    WwapiEsimUpdate:
Component=LpaServiceEsimManager,fIsEsimInterface=True,fIsInterfaceRemoved=Fa
lse,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::EsimManager::ProcessWwapiSimUpdate,readyState=0
24 6343    None    2020-03-04T09:06:12.7870313    0.0218465    11720    4992
Microsoft_Windows_Cellcore_LPA_Service    WwapiAsyncResponseFailure:
Component=LpaApduHelper,hr=-1073479677,Location=LPA::ApduHelper::ApduOperati
on::OnWwapiSendApduComplete
27 11067    None    2020-03-04T09:06:13.1843659    0.0000080    11720
2720    Microsoft_Windows_Cellcore_LPA_Service    WwapiEsimUpdate:
```

```
Component=LpaServiceEsimManager,fIsEsimInterface=True,fIsInterfaceRemoved=Fa
lse,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::EsimManager::ProcessWwapiSimUpdate,readyState=6
28 11070    None    2020-03-04T09:06:13.1843928    0.0000269    11720
2720    Microsoft_Windows_Cellcore_LPA_Service    CardResetComplete:
Component=LpaServiceEsimManager,count=0,guidInterface=a549349a-2a86-4703-
bebe-
6f0d034f0ff3,hr=0,Location=LPA::EsimManager::ProcessWwapiSimUpdate,midoperat
ion=True,readyState=6
29 13752    None    2020-03-04T09:06:13.2040341    0.0196413    11720
2720    Microsoft_Windows_Cellcore_LPA_Service    OpenChannel:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::OpenChannelOperation::Execute
31 17057    None    2020-03-04T09:06:13.2232087    0.0133736    11720
2720    Microsoft_Windows_Cellcore_LPA_Service    WwapiEsimUpdate:
Component=LpaServiceEsimManager,fIsEsimInterface=True,fIsInterfaceRemoved=Fa
lse,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::EsimManager::ProcessWwapiSimUpdate,readyState=6
33 33469    None    2020-03-04T09:06:13.3307557    0.0000034    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    WwapiEsimUpdate:
Component=LpaServiceEsimManager,fIsEsimInterface=True,fIsInterfaceRemoved=Fa
lse,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::EsimManager::ProcessWwapiSimUpdate,readyState=6
34 36738    None    2020-03-04T09:06:13.3658938    0.0351381    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    ChannelOpened:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::OnChannelOperationOpenChannelComplete
,ulChannel=1
35 37169    None    2020-03-04T09:06:13.3699437    0.0040499    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,6,191,62,3,92,1,90,0],vCommandApdu.data().Count=12
36 41856    None    2020-03-04T09:06:13.4055017    0.0355580    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=
[191,62,18,90,16,137,3,48,35,66,67,32,0,0,0,0,4,86,35,54,6],pApduCompleteInf
o->ApduInfo.Response.Count=21
37 42027    None    2020-03-04T09:06:13.4073896    0.0018879    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,3,191,60,0,0],vCommandApdu.data().Count=9
38 44071    None    2020-03-04T09:06:13.4249377    0.0175481    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=
[191,60,17,129,15,108,112,97,46,100,115,46,103,115,109,97,46,99,111,109],pAp
duCompleteInfo->ApduInfo.Response.Count=20
39 44341    None    2020-03-04T09:06:13.4353474    0.0104097    11720
```

```
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,3,191,34,0,0],vCommandApdu.data().Count=9
40 45644    None    2020-03-04T09:06:13.4574746    0.0221272    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=
[191,34,129,134,129,3,2,1,0,130,3,2,2,0,131,3,3,2,0,132,15,129,1,0,130,4,0,1
3,186,0,131,4,0,0,105,233,133,4,5,127,50,224,134,3,9,2,0,135,3,2,3,0,136,2,4
,208,169,22,4,20,129,55,15,81,37,208,177,212,8,212,195,178,50,230,210,94,121
,91,235,251,170,22,4,20,129,55,15,81,37,208,177,212,8,212,195,178,50,230,210
,94,121,91,235,251,139,1,2,153,2,6,64,4,3,0,0,2,12,20,71,79,45,80,65,45,48,5
2,49,57,32,32,32,32,32,32,32,32,32,32],pApduCompleteInfo-
>ApduInfo.Response.Count=138
41 45763    None    2020-03-04T09:06:13.4628539    0.0053793    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,16,191,45,13,92,11,90,79,159,112,144,145,146,147,148,149,153,
0],vCommandApdu.data().Count=22
42 47274    None    2020-03-04T09:06:13.5033394    0.0404855    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=
[191,45,130,1,143,160,130,1,139,227,73,90,10,152,16,66,7,0,0,0,0,0,249,79,16
,160,0,0,5,89,16,16,255,255,255,255,137,0,0,16,0,159,112,1,0,145,9,83,80,71,
116,111,69,122,67,120,146,23,69,90,32,67,111,110,110,101,99,116,32,80,114,11
1,118,105,115,105,111,110,105,110,103,149,1,1,227,78,90,10,152,16,0,0,0,0,0,
0,0,6,79,16,160,0,0,5,89,16,16,255,255,255,255,137,0,160,0,0,159,112,1,0,145
,9,77,105,99,114,111,115,111,102,116,146,28,78,101,116,119,111,114,107,32,83
,105,109,117,108,97,116,111,114,32,45,32,77,105,108,101,110,97,103,101,149,1
,0,227,76,90,10,152,16,0,0,0,0,0,0,0,135,79,16,160,0,0,5,89,16,16,255,255,25
5,255,137,0,160,1,0,159,112,1,0,145,9,77,105,99,114,111,115,111,102,116,146,
26,78,101,116,119,111,114,107,32,83,105,109,117,108,97,116,111,114,32,45,32,
88,79,82,32,54,52,149,1,0,227,77,90,10,152,16,0,0,0,0,0,0,0,104,79,16,160,0,
0,5,89,16,16,255,255,255,255,137,0,160,2,0,159,112,1,0,145,9,77,105,99,114,1
11,115,111,102,116,146,27,78,101,116,119,111,114,107,32,83,105,109,117,108,9
7,116,111,114,32,45,32,88,79,82,32,49,50,56,149,1,0,227,81,90,10,152,83,4,1,
0,0,0,0,85,101,79,16,160,0,0,5,89,16,16,255,255,255,255,137,0,0,17,0,159,112
,1,1,145,4,101,67,84,67,146,36,101,67,84,67,32,80,114,111,102,105,108,101,32
,102,111,114,32,68,117,109,109,121,32,83,117,98,115,99,114,105,112,116,105,1
11,110,115,149,1,2],pApduCompleteInfo->ApduInfo.Response.Count=404
57 47329    None    2020-03-04T09:06:13.5038992    0.0000179    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    LuiAsyncResult:
Component=LpaServiceLui,error=Dynamic:
baseType=LPA_ERROR_DETAILS,dwParams=0,error=0,violation=0,hrResult=0,Locatio
n=LPA::Lui::CompleteLuiRpcOperation,ulTransactionId=5
58 47419    None    2020-03-04T09:06:13.5057078    0.0018086    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
```

```
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,3,191,43,0,0],vCommandApdu.data().Count=9
74 50078    None    2020-03-04T09:06:13.7457319    0.2395537    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=
[191,43,130,5,15,160,130,5,11,48,130,5,7,191,47,63,128,1,53,129,2,6,64,12,42
,101,99,116,99,45,108,105,118,101,108,97,98,46,112,114,111,100,46,111,110,10
0,101,109,97,110,100,99,111,110,110,101,99,116,105,118,105,116,121,46,99,111
,109,90,10,152,83,4,1,0,0,0,0,85,101,95,55,64,42,56,96,255,201,144,175,106,1
13,19,175,113,18,210,3,109,79,148,57,237,59,174,37,216,33,105,134,155,220,17
2,95,224,174,24,219,51,85,61,97,207,2,255,147,95,163,16,82,210,218,197,4,207
,189,214,187,212,199,133,250,29,52,5,42,72,48,130,1,221,48,130,1,132,160,3,2
,1,2,2,17,0,128,80,86,187,39,45,44,80,66,120,232,145,219,50,27,96,48,10,6,8,
42,134,72,206,61,4,3,2,48,50,49,19,48,17,6,3,85,4,10,12,10,71,69,77,65,76,84
,79,32,83,65,49,27,48,25,6,3,85,4,3,12,18,71,69,77,65,76,84,79,32,69,85,77,3
2,67,69,32,80,65,85,48,32,23,13,49,57,48,51,50,53,48,48,48,48,48,48,90,24,15
,57,57,57,57,49,50,51,49,50,51,53,57,53,57,90,48,64,49,19,48,17,6,3,85,4,10,
12,10,71,69,77,65,76,84,79,32,83,65,49,41,48,39,6,3,85,4,5,19,32,56,57,48,51
,51,48,50,51,52,50,52,51,50,48,48,48,48,48,48,48,48,48,48,52,53,54,50,51,51,
54,48,54,48,89,48,19,6,7,42,134,72,206,61,2,1,6,8,42,134,72,206,61,3,1,7,3,6
6,0,4,220,59,239,236,155,254,120,18,18,152,140,158,193,12,233,47,93,241,4,16
4,84,133,39,169,12,11,226,18,112,94,39,144,174,227,237,242,37,73,25,64,210,4
,184,60,17,114,103,226,235,246,229,229,226,54,4,208,245,174,180,198,239,7,98
,176,163,107,48,105,48,31,6,3,85,29,35,4,24,48,22,128,20,220,222,26,105,31,2
33,175,117,221,161,187,146,153,49,139,41,177,212,22,178,48,29,6,3,85,29,14,4
,22,4,20,200,220,224,111,237,248,116,35,250,198,112,246,190,164,149,129,214,
2,198,159,48,14,6,3,85,29,15,1,1,255,4,4,3,2,0,128,48,23,6,3,85,29,32,1,1,25
5,4,13,48,11,48,9,6,7,103,129,18,1,2,1,1,48,10,6,8,42,134,72,206,61,4,3,2,3,
71,0,48,68,2,32,63,48,8,73,147,118,41,144,127,5,77,200,133,217,208,20,34,133
,6,19,182,144,58,15,70,229,67,72,111,167,73,240,2,32,37,234,11,51,98,141,182
,118,16,21,38,167,115,14,72,111,47,143,45,213,157,184,236,156,11,34,63,177,2
07,73,42,251,48,130,2,157,48,130,2,67,160,3,2,1,2,2,16,38,116,211,243,157,55
,39,121,8,126,27,89,53,236,251,82,48,10,6,8,42,134,72,206,61,4,3,2,48,68,49,
24,48,22,6,3,85,4,10,19,15,71,83,77,32,65,115,115,111,99,105,97,116,105,111,
110,49,40,48,38,6,3,85,4,3,19,31,71,83,77,32,65,115,115,111,99,105,97,116,10
5,111,110,32,45,32,82,83,80,50,32,82,111,111,116,32,67,73,49,48,30,23,13,49,
55,48,53,50,53,48,48,48,48,48,48,90,23,13,52,55,48,53,50,52,50,51,53,57,53,5
7,90,48,50,49,19,48,17,6,3,85,4,10,12,10,71,69,77,65,76,84,79,32,83,65,49,27
,48,25,6,3,85,4,3,12,18,71,69,77,65,76,84,79,32,69,85,77,32,67,69,32,80,65,8
5,48,89,48,19,6,7,42,134,72,206,61,2,1,6,8,42,134,72,206,61,3,1,7,3,66,0,4,1
78,83,182,218,149,17,229,225,138,112,30,182,68,55,158,224,64,192,174,31,121,
186,38,129,24,66,117,75,251,252,97,44,56,126,103,229,223,178,243,184,125,24,
62,148,245,200,21,56,225,114,221,60,21,125,154,123,136,205,30,47,187,198,82,
182,163,130,1,39,48,130,1,35,48,18,6,3,85,29,19,1,1,255,4,8,48,6,1,1,255,2,1
,0,48,23,6,3,85,29,32,1,1,255,4,13,48,11,48,9,6,7,103,129,18,1,2,1,2,48,77,6
,3,85,29,31,4,70,48,68,48,66,160,64,160,62,134,60,104,116,116,112,58,47,47,1
03,115,109,97,45,99,114,108,46,115,121,109,97,117,116,104,46,99,111,109,47,1
11,102,102,108,105,110,101,99,97,47,103,115,109,97,45,114,115,112,50,45,114,
111,111,116,45,99,105,49,46,99,114,108,48,14,6,3,85,29,15,1,1,255,4,4,3,2,1,
6,48,60,6,3,85,29,30,1,1,255,4,50,48,48,160,46,48,44,164,42,48,40,49,19,48,1
7,6,3,85,4,10,12,10,71,69,77,65,76,84,79,32,83,65,49,17,48,15,6,3,85,4,5,19,
8,56,57,48,51,51,48,50,51,48,23,6,3,85,29,17,4,16,48,14,136,12,43,6,1,4,1,12
9,248,2,135,106,4,3,48,29,6,3,85,29,14,4,22,4,20,220,222,26,105,31,233,175,1
17,221,161,187,146,153,49,139,41,177,212,22,178,48,31,6,3,85,29,35,4,24,48,2
2,128,20,129,55,15,81,37,208,177,212,8,212,195,178,50,230,210,94,121,91,235,
251,48,10,6,8,42,134,72,206,61,4,3,2,3,72,0,48,69,2,32,66,41,14,233,224,150,
13,19,243,91,225,69,204,238,251,34,51,242,209,112,206,71,158,159,79,106,47,2
10,84,210,156,187,2,33,0,205,238,69,205,202,163,47,35,70,41,24,250,125,165,2
33,16,22,196,115,13,165,10,153,178,254,108,220,88,138,37,17,163],pApduComple
teInfo->ApduInfo.Response.Count=1300
76 50290    None    2020-03-04T09:06:13.7503562    0.0032197    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,3,191,32,0,0],vCommandApdu.data().Count=9
77 51346    None    2020-03-04T09:06:13.7754287    0.0250725    11720
4992    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=
[191,32,53,130,3,2,2,0,169,22,4,20,129,55,15,81,37,208,177,212,8,212,195,178
,50,230,210,94,121,91,235,251,170,22,4,20,129,55,15,81,37,208,177,212,8,212,
195,178,50,230,210,94,121,91,235,251],pApduCompleteInfo-
>ApduInfo.Response.Count=56
82 51796    None    2020-03-04T09:06:14.9911183    0.0020754    11720
7520    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,6,191,48,3,128,1,53,0],vCommandApdu.data().Count=12
83 52474    None    2020-03-04T09:06:15.0374406    0.0463223    11720
7520    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=[191,48,3,128,1,0],pApduCompleteInfo-
>ApduInfo.Response.Count=6
86 52736    None    2020-03-04T09:06:15.0451263    0.0076479    11720
7520    Microsoft_Windows_Cellcore_LPA_Service    SendApdu:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::SendApdu,m_ulChannel=1
,vCommandApdu.data()=
[129,226,145,0,3,191,43,0,0],vCommandApdu.data().Count=9
87 53507    None    2020-03-04T09:06:15.0622867    0.0171604    11720
7520    Microsoft_Windows_Cellcore_LPA_Service    SendApduResponse:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::ApduOperation::OnWwapiSendApduComplet
e,pApduCompleteInfo->ApduInfo.Response=[191,43,2,160,0],pApduCompleteInfo-
>ApduInfo.Response.Count=5
90 54169    None    2020-03-04T09:06:45.0792254    0.0166565    11720
2720    Microsoft_Windows_Cellcore_LPA_Service    CloseChannel:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
6f0d034f0ff3,Location=LPA::ApduHelper::CloseChannelOperation::Execute,ulChan
nel=1
91 54435    None    2020-03-04T09:06:45.0965610    0.0173356    11720
2720    Microsoft_Windows_Cellcore_LPA_Service    ChannelClosed:
Component=LpaApduHelper,guidInterface=a549349a-2a86-4703-bebe-
```

```
6f0d034f0ff3,Location=LPA::ApduHelper::OnChannelOperationCloseChannelComplet
e
```

See Also

MB eSIM MBIM Ready-State Guidance

Firmware Upgrade for eSIM

Article • 03/14/2023

Overview

This document describes the Windows OS changes that support firmware updates for eSIM devices.

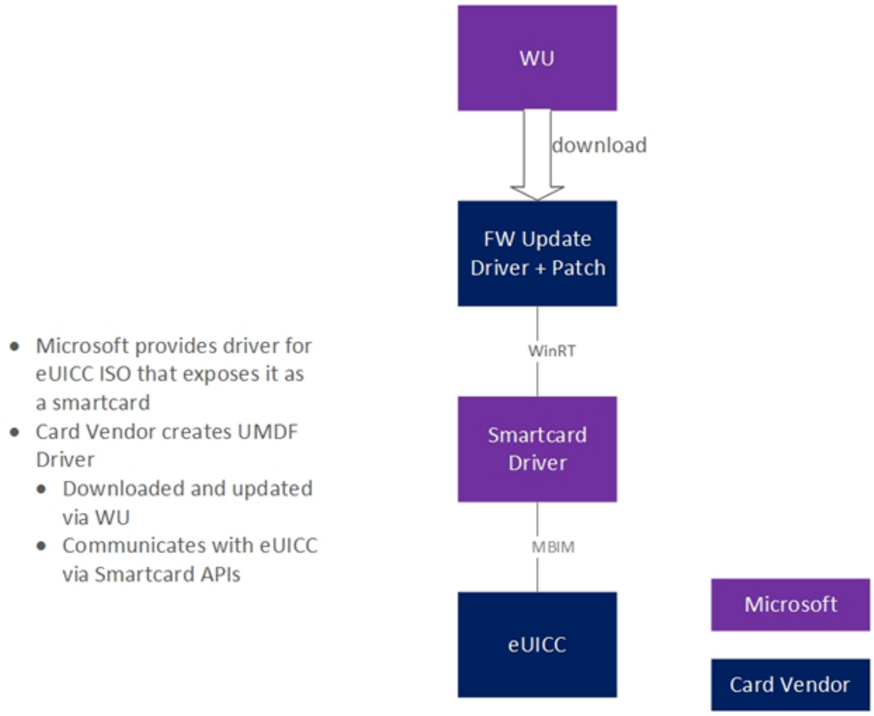
Windows Update (WU) is the model used for the firmware patches. In this model, the eSIM device vendor authors a UMDF driver and adds it to a WU package along with the firmware patch. The package is published to the WU and will be downloaded & installed on Windows devices containing the card vendor's eSIM device. Once installed the card vendor's driver writes the firmware patch using the Smart Card WinRT API. Microsoft enables this by providing a UMDF driver for the ISO interface of the eSIM and exposing it as a smart card via the existing Smart Card WinRT API.

The modem ISO interface has a 255-byte limitation as the max Application Protocol Data Unit (APDU) size and is therefore too slow for full firmware OS updates (TRC/PBL image). For full firmware OS updates, Microsoft provides a separate smart card UMDF driver which uses the SPB interface of the eSIM as the transport.

The following acronyms are used in this section:

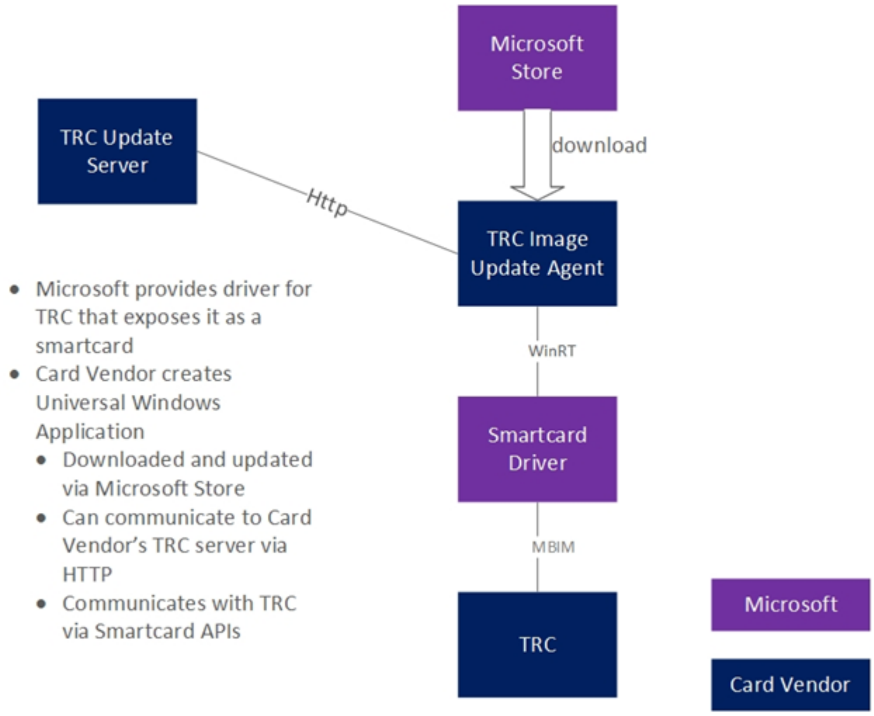
- **ACPI:** Advanced Configuration and Power Interface
- **APDU:** Application Protocol Data Unit
- **ATR:** Answer to Reset
- **eUICC:** Embedded Universal Integrated Circuit Card
- **FW:** Firmware
- **HCP:** Host Controller Protocol
- **HWID:** Hardware ID
- **ISO:** International Organization for Standardization
- **MF:** Master File
- **MSFT:** Microsoft
- **OEM:** Original Equipment Manufacturer
- **PBL:** Primary bootloader
- **RPC:** Remote Procedure Call
- **ScardSvr:** Smart Cards for Windows Service
- **Scard WinRT:** Smart Card Windows Runtime
- **SC DDI:** Smart Card DDI
- **sHDLC:** Simplified High Level Data Link Control
- **SPB:** Simple Peripheral Bus
- **SPI:** Serial Peripheral Interface
- **TRC:** Tamper Resistant Chip
- **WwanSvc:** WWAN Service

High Level Design for Firmware Patch Update



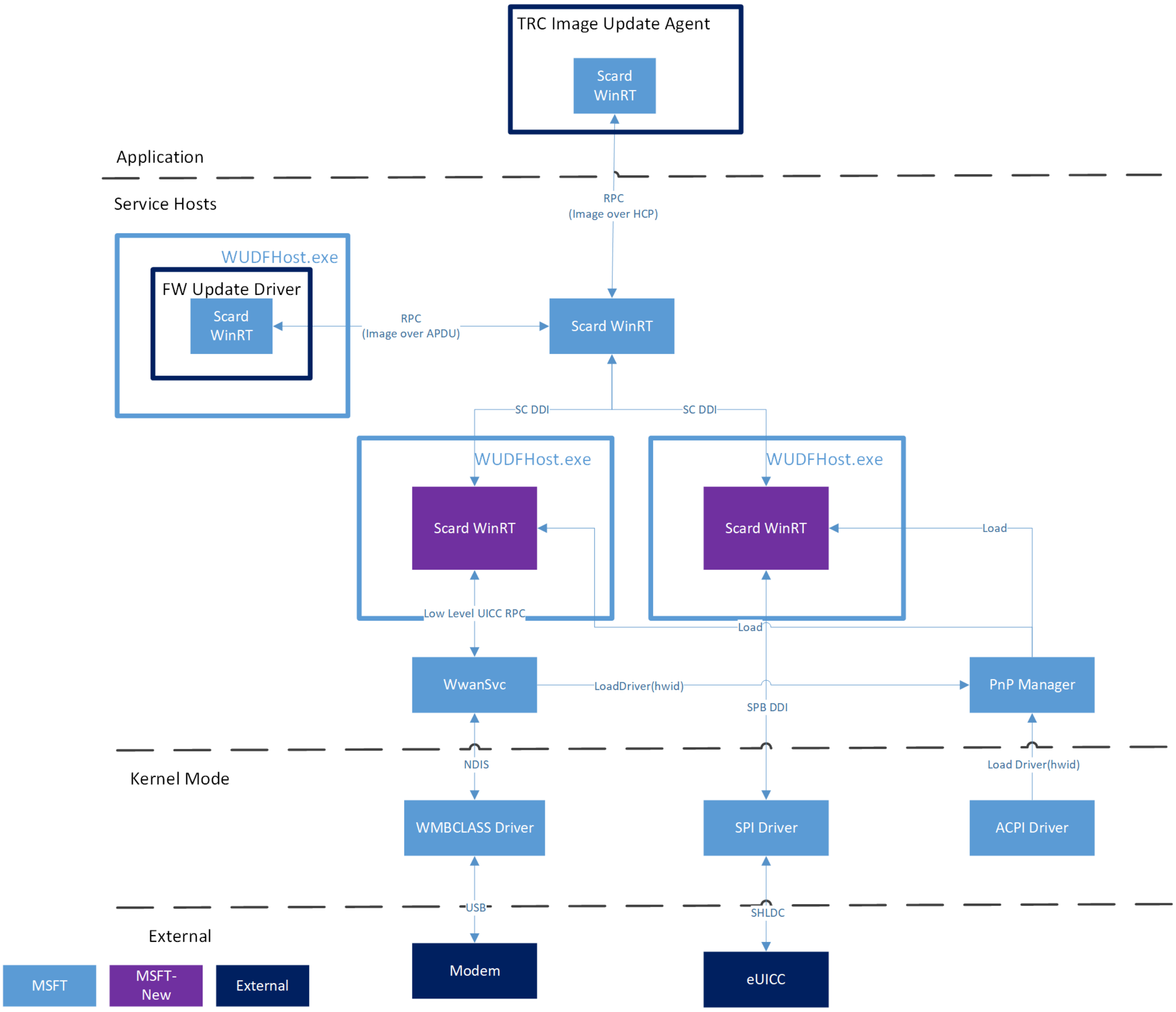
- Microsoft provides driver for eUICC ISO that exposes it as a smartcard
- Card Vendor creates UMDF Driver
  - Downloaded and updated via WU
  - Communicates with eUICC via Smartcard APIs

High Level Design for Firmware OS Update



- Microsoft provides driver for TRC that exposes it as a smartcard
- Card Vendor creates Universal Windows Application
  - Downloaded and updated via Microsoft Store
  - Can communicate to Card Vendor's TRC server via HTTP
  - Communicates with TRC via Smartcard APIs

Firmware Upgrade Architecture



For Windows 10, version 1703 the card vendor will deliver firmware updates on APDUs over the ISO interface. Updates for full images (PBL HCP) on HCP packets over the SPI interface are planned for the Windows 10, version 1709 time frame.

The ISO UMDF driver is loaded by the WWAN Service when the WWAN Service detects an eSIM based on the ATR information. The ISO UMDF driver sends APDUs to the eUICC via the modem using the low-level UICC access RPC of the WWAN Service.

The SPI UMDF driver is loaded by PnP based on the hardware ID in the ACPI entry for the eSIM card. The SPI UMDF driver sends sHDLC frames to the TRC on the card via the SPB IOCTL interface.

On the upper layer both drivers will implement the Smart Card DDI that provides low-level access for interacting with smart cards. This will expose both the ISO and SPI interfaces of the eSIM as a smart card via the Smart Card WinRT API.

In devices where the SPI interface to eSIM is available, OEMs are expected to add the Microsoft UICC SPI driver HWID to the ACPI table as a hardware compatible ID. The HWID for the Microsoft UICC SPI driver is ACPI\MSFTUICCSPB.

UMDF drivers

UMDF drivers will implement the following Smart Card IOCTLs:

Usage	DDI
Smart card states	IOCTL_SMARTCARD_GET_STATE IOCTL_SMARTCARD_IS_ABSENT IOCTL_SMARTCARD_IS_PRESENT IOCTL_SMARTCARD_POWER
Smart card attributes	IOCTL_SMARTCARD_GET_ATTRIBUTE IOCTL_SMARTCARD_SET_ATTRIBUTE

Usage	DDI
Smart card communication	IOCTL_SMARTCARD_SET_PROTOCOL IOCTL_SMARTCARD_TRANSMIT

Requirements for smart card devices

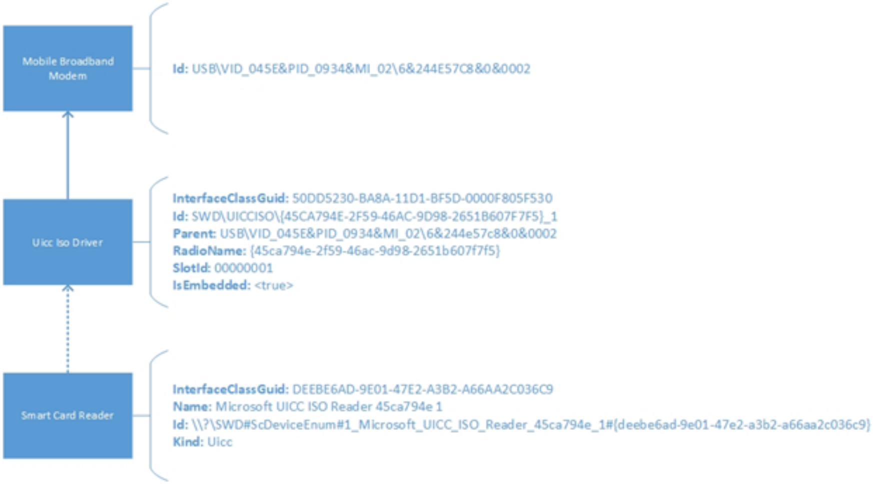
Define the following device properties:

Define	Name	Type	FormatID	Value
Device interface guid	System.Devices.InterfaceClassGuid -- PKEY_Devices_InterfaceClassGuid	Guid -- VT_CLSID	{026E516E-B814-414B-83CD-856D6FEF4822}, 4, DEVPROP_TYPE_GUID	{DEEBE6AD-9E01-47E2-A3B2-A66AA2C036C9}
ReaderKind	System.Devices.SmartCards.ReaderKind -- PKEY_Devices_SmartCards_ReaderKind	Byte -- VT_UI1 (should be INT16 Bug 9550228)	{D6B5B883-18BD-4B4D-B2EC-9E38AFFEDA82}, 2, DEVPROP_TYPE_BYTE	SmartCardReaderKind_Uicc
ReaderName	DEVPKEY_Device_ReaderName (0xD6B5B883, 0x18BD, 0x4B4D, 0xB2, 0xEC, 0x9E, 0x38, 0xAF, 0xFE, 0xDA, 0x82, 0x03)	String -- VT_LPWSTR (For variants: VT_BSTR)	{D6B5B883-18BD-4B4D-B2EC-9E38AFFEDA82}, 3, DEVPROP_TYPE_STRING	CustomName
AppAccessRestrictionsFlags	System.Devices.SmartCards.ReaderKind -- PKEY_Devices_SmartCards_AppAccessRestrictionsFlags	Byte -- VT_UI1	{D6B5B883-18BD-4B4D-B2EC-9E38AFFEDA82}, 4, DEVPROP_TYPE_BYTE	PrivilegedAppOnly (1)

The ISO UMDF driver sets additional custom dev properties on the Smart Card reader devnode:

Define	Name	Type	FormatID	Value
RadioName	DEVPKEY_MbbDevice_RadioName	: DEVPROP_TYPE_GUID	{41e061f2-9999-4b33-bf42-f950cbfd5f2e}, 1, DEVPROP_TYPE_GUID	RadioInterfaceGuid
SlotId	DEVPKEY_MbbDevice_SlotId	DEVPROP_TYPE_UINT32	{c4c66992-3bcc-4f96-9a85-bd807235fbe1}, 2, DEVPROP_TYPE_UINT32	SlotId
IsEmbedded	DEVPKEY_MbbDevice_IsEmbedded	DEVPROP_TYPE_BOOLEAN	{7d08a710-b448-4148-8049-0aa12e5fd2dd}, 3, DEVPROP_TYPE_BOOLEAN	IsEmbedded

These properties are used to uniquely name the smart card readers and identify the reader that is attached to the correct eSIM card. For example: IsEmbedded=True and SlotId=1.



Id: USB\VID_045E&PID_0934&MI_02\6&244E57C8&0&0002

InterfaceClassGuid: 50DD5230-BA8A-11D1-BF5D-0000F805F530
Id: SWD\UICCISO\{45CA794E-2F59-46AC-9D98-2651B607F7F5}_1
Parent: USB\VID_045E&PID_0934&MI_02\6&244e57c8&0&0002
RadioName: {45ca794e-2f59-46ac-9d98-2651b607f7f5}
SlotId: 00000001
IsEmbedded: <true>

InterfaceClassGuid: DEEBE6AD-9E01-47E2-A3B2-A66AA2C036C9
Name: Microsoft UICC ISO Reader 45ca794e 1
Id: \\?\SWD#ScDeviceEnum#1_Microsoft_UICC_ISO_Reader_45ca794e_1#{deebe6ad-9e01-47e2-a3b2-a66aa2c036c9}
Kind: Uicc

The TRC Image Update Agent is required to have sharedUserCertificates capability which allows access to the Smart Card WinRT API. The sharedUserCertificates capability is a restricted capability that is only given to enterprises with certain credentials. Once the access is granted, the app can connect to the TRC on the device via the Smart Card API and send commands to the card.

It is expected that the firmware patch is carried over APDUs. Because the Smart Card WinRT API only exposes transmitting APDUs and not channel management functions such as open/close, the UMDF driver will inspect the APDUs and look for a SELECT by AID command. If the driver finds a SELECT by AID command, it will be interpreted as opening a logical channel using the Wwan RPC API. The UMDF driver will always validate that the AID is allowlisted, otherwise it will deny the request. Because there is no close or disconnect IOCTL in the SC DDI, the UMDF driver has no way of knowing when the transmission ends and when to close the logical channel. To prevent leaking of the logical channels, the UMDF driver will set a 5 minute timer when a logical channel is opened and will close the channel when the timer expires. The 5 minutes should be long enough as a firmware update is expected to run up to 2.5 minutes maximum. If the UMDF driver detects a new SELECT by AID command, then it will close the previously opened channel and reset the timer for the new logical channel. Note that 5 minute timeout is measured from the last transmitted APDU. In other words, transmitting an APDU over an existing channel resets the timer.

To prevent from opening channels against random UICC apps, the ISO UMDF driver will permit app IDs that are required for an update and restrict access to only these apps. The card vendor helps identify the app IDs and the OEM adds the IDs as registry entries. It is also expected that the UICC app in the card will perform digital signature checks on the firmware to protect against malicious apps sending data.

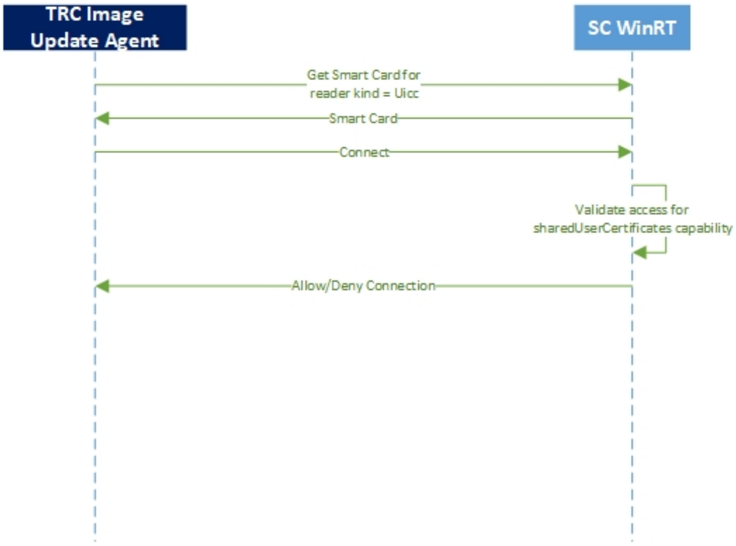
COSA Setting

`CellCore/PerDevice/eSIM/FwUpdate/AllowedAppIdList`

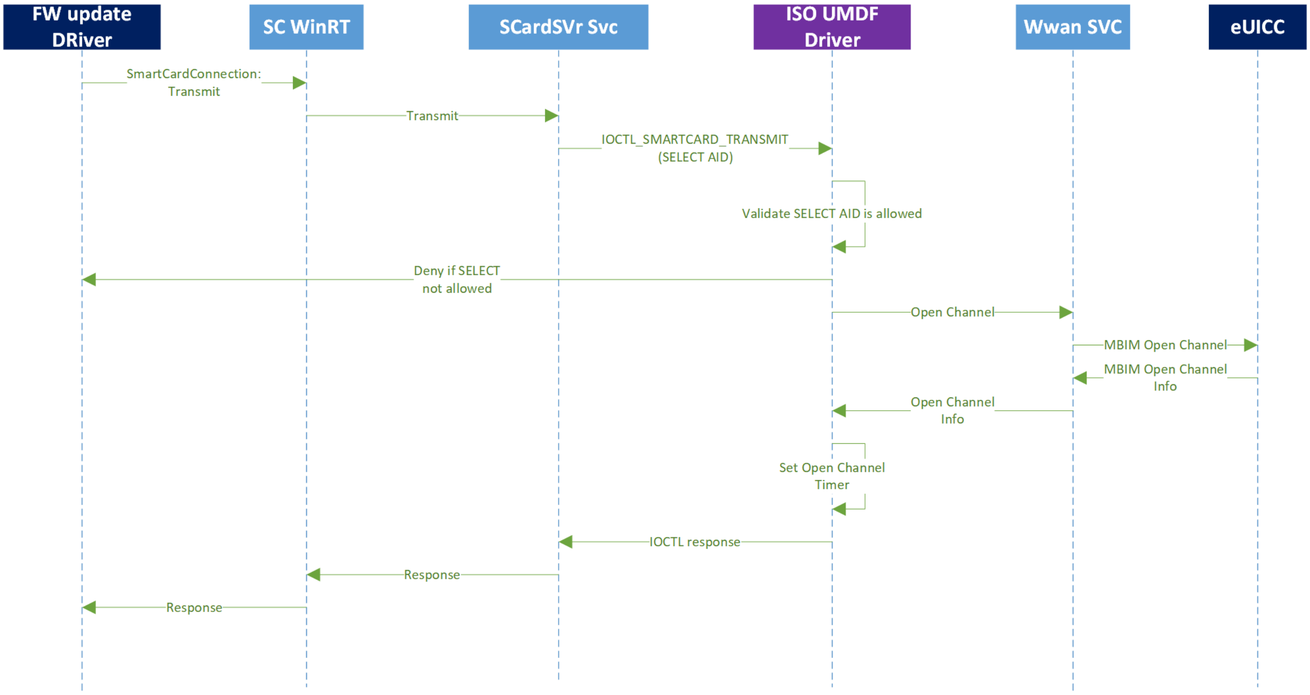
During a full firmware OS update it is important that the UICC apps that are involved are not accessed by the modem. To achieve this the TRC Image Update Agent will send a special APDU that instructs the eUICC to go in to the TRC/PBL mode. The TRC app will then ask the modem to go into the passthrough mode and reset the card. The card will boot as an empty MF. Once the update is done, the modem will be asked to reset the card again. This time both the modem and the card will get back to normal mode.

Flow Diagrams

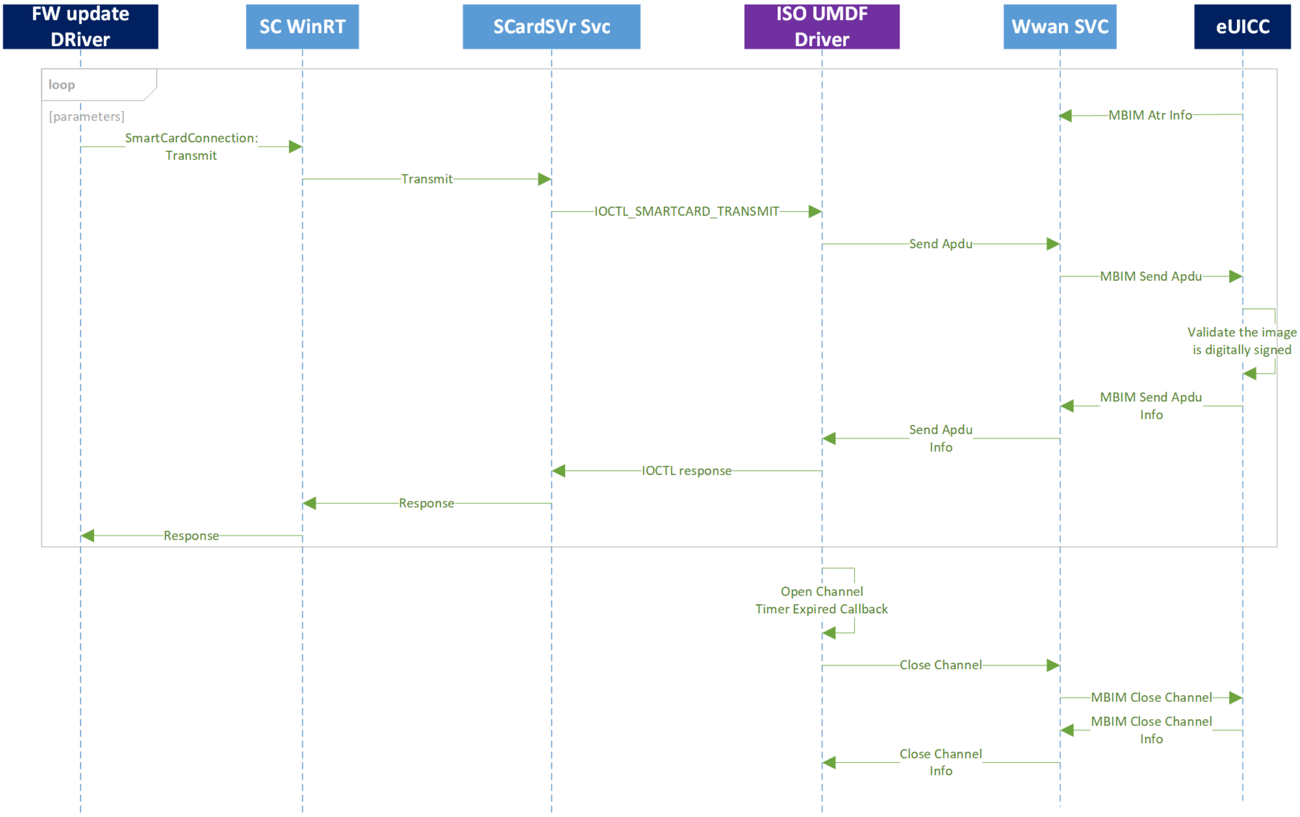
Connect


TRC Image Update Agent
SC WinRT
Get Smart Card for reader kind = Uicc
Smart Card
Connect
Validate access for sharedUserCertificates capability
Allow/Deny Connection

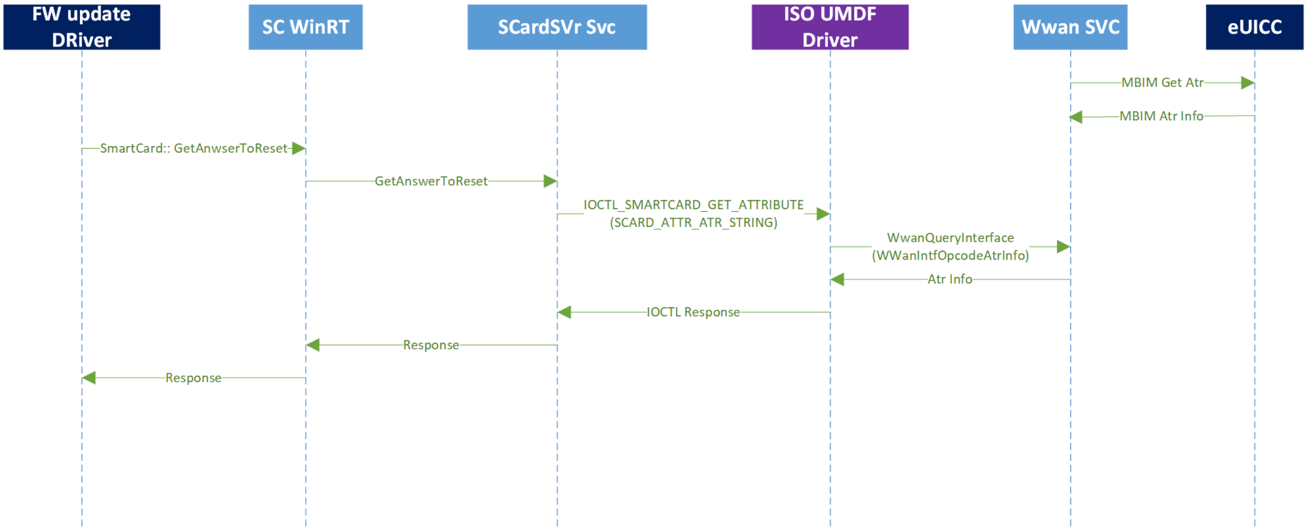
Transmit (Open Channel)



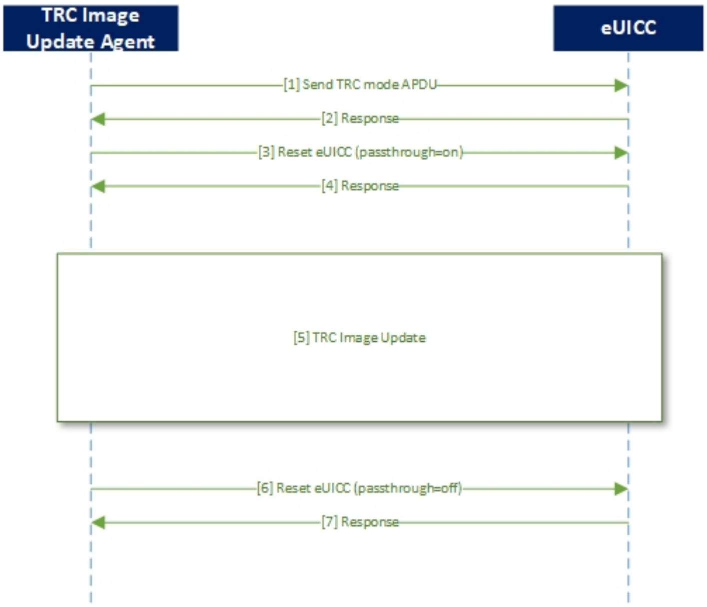
Transmit (send APDU and Close Channel)



Get ATR



Pass-through mode



Related

Preinstall Apps Using DISM Published

To add a preinstalled app to a desktop image

Preinstallable apps for mobile devices

Preinstall Task

MB modem reset operations

Article • 03/14/2023

This section defines MBIM CID commands and data structures, as well as NDIS OID commands and data structures, for resetting the modem in a mobile broadband (MB) device. These commands and data structures are available in Windows 10, version 1709 and later.

MBIM_CID_MS_DEVICE_RESET

The host sends MBIM_CID_MS_DEVICE_RESET to the MBIM function to reset the modem device.

Service name	UUID	UUID value
Microsoft Basic IP Connectivity Extensions	UUID_BASIC_CONNECT_EXTENSIONS	3d01dcc5-fef5-4d05-0d3a-bef7058e9aaf

CID	Command code	Set	Query	Notify
MBIM_CID_MS_DEVICE_RESET	10	Y	N	N

Parameters

Type	Set	Query	Notification
Command	Empty	Not applicable	Not applicable
Response	Empty	Not applicable	Not applicable

Query

Not applicable.

Set

The InformationBuffer shall be NULL and *InformationBufferLength* shall be zero.

Response

The InformationBuffer shall be NULL and *InformationBufferLength* shall be zero.

Notification

Not applicable.

Status codes

The following status codes are applicable. Status is returned as an asynchronous response to a set operation after reset is complete.

Status code	Description
MBIM_STATUS_SUCCESS	The operation succeeded.
MBIM_STATUS_BUSY	The device is busy.
MBIM_STATUS_FAILURE	The operation failed.
MBIM_STATUS_NO_DEVICE_SUPPORT	The device does not support this operation.

OID_WWAN_DEVICE_RESET

The NDIS equivalent for MBIM_CID_MS_DEVICE_RESET is OID_WWAN_DEVICE_RESET.

eSIM Download and Install Log Filter

Article • 12/15/2021

To make searching log files easier, below is an eSIM download and install filter file for the TextAnalysisTool .

To use the eSIM download and install log filter:

1. Copy and paste the lines below and save them into a text file named "esimdownload.tat."

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<TextAnalysisTool.NET version="2015-08-17" showOnlyFilteredLines="True">
  <filters>
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="RpcDownloadProfile" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="LuiAsyncResult" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="DownloadSequenceEvent" />
  </filters>
</TextAnalysisTool.NET>
```

2. Load the filter file into the TextAnalysisTool by clicking File > Load Filters.

eSIM Profile Operations Log Filter

Article • 12/15/2021

To make searching log files easier, below is an eSIM Profile Operation filter file for the TextAnalysisTool.

To use the eSIM Profile Operation log filter:

1. Copy and paste the lines below and save them into a text file named "esimoperation.tat." This filter is for the "EnableProfile" operation. For other operations, replace "RpcEnableProfile" with one of the strings below that is specific to the operation.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<TextAnalysisTool.NET version="2015-08-17" showOnlyFilteredLines="True">
  <filters>
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="RpcEnableProfile" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="OpenChannel" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="CloseChannel" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="CardResetComplete" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="LuiAsyncResult" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="WwapiEsimUpdate" />
    <filter enabled="y" excluding="n" description="" type="matches_text"
case_sensitive="n" regex="n" text="SendApdu" />
  </filters>
</TextAnalysisTool.NET>
```

```
Enable Profile  - RpcEnableProfile
Disable Profile - RpcDisableProfile
Delete Profile  - RpcDeleteProfile
Set Nick Name   - RpcSetProfileNickname
Reset eSim      - RpcWipeEsim
```

2. Load the filter file into the TextAnalysisTool by clicking File > Load Filters.

MB eSIM MBIM ready state guidance

Article • 03/14/2023

This topic provides guidance on the expected MBIM ready state for eSIM scenarios on Windows 10, version 1709 and later. Conforming to the correct ready state ensures that the OS handles all changes properly for that scenario.

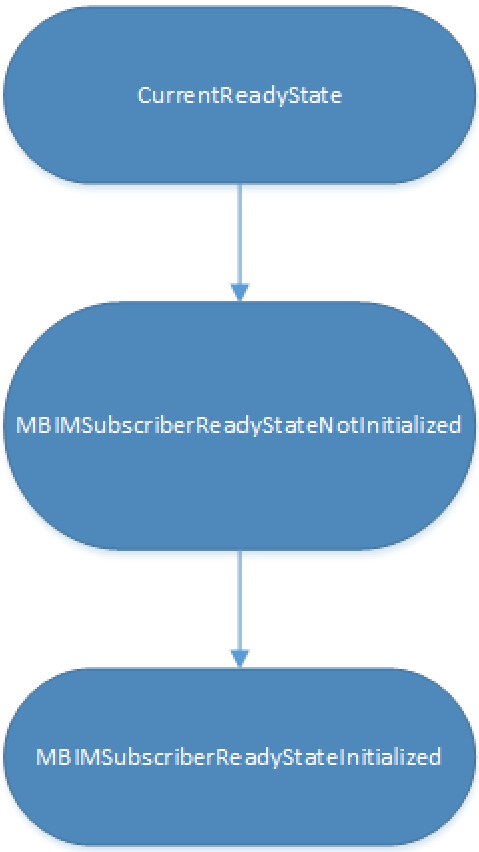
ⓘ Important

All scenarios and states in this topic assume that the eSIM capable card is mapped to the default executor, executor 0, and is powered ON.

Scenario	MBIM_MS_UICCSLOT_STATE	MBIM_SUBSCRIBER_READY_STATE
eSIM with MF only (no profiles)	MBIMMsUICCSlotStateActiveEsimNoProfiles	MBIMSubscriberReadyStateNoEsimProfile
eSIM with no enabled profiles	MBIMMsUICCSlotStateActiveEsimNoProfiles	MBIMSubscriberReadyStateNoEsimProfile
eSIM with profile enabled	MBIMMsUICCSlotStateActiveEsim	MBIMSubscriberReadyStateInitialized
eSIM in passthrough mode	MBIMMsUICCSlotStateActiveEsimNoProfiles	MBIMSubscriberReadyStateNotInitialized

When a change in both MBIM_MS_UICCSLOT_STATE and MBIM_SUBSCRIBER_READY_STATE is needed, the slot state change should precede the ready state change.

When enabling a new profile or switching between profiles, the ready state should have the following flow:


CurrentReadyState
MBIMSubscriberReadyStateNotInitialized
MBIMSubscriberReadyStateInitialized

For more info about MBIM_MS_UICCSLOT_STATE, see the MBIM_MS_UICCSLOT_STATE table on MB Multi-SIM Operations (MBIM_CID_MS_SLOT_INFO_STATUS).

For more info about MBIM_SUBSCRIBER_READY_STATE, see Section 10.5.2.3 of the public USB MBIM standard.

Access an eSIM in the inactive SIM slot

Article • 05/23/2022

Before the Windows 11, version 22H2 release, all eSIM access related CIDs in the MBIM interface target the SIM in the active SIM slot in DSSA modems. Active SIM slot refers to the SIM slot in a DSSA modem whose SIM is mapped to the device for active use for registration and data connection (or the only SIM slot in a single-SIM modem). As a result, eSIM functionality is applicable only if an eSIM resides in the active SIM slot in a DSSA modem. If an eSIM resides in the inactive slot in a DSSA modem, there is no access to the eSIM.

In the Windows 11, version 22H2 release, MBIMEx 4.0 introduces access to an eSIM in the inactive SIM slot. MBIMEx 4.0 extends the following CIDs with a slot ID element (and other necessary information) to support full access to an eSIM in the inactive slot of a DSSA modem, in addition to the eSIM in the active SIM slot.

- MBIM_CID_MS_UICC_ATR
- MBIM_CID_MS_UICC_OPEN_CHANNEL
- MBIM_CID_MS_UICC_CLOSE_CHANNEL
- MBIM_CID_MS_UICC_APDU
- MBIM_CID_MS_UICC_TERMINAL_CAPABILITY
- MBIM_CID_MS_UICC_RESET
- MBIM_CID_SUBSCRIBER_READY_STATUS

Download the MBIMEx 4.0 specification here .

For general information about MBIMEx 4.0, see MBIMEx 4.0 – 5G SA Phase 2 support.

MB Device-based Reset and Recovery

Article • 03/14/2023

Mobile Broadband (MB or MBB) Device-based Reset and Recovery is a technology in Windows 10, version 1809 and later that introduces a robust reset recovery mechanism for MBB devices and drivers. This mechanism enables MBB devices to avoid failures that cause malfunction, loss of connectivity, or unresponsiveness to operational commands, ultimately making the user experience seamless if errors do occur and reducing the chance of having to restart the system.

Device-based Reset and Recovery can be implemented with or without firmware dependencies. IHV or OEM partners can extend the software-based reset mechanisms available on all Windows PCs with supported device or firmware-level reset mechanisms to increase the rate of successful recovery.

MB Device-based Reset and Recovery architecture overview

The Windows MBB service initializes and controls cellular devices using the standard Mobile Broadband Interface Model (MBIM) interface, which is built on top of the USB transport extending the NCM specifications. Each command sent to the IHV device is sent as an MBIM command via the inbox Windows MBB class driver. Once the cellular modem is initialized using the MBIM protocol exchange, the data path can start between the host, modem, and the network. Any additional controls sent to the cellular device continue via the MBIM protocol exchange in parallel.

There are a number of failures that can occur on both the MBIM control path and the data path. In versions of Windows before Windows 10, version 1809, a simple error handling mechanism is built in. Whenever a MBIM command is sent and device becomes unresponsive, the mechanism attempts to reset the device by sending an MBIM reset command. However, because the failure is often due to an unresponsive MBIM interface in the first place, this reset does not always work. Furthermore, the mechanism doesn't address other failures that might occur such as the loss of connectivity due to data path failures.

MB Device-based Reset and Recovery introduces a centralized framework to detect a larger set of failures and coordinates recovery with a set of progressively impactful resets. If the device supports device-level reset, MB Device-based Reset and Recovery will incorporate the device-level resets after all software-based resets are exhausted. The reset and recovery framework redefines and replaces the existing reset mechanism based on MBIM command responsiveness.

MB Device-based Reset and Recovery detects and attempts to remedy the following types of failures:

Area of failure	Failure description
Control path	• A hang condition detected on MBIM protocol path. For more information about hang detection, see MB hang detection. • Failures due to MBB responding with incorrect state and/or information.
Data path	• Device-side failure resulting in data path failures. For example, endpoints not responding to data traffic, corrupted data from PHY, etc. • Modem/network-side failure. For example, the network not responding to IP traffic, DNS failure, packet loss, etc.

Some failures are not actionable from a recovery perspective, including but not limited to:

- Provisioning- or activation-related issues such as missing COSA settings or MO-initiated service denial
- Control path issues due to driver initialization failure (power- or hardware-related) or software bugs

However, once an actionable failure is detected, MB Device-based Reset and Recovery will attempt the following reset mechanisms. The reset options are listed in the order Windows will perform, from least to most impactful.

Software-based reset options, in the following table are available on all Windows 10, version 1809 MBB devices and can be disabled or configured by OEM patners.

Reset sequence	Reset type	Reset mechanism
1	Software only	Deactivate and activate the PDP context
2	Software only	Toggle Airplane Mode (APM) ON/OFF
3	Sofware only	Enable/disable the device at the Plug and Play (PnP) level

The following device-based reset options are enabled by OEMs with MBB device/firmware capabilities.

Reset sequence	Reset type	Reset mechanism

Reset sequence	Reset type	Reset mechanism
4	Device-based	Functional-level Device Reset (FLDR)
5	Device-based	Platform-level Device Reset (PLDR)

The order of recovery is altered, and in some cases certain reset mechanisms bypassed altogether, for certain types of failures. For example, if a command timeout occurs while toggling airplane mode, the OS does not toggle Airplane Mode to fix it. If the MBB device does not respond to any MBIM commands, then the OS will engage the Device-based reset mechanisms directly.

For UDE client drivers that enable an MBIM function, Windows 10, version 1809 contains a new API that can be used to request a reset whenever the UDECx client driver detects an error. The following section describes these new device-based reset mechanims including FLDR, PLDR, and UDECx reset for PCI.

Device-based resets

Function-level Device Reset (FLDR)

Function-level Device Reset is the lightest device-based reset in terms of system impact. It happens inside the device and is not visible to other devices, and the device stays connected to the bus throughout the reset and returns to a valid state (in other words, an initial state) after the process. This can be provided by either the bus driver or by the firmware. The bus driver implements an FLDR handler if the bus specification defines an in-band reset mechanism that fits the requirement. Firmware writers might override a bus-defined FLDR with their own implementation of FLDR that uses out-of-band signals, such as reset line or power toggling, while still following the requirements of FLDR.

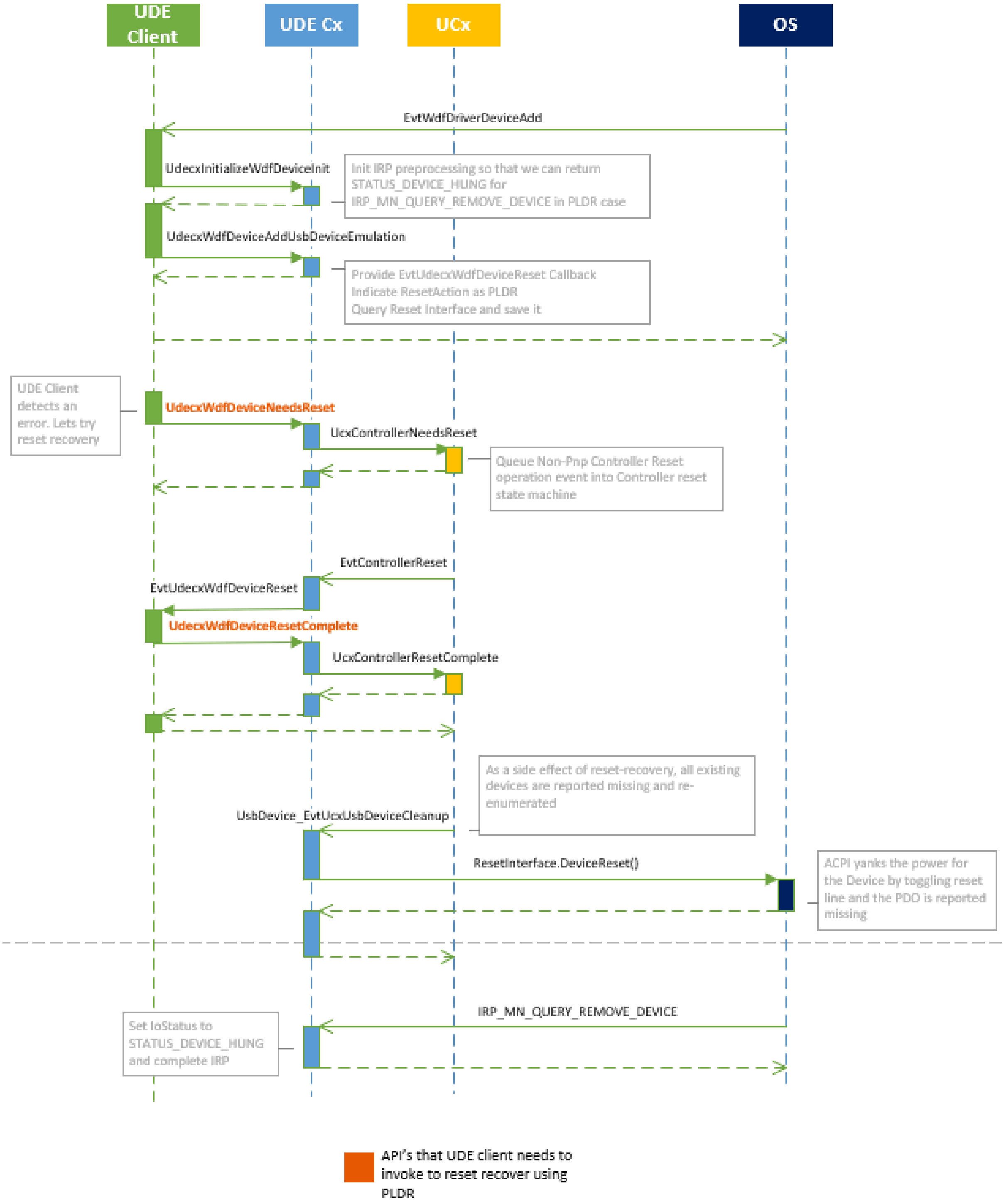
Platform-level Device Reset (PLDR)

Platform-level Device Reset is for cases where FLDR cannot be used, or as a last-resort supplement to FLDR. This reset mechanism causes the device to be reported as missing from the bus (during a power cycle, for example) or affects multiple devices (such as a shared power rail or reset line among devices). The reset method is specified in the ACPI table, which might be implemented as toggling a dedicated reset line or power-cycling the D3 power resource. When PLDR is performed, the OS tears down and rebuilds the stacks of all affected devices to ensure everything starts from a pristine state.

Reset recovery for UDE devices

For UDE client drivers that enable an MBIM function, Windows 10, version 1809 includes an API that can be used to request a reset whenever the UDECx client driver detects an error. The client driver requests a reset by calling a new method, **UdecxWdfDeviceNeedsReset**, specifying the reset type that it wants the UDECx to attempt for the device (if supported). These reset types are **PlatformLevelDeviceReset** and **FunctionLevelDeviceReset** and are values of the **UDECX_WDF_DEVICE_RESET_TYPE** enumeration. Once a reset is initiated, UDECx invokes the driver's *EVT_UDECX_WDF_DEVICE_RESET* callback function and ensures that no other callback is invoked during this process. The client driver is expected to perform any reset related operations such as releasing any resources, then signal reset completion by invoking **UdecxWdfDeviceResetComplete**.

The following flow diagram illustrates the UDE device reset process.


UDE Client
UDE Cx
UCx
OS
EvtWdfDriverDeviceAdd
UdecxInitializeWdfDeviceInit
Init IRP preprocessing so that we can return STATUS_DEVICE_HUNG for IRP_MN_QUERY_REMOVE_DEVICE in PLDR case
UdecxWdfDeviceAddUsbDeviceEmulation
Provide EvtUdecxWdfDeviceReset Callback
Indicate ResetAction as PLDR
Query Reset Interface and save it
UDE Client detects an error. Lets try reset recovery
UdecxWdfDeviceNeedsReset
UcxControllerNeedsReset
Queue Non-Pnp Controller Reset operation event into Controller reset state machine
EvtControllerReset
EvtUdecxWdfDeviceReset
UdecxWdfDeviceResetComplete
UcxControllerResetComplete
As a side effect of reset-recovery, all existing devices are reported missing and re-enumerated
UsbDevice_EvtUcxUsbDeviceCleanup
ResetInterface.DeviceReset()
ACPI yanks the power for the Device by toggling reset line and the PDO is reported missing
IRP_MN_QUERY_REMOVE_DEVICE
Set IoStatus to STATUS_DEVICE_HUNG and complete IRP
🟧 API's that UDE client needs to invoke to reset recover using PLDR

RnR triggers

WWAN Service organizes actionable failures into RnR triggers:

1. Bad connectivity
2. Radio state set/query failures/time-outs
3. Consecutive OID request time-outs
4. Initialization failures

RnR Trigger #1 - Bad connectivity

- Bad connectivity
- Limited Internet connectivity
- Lost Internet connectivity
- Routes not received correctly
- Routes unreachable
- Dead gateway
- DNS query failing

WCM detects based on various sources (NCSI etc.).

WCM publishes WNF_WCM_INTERFACE_CONNECTION_STATE.

```
struct WCM_WNF_INTERFACE_CONNECTION_STATE_INFO
{
    GUID InterfaceGuid;
    WCM_MEDIA_TYPE MediaType = wcm_media_unknown;
    // ConnectionState is one of the WCM_WNF_INTERFACE_CONNECTIVITY_STATE_*
values
    DWORD ConnectionState = 0;
    // TimeInBadStateMs tracks how long a connection is in a Bad state
    // It will reset back to zero when in a good state
    DWORD TimeInBadStateMs = 0;
    // ConnectivityTriggers is a bitmask of
WCM_WNF_INTERFACE_CONNECTIVITY_TRIGGER_* flags
    DWORD ConnectivityTriggers = 0;
    // fWasConnectedGood will be TRUE if a connection is ever in a good
state over the lifetime of an L2 connection
    // Once it is set to TRUE, it will never go FALSE until the interface
disconnects
    BOOLEAN fWasConnectedGood = FALSE;
    // When processing the WNF, walk the array of
WCM_WNF_INTERFACE_CONNECTION_STATE_INFO structs
    // until you reach the struct with afLastArrayValues == TRUE
    BOOLEAN fLastArrayValue = TRUE;
};
```

Recovery Process:

- Reset PDP context up to three times (see FSM transition diagram)
- Toggle APM once
- PnP disable and enable MBB device once
- Invoke FLDR once if supported
- Invoke PLDR once if supported

The process stops as soon as L3 connectivity is good.

Verification of outcome: L3 connectivity is good.

RnR trigger #2 - Radio state set/query failure or time-out

- No response or failure response for setting or querying radio state.
- OID_WWAN_RADIO_STATE set or query requests.
- Should never happen.
- Once it happens, OS and modem may end up in an inconsistent state.
- Indicates serious problem(s) in the modem.
- CWwanExecutor detects it and internally reports to CWwanResetRecovery.

Recovery process:

- Invoke PLDR if supported
- Otherwise, invoke PnP disable/enable

Verification of outcome: Send OID_WWAN_RADIO_STATE query and verify response.

RnR trigger #3 - Time-out of consecutive OID requests

- TXM times all outstanding OID requests and expects responses for each.
- If a “configurable” number of consecutive OID requests receive no response in time, TXM detects it and internally reports to CWwanResetRecovery.
- OIDs may be grouped in high/medium/low latency groups:
  - OID requests that are have no interaction with MO will have lower latency.
  - OID requests that result in interaction with MO will have medium latency.
  - OID_WWAN_CONNECT activation/deactivation request: ~180 seconds.

Recovery process:

- Invoke PLDR if supported
- Otherwise, invoke PnP disable/enable

Verification of outcome: Send OID_WWAN_RADIO_STATE query and verify response.

RnR Trigger #4 - Initialization failures

- Time-out of the device caps or device capsEx query during the initialization upon MB device arrival
- CWwanManager detects and acts on it

Recovery process:

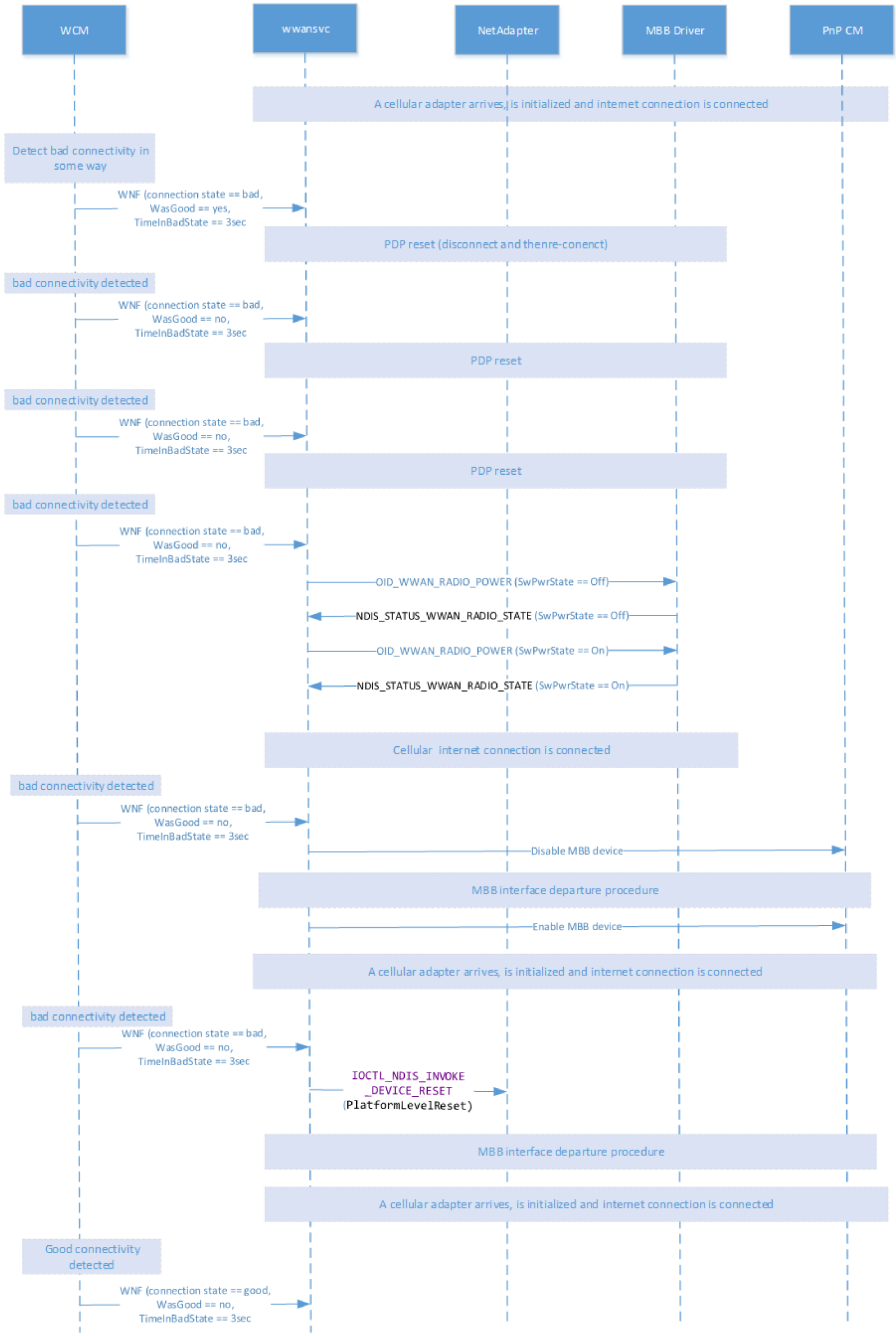
- Invoke PLDR if supported
- Otherwise, invoke PnP disable/enable

Verification of outcome: none

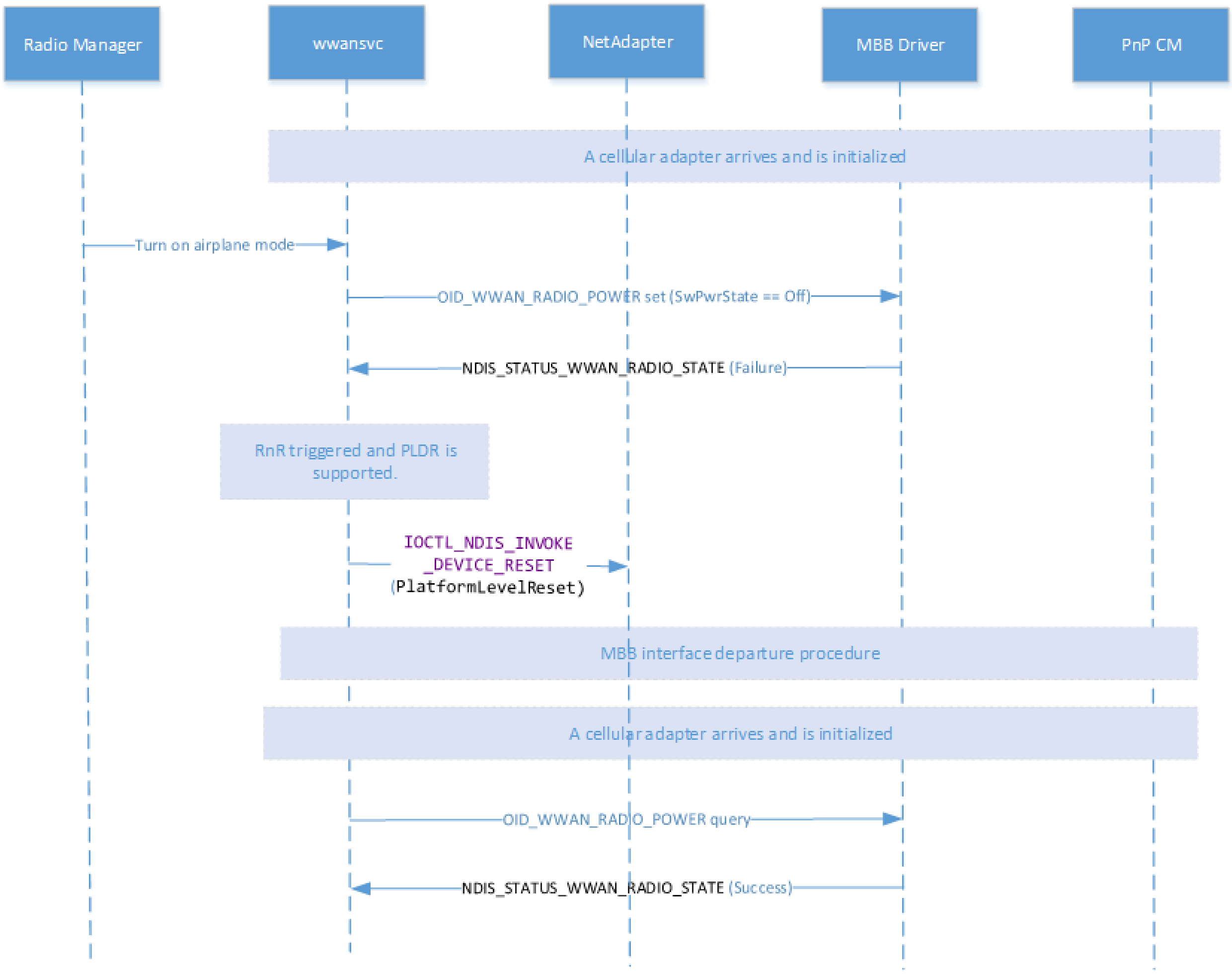
After PLDR or PnP disable/enable, the device departs and then re-arrives. Initialization upon arrival follows.

Primary flows

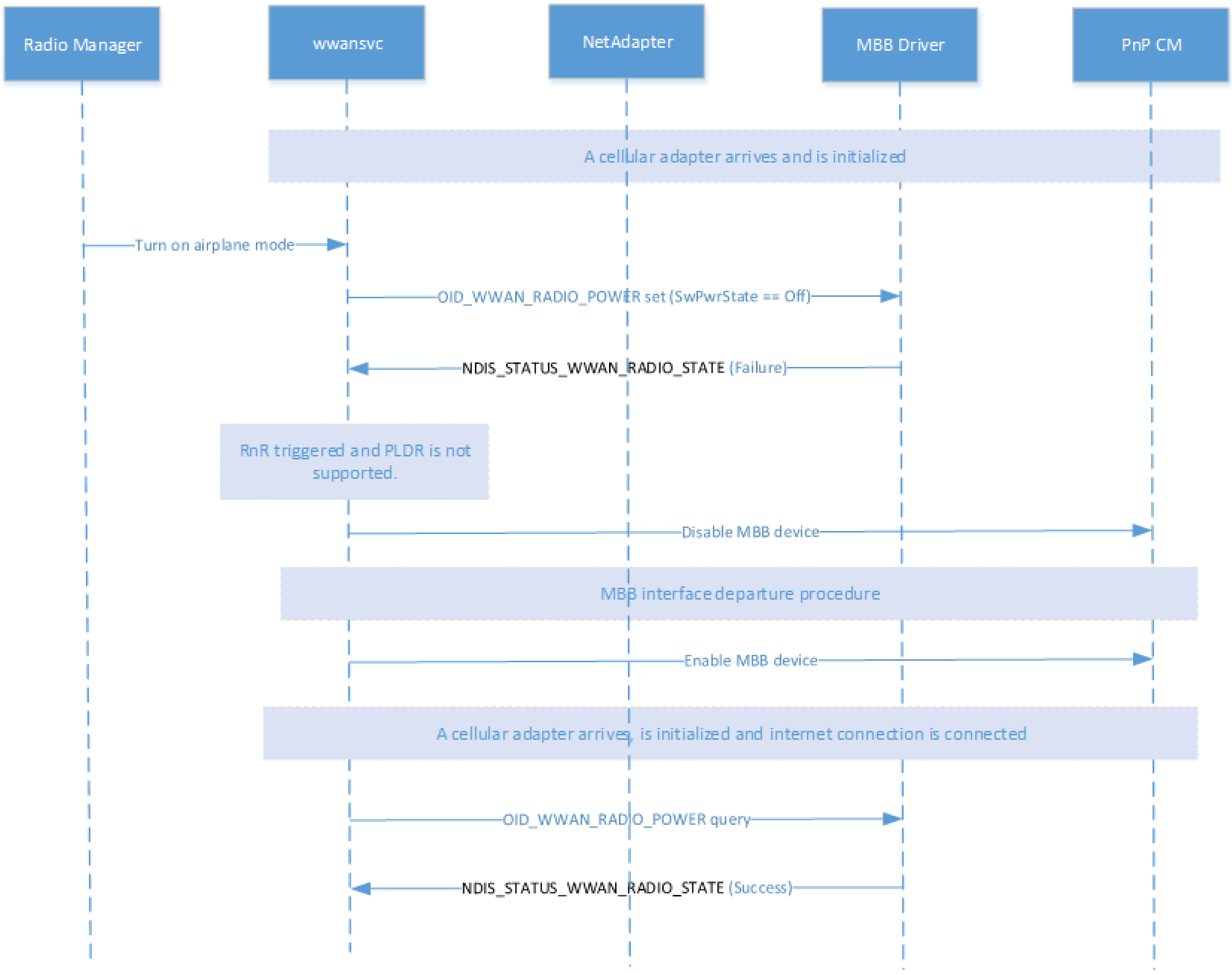
RnR for bad connectivity


WCM
wwansvc
NetAdapter
MBB Driver
PnP CM
A cellular adapter arrives, is initialized and internet connection is connected
Detect bad connectivity in some way
WNF (connection state == bad, WasGood == yes, TimeInBadState == 3sec
PDP reset (disconnect and thenre-conenct)
bad connectivity detected
WNF (connection state == bad, WasGood == no, TimeInBadState == 3sec
PDP reset
bad connectivity detected
WNF (connection state == bad, WasGood == no, TimeInBadState == 3sec
PDP reset
bad connectivity detected
WNF (connection state == bad, WasGood == no, TimeInBadState == 3sec
OID_WWAN_RADIO_POWER (SwPwrState == Off)
NDIS_STATUS_WWAN_RADIO_STATE (SwPwrState == Off)
OID_WWAN_RADIO_POWER (SwPwrState == On)
NDIS_STATUS_WWAN_RADIO_STATE (SwPwrState == On)
Cellular internet connection is connected
bad connectivity detected
WNF (connection state == bad, WasGood == no, TimeInBadState == 3sec
Disable MBB device
MBB interface departure procedure
Enable MBB device
A cellular adapter arrives, is initialized and internet connection is connected
bad connectivity detected
WNF (connection state == bad, WasGood == no, TimeInBadState == 3sec
IOCTL_NDIS_INVOKE _DEVICE_RESET (PlatformLevelReset)
MBB interface departure procedure
A cellular adapter arrives, is initialized and internet connection is connected
Good connectivity detected
WNF (connection state == good, WasGood == no, TimeInBadState == 3sec

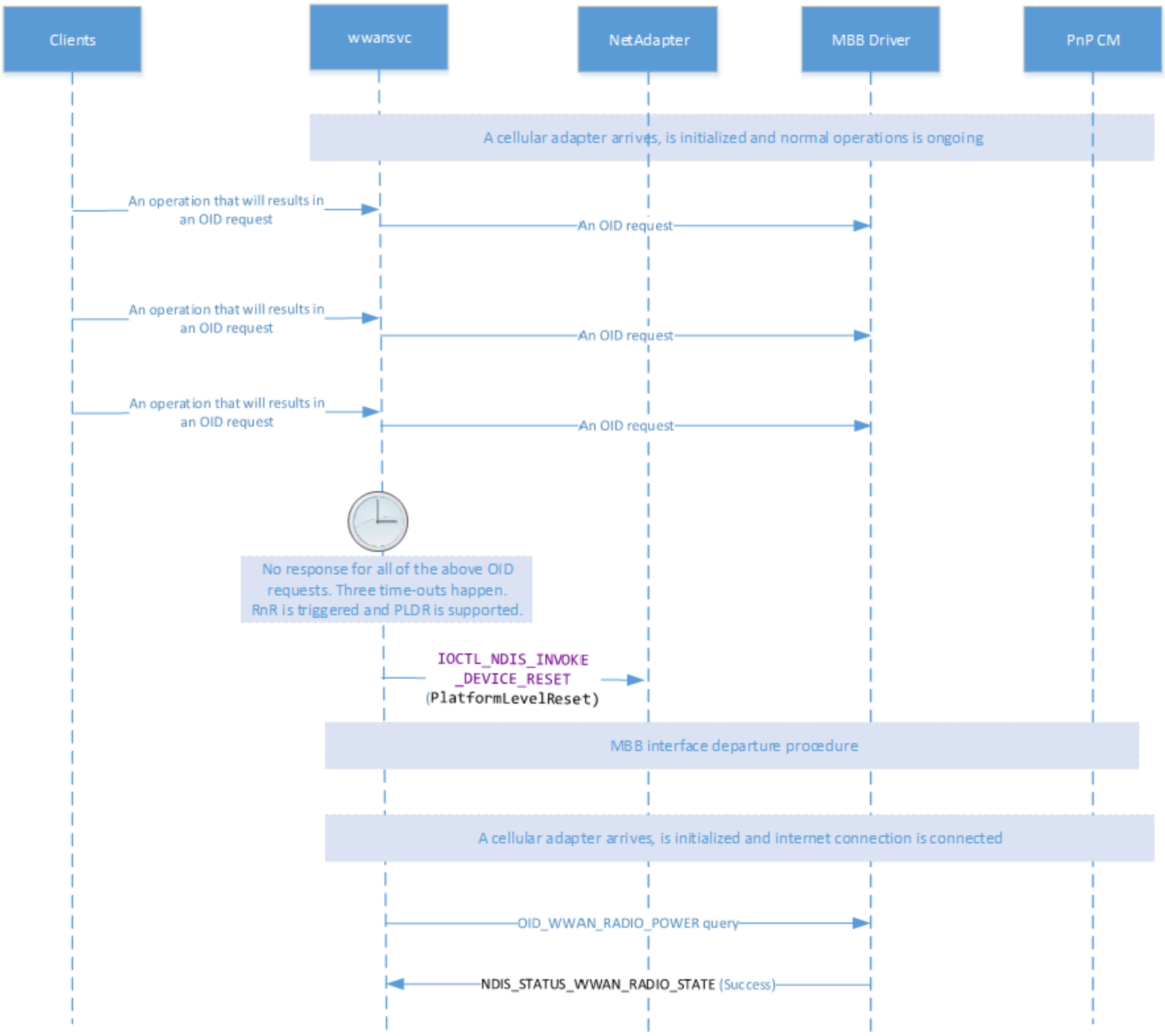
PLDR for radio power set failure


Radio Manager
wwansvc
NetAdapter
MBB Driver
PnP CM
A cellular adapter arrives and is initialized
Turn on airplane mode
OID_WWAN_RADIO_POWER set (SwPwrState == Off)
NDIS_STATUS_WWAN_RADIO_STATE (Failure)
RnR triggered and PLDR is supported.
IOCTL_NDIS_INVOKE_DEVICE_RESET (PlatformLevelReset)
MBB interface departure procedure
A cellular adapter arrives and is initialized
OID_WWAN_RADIO_POWER query
NDIS_STATUS_WWAN_RADIO_STATE (Success)

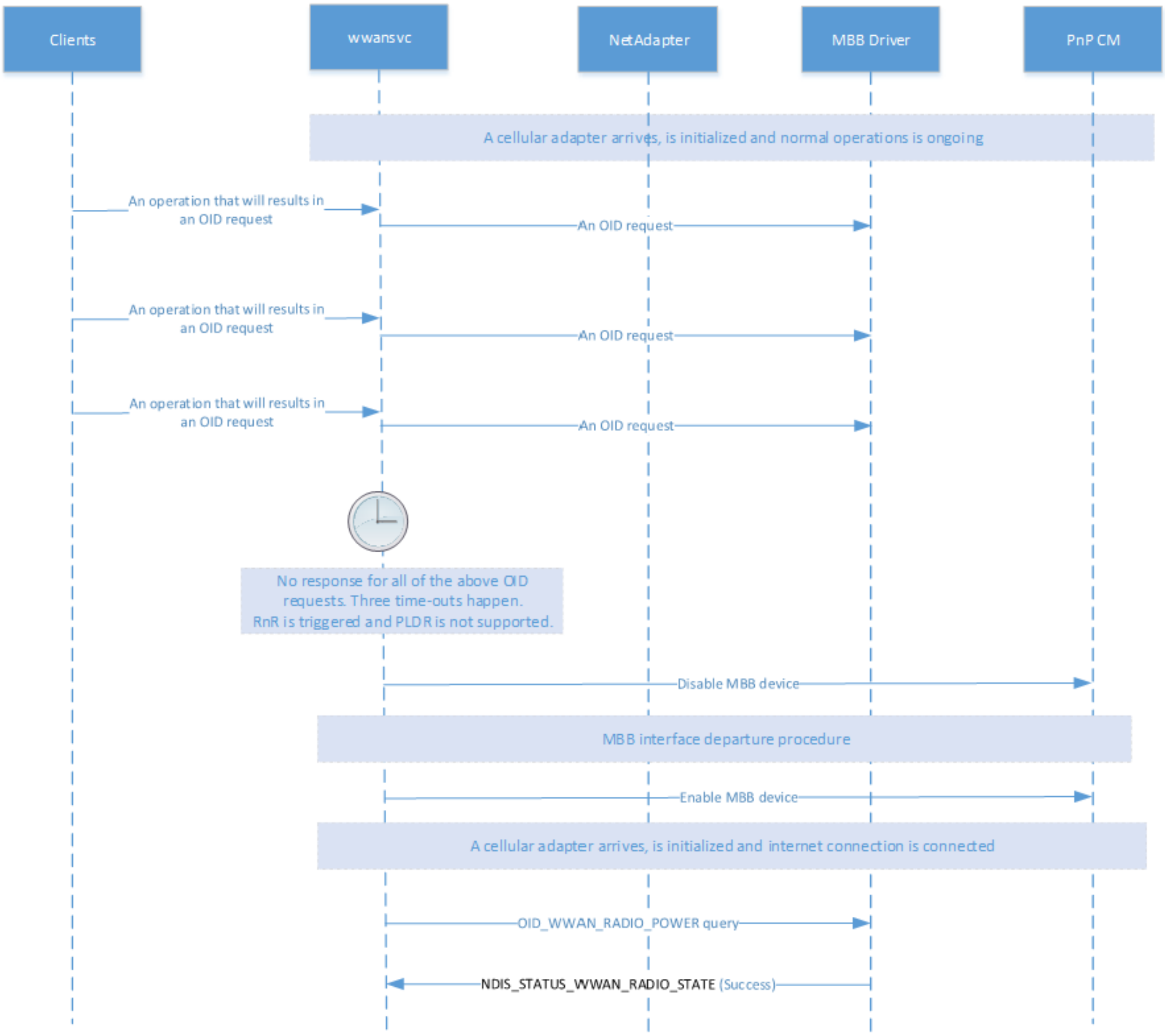
PnP disable/enable for radio state set failure


Radio Manager
wwansvc
NetAdapter
MBB Driver
PnP CM
A cellular adapter arrives and is initialized
Turn on airplane mode
OID_WWAN_RADIO_POWER set (SwPwrState == Off)
NDIS_STATUS_WWAN_RADIO_STATE (Failure)
RnR triggered and PLDR is not supported.
Disable MBB device
MBB interface departure procedure
Enable MBB device
A cellular adapter arrives, is initialized and internet connection is connected
OID_WWAN_RADIO_POWER query
NDIS_STATUS_WWAN_RADIO_STATE (Success)

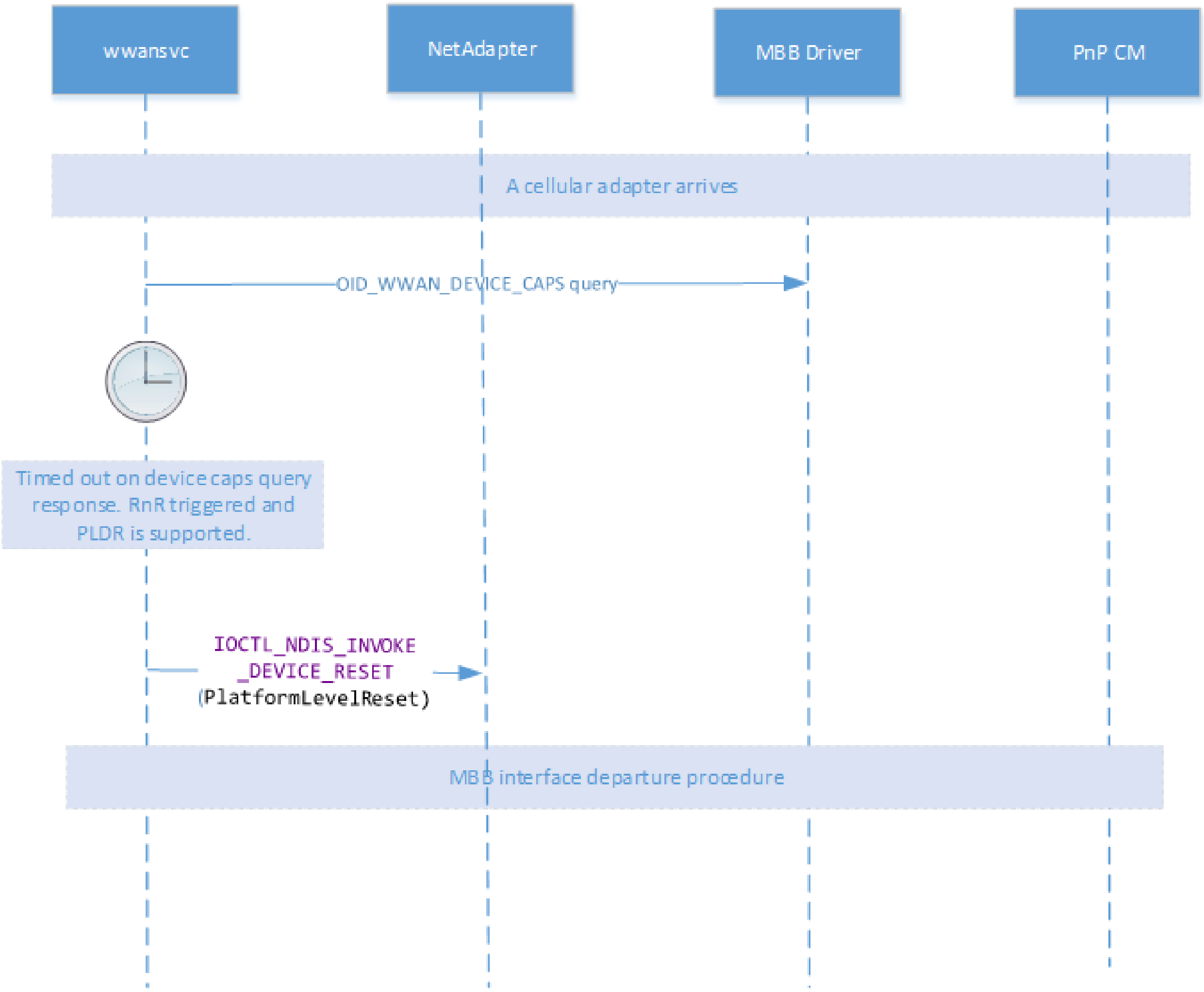
PLDR for time-outs of consecutive OID requests


Clients
wwansvc
NetAdapter
MBB Driver
PnP CM
A cellular adapter arrives, is initialized and normal operations is ongoing
An operation that will results in an OID request
An OID request
An operation that will results in an OID request
An OID request
An operation that will results in an OID request
An OID request
No response for all of the above OID requests. Three time-outs happen. RnR is triggered and PLDR is supported.
IOCTL_NDIS_INVOKE _DEVICE_RESET (PlatformLevelReset)
MBB interface departure procedure
A cellular adapter arrives, is initialized and internet connection is connected
OID_WWAN_RADIO_POWER query
NDIS_STATUS_WWAN_RADIO_STATE (Success)

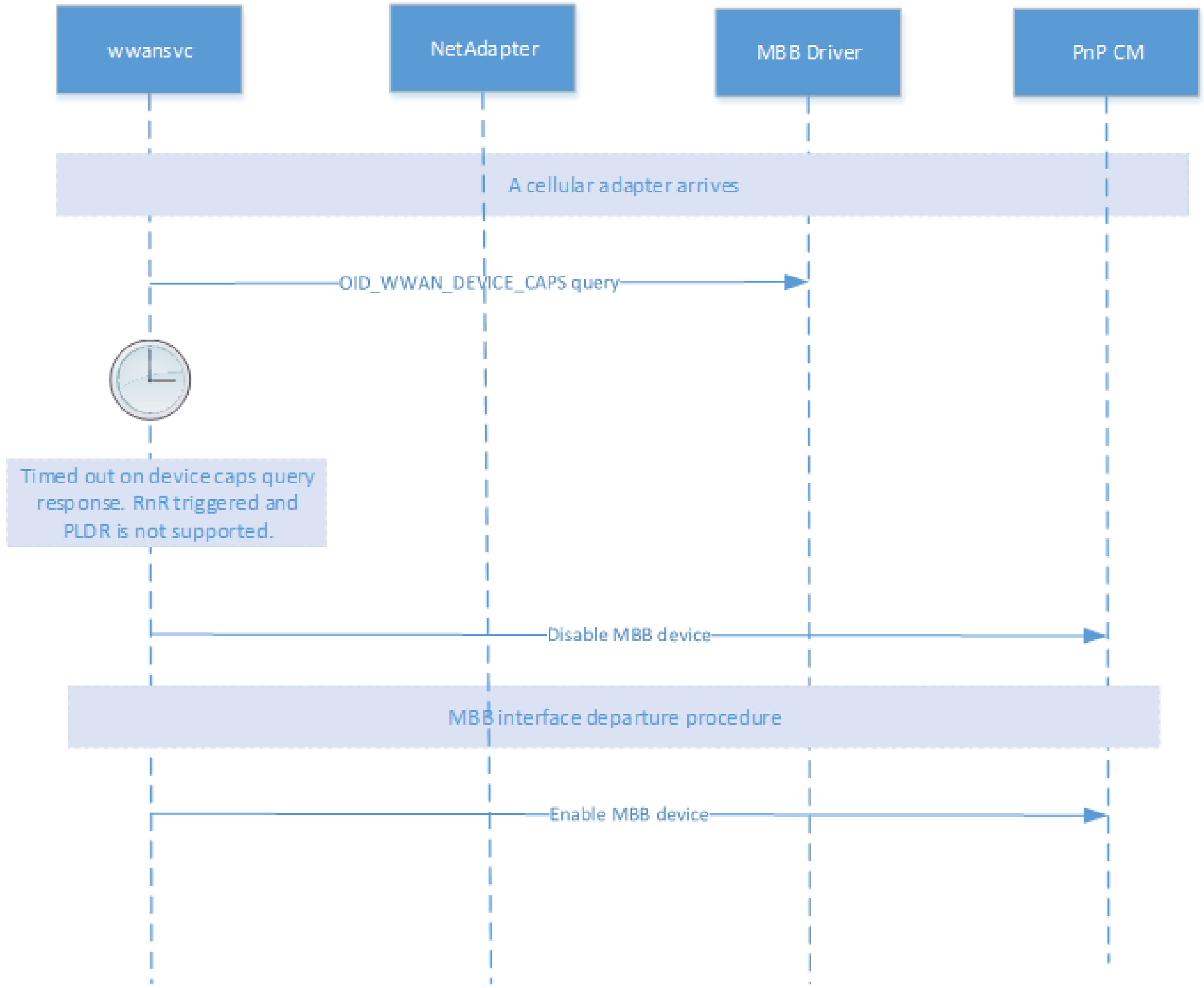
PnP disable/enable for time-outs of consecutive OID requests


Clients
wwansvc
NetAdapter
MBB Driver
PnP CM
A cellular adapter arrives, is initialized and normal operations is ongoing
An operation that will results in an OID request
An OID request
An operation that will results in an OID request
An OID request
An operation that will results in an OID request
An OID request
No response for all of the above OID requests. Three time-outs happen. RnR is triggered and PLDR is not supported.
Disable MBB device
MBB interface departure procedure
Enable MBB device
A cellular adapter arrives, is initialized and internet connection is connected
OID_WWAN_RADIO_POWER query
NDIS_STATUS_WWAN_RADIO_STATE (Success)

PLDR for initialization failure


wwansvc
NetAdapter
MBB Driver
PnP CM
A cellular adapter arrives
OID_WWAN_DEVICE_CAPS query
Timed out on device caps query response. RnR triggered and PLDR is supported.
IOCTL_NDIS_INVOKE_DEVICE_RESET (PlatformLevelReset)
MBB interface departure procedure

PnP disable/enable for initialization failure


wwansvc
NetAdapter
MBB Driver
PnP CM
A cellular adapter arrives
OID_WWAN_DEVICE_CAPS query
Timed out on device caps query response. RnR triggered and PLDR is not supported.
Disable MBB device
MBB interface departure procedure
Enable MBB device

Requirements for MB Device-based Reset and Recovery

Requirements for FLDR

To support FLDR on a device, inside the Device() scope there must be a `_RST` method defined. When executed, the method must reset only that device, and should not touch another device. The device must also stay on the bus, connected.

```C++
Device(PCI0)
{
Device(USB0)
{
    Name(_ADR, 0x1d0000)
    Name(_S4D, 0x2)
    Name(_S3D, 0x2)
    …
    Method(_RST, 0x0, NotSerialized)
    {
            //
            // Perform reset of the USB0 device
            //
    }
}
}
```

Requirements for PLDR

In PLDR, devices that are affected by the reset of other device are expressed as sharing a `PowerResource` for reset. The devices declare their dependency on the `PowerResource` for reset, and that `PowerResource` implements the `_RST` method.

```C++
Device(PCI0)
{
PowerResource(URST, 0x5, 0x0)
{
    //
    // Dummy _ON and _OFF methods. All power resources must have these
    // two defined.
    // Method(_ON, 0x0, NotSerialized)
    {
    }
    Method(_OFF, 0x0, NotSerialized)
    {
    }
    Method(_RST, 0x0, NotSerialized)
    {
            //
            // Perform reset of the USB0 and USB1 devices
            //
    }
}
Device(USB0)
{
    Name(_ADR, 0x1d0000)
    Name(_S4D, 0x2)
    Name(_S3D, 0x2)
    …
    Name(_PRR, Package(0x1) { ^URST })
}
Device(USB1)
{
    Name(_ADR, 0x1d0001)
    Name(_S4D, 0x2)
    Name(_S3D, 0x2)
    …
    Name(_PRR, Package(0x1) { ^URST })
}
}
```

Alternatively, PLDR can be achieved by putting the device into the D3Cold power state and back to D0, essentially power cycling the device. In this case, having `_PR3` declared in the device scope is sufficient to support PLDR. ACPI will use `_PR3` to determine reset dependencies between devices if no `_PRR` is referenced in the device scope. For more information, see Resetting and recovering a device.

Sample Log

```
NTS]WWAN Service event: [Info] WwanTimerWrapper::StartTimer:  Timer (ID = 0)
Start Completed
[0]0E98.34E4::11/27/2019-05:37:55.622 [Microsoft-Windows-WWAN-SVC-
EVENTS]WWAN Service event: [Info] WwanTxmEvaluateArmTimer: TXM timer armed
for 60 seconds Interface: {{8a664721-db25-4157-8395-5d21e0560fa4}}
[0]0E98.34E4::11/27/2019-05:37:55.622 [Microsoft-Windows-WWAN-SVC-
EVENTS]WWAN Service event: [Info] _sendReq: ASYNC OID (pTx->handle:
00000000000000B0 Code: 1) sent
[0]0E98.34E4::11/27/2019-05:37:55.622 [Microsoft-Windows-WWAN-SVC-
EVENTS]WWAN Service event: [Info] CWwanExecutor::RegWriteStoredRadioState:
Try to set the subkey to 0x0 for ArrivalRadioState Interface: {{8a664721-
db25-4157-8395-5d21e0560fa4}}
[0]0E98.34E4::11/27/2019-05:37:55.623 [Microsoft-Windows-WWAN-SVC-
EVENTS]WWAN Service event: [Info]
CWwanResetRecovery::EvaluateAndTryHighImpactRnRMethod:  Attempted to turn
off radio via MBB (reqId 0x10c): request ID 0x1 prev stage 0 APMToggling 0;
PnPDisabling 0; PLDR 0; FLDR 0 Interface: {{8a664721-db25-4157-8395-
5d21e0560fa4}}
[0]0E98.34E4::11/27/2019-05:37:55.623 [Microsoft-Windows-WWAN-SVC-
EVENTS]WWAN Service event: [Info] WwanTimerWrapper::StartTimer:  Timer (ID =
6) Start Completed
[0]0E98.34E4::11/27/2019-05:37:55.623 [Microsoft-Windows-WWAN-SVC-
EVENTS]WWAN Service event: [Info] CWwanResetRecovery::fsmEventHandler:  exit
with state: 7, event: 4, RnR stage: 2 Potent RnR: 0 Interface: {{8a664721-
db25-4157-8395-5d21e0560fa4}}
```

Related links

MB hang detection

UDECX_WDF_DEVICE_RESET_TYPE

UdecxWdfDeviceNeedsReset

EVT_UDECX_WDF_DEVICE_RESET

UdecxWdfDeviceResetComplete

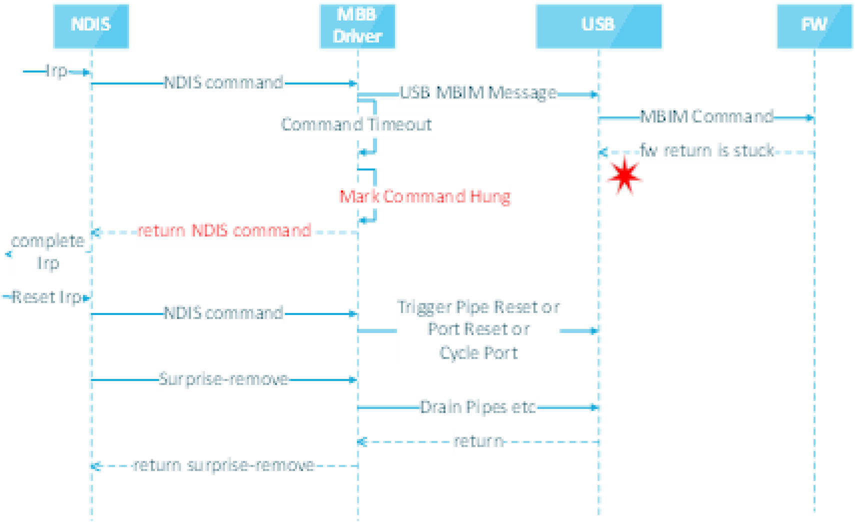
MB hang detection

Article • 03/14/2023

Mobile Broadband (MB or MBB) hang detection is a technology in Windows 10, version 1809 and later that assists MBB client drivers in detecting hang conditions in the control path and recovering from them. It is a part of the Device-based Reset and Recovery feature that is aimed at recovering from a variety of possible error conditions for MBB devices and drivers.

The flow diagrams in this topic use USB as the underlying bus, although the reset mechanism is bus agnostic if the ACPI and bus stack define the interfaces described on this page.

The following flow diagram applies generically to all NDIS object identifiers (OIDs) and callbacks to miniport drivers. There might be cases where the recovery part of this process does not work if NDIS does not fully support reset recovery.



This hang detection and reset flow sequence consists of 3 phases:

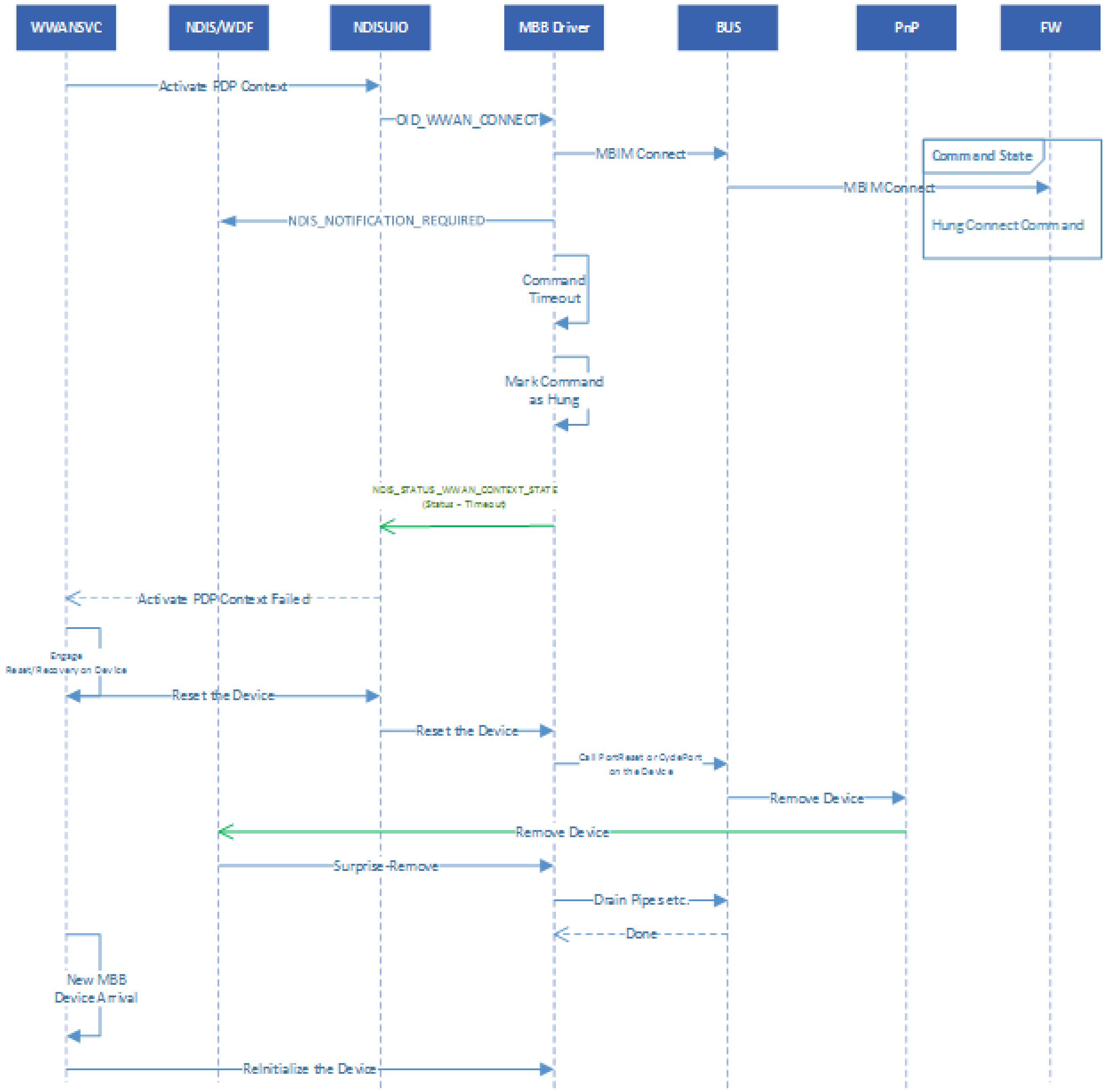
1. Hang detection
2. Any potential logging to get the states for further debugging
3. Reset – surprise removal handling

Hang detection

The service layer provides a hint to the driver to initiate a recovery whenever something wrong is detected at the user layer, since `WWANSvc` has a state machine managing the connectivity state of the cellular adapter. To support this, a private interface is defined that the driver uses to trigger a reset operation. There are a few cases where the driver originates its own commands down to the device to make sure the state machine is valid. If any of these commands time out, then the reset/recovery is triggered from the driver itself without having to communicate the operation back to user mode to initiate a recovery operation.

For more information about the private interface that UDE client drivers can use to trigger a reset operation, see MB Device-based Reset and Recovery.

This example uses OID_WWAN_CONNECT as an example for walking through the hang detection flow.



1. NDIS (via the protocol driver) receives an OID_WWAN_CONNECT.
2. NDIS passes OID_WWAN_CONNECT down to the class driver.
3. The class driver constructs an MBIM message for the Connect request.
4. The class driver sends the MBIM message to the MBIM function via the USB bus.

5. The firmware command times out, which could be because the firmware is hung or the MBIM command is taking a long time to complete.
6. The class driver returns the NDIS command without the firmware completion with NDIS_NOTIFICATION_REQUIRED. The result of OID_WWAN_CONNECT is returned with a solicited notification from the driver via NDIS_STATUS_WWAN_CONTEXT_STATE with **Status** set to **Timeout**, indicating that the underlying device didn't respond to the command.
7. NDIS completes the OID request to the protocol driver.
8. The protocol driver returns the call back to the service, which sees that the command failed.
9. The service triggers a reset operation on the device using the new OID interface.
10. After this point, the FDO calls the bus to surprise-remove and re-enumerate the MBB device. If the underlying bus is USB, then the FDO will call appropriate functions to reset the device.
11. If the appropriate ACPI methods are defined in UEFI, then either FLDR or PLDR will be triggered.

For more information about FLDR and PLDR, see MB Device-based Reset and Recovery.

Reset (Surprise-Removal)

Once the reset recovery can proceed, the bus causes the Plug and Play (PnP) manager to generate a surprise-remove IRP, provided the support is present at the ACPI/UEFI level. NDIS, on receiving the surprise-remove IRP, calls back into `WMBCLASS` for a surprise-remove PnP event callback. `WMBCLASS` handles the surprise-removal operation. At this point, all the commands, etc. must be completed and the data packets must be returned successfully back to NDIS. Otherwise, the surprise-removal operation will not complete. The rest of flow is identical to a real device surprise-remove on a bus, for example USB.

1. NDIS calls the PnP event for surprise-removal.
2. `WMBCLASS` ignores the return of hung MBIM command and returns the original NDIS command.
3. `WMBCLASS` returns the NDIS PnP callback for surprise-removal.

Recovery

After the surprise-removal, all drivers in the stack including `WMBCLASS` must release all resources so that the device object can be removed and re-enumerated by the bus. Failing to do so, the device will not be re-enumerated and will not be recovered.

Related links

MB Device-based Reset and Recovery

MB Device-based Reset and Recovery Trace

MB Device Reset and Recovery (RnR) trace

Article • 03/14/2023

Useful keywords/regexp for filtering traces

- Attempt connection reset
- current RnR state
- OnConnectionStateInfoChanged
- EvaluateAndTryHighImpactRnRMethod
- CWwanResetRecovery::Trigger
- CWwanResetRecovery::Initialize
- CWwanResetRecovery::OnNdisNotification
- CWwanResetRecovery::OnWwanNotification
- CWwanResetRecovery::OnInterfaceReArrival
- CWwanResetRecovery::OnInterfaceRemoval
- CWwanResetRecovery::fsmEventHandler:
- CWwanResetRecovery::OnWwanNotification
- CWwanResetRecovery::fsmEventHandler:
- CWwanDeviceEnumerator::
- CWwanResetRecovery
- SET OID_WWAN_RADIO_STATE (e010103), RequestId
- : OID_WWAN_CONNECT (
- [NH] Dispatch WwanNotificationSourceMsm\WwanMsmEventTypeConnectionIStreamUpdated
-]RouteManagement::BadConnectionState
- WNF_WCM_INTERFACE_CONNECTION_STATE
- WNF_PHN_CALL_STATUS
- L3ConnnectivityGood
- WaitL3ConnnectivityGood
- CONTEXT_STATE Resp (Set)
- found valid
- _ReadRnRPolicies
- Opportunistic Internet
- Active probe result code on interface

Investigation tips

- Ensure the necessary ETW providers are included in log:

    1. netsh trace start wireless_dbg,provisioning persist=yes
    2. repro the scenario
    3. netsh trace stop
    4. Collect the files generated and attach them to the bug

- Follow the RnR trigger guidelines to identify the case that caused the RnR trigger.

MB SAR Platform Support

Article • 03/14/2023

Overview

Specific Absorption Rate (SAR) is the capability to change the MBB radio transmitter power in reaction to the proximity of the MBB antenna to the user. Traditionally, OEMs have implemented proprietary solutions for SAR. This requires the OEM to implement a device service command that is either only identified between their User Mode Driver (UMDF) and the modem or requires kernel mode components to directly interact with the modem. Some OEMs may even have a hybrid solution where they have both UMDF-modem and kernel mode-modem components. As radio radiation awareness has increased, standardizing the interface for OEM software components to pass through the SAR command to the modem introduces the following benefits:

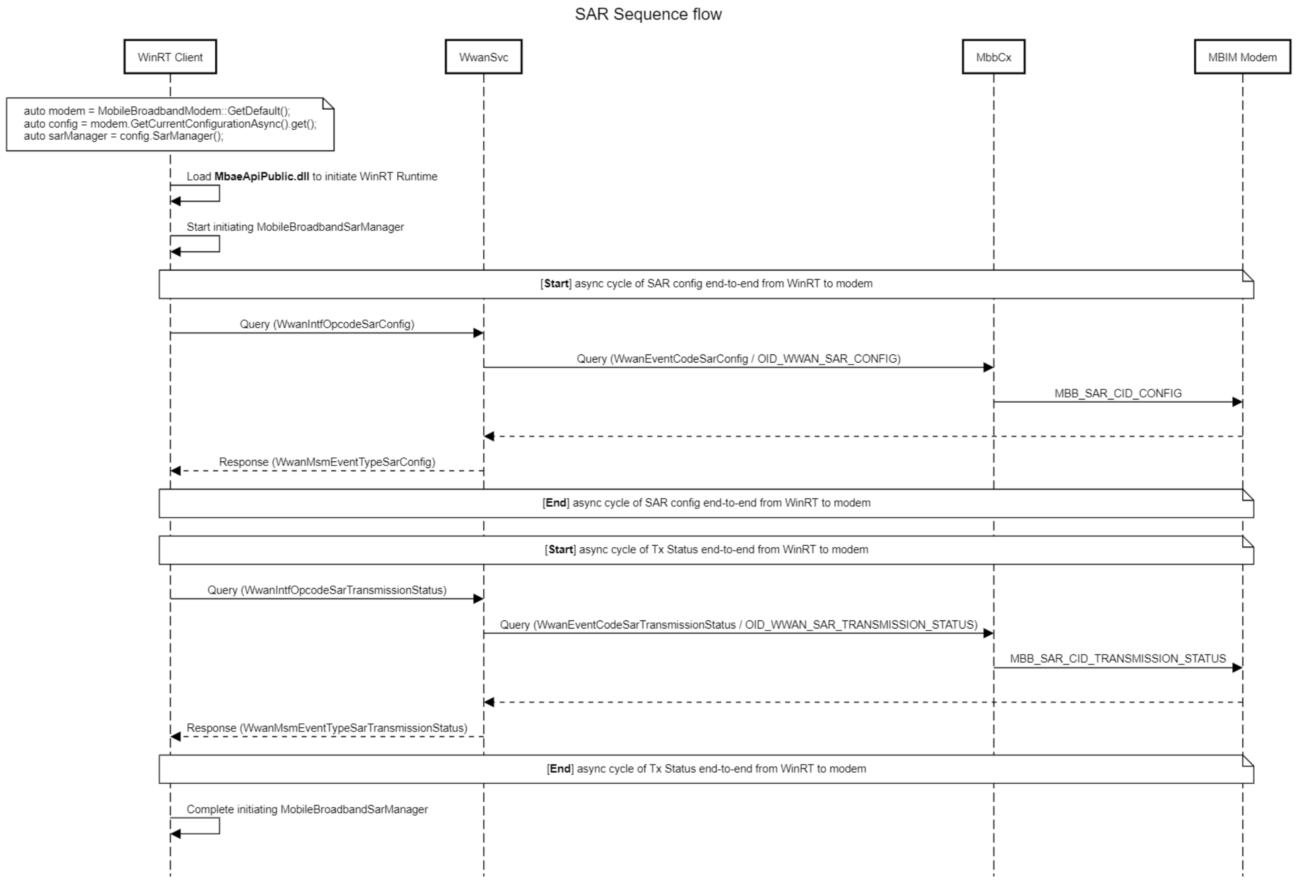
1. OEMs can move toward user mode components and makes the system more stable, as errors in user mode are not fatal to the system compared to kernel mode.
2. Windows provides a platform standard interface and reduces the proprietary implementation from OEMs.
3. Services in the platform that want to take advantage of SAR can retrieve the information from the modem.

Starting in Windows 10, version 1703, Windows supports passing through SAR configuration and modem transmission status. Windows will continue to leave the SAR business logic to IHVs and OEMs to use as a self-differentiating factor but will provide an interface to streamline the platform. Two new NDIS OIDs and two new MBIM CIDs have been defined to support this interface. Devices that want to take advantage of OS support must implement both commands.

This feature is supported by adding in two new OIDs and CIDs. For IHV partners that implement MBIM, only the CID version needs to be supported.

> **Note**
>
> This topic defines the interface for IHV partners to implement SAR platform support in their modem device drivers. If you are looking for info about customizing the SAR mapping table for a device, see **Customize a Specific Absorption Rate (SAR) mapping table**.

Flow



```
auto modem = MobileBroadbandModem::GetDefault();
auto config = modem.GetCurrentConfigurationAsync().get();
auto sarManager = config.SarManager();
```

SAR Sequence flow
WinRT Client
WwanSvc
MbbCx
MBIM Modem
Load MbaeApiPublic.dll to initiate WinRT Runtime
Start initiating MobileBroadbandSarManager
[Start] async cycle of SAR config end-to-end from WinRT to modem
Query (WwanIntfOpcodeSarConfig)
Query (WwanEventCodeSarConfig / OID_WWAN_SAR_CONFIG)
MBB_SAR_CID_CONFIG
Response (WwanMsmEventTypeSarConfig)
[End] async cycle of SAR config end-to-end from WinRT to modem
[Start] async cycle of Tx Status end-to-end from WinRT to modem
Query (WwanIntfOpcodeSarTransmissionStatus)
Query (WwanEventCodeSarTransmissionStatus / OID_WWAN_SAR_TRANSMISSION_STATUS)
MBB_SAR_CID_TRANSMISSION_STATUS
Response (WwanMsmEventTypeSarTransmissionStatus)
[End] async cycle of Tx Status end-to-end from WinRT to modem
Complete initiating MobileBroadbandSarManager

MB Interface Update for SAR Platform Support

An MBIM-compliant device implements and reports the following device service when queried by CID_MBIM_DEVICE_SERVICES. The existing well-known services are defined in section 10.1 of the USB NCM MBIM 1.0 specification. Microsoft extends this to define the following service.

Service Name = **Microsoft SAR Control**

UUID = **UUID_MS_SARControl**

UUID Value = **68223D04-9F6C-4E0F-822D-28441FB72340**

CID	Minimum OS Version
MBIM_CID_MS_SAR_CONFIG	Windows 10, version 1703
MBIM_CID_MS_TRANSMISSION_STATUS	Windows 10, version 1703

MBIM_CID_MS_SAR_CONFIG

Description

This command sets or returns information about a MB device’s SAR back off mode and level. The MB device must act on the SAR back off command immediately by overwriting the current Transmit power limits and applying them to the transmitting antennas. If an antenna’s SAR configuration was not changed by the operating system, it should maintain its current setting. For example, if the operating system sets antenna 1 to be SAR back off index 1, then antenna 2’s configuration should be kept the same without any changes.

It is expected for devices that support this command to implement Query so they provide device information to the OS and its clients. For the Set command, it is between the IHV and the OEM to define which value of each field is acceptable. The typical expectation is that the SAR back off index is configurable for all antennas as a minimum baseline. If a Set request is sent with fields that are not supported by the device, then MBIM_STATUS_INVALID_PARAMETERS must be returned as the status code.

After each Query or Set response, the modem should return a MBIM_MS_SAR_CONFIG structure that contains information for all antennas on the device associated with Mobile Broadband.

Query

The InformationBuffer on MBIM_COMMAND_MSG is not used. MBIM_MS_SAR_CONFIG is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Set

The InformationBuffer on MBIM_COMMAND_MSG contains a MBIM_MS_SAR_CONFIG. MBIM_MS_SAR_CONFIG is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Unsolicited Events

Not applicable.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_SET_SAR_CONFIG	Not applicable	Not applicable
Response	MBIM_MS_SAR_CONFIG	MBIM_MS_SAR_CONFIG	Not applicable

Data Structures

Query

The InformationBuffer shall be NULL and InformationBufferLength shall be zero.

Set

The following MBIM_MS_SET_SAR_CONFIG structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description
0	4	SARMode	MBIM_MS_SAR_CONTROL_MODE	For more information, see the MBIM_MS_SAR_CONTROL_MODE table.
4	4	SARBackOffStatus	MBIM_MS_SAR_BACKOFF_STATE	For more information, see the MBIM_MS_SAR_BACKOFF_STATE table. If MBIM_MS_SAR_CONTROL_MODE is set to be device-controlled, then the OS will not be able set this field.
8	4	ElementCount (EC)	UINT32	Count of MBIM_MS_SAR_CONFIG structures that follow in the DataBuffer.

Offset	Size	Field	Type	Description
12	8 * EC	SARConfigStatusRefList	OL_PAIR_LIST	The first element of the pair is a 4-byte offset, calculated from the beginning (offset 0) of this MBIM_MS_SET_SAR_CONFIG structure, to an MBIM_MS_SAR_CONFIG_STATE structure. For more information, see the MBIM_MS_SAR_CONFIG_STATE table. The second element of the pair is a 4-byte size of a pointer to the corresponding MBIM_MS_SAR_CONFIG_STATE structure.
12 + (8 * EC)		DataBuffer	DATABUFFER	Array of MBIM_MS_SAR_CONFIG_STATE structures.

The following structures are used in the preceding table.

MBIM_MS_SAR_CONTROL_MODE specifies how the SAR back off mechanism is controlled.

Type	Value	Description
MBIMMsSARControlModeDevice	0	SAR back off mechanism is controlled by the modem device directly.
MBIMMsSARControlModeOS	1	SAR back off mechanism is controlled and managed by the operating system.

MBIM_MS_SAR_BACKOFF_STATE describes the state of SAR back off.

Type	Value	Description
MBIMMsSARBackOffStatusDisabled	0	SAR back off is disabled in the modem.
MBIMMsSARBackOffStatusEnabled	1	SAR back off is enabled in the modem.

MBIM_MS_SAR_CONFIG_STATE describes the possible states for SAR backoff for the antennas.

Offset	Size	Field	Type	Description
0	4	SARAntennaIndex	UINT32	An antenna index that corresponds to the **SARBackOffIndex** field in this table. It corresponds to the antenna number and is left to OEM implementation to index each antenna on the device. Any index is valid for this value. If this value is set to **0xFFFFFFFF** in a *Set* command, the **SARBackOffIndex** should be applied to all antennas. If this value is set to **0xFFFFFFFF** in response, it indicates that **SARBackOffIndex** is applied to all antennas.
4	4	SARBAckOffIndex	UINT32	A back off index that corresponds to the back off table that is defined by the OEM or modem vendor. The table has individual bands and associated back off parameters.

Response

The following MBIM_MS_SAR_CONFIG structure shall be used in the InformationBuffer. MBIM_MS_SAR_CONFIG specifies the configuration for SAR.

Offset	Size	Field	Type	Description
0	4	SARMode	MBIM_MS_SAR_MODE	For more information, see the MBIM_MS_SAR_CONTROL_MODE table.
4	4	SARBackOffStatus	MBIM_MS_SAR_BACKOFF_STATE	For more information, see the MBIM_MS_SAR_BACKOFF_STATE table.

Offset	Size	Field	Type	Description
8	4	SARWifiIntegration	MBIM_MS_SAR_ WIFI_HARDWARE_INTEGRATION	For more information, see the MBIM_MS_SAR_HARDWARE_WIFI_INTEGRATION table. This implies the device's Wi-Fi and Cellular SAR is integrated at the hardware layer and the device will automatically adjust SAR control for both radios.
12	4	ElementCount (EC)	UINT32	Count of MBIM_MS_SAR_CONFIG_STATE structures that follow in the DataBuffer.
16	8 * EC	SARConfigStatusRefList	OL_PAIR_LIST	The first element of the pair is a 4 byte offset, calculated from the beginning (offset 0) of this MBIM_MS_SAR_CONFIG structure, to an MBIM_MS_SAR_CONFIG_STATE structure. For more information, see the MBIM_MS_SAR_CONFIG_STATE table. The second element of the pair is a 4 -byte size of a pointer to the corresponding MBIM_MS_SAR_CONFIG_STATE structure.
16 + (8 * EC)		DataBuffer	DATABUFFER	Array of MBIM_MS_SAR_CONFIG_STATE structures.

The following MBIM_MS_SAR_HARDWARE_WIFI_INTEGRATION structure is used in the preceding table. It specifies whether Wi-Fi and Cellular are integrated at the hardware level.

Type	Value	Description
MBIMMsSARWifiHardwareIntegrated	0	Wi-Fi and Cellular modem SAR is integrated in the device.
MBIMMsSARWifiHardwareNotIntegrated	1	Wi-Fi and Cellular modem SAR is not integrated in the device.

Notification

Not applicable.

Status Codes

Error Code	Description
MBIM_STATUS_SUCCESS	The request was successfully processed.
MBIM_STATUS_BUSY	The device is currently busy.
MBIM_STATUS_FAILURE	The request failed.
MBIM_STATUS_NO_DEVICE_SUPPORT	Device does not support this command.
MBIM_STATUS_INVALID_PARAMETERS	The operation failed because of invalid parameters.
MBIM_STATUS_OPERATION_NOT_ALLOWED	The operation failed because the operation is not allowed.

MBIM_CID_MS_TRANSMISSION_STATUS

Description

This command is used to enable or disable the notification from the modem on transmit state. It is a per-executor command as each executor can have different channel transmit state. For example, a dual SIM modem might have one on LTE and the other on GSM. At the same time, it can be used to provide the transmit status of the modem. This notification could be used for clients that are interested in whether the modem is transmitting data or not. The modem should provide notification any time there is a start or end of TX traffic. If the duty cycle is too small and cannot be provided in real time to the host, then the TX state can be kept as active for a set time with a hysteresis timer before it sends an update of the state. As an example, it might be that there was a short burst of TX and the modem could not provide the start and end notification in time. The modem should send up notification when the TX traffic starts and should continue to monitor its TX traffic during the hysteresis timer. If no more TX traffic was generated within the timer's timeframe, then it should report that TX traffic has ended.

This is very useful in scenarios where both Wi-Fi and LTE are connected. If both LTE and Wi-Fi are in a transmitting state and proximity was detected, then Wi-Fi back off may be required. If LTE is not in transmitting state but Wi-Fi is, then Wi-Fi back off may not be required. This applies to general Wi-Fi/LTE connection and mobile hot spot scenarios.

The Wi-Fi back off mechanism and command is out of scope of this specification.

OEMs that use this command should be aware of the potential power impact as the modem may be sending up transmission-related notifications at all times, including reduced power states. The OS, by default, will not allow this notification to awake the AP during Modern Standby to improve power performance.

Query

The InformationBuffer on MBIM_COMMAND_MSG is not used. MBIM_MS_TRANSMISSION_STATUS_INFO is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Set

The InformationBuffer on MBIM_COMMAND_MSG contains MBIM_MS_SET_TRANSMISSION_STATUS. MBIM_MS_TRANSMISSION_STATUS_INFO is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Unsolicited Events

Unsolicited events contain MBIM_MS_TRANSMISSION_STATUS_INFO and are sent when there is a change to the active over-the-air (OTA) channels. For example, if a modem started uploading packet data, it would be required to set up uplink channels when it uses the network data channel so that it can upload payloads. This would trigger the notification to be provided to the operating system.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MS_SET_TRANSMISSION_STATUS	Not applicable	Not applicable
Response	MBIM_MS_TRANSMISSION_STATUS_INFO	MBIM_MS_TRANSMISSION_STATUS_INFO	MBIM_MS_TRANSMISSION_STATUS_INFO

Data Structures

Query

The InformationBuffer on MBIM_COMMAND_MSG is not used. MBIM_MS_TRANSMISSION_STATUS_INFO is returned in the InformationBuffer of MBIM_COMMAND_DONE.

Set

The following MBIM_MS_SET_TRANSMISSION_STATUS structure shall be used in the InformationBuffer.

Offset	Size	Field	Type	Description

Offset	Size	Field	Type	Description
0	4	ChannelNotification	MBIM_MS_TRANSMISSION_STATUS_NOTIFICATION	For more information, see the MBIM_MS_TRANSMISSION_STATUS_NOTIFICATION table.
4	4	HysteresisTimer	UINT32	Hysteresis Indicator that is used by the modem to determine when to send the MBIMMsTransmissionStateInactive to the host. This value is the timer the modem sees as a continuous no-transmit activity before it sends an OFF indicator to host. This timer should be set in seconds, ranging from 1 second to 5 seconds.

The following MBIM_MS_TRANSMISSION_STATUS_NOTIFICATION structure is used in the preceding table. It specifies whether modem channel transmission is disabled or enabled.

Type	Value	Description
MBIMMsTransmissionNotificationDisabled	0	Modem channel transmission status notification disabled.
MBIMMsTransmissionNotificationEnabled	1	Modem channel transmission status notification enabled.

Response

The following MBIM_MS_TRANSMISSION_STATUS_INFO structure is used for response.

Offset	Size	Field	Type	Description
0	4	ChannelNotification	MBIM_MS_TRANSMISSION_STATUS_NOTIFICATION	For more information, see the MBIM_MS_TRANSMISSION_STATUS_NOTIFICATION table.
4	4	TransmissionStatus	MBIM_MS_TRANSMISSION_STATUS	For more information, see the MBIM_MS_TRANSMISSION_STATUS table. This indicates whether the modem has TX traffic every 5 seconds.
8	4	HysteresisTimer	UINT32	Hysteresis Indicator that is used by the modem to determine when to send the MBIMMsTransmissionStateInactive to the host. This value is the timer the modem sees as a continuous no-transmit activity before it sends an OFF indicator to host. This timer should be set in seconds, ranging from 1 second to 5 seconds.

The following MBIM_MS_TRANSMISSION_STATUS structure is used in the preceding table. It indicates whether modem is having TX traffic every 5 seconds.

Type	Value	Description
MBIMMsTransmissionStateInactive	0	The modem was not actively transmitting data without any continuous lapse of transmission for the last HysteresisTimer value.
MBIMMsTransmissionStateActive	1	The modem was actively transmitting data.

Notification

For more information, see the MBIM_MS_TRANSMISSION_STATUS_INFO table.

Status Codes

Error Code	Description
MBIM_STATUS_SUCCESS	The request was successfully processed.
MBIM_STATUS_BUSY	The device is currently busy.
MBIM_STATUS_FAILURE	The request failed.
MBIM_STATUS_NO_DEVICE_SUPPORT	Device does not support this command.
MBIM_STATUS_INVALID_PARAMETERS	The operation failed because of invalid parameters.
MBIM_STATUS_OPERATION_NOT_ALLOWED	The operation failed because the operation is not allowed.

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to the device Cellular modem driver and run the test: Win6_4.MB.GSM.Data.TestSAR.

This test contains the following tests:

Test name	Description
QuerySarConfig	This test verifies the test can successfully query SAR configurations.
SetSarConfig	This test verifies the test can successfully set SAR configurations.
QuerySarTransmissionStatus	This test verifies the test can successfully query SAR transmission status.
SetSarTransmissionStatus	This test verifies the test can successfully set SAR transmission status.

WinRT API

MobileBroadbandSarManager

Log Analysis

Logs can be collected and decoded using these instructions: MB Collecting Logs.

Important providers and corresponding keywords

Microsoft-Windows-WWAN-SVC-EVENTS (3cb40aaa-1145-4fb8-b27b-7e30f0454316)

Keywords for filtering:

1. SarConfig
2. CWwanSar::OnNdisNotification
3. LoadSemiStaticOEMSARTable
4. AttemptAutoConfigureSAR
5. PreCheckSemiStaticOEMSARTable
6. WwanIntfOpcodeSarConfig
7. WwanIntfOpcodeSarTransmissionStatus
8. WwanMsmEventTypeSarConfig
9. WwanMsmEventTypeSarTransmissionStatus

MobileBroadband WinRT WPP (56dd9c57-06cc-48ba-b123-876a6495ba13)

Keywords for filtering: MobileBroadbandSarManager

WwanProtDIM (3a07e1ba-3a6b-49bf-8056-c105b54dd7fb)

Keywords for filtering:

1. NDIS_WWAN_SAR_CONFIG_INFO
2. SarMode
3. NDIS_WWAN_SAR_TRANSMISSION_STATUS_INFO
4. HysteresisTimer

See Also

Customize a Specific Absorption Rate (SAR) mapping table

MB base stations information query support

Article • 03/14/2023

Overview

The base stations information query interface is used to provide location based services with cellular base station information, such as *Base Station ID*, *Time Advance*, and other parameters that can be used to compute the geographical position of the mobile subscriber. The information gathered pertains to the cellular base station currently serving the subscriber, as well as neighboring cellular base stations.

This topic defines the base stations information query interface for Windows, as the MBIM 1.0 specification does not provide this information through any existing CIDs. This interface is available in Windows 10, version 1709 and later.

Serving and neighbor cell parameters are retrieved via Query/Response operations. A notification is also defined in this topic to indicate that the location of the device within the cellular network has changed.

MBIM_CID_BASE_STATIONS_INFO

This command retrieves information about the serving and neighbor cells known to the modem.

Service: **MBB_UUID_BASIC_CONNECT_EXTENSIONS**

Service UUID: **3d01dcc5-fef5-4d05-0d3a-bef7058e9aaf**

CID	Command code	Minimum OS version
MBIM_CID_BASE_STATIONS_INFO	11	Windows 10, version 1709

Parameters

Type	Set	Query	Notification
Command	Not applicable	MBIM_BASE_STATIONS_INFO_REQ	Not applicable
Response	Not applicable	MBIM_BASE_STATIONS_INFO	Not applicable

Query

The InformationBuffer of MBIM_COMMAND_MSG contains an MBIM_BASE_STATIONS_INFO_REQ struture. The InformationBuffer of MBIM_COMMAND_DONE contains an MBIM_BASE_STATIONS_INFO structure.

MBIM_BASE_STATIONS_INFO_REQ

The MBIM_BASE_STATIONS_INFO_REQ structure shall be used in the InformationBuffer for queries. It is used to configure aspects of the cell information, such as the maximum number of neighbor cell measurements, to send in response.

Offset	Size	Field	Type	Description
0	4	MaxGSMCount	SIZE	The maximum number of entries of GSM neighboring cells returned in the GSM network measurement reports of MBIM_GSM_NMR. Default capacity is 15.
4	4	MaxUMTSCount	SIZE	The maximum number of entries of UMTS neighboring cells returned in the UMTS measured results list in MBIM_UMTS_MRL. Default capacity is 15.
8	4	MaxTDSCDMACount	SIZE	The maximum number of entries of TDSCDMA neighboring cells returned in the TDSCDMA measured results list in MBIM_TDSCDMA_MRL. Default capacity is 15.
12	4	MaxLTECount	SIZE	The maximum number of entries of LTE neighboring cells returned in the LTE measured results list of MBIM_LTE_MRL. Default capacity is 15.
16	4	MaxCDMACount	SIZE	The maximum number of entries of CDMA cells returned in the CDMA measured results list in MBIM_CDMA_MRL. This list includes both serving and neighboring cells. Default capacity is 12.

Set

Not applicable.

Response

The MBIM_BASE_STATIONS_INFO structure shall be used in the InformationBuffer of MBIM_COMMAND_DONE for responses.

MBIM_BASE_STATIONS_INFO

The MBIM_BASE_STATIONS_INFO structure contains information about both serving and neighboring base stations.

Offset	Size	Field	Type	Description
0	4	SystemType	MBIM_DATA_CLASS	Indicates the system type (or types) for which serving cell information is valid. This member is a bitmask of one or more system types as defined in the MBIM_DATA_CLASS.
4	4	GSMServingCellOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing the GSM serving cell information. This member can be NULL when the technology of the serving cell is not GSM.
8	4	GSMServingCellSize	SIZE(0-44)	The size, in bytes, used for MBIM_GSM_SERVING_CELL_INFO.
12	4	UMTSServingCellOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing the UMTS serving cell information. This member can be NULL when the technology of serving cell is not UMTS.
16	4	UMTSServingCellSize	SIZE(0-60)	The size, in bytes, used for MBIM_UMTS_SERVING_CELL_INFO.
20	4	TDSCDMAServingCellOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing the TDSCDMA serving cell information. This member can be NULL when the technology of serving cell is not TDSCDMA.
24	4	TDSCDMAServingCellSize	SIZE(0-48)	The size, in bytes, used for MBIM_TDSCDMA_SERVING_CELL_INFO.
28	4	LTEServingCellOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing the LTE serving cell information. This member can be NULL when the technology of serving cell is not LTE.

Offset	Size	Field	Type	Description
32	4	LTEServingCellSize	SIZE(0-48)	The size, in bytes, used for MBIM_LTE_SERVING_CELL_INFO.
36	4	GSMNmrOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing the GSM Network Measurement report. This member can be NULL when no GSM neighboring network is returned in the measurement report.
40	4	GSMNmrSize	SIZE	The total size, in bytes, of the buffer containing the GSM Network Measurement report in the format of MBIM_GSM_NMR.
44	4	UMTSMrlOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing UMTS measured results list. This member can be NULL when no UMTS neighboring network is returned in the measurement report.
48	4	UMTSMrlSize	SIZE	The total size, in bytes, of the buffer containing the UMTS measured results list in the format of MBIM_UMTS_MRL.
52	4	TDSCDMAMrlOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing TDSCDMA measured results list. This member can be NULL when no TDSCDMA neighboring network is returned in the measurement report.
56	4	TDSCDMAMrlSize	SIZE	The total size, in bytes, of the buffer containing the TDSCDMA measured results list in the format of MBIM_TDSCDMA_MRL.
60	4	LTEMrlOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing the LTE measured results list. This member can be NULL when no LTE neighboring network is returned in the measurement report.

Offset	Size	Field	Type	Description
64	4	LTEMrlSize	SIZE	The total size, in bytes, of the buffer containing the LTE measured results list in the format of MBIM_LTE_MRL.
68	4	CDMAMrlOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing CDMA measured results list. This member can be NULL when no CDMA neighboring network is returned in the measurement report.
72	4	CDMAMrlSize	SIZE	The total size, in bytes, of the buffer containing the CDMA measured results list in the format of MBIM_CDMA_MRL.
76		DataBuffer	DATABUFFER	The data buffer containing *GSMServingCell*, *UMTSServingCell*, *TDSCDMAServingCell*, *LTEServingCell*, *GSMNmr*, *UMTSMrl*, *TDSCDMAMrl*, *LTEMrl*, and *CDMAMrl*.

GSM cell data structures

MBIM_GSM_SERVING_CELL_INFO

The MBIM_GSM_SERVING_CELL_INFO structure contains information about the GSM serving cell.

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.
8	4	LocationAreaCode	UINT32	The Location Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.

Offset	Size	Field	Type	Description
12	4	CellID	UINT32	The Cell ID (0-65535). Use 0xFFFFFFFF when this information is not available.
16	4	TimingAdvance	UINT32	The Timing Advance (0-255) in bit periods, where a bit period is 48/13µs. Use 0xFFFFFFFF when this information is not available.
20	4	ARFCN	UINT32	The Absolute Radio Frequency Channel Number of the serving cell (0-1023). Use 0xFFFFFFFF when this information is not available.
24	4	BaseStationId	UINT32	The Base Station ID - the base station color code and the network identity code. Use 0xFFFFFFFF when this information is not available.

28	4	RxLevel	UINT32	The received signal strength of the serving cell (0-63), where
				`X = 0, if RSS < -110 dBm`
				`X = 63, if RSS > -47 dBm`
				`X = integer [RSS + 110], if -110 <= RSS <= -47`
				Use 0xFFFFFFFF when this information is not available.

32		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

MBIM_GSM_NMR

The MBIM_GSM_NMR structure contains the network measurement report (NMR) of neighboring GSM cells.

Offset	Size	Field	Type	Description
0	4	ElementCount (EC)	UINT32	The count of NMR entries following this element.
4		DataBuffer	DATABUFFER	The array of NMR records, each specified as an MBIM_GSM_NMR_INFO structure.

MBIM_GSM_NMR_INFO

The MBIM_GSM_NMR_INFO structure contains information about a neighboring GSM cell.

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.
8	4	LocationAreaCode	UINT32	The Location Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.
12	4	CellID	UINT32	The Cell ID (0-65535). Use 0xFFFFFFFF when this information is not available.
16	4	ARFCN	UINT32	The Absolute Radio Frequency Channel Number of the serving cell (0-1023). Use 0xFFFFFFFF when this information is not available.
20	4	BaseStationId	UINT32	The radio base station ID of the serving cell (0-63). Use 0xFFFFFFFF when this information is not available.

24	4	RxLevel	UINT32	The received signal strength of the serving cell (0-63), where
				`X = 0, if RSS < -110 dBm`
				`X = 63, if RSS > -47 dBm`
				`X = integer [RSS + 110], if -110 <= RSS <= -47`
				Use 0xFFFFFFFF when this information is not available.

28		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

UMTS cell data structures

MBIM_UMTS_SERVING_CELL_INFO

The MBIM_UMTS_SERVING_CELL_INFO structure contains information about the UMTS serving cell.

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.
8	4	LocationAreaCode	UINT32	The Location Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.
12	4	CellID	UINT32	The Cell ID (0-268435455). Use 0xFFFFFFFF when this information is not available.
16	4	FrequencyInfoUL	UINT32	The Frequency Info Uplink (0-16383). Use 0xFFFFFFFF when this information is not available.
20	4	FrequencyInfoDL	UINT32	The Frequency Info Downlink (0-16383). Use 0xFFFFFFFF when this information is not available.
24	4	FrequencyInfoNT	UINT32	The Frequency Info for TDD (0-16383). Use 0xFFFFFFFF when this information is not available.
28	4	UARFCN	UINT32	The UTRA Absolute Radio Frequency Channel Number for the serving cell (0-16383). Use 0xFFFFFFFF when this information is not available.
32	4	PrimaryScramblingCode	UINT32	The Primary Scrambling Code of the serving cell (0-511). Use 0xFFFFFFFF when this information is not available.
36	4	RSCP	INT32	The Received Signal Code Power of the serving cell. The range is -120 to -25, in units of 1dBm. Use 0 when this information is not available.

Offset	Size	Field	Type	Description
40	4	ECNO	INT32	The signal to noise ratio of the serving cell; the ratio of the received energy per PN chip for the CPICH to the total received. The range is -50 to 0, in units of 1dBm. Use 1 when this information is not available.
44	4	PathLoss	UINT32	The path loss of the serving cell (46-173). Use 0xFFFFFFFF when this information is not available.
48		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

MBIM_UMTS_MRL

The MBIM_UMTS_MRL structure contains the measured results list (MRL) of neighboring UMTS cells.

Offset	Size	Field	Type	Description
0	4	ElementCount (EC)	UINT32	The count of MRL entries following this element.
4		DataBuffer	DATABUFFER	The array of MRL records, each specified as an MBIM_UMTS_MRL_INFO structure.

MBIM_UMTS_MRL_INFO

The MBIM_UMTS_MRL_INFO structure contains information about a neighboring UMTS cell.

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.

Offset	Size	Field	Type	Description
8	4	LocationAreaCode	UINT32	The Location Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.
12	4	CellID	UINT32	The Cell ID (0-268435455). Use 0xFFFFFFFF when this information is not available.
16	4	UARFCN	UINT32	The UTRA Absolute Radio Frequency Channel Number for the serving cell (0-16383). Use 0xFFFFFFFF when this information is not available.
20	4	PrimaryScramblingCode	UINT32	The Primary Scrambling Code of the serving cell (0-511). Use 0xFFFFFFFF when this information is not available.
24	4	RSCP	INT32	The Received Signal Code Power of the serving cell. The range is -120 to -25, in units of 1dBm. Use 0 when this information is not available.
28	4	ECNO	INT32	The signal to noise ratio of the serving cell; the ratio of the received energy per PN chip for the CPICH to the total received. The range is -50 to 0, in units of 1dBm. Use 1 when this information is not available.
32	4	PathLoss	UINT32	The path loss of the serving cell (46-173). Use 0xFFFFFFFF when this information is not available.
36		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

TDSCDMA cell data structures

MBIM_TDSCDMA_SERVING_CELL_INFO

The MBIM_TDSCDMA_SERVING_CELL_INFO structure contains information about the TDSCDMA serving cell.

Offset	Size	Field	Type	Description

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.
8	4	LocationAreaCode	UINT32	The Location Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.
12	4	CellID	UINT32	The Cell ID (0-268435455). Use 0xFFFFFFFF when this information is not available.
16	4	UARFCN	UINT32	The UTRA Absolute Radio Frequency Channel Number for the serving cell (0-16383). Use 0xFFFFFFFF when this information is not available.
20	4	CellParameterID	UINT32	The Cell parameter ID (0-127). Use 0xFFFFFFFF when this information is not available.
24	4	TimingAdvance	UINT32	The Timing Advance (0-1023). This member is the same value for all timeslots. Use 0xFFFFFFFF when this information is not available.
28	4	RSCP	INT32	The Received Signal Code Power of the serving cell. The range is -120 to -25, in units of 1dBm in Q8 L3 filtered. Use 0xFFFFFFFF when this information is not available.
32	4	PathLoss	UINT32	The path loss of the serving cell (46-158). Use 0xFFFFFFFF when this information is not available.
36		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

MBIM_TDSCDMA_MRL

The MBIM_TDSCDMA_MRL structure contains the measured results list (MRL) of neighboring TDSCDMA cells.

Offset	Size	Field	Type	Description

Offset	Size	Field	Type	Description
0	4	ElementCount (EC)	UINT32	The count of MRL entries following this element.
4		DataBuffer	DATABUFFER	The array of MRL records, each specified as an MBIM_TDSCDMA_MRL_INFO structure.

MBIM_TDSCDMA_MRL_INFO

The MBIM_TDSCDMA_MRL_INFO structure contains information about a neighboring TDSCDMA cell.

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.
8	4	LocationAreaCode	UINT32	The Location Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.
12	4	CellID	UINT32	The Cell ID (0-268435455). Use 0xFFFFFFFF when this information is not available.
16	4	UARFCN	UINT32	The UTRA Absolute Radio Frequency Channel Number for the serving cell (0-16383). Use 0xFFFFFFFF when this information is not available.
20	4	CellParameterID	UINT32	The Cell parameter ID (0-127). Use 0xFFFFFFFF when this information is not available.
24	4	TimingAdvance	UINT32	The Timing Advance (0-1023). This member is the same value for all timeslots. Use 0xFFFFFFFF when this information is not available.
28	4	RSCP	INT32	The Received Signal Code Power of the serving cell. The range is -120 to -25, in units of 1dBm in Q8 L3 filtered. Use 0xFFFFFFFF when this information is not available.

Offset	Size	Field	Type	Description
32	4	PathLoss	UINT32	The path loss of the serving cell (46-158). Use 0xFFFFFFFF when this information is not available.
36		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

LTE cell data structures

MBIM_LTE_SERVING_CELL_INFO

The MBIM_LTE_SERVING_CELL_INFO structure contains information about the LTE serving cell.

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.
8	4	CellID	UINT32	The Cell ID (0-268435455). Use 0xFFFFFFFF when this information is not available.
12	4	EARFCN	UINT32	The Radio Frequency Channel Number of the serving cell (0-65535). Use 0xFFFFFFFF when this information is not available.
16	4	PhysicalCellID	UINT32	The Physical Cell ID (0-503). Use 0xFFFFFFFF when this information is not available.
20	4	TAC	UINT32	The Tracking Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.
24	4	RSRP	INT32	The Average Reference Signal Received Power. The range is -140 to -44, in units of 1dBm. Use 0xFFFFFFFF when this information is not available.
28	4	RSRQ	INT32	The Average Reference Signal Received Quality. The range is -20 to -3, in units of 1dBm. Use 0xFFFFFFFF when this information is not available.

Offset	Size	Field	Type	Description
32	4	TimingAdvance	UINT32	The Timing Advance (0-255). Use 0xFFFFFFFF when this information is not available.
36		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

MBIM_LTE_MRL

The MBIM_LTE_MRL structure contains the measured results list (MRL) of neighboring LTE cells.

Offset	Size	Field	Type	Description
0	4	ElementCount (EC)	UINT32	The count of MRL entries following this element.
4		DataBuffer	DATABUFFER	The array of MRL records, each specified as an MBIM_LTE_MRL_INFO structure.

MBIM_LTE_MRL_INFO

The MBIM_LTE_MRL_INFO structure contains information about a neighboring LTE cell.

Offset	Size	Field	Type	Description
0	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called *ProviderId* that represents the network provider identity. This string is a concatenation of a three-digit Mobile Country Code (MCC) and a two or three-digit Mobile Network Code (MNC). This member can be NULL when no *ProviderId* information is returned.
4	4	ProviderIdSize	SIZE(0-12)	The size used for *ProviderId*.
8	4	CellID	UINT32	The Cell ID (0-268435455). Use 0xFFFFFFFF when this information is not available.
12	4	EARFCN	UINT32	The Radio Frequency Channel Number of the serving cell (0-65535). Use 0xFFFFFFFF when this information is not available.
16	4	PhysicalCellID	UINT32	The Physical Cell ID (0-503). Use 0xFFFFFFFF when this information is not available.

Offset	Size	Field	Type	Description
20	4	TAC	UINT32	The Tracking Area Code (0-65535). Use 0xFFFFFFFF when this information is not available.
24	4	RSRP	INT32	The Average Reference Signal Received Power. The range is -140 to -44, in units of 1dBm. Use 0xFFFFFFFF when this information is not available.
28	4	RSRQ	INT32	The Average Reference Signal Received Quality. The range is -20 to -3, in units of 1dBm. Use 0xFFFFFFFF when this information is not available.
32		DataBuffer	DATABUFFER	The data buffer containing *ProviderId*.

CDMA cell data structures

MBIM_CDMA_MRL

The MBIM_CDMA_MRL structure contains the measured results list (MRL) of both serving and neighboring CDMA cells.

Offset	Size	Field	Type	Description
0	4	ElementCount (EC)	UINT32	The count of MRL entries following this element.
4		DataBuffer	DATABUFFER	The array of MRL records, each specified as an MBIM_CDMA_MRL_INFO structure.

MBIM_CDMA_MRL_INFO

The MBIM_CDMA_MRL_INFO data structure is designed for the CDMA2000 network type. There can be more than one CDMA2000 serving cell at the same time. Both serving cells and neighboring cells will be returned in the same list. The **ServingCellFlag** field indicates whether a cell is a serving cell or not.

Offset	Size	Field	Type	Description
0	4	ServingCellFlag	UINT32	Indicates whether this is a serving cell. A value of 1 indicates a serving cell, while a value of 0 indicates a neighboring cell. There may be more than one serving cell at a time (notably while in a call).
4	4	NID	UINT32	The Network ID (0-65535). Use 0xFFFFFFFF when this information is not available.

Offset	Size	Field	Type	Description
8	4	SID	UINT32	The System ID (0-32767). Use 0xFFFFFFFF when this information is not available.
12	4	BaseStationId	UINT32	The Base Station ID (0-65535). Use 0xFFFFFFFF when this information is not available.
16	4	BaseLatitude	UINT32	The Base Station Latitude (0-4194303). This is encoded in units of 0.25 seconds, expressed in two's complement representation within the low 22 bits of the DWORD. As a signed value, North latitudes are positive. Use 0xFFFFFFFF when this information is not available.
20	4	BaseLongitude	UINT32	The Base Station Longitude (0-8388607). This is encoded in units of 0.25 seconds, expressed in two's complement representation within the low 23 bits of the DWORD. As a signed value, East longitudes are positive. Use 0xFFFFFFFF when this information is not available.
24	4	RefPN	UINT32	The Base Station PN Number (0-511). Use 0xFFFFFFFF when this information is not available.
28	4	GPSSeconds	UINT32	The GPS seconds, or the time at which this arrived from the base station. Use 0xFFFFFFFF when this information is not available.
32	4	PilotStrength	UINT32	The Signal strength of the pilot (0-63). Use 0xFFFFFFFF when this information is not available.

Unsolicited Event

Not applicable.

Status codes

This CID uses Generic Status Codes (see Use of Status Codes in Section 9.4.5 of the public USB MBIM standard).

MBIM_CID_LOCATION_INFO_STATUS

This CID retrieves the status of the cellular information which indicates the location of the device. It may also be used to deliver an unsolicited notification when the location information changes.

Service: **MBB_UUID_BASIC_CONNECT_EXTENSIONS**

Service UUID: **3d01dcc5-fef5-4d05-0d3a-bef7058e9aaf**

CID	Command code	Minimum OS version
MBIM_CID_LOCATION_INFO_STATUS	12	Windows 10, version 1709

> ⓘ Note
>
> MBIM_CID_LOCATION_INFO_STATUS is defined starting in Windows 10, version 1709, but is not currently supported by the OS. A modem can send this command as a notification, but the OS does not currently handle it.

Parameters

Type	Set	Query	Notification
Command	Not applicable	Not applicable	Not applicable
Response	Not appliable	MBIM_LOCATION_INFO	MBIM_LOCATION_INFO

Query

The InformationBuffer of the MBIM_COMMAND_MSG is not used. The InformationBuffer of the MBIM_COMMAND_DONE contains an MBIM_LOCATION_INFO structure.

Set

Not applicable.

Response

MBIM_LOCATION_INFO

Offset	Size	Field	Type	Description
0	4	LocationAreaCode	UINT32	The GSM/UMTS area code of the current location. Return 0xFFFFFFFF when the current system type is not applicable.

Offset	Size	Field	Type	Description
4	4	TrackingAreaCode	UINT32	The LTE tracking area code of the current location. Return 0xFFFFFFFF when the current system type is not applicable.
8	4	CellID	UINT32	The ID of the cellular tower. Return 0xFFFFFFFF when *CellID* is not available.

Unsolicited Events

The event InformationBuffer contains an MBIM_LOCATION_INFO structure.

This event is sent if the value of *Location Area Code/Tracking Area Code* changes to a valid value. This event is not sent when *CellID* changes or when *Location Area Code/Tracking Area Code* becomes invalid.

Status codes

This CID uses Generic Status Codes (see Use of Status Codes in Section 9.4.5 of the public USB MBIM standard).

OID_WWAN_BASE_STATIONS_INFO

The NDIS equivalent for MBIM_CID_BASE_STATIONS_INFO is OID_WWAN_BASE_STATIONS_INFO.

MB NITZ support

Article • 03/14/2023

Overview

Starting in Windows 10, version 1903, Windows supports Network Identity and Time Zone (NITZ) at the OS level for mobile broadband (MBB) devices. In previous versions of Windows, the only network time available at the OS level was Network Time Protocol (NTP), even though NITZ was supported at the modem level by all 3GPP-compliant modems. With NITZ support, Windows is able to receive unsolicited NITZ notifications from modems and publish necessary events to notify consumers of the NITZ timestamps.

For MBIM functions, no additional NITZ-related setup and provisioning is required. As long as a data connection is established over a cellular bearer, a modem can notify the OS any time it has received a NITZ timestamp from the network. Modems can receive NITZ notifications from the network infrastructure based on the mobile operator's own defined cadence and schedule, within the 3GPP specifications. NITZ notifications are unsolicited. Upon receiving the NITZ notification, the OS publishes the notification that NITZ data is available.

NDIS interface extension

The following OID has been defined to support NITZ.

- OID_WWAN_NITZ

MBIM service and CID values

Service name	UUID	UUID value
Microsoft Voice Extensions	UUID_VOICEEXTENSIONS	8d8b9eba-37be-449b-8f1e-61cb034a702e

The following table specifies the UUID and command code for each CID, as well as whether the CID supports Set, Query, or Event (notification) requests. See each CID’s individual Section within this topic for more info about its parameters, data structures, and notifications.

CID	UUID	Command code	Set	Query	Notify
MBIM_CID_NITZ	UUID_VOICEEXTENSIONS	10	N	Y	Y

MBIM_CID_NITZ

Parameters

Operation	Set	Query	Notification
Command	Not applicable	Not applicable	Not applicable
Response	Not applicable	MBIM_NITZ_INFO	MBIM_NITZ_INFO

Query

Queries the current network time. The InformationBuffer of MBIM_COMMAND_MSG is not used. The following MBIM_NITZ_INFO structure is used in the InformationBuffer of MBIM_COMMAND_DONE.

MBIM_NITZ_INFO

Offset	Size	Field	Type	Description
0	4	Year	UINT32	The year as an integer. For example, **2014**.
4	4	Month	UINT32	The month (1..12), where January == 1.
8	4	Day	UINT32	The day of the month, (1..31).
12	4	Hour	UINT32	The hour, (0..23).
16	4	Minute	UINT32	The minute, (0..59).
20	4	Second	UINT32	The second, (0..59).
24	4	TimeZoneOffsetMinutes	UINT32	The time zone offset, in minutes, from UTC. This value includes any adjustment for the current state of daylight saving time. This value should be set to 0xFFFFFFFF when time zone info is not available.

Offset	Size	Field	Type	Description
28	4	DaylightSavingTimeOffsetMinutes	UINT32	The offset for daylight saving time, in minutes. This value should be set to 0xFFFFFFFF when daylight saving time is not available.
32	4	DataClasses	UINT32	Data classes supported by this network. If this information is not available, this field should be set to MBIMDataClassNone.

Set

Not applicable.

Response

The InformationBuffer in MBIM_COMMAND_DONE contains an MBIM_NITZ_INFO structure.

Unsolicited Events

This unsolicited event provides the current network time and time zone information.

Status Codes

This CID only uses generic status codes defined in Section 9.4.5 of the MBIM specification revision 1.0 .

OID Definitions

- OID_WWAN_NITZ

Hardware Lab Kit (HLK) Tests

See Steps for installing HLK .

In HLK Studio connect to hte device Cellular modem driver and run the test: TestNitzInfo - GSM.

Manual Tests

[NITZ] Time update while roaming on Cellular

1. Place the Cobalt device in an RF cage with Cellular disabled.
2. Enable Airplane mode.
3. Disable Ethernet and all other connections.
4. Set the time mode to manual.
5. Set the time to 11:15AM 10/15/2016 UTC.
6. Verify that the time is set to the value given in the system tray.
7. Set the time mode to automatic.
8. Turn on Cellular.
9. Wait for the device to receive the NITZ information from the simulated cellular base station.
10. Verify that the time is set to the value sent by the simulated base station.

MB modem logging with DSS

Article • 03/14/2023

ⓘ Note

If you are planning for your modem to support `MBIM_CID_MODEM_LOGGING_CONFIG`, please provide feedback on this page so that we can best support you. This CID is currently experimental and has not been tested with a modem yet, as none support it.

This topic describes a new standard Windows mobile broadband (MBB) logging interface through Microsoft extensions to the USB MBIM 1.0 specification, available in Windows 10, version 1903 and later.

With this new logging interface, the OS can inform the MBB device to start, stop, and flush the logs to the OS file system through MBIM CID commands. Given the non-IP nature of the modem's logging payload, the data channel that the MBB service uses to transmit logging payloads to the OS uses the MBB Data Service Stream (DSS). DSS is defined in the Mobile Broadband Interface Model (MBIM) 1.0 specification.

The OS abstracts the modem's diagnostic functionalities and configurations across the entire MBB ecosystem with a set of Windows-specific MBB logging configurations. These MBB logging configs enable a modem's vendor to map the OS MBB logging requirements to the appropriate internal logging configurations. The logging configurations abstracted and defined by the OS include the MBB logging verbosity levels and the maximum flush time.

A modem keeps filling its logging buffer, up to the max buffer size, until the segment is filled and the MBIM framework transmits the segment to the OS, or it flushes the content of its buffer when the maximum flush time is reached (even if the segment is not filled). The OS defines a set of standard Windows MBB logging configuration levels, described later in this topic. Each config level specifies an OS abstraction of MBB logging details and verbosity.

OS abstraction of MBB config levels is mapped to the appropriate internal modem configuration by modems. The OS does not provide any additional configuration payloads, such as logging filters or masks, to modems other than the OS MBB configuration level.

For modems that support MBB logging, all MBB logging config levels except for MBIMLoggingLevelOem must be present on all BSP variants. In other words, the IHV or OEM must support PROD or LAB levels of MBB logging in both production and R&D versions of the BSP. LAB levels of MBB logging can only be disabled from the OS.

This new logging interface's design uses the control channel to set the logging parameters, and uses the data channel to receive modem logs because the data channel is designed to transfer bulk modem data. The advantage of this design is that bulk data does not need to be transferred over the control channel, thus keeping device performance consistent. It also scales well for higher throughput. The data channel is operated by DSS commands. An example flow for a modem might look like this:

1. The OS sends the MBIM_CID_MODEM_LOGGING_CONFIG CID to the modem to configure logging parameters such as MaxSegmentSize, MaxFlushTime, and the LoggingLevel.

2. Once the OS receives a successful response from the modem, it sends the MBIM_CID_DSS_CONNECT DSS command to the modem with a specific GUID for modem logging, the MBIMDssLinkActivate state, and a unique DSS session ID.
3. Once it receives a success status code, the OS prepares to receive fragments from the modem. These fragments are called DataServiceSessionRead packets.
4. DataServiceSessionRead packets continue to arrive until the OS issues another MBIM_CID_DSS_CONNECT command with the same DSS session ID and an MBIMDSSLinkDeactivate state.

Once the modem writes any logs to the newly created data channel, the modem calls **MbbDeviceReceiveDeviceServiceSessionData**, the data from which is available to apps via the WinRT layer : **MobileBroadbandDeviceService**. The modem logs should be formatted as printable string data that can be redirected to an ETW session.

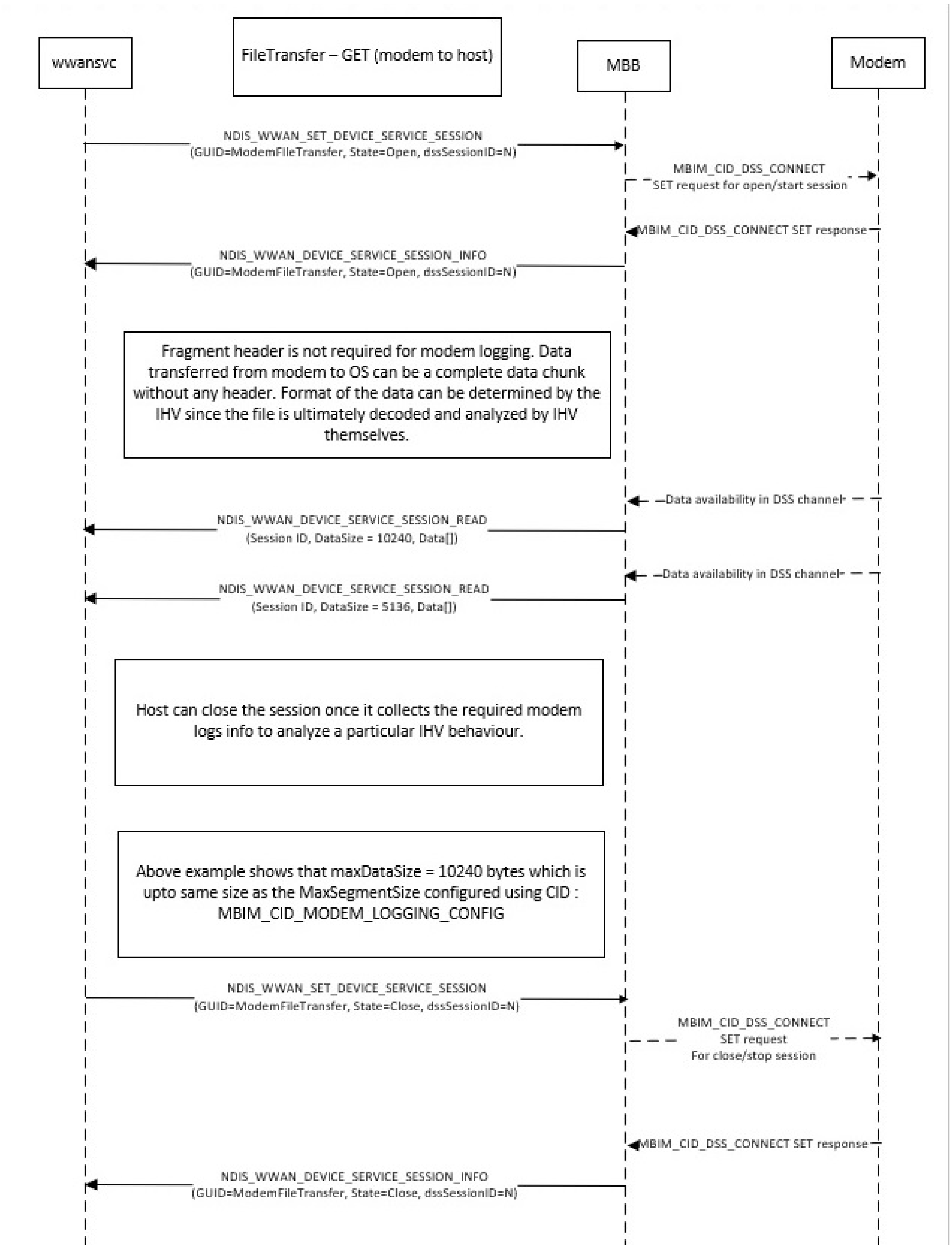
Modem logging data path

Moddem logging uses the MBIM Data Service Stream (DSS) to transfer the data for logging payloads. For more information about DSS, see Section 10.5.38 of the MBIM 1.0 specification .

When connecting or disconnecting from DSS, the following GUID is used for modem logging:

GUID	Value
ModemFileTransfer GUID	0EBB1CEB-AF2D-484D-8DF3-53BC51FD162C

The following flow diagram illustrates the DSS setup and tear down process.


wwansvc
FileTransfer – GET (modem to host)
MBB
Modem
NDIS_WWAN_SET_DEVICE_SERVICE_SESSION (GUID=ModemFileTransfer, State=Open, dssSessionID=N)
MBIM_CID_DSS_CONNECT SET request for open/start session
MBIM_CID_DSS_CONNECT SET response
NDIS_WWAN_DEVICE_SERVICE_SESSION_INFO (GUID=ModemFileTransfer, State=Open, dssSessionID=N)
Fragment header is not required for modem logging. Data transferred from modem to OS can be a complete data chunk without any header. Format of the data can be determined by the IHV since the file is ultimately decoded and analyzed by IHV themselves.
Data availability in DSS channel
NDIS_WWAN_DEVICE_SERVICE_SESSION_READ (Session ID, DataSize = 10240, Data[])
Data availability in DSS channel
NDIS_WWAN_DEVICE_SERVICE_SESSION_READ (Session ID, DataSize = 5136, Data[])
Host can close the session once it collects the required modem logs info to analyze a particular IHV behaviour.
Above example shows that maxDataSize = 10240 bytes which is upto same size as the MaxSegmentSize configured using CID : MBIM_CID_MODEM_LOGGING_CONFIG
NDIS_WWAN_SET_DEVICE_SERVICE_SESSION (GUID=ModemFileTransfer, State=Close, dssSessionID=N)
MBIM_CID_DSS_CONNECT SET request For close/stop session
MBIM_CID_DSS_CONNECT SET response
NDIS_WWAN_DEVICE_SERVICE_SESSION_INFO (GUID=ModemFileTransfer, State=Close, dssSessionID=N)

NDIS interface extension

The following OID has been defined in Windows 10, version 1903, to support modem logging.

- OID_WWAN_MODEM_LOGGING_CONFIG

MBIM service and CID values

Service name	UUID	UUID value
Microsoft Basic IP Connectivity Extensions	UUID_BASIC_CONNECT_EXTENSIONS	3d01dcc5-fef5-4d05-9d3a-bef7058e9aaf

The following table specifies the UUID and command code for each CID, as well as whether the CID supports Set, Query, or Event (notification) requests. See each CID's individual Section within this topic for more info about its parameters, data structures, and notifications.

CID	UUID	Command code	Set	Query	Notify
MBIM_CID_MODEM_LOGGING_CONFIG	UUID_BASIC_CONNECT_EXTENSIONS	TBD	Y	Y	Y

MBIM_CID_MODEM_LOGGING_CONFIG

This CID is used to configure the logs that are collected by the modem and how often they will be sent from the modem to the host over DSS. Logging must be configured before a logging session is started. Because this CID is part of connect extensions, it is optional for IHVs to support this CID. If an IHV supports modem logging via the DSS data channel, it must specify this as a capability. The capability can be advertised using the MBIM_BASIC_CID_DEVICE_SERVICES CID.

Parameters

Operation	Set	Query	Notification
Command	MBIM_MODEM_LOGGING_CONFIG	Not Applicable	Not applicable
Response	MBIM_MODEM_LOGGING_CONFIG	MBIM_MODEM_LOGGING_CONFIG	MBIM_MODEM_LOGGING_CONFIG

Query

Queries the current modem logging configuration. The InformationBuffer of MBIM_COMMAND_MSG is not used. The following MBIM_MODEM_LOGGING_CONFIG structure is used in the InformationBuffer of MBIM_COMMAND_DONE.

MBIM_MODEM_LOGGING_CONFIG

Offset	Size	Field	Type	Description
0	4	Version	UINT32	The version number of this structure. This field must be set to **1** for version 1 of this structure.
4	4	MaxSegmentSize	UINT32	Specifies the segment size, in kilobytes, for each fragment sent by the modem. If the maximum fragment size supported by the modem for Device Service Command exceeds the value set, then this value is set to the maximum supported segment size.

Offset	Size	Field	Type	Description
8	4	MaxFlushTime	UINT32	The time, in milliseconds, indicating the maximum time the modem waits before sending a log fragment. If the logs collected don't reach **MaxSegmentSize** within the **MaxFlushTime** duration since the last log fragment sent, then a log fragment is sent regardless of its size. If there is no logging data, then no notification is sent. If the device cannot handle smaller flush times, then the device returns the time that it can handle in the response. The response to a query or set contains the currently configured **MaxFlushTime**.
12	4	LevelConfig	MBIM_LOGGING_LEVEL_CONFIG	Configures the level for which logs are collected. The response to a query or set contains the currently configured **LevelConfig**.

> ⓘ Note
>
> If the modem is not able to provide log data to the OS at the requested **MaxSegmentSize** and **MaxFlushTimer**, it can choose its own values for these parameters and update the OS as a set response or an unsolicited event. The OS behavior does not change if **MaxSegmentSize** or **MaxFlushTimer** change, as it receives the data packets regardless and dumps them to a file.

The following MBIM_LOGGING_LEVEL_CONFIG enumeration is used in the preceding MBIM_MODEM_LOGGING_CONFIG structure.

Type	Value	Description
MBIMLoggingLevelProd	0	Intended for telemetry collection from a retail or production population. The resulting log should be capsule-sized and contains key modem or MBB state or failure information only.
MBIMLoggingLevelLabVerbose	1	Intended for the development of MBB products with low maturity. Verbose full-stack capture of modems. The resulting modem capture should enable the IHV to replay and fully recover the capture during the log.
MBIMLoggingLevelLabMedium	2	Intended for verification and field testing of MBB products with relative maturity and stability. The level of detail and verbosity provides enough data points for IHV engineers to triage most MBB failures.
MBIMLoggingLevelLabLow	3	Intended for self-host-level logging. Summary-level capture of full-stack capture modems. Enables a highlight-level understanding of the modem's state and OS interactions.
MBIMLoggingLevelOem	4	Reserved for OEM and IHV internal usage.

Set

A set command is used to set to configure the level, segment size, and maximum flush time for modem logging. An MBIM_MODEM_LOGGING_CONFIG structure is used in the InformationBuffer.

Response

The InformationBuffer in MBIM_COMMAND_DONE contains an MBIM_MODEM_LOGGING_CONFIG structure.

Unsolicited Events

Unsolicited events are supported for scenarios where the modem needs to inform the OS about internal changes. Currently, in Windows 10, version 1903, these scenarios do not occur.

Status Codes

This CID only uses generic status codes defined in Section 9.4.5 of the MBIM specification revision 1.0 .

DSS session behavior during inactivity

The following table describes how the DSS session behaves during various stages of inactivity:

Scenario	DSS session state
System sleep, modem-only sleep, reset and recovery	DSS session kept open
System shutdown, restart, hibernation	DSS session closed

MB 5G Operations Overview

Article • 03/14/2023

Windows 10, version 1903 is the first version of Windows to support a preview release of 5G mobile broadband driver development by IHV partners. *5G* is the friendly name for New Radio (NR), which was introduced in the 3GPP Release 15 specification. NR is a comprehensive set of standards that is envisioned to provide true long-term evolution to existing 4th generation LTE technologies, potentially covering all cellular communication needs from narrowband to ultra-broadband, and from nominal to mission-critical latency requirements. As a technology, 5G is expected to develop over a decade-long time frame.

This section describes the MBIM extensions first released in Windows 10 version 1903, which enable hardware partners to develop an MBB driver with data-class support for enhanced mobile broadband (eMBB) over 5G “non-standalone” EPC-based NR networks. The data-plane support and enablement for 5G throughput and commercialization requirements are not part of this Windows release and not described in this section.

In this section

MB 5G Operations Terminology

Windows 5G MBIM Interface

NDIS Interface for 5G Data Class Support

MBIMEx 2.0 – 5G NSA support

MBIMEx 3.0 – 5G SA Phase 1 support

MB 5G Operations Terminology

Article • 03/14/2023

This section uses the following terms:

Term	Definition
NR	New Radio. NR is the term used in 3GPP when referring to 5G.
MBB	Mobile broadband.
EPC	Enhanced Packet Core. The term used in 3GPP when referring to the LTE core network.
NGC	Next Generation Core. The term used in 3GPP when referring to the 5G core network. The NR-equivalent of EPC.
DC	Dual Connectivity. The network can support both LTE and 5G NR, including dual connectivity with which devices have simultaneous connections to LTE and NR.
SA	Standalone 5G. Refers to any NGC-based NR networks.
NSA	Non-standalone 5G. Refers to any EPC-based NR networks.
gNB	An NR radio base station that supports the NR air interface as well as connectivity to NGC.
RAT	Radio Access Technology.

Windows 5G MBIM Interface

Article • 03/14/2023

As of Windows 10, version 1903, 5G on the whole is still developing. From a network deployment perspective, 5G is expected to be deployed in two major phases:

- In Phase 1, most mobile network operators are expected to deploy 5G with the addition of 5G radio to the existing LTE radio and EPC core deployments, commonly known “nonstandalone 5G” networks.
- In Phase 2, mobile network operators are expected to replace EPCs and NGCs and densify the 5G radio deployment in parallel to enable true “standalone”, or NR-NGC-based 5G networks. Phase 2 interface extensions are not in scope in this topic or Windows release.

Interface extensions to support basic Phase 1 network requirements, or ”nonstandalone” EPC-based 5G networks, was introduced in Windows 10, version 1903. In order to be extensible and fully backward compatible with legacy modems, a new Microsoft MBIM extension version (2.0) is introduced.

The new Microsoft MBIM extension version is required because the MBIM 1.0 errata specification has a mechanism to add and advertise optional CIDs, but it lacks a mechanism to change the existing CIDs (new payloads or modified payload) or to introduce changes in any aspect that cannot be accommodated by optional CIDs. Each payload may consist of fixed sized members or dynamic sized (offset/size pairs) members. If one or more dynamically sized members exist, then the last member has a variable size buffer.

This spec also adds a new CID for the host to advertise its MBIM Release version and Extensions Release version to MBIM devices. For legacy drivers that are already in the field, this CID is optional so backward compatibility is fully maintained. For more details, see MBIMEx 2.0 – 5G NSA support.

NDIS Interface for 5G Data Class Support

Article • 03/14/2023

NDIS supports a revision number in the NDIS_HEADER. This permits adding new members to an OID message, which NDIS uses the optional service caps table in OID_WWAN_DEVICE_CAPS_EX.

The following NDIS OIDs and their data structures have been updated for 5G data class support.

- OID_WWAN_DEVICE_CAPS_EX
- OID_WWAN_REGISTER_STATE
- OID_WWAN_PACKET_SERVICE
- OID_WWAN_SIGNAL_STATE

The equivalent MBIM CID messages for these OIDs are described in MBIMEx 2.0 – 5G NSA support.

MBIMEx 2.0 – 5G NSA support

Article • 03/14/2023

Because the MBIM 1.0 errata specification lacks a mechanism to change existing CIDs with new or modified payloads, Windows 10, version 1903 introduces MBIM 1.0 Extension 2.0 to extend the interface to support 5G.

Versioning scheme

> **Note**
>
> In this section, the term *MBIMEx version* refers to the MBIM Extensions release number.

The host learns a device's MBIMEx version through two ways:

1. The MBIM EXTENDED FUNCTIONAL DESCRIPTOR.
2. The optional MBM_CID_VERSION message, if the device supports it and declares support for it.

If these two are different, the higher version dictates the MBIMEx version for the duration that the device stays enumerated to the host. The higher MBIMEx version is referred to as the device's *announced MBIMEx version*. A device's announced MBIMEx version can be lower than its native MBIMEx version, which is the highest MBIMEx version that the device supports. Devices can learn the host's MBIMEx version explicitly only via the MBIM_CID_VERSION message.

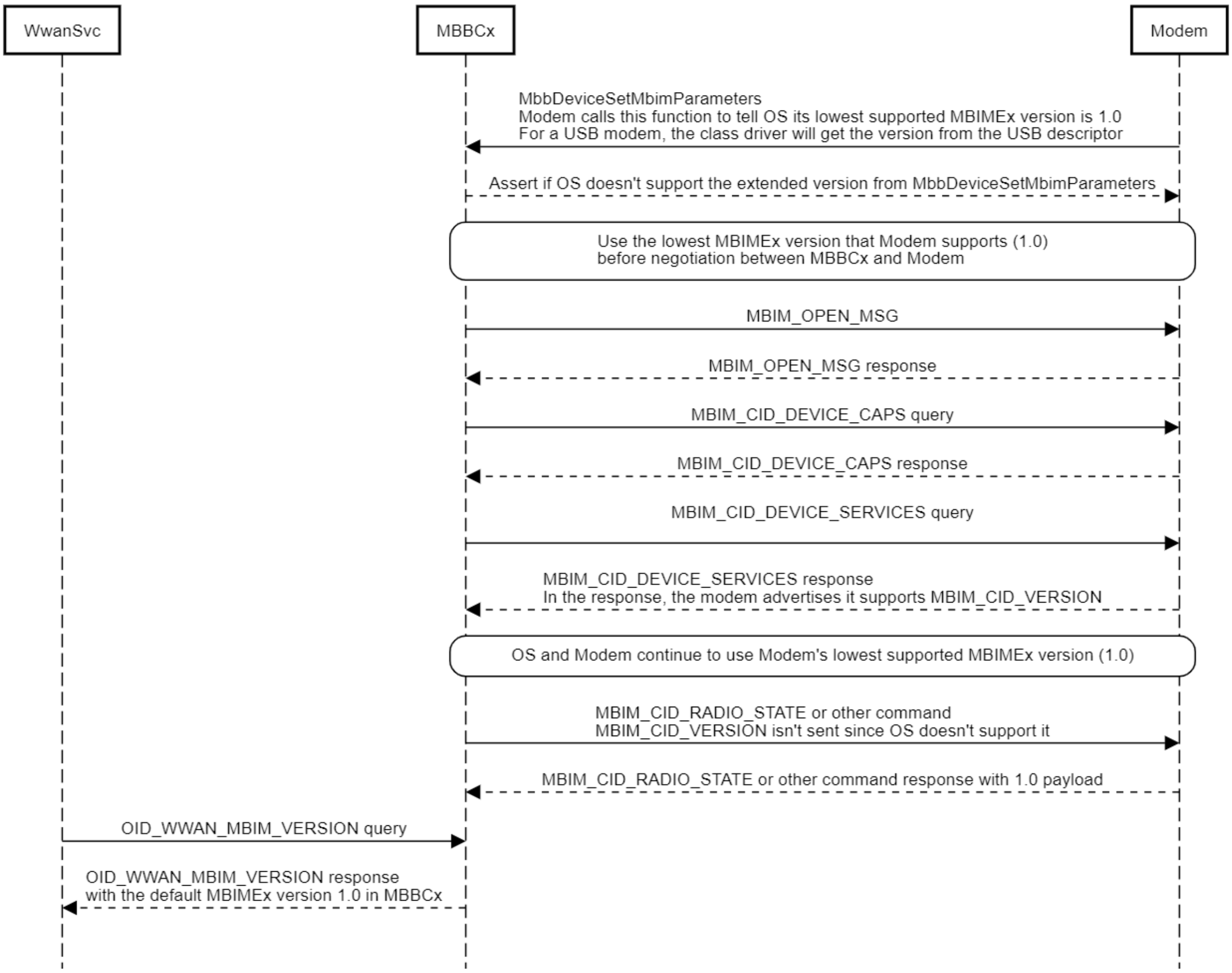
In any release, the host always queries the device for supported services and CIDs using MBIM_CID_DEVICE_SERVICES at the beginning of the device initialization sequence.

If a device supports MBIM_CID_VERSION and advertises its support in the MBIM_CID_DEVICE_SERVICES query response, then a host that does not understand MBIM_CID_VERSION or has an MBIMEx version lower than 2.0 ignores it. Meanwhile, a host that does understand MBIM_CID_VERSION and has a native MBIMEx version of 2.0 or higher sends a MBIM_CID_VERSION message to the device with the host's native MBIMEx version, and the CID is the first CID that is sent to the device after receiving the MBIM_CID_DEVICE_SERVICES response.

If the first CID that the device receives from the host after it responds to the MBIM_CID_DEVICE_SERVICES query is MBIM_CID_VERSION, the device knows the host's MBIMEx version.

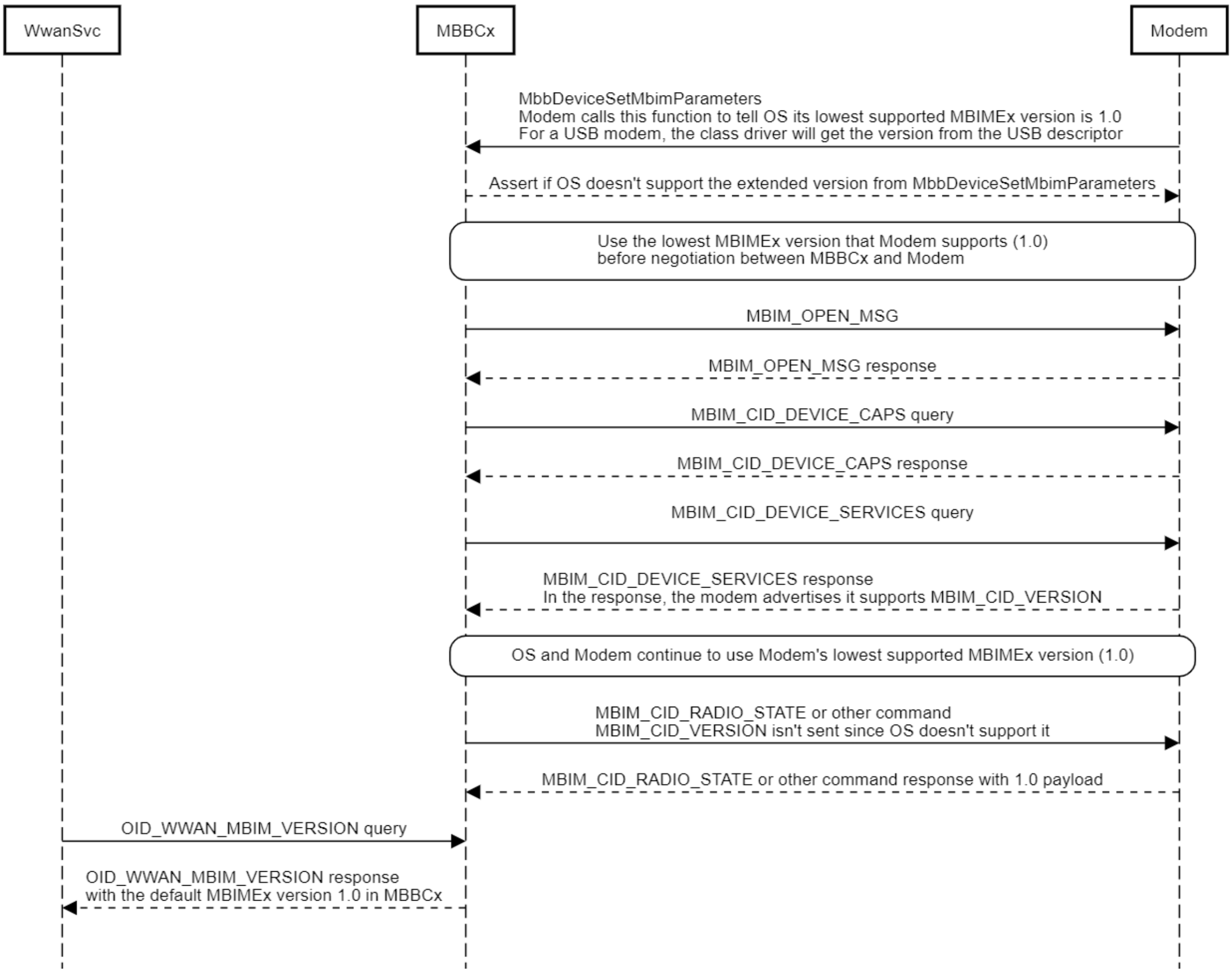
If the first CID that the device receives from the host after it responds to the MBIM_CID_DEVICE_SERVICES query is any other CID, then the device assumes that the host's native MBIMEx version is 1.0.

OS doesn't support MBIM_CID_VERSION and Modem's highest supported MBIMEx version is 3.0


WwanSvc
MBBCx
Modem
MbbDeviceSetMbimParameters
Modem calls this function to tell OS its lowest supported MBIMEx version is 1.0
For a USB modem, the class driver will get the version from the USB descriptor
Assert if OS doesn't support the extended version from MbbDeviceSetMbimParameters
Use the lowest MBIMEx version that Modem supports (1.0)
before negotiation between MBBCx and Modem
MBIM_OPEN_MSG
MBIM_OPEN_MSG response
MBIM_CID_DEVICE_CAPS query
MBIM_CID_DEVICE_CAPS response
MBIM_CID_DEVICE_SERVICES query
MBIM_CID_DEVICE_SERVICES response
In the response, the modem advertises it supports MBIM_CID_VERSION
OS and Modem continue to use Modem's lowest supported MBIMEx version (1.0)
MBIM_CID_RADIO_STATE or other command
MBIM_CID_VERSION isn't sent since OS doesn't support it
MBIM_CID_RADIO_STATE or other command response with 1.0 payload
OID_WWAN_MBIM_VERSION query
OID_WWAN_MBIM_VERSION response
with the default MBIMEx version 1.0 in MBBCx

If the device doesn't support MBIM_CID_VERSION, it will not respond to the MBIM_CID_DEVICE_SERVICES query with MBIM_CID_VERSION. Therefore the host will not send a MBIM_CID_VERSION message and assumes that the device's native MBIMEx version is 1.0.

OS' highest supported MBIMEx version is 3.0 and Modem doesn't support MBIM_CID_VERSION

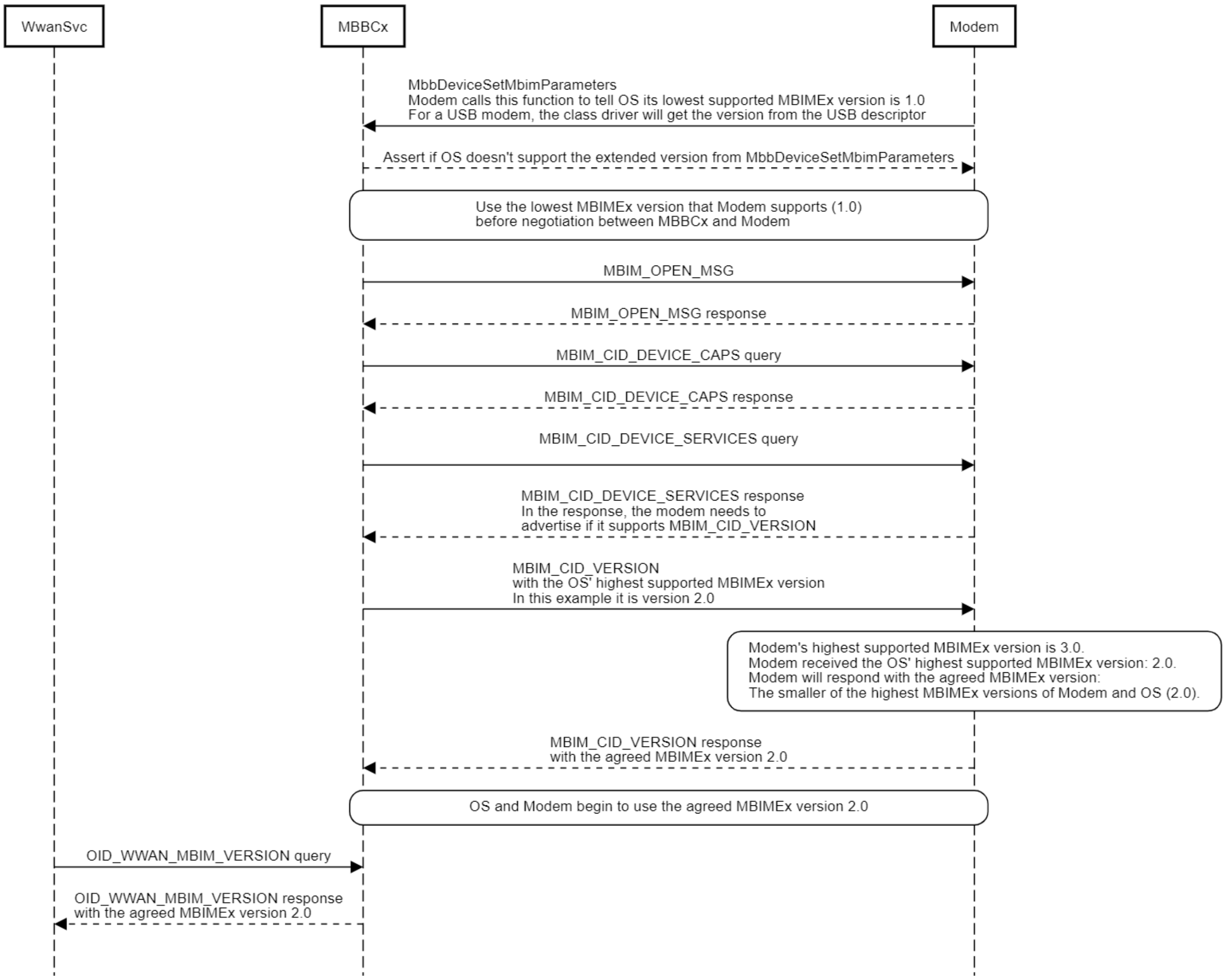

WwanSvc
MBBCx
Modem
MbbDeviceSetMbimParameters
Modem calls this function to tell OS its lowest supported MBIMEx version is 1.0
For a USB modem, the class driver will get the version from the USB descriptor
Assert if OS doesn't support the extended version from MbbDeviceSetMbimParameters
Use the lowest MBIMEx version that Modem supports (1.0)
before negotiation between MBBCx and Modem
MBIM_OPEN_MSG
MBIM_OPEN_MSG response
MBIM_CID_DEVICE_CAPS query
MBIM_CID_DEVICE_CAPS response
MBIM_CID_DEVICE_SERVICES query
MBIM_CID_DEVICE_SERVICES response
In the response, the modem advertises it supports MBIM_CID_VERSION
OS and Modem continue to use Modem's lowest supported MBIMEx version (1.0)
MBIM_CID_RADIO_STATE or other command
MBIM_CID_VERSION isn't sent since OS doesn't support it
MBIM_CID_RADIO_STATE or other command response with 1.0 payload
OID_WWAN_MBIM_VERSION query
OID_WWAN_MBIM_VERSION response
with the default MBIMEx version 1.0 in MBBCx

Feature-wise, a higher MBIMEx version is a superset of all lower MBIMEx versions. A host supports all devices with an announced MBIMEx version at or below the host's native MBIMEx version. If a device's announced MBIMEx version is higher than a host's native MBIMEx version, the host is not expected to support the device and the exact behavior of the host in this situation is undefined.

A device that intends to work with older hosts should initially advertise MBIMEx version 1.0 or the lowest host MBIMEx version with which the device is intended to work in an MBIM extended functional descriptor.

If the host sends MBIM_CID_VERSION with a higher MBIMEx version than the device initially advertised, then the device should indicate a higher MBIMEx version in the MBIM_CID_VERSION response up to the smaller of the host's native MBIMEx version and the device's native MBIMEx version.

OS' highest supported MBIMEx version is lower than Modem's



MbbDeviceSetMbimParameters
Modem calls this function to tell OS its lowest supported MBIMEx version is 1.0
For a USB modem, the class driver will get the version from the USB descriptor

Assert if OS doesn't support the extended version from MbbDeviceSetMbimParameters

Use the lowest MBIMEx version that Modem supports (1.0)
before negotiation between MBBCx and Modem

MBIM_OPEN_MSG

MBIM_OPEN_MSG response

MBIM_CID_DEVICE_CAPS query

MBIM_CID_DEVICE_CAPS response

MBIM_CID_DEVICE_SERVICES query

MBIM_CID_DEVICE_SERVICES response
In the response, the modem needs to
advertise if it supports MBIM_CID_VERSION

MBIM_CID_VERSION
with the OS' highest supported MBIMEx version
In this example it is version 2.0

Modem's highest supported MBIMEx version is 3.0.
Modem received the OS' highest supported MBIMEx version: 2.0.
Modem will respond with the agreed MBIMEx version:
The smaller of the highest MBIMEx versions of Modem and OS (2.0).

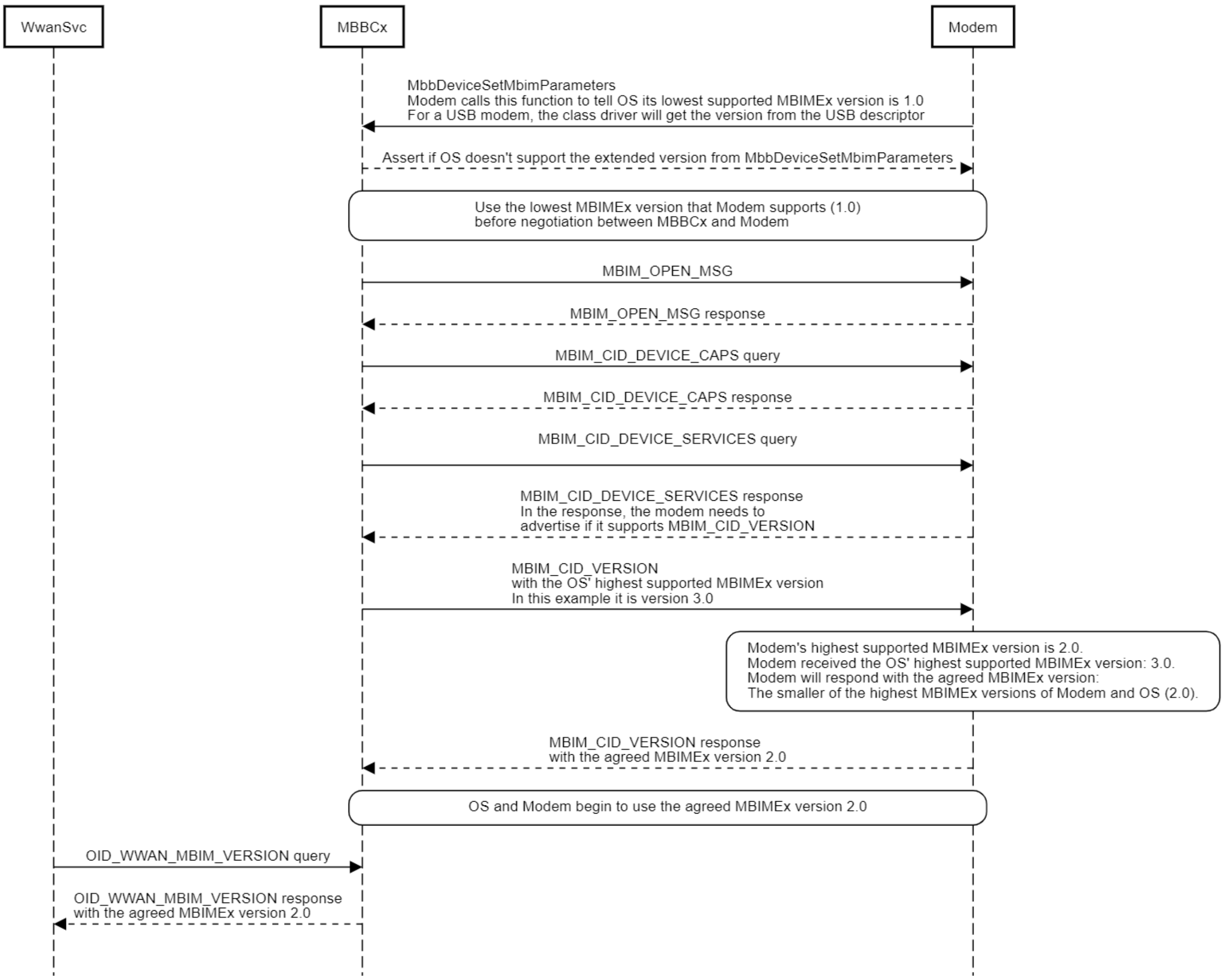
MBIM_CID_VERSION response
with the agreed MBIMEx version 2.0

OS and Modem begin to use the agreed MBIMEx version 2.0

OID_WWAN_MBIM_VERSION query

OID_WWAN_MBIM_VERSION response
with the agreed MBIMEx version 2.0

OS' highest supported MBIMEx version is higher than Modem's


WwanSvc
MBBCx
Modem
MbbDeviceSetMbimParameters
Modem calls this function to tell OS its lowest supported MBIMEx version is 1.0
For a USB modem, the class driver will get the version from the USB descriptor
Assert if OS doesn't support the extended version from MbbDeviceSetMbimParameters
Use the lowest MBIMEx version that Modem supports (1.0)
before negotiation between MBBCx and Modem
MBIM_OPEN_MSG
MBIM_OPEN_MSG response
MBIM_CID_DEVICE_CAPS query
MBIM_CID_DEVICE_CAPS response
MBIM_CID_DEVICE_SERVICES query
MBIM_CID_DEVICE_SERVICES response
In the response, the modem needs to
advertise if it supports MBIM_CID_VERSION
MBIM_CID_VERSION
with the OS' highest supported MBIMEx version
In this example it is version 3.0
Modem's highest supported MBIMEx version is 2.0.
Modem received the OS' highest supported MBIMEx version: 3.0.
Modem will respond with the agreed MBIMEx version:
The smaller of the highest MBIMEx versions of Modem and OS (2.0).
MBIM_CID_VERSION response
with the agreed MBIMEx version 2.0
OS and Modem begin to use the agreed MBIMEx version 2.0
OID_WWAN_MBIM_VERSION query
OID_WWAN_MBIM_VERSION response
with the agreed MBIMEx version 2.0

ⓘ Note

For example, a device supports MBIMEx version 2.0, but is intended to work with older versions of the OS that do not support MBIMEx 2.0. The device initially advertises MBIMEx version 1.0 in the USB descriptors and advertises support for the optional MBIM_CID_VERSION. When inserted into a host running Windows 10, version 1803, the host does not understand MBIM_CID_VERSION and does not send MBIM_CID_VERSION to the device. To the host, the device's MBIMEx version is 1.0. The host continues to send other CIDs in the initialization sequence. Upon receiving CIDs other than MBIM_CID_VERSION, the device knows that the host supports MBIMEx version 1.0. Both sides proceed to conform to MBIMEx version 1.0. Later, when the same device is inserted into a host running Windows 10, version 1903 with a native MBIMEx version of 2.0, the host sends MBIM_CID_VERSION to the device to inform it that the host's native MBIMEx version is 2.0. The device sends MBIM_CID_VERSION back in response with the device's announced MBIMEx version 2.0. From there, both sides proceed to conform to MBIMEx version 2.0.

The following table shows a compatibility matrix with three hypothetical hosts and three hypothetical devices, each with its native MBIMEx version stated. The devices advertise MBIMEx version 1.0 initially in the USB descriptor. The matrix shows how each of the devices behaves with each of the hosts.

Device (below) / Host (right)	Windows 10, version 1809 or earlier (native MBIMEx version 1.0)	Windows 10, version 1903 and later (MBIMEx version 2.0)

Device (below) / Host (right)	Windows 10, version 1809 or earlier (native MBIMEx version 1.0)	Windows 10, version 1903 and later (MBIMEx version 2.0)
4G device Native MBIMEx version 1.0	Device initially advertises MBIMEx 1.0. No MBIM_CID_VERSION exchange. Compatible device and host. Works by default with MBIMEx version 1.0.	Device initially advertises MBIMEx 1.0. No MBIM_CID_VERSION exchange. The host works with the device using MBIMEx 1.0.
5G NSA device Native MBIMEx version 2.0	Device initially advertises MBIMEx 1.0. No MBIM_CID_VERSION exchange. Device knows that the host has MBIMEx 1.0 and proceeds with MBIMEx 1.0.	Device initially advertises MBIMEx 1.0. Host sends MBIM_CID_VERSION to inform the device that the host supports MBIMEx 2.0. Device responds with MBIMEx 2.0. Both sides proceed with MBIMEx 2.0.

The following table lists all existing CIDs that are modified in MBIMEx version 2.0, and their modified payloads. All unmentioned payloads in these CIDs and all other CIDs not mentioned in the table carry over from MBIMEx version 1.0 and remain unchanged.

CID	Payload
MBIM_CID_REGISTER_STATE	MBIM_REGISTRATION_STATE_INFO_V2
MBIM_CID_PACKET_SERVICE	MBIM_PACKET_SERVICE_INFO_V2
MBIM_CID_SIGNAL_STATE	MBIM_SIGNAL_STATE_INFO_V2

MBIM service

Service name	UUID	UUID value
Microsoft Basic IP Connectivity Extensions	UUID_BASIC_CONNECT_EXTENSIONS	3D01DCC5-FEF5-4D05-9D3A-BEF7058E9AAF

MBIM_CID_VERSION

For MBB drivers that support MBIM Microsoft extension 2.0 or above, MBIM_CID_VERSION is a mandatory command for exchanging MBIM version information between the host and the device. For in-market devices with drivers that do not recognize this CID, the host will assume and provide backward compatibility.

The host sends this command as a query if it is supported by the device. The query contains the MBIM release number and MBIM Extensions release number that the host currently supports.

On the device side, the device adjusts its announced MBIM release number and MBIM Extensions release number based on the rules defined in versioning scheme, then sends them in the response to the host.

This command is defined under the **Basic Connect Extensions** service.

CID	Command code	UUID
MBIM_CID_VERSION	15	3d01dcc5-fef5-4d05-0d3abef7058e9aaf

Parameters

Operation	Set	Query	Notification
Command	Not applicable	MBIM_VERSION_INFO	Not applicable
Response	Not applicable	MBIM_VERSION_INFO	Not applicable

Query

Informs the device of the host's native MBIM release number and MBIM Extensions release number. The InformationBuffer contains the following MBIM_VERSION_INFO structure.

Offset	Size	Field	Type	Description
0	2	bcdMBIMVersion	UINT16	The MBIM release number of the sender in BCD, with an implied decimal point between bits 7 and 8. For example, `0x0100 == 1.00 == 1.0`. This is a little-endian constant, so the bytes are 0x00, then 0x01.
2	2	bcdMBIMExtendedVersion	UINT16	The MBIM Extensions release number of the sender in BCD, with an implied decimal point between bits 7 and 8. For example, `0x0100 == 1.00 == 1.0`. This is a little-endian constant, so the bytes are 0x00, then 0x01.

Set

Not applicable.

Response

The InformationBuffer in MBIM_COMMAND_DONE contains an MBIM_VERSION_INFO structure.

Unsolicited Events

Not applicable.

Status Codes

This CID only uses generic status codes defined in Section 9.4.5 of the MBIM specification revision 1.0.

MBIM_CID_MS_DEVICE_CAPS_V2

This CID is the same as defined on MB Multi-SIM operations, which itself is an extension of MBIM_CID_MS_DEVICE_CAPS as defined in Section 10.5.1 of the MBIM specification revision 1.0. For MBIM Extensions release 2.0, there are new data classes defined in the MBIM_DATA_CLASS table that enable the device to report its 5G capabilities. MBIMDataClass5G_NSA denotes that the device supports 5G Non-standalone (NSA), defined in 3GPP TS 37.340, and MBIMDataClass5G_SA denotes that the device supports 5G Standalone (SA), also defined in 3GPP TS 37.340.

If the device supports both new data classes, then both bits shall be set.

MBIM_DATA_CLASS

Types	Mask
MBIMDataClassNone	0h
MBIMDataClassGPRS	1h
MBIMDataClassEDGE	2h
MBIMDataClassUMTS	4h
MBIMDataClassHSDPA	8h
MBIMDataClassHSUPA	10h
MBIMDataClassLTE	20h
MBIMDataClass5G_NSA	**40h**
MBIMDataClass5G_SA	**80h**
Reserved	100h-8000h
MBIMDataClass1XRTT	10000h
MBIMDataClass1XEVDO	20000h
MBIMDataClass1XEVDORevA	40000h
MBIMDataClass1XEVDV	80000h
MBIMDataClass3XRTT	100000h
MBIMDataClass1XEVDORevB	200000h
MBIMDataClassUMB	400000h
Reserved	800000-40000000h
MBIMDataClassCustom	80000000h

MBIM_CID_REGISTER_STATE

This command is an extension for the MBIM_CID_REGISTER_STATE CID already defined in the MBIM specification revision 1.0. This extension adds a new member called **PreferredDataClasses** for the response structure.

Parameters

Operation	Set	Query	Notification
Command	MBIM_SET_REGISTRATION_STATE	Empty	Not applicable
Response	MBIM_REGISTRATION_STATE_INFO_V2	MBIM_REGISTRATION_STATE_INFO_V2	MBIM_REGISTRATION_STATE_INFO_V2

Query

The InformationBuffer is null and the InformationBufferLength is zero.

Set

Sets the registration state. The information is the same as described in the MBIM specification revision 1.0 .

Response

The InformationBuffer in MBIM_COMMAND_DONE contains the following MBIM_REGISTRATION_STATE_INFO_V2 structure. Compared with the MBIM_REGISTRATION_STATE_INFO structure defined in Section 10.5.10.6 of the MBIM specification revision 1.0 , the following structure has a new **PreferredDataClasses** field. Unless stated here, field descriptions in table 10-55 of the MBIM specification revision 1.0 apply to this structure.

MBIM_REGISTRATION_STATE_INFO_V2

Offset	Size	Field	Type	Description
0	4	NwError	UINT32	A network-specific error. Table 10-44 in the MBIM specification revision 1.0 documents the cause codes for NwError.
4	4	RegisterState	MBIM_REGISTER_STATE	See Table 10-46 in the MBIM specification revision 1.0 .
8	4	RegisterMode	MBIM_REGISTER_MODE	See Table 10-47 in the MBIM specification revision 1.0 .
12	4	AvailableDataClass	UINT32	A bitmap of the values in MBIM_DATA_CLASS that represents the supported data classes on the registered network, for the cell in which the device is registered. This value is set to MBIMDataClassNone if the **RegisterState** is not **MBIMRegisterStateHome**, **MBIMRegisterStateRoaming**, or **MBIMRegisterStatePartner**.
16	4	CurrentCellularClass	MBIM_CELLULAR_CLASS	Indicates the current cellular class in use for a multi-mode function. See Table 10-8 in the MBIM specification revision 1.0 for more information. For a single-mode function, this is the same as the cellular class reported in MBIM_CID_DEVICE_CAPS. For multi-mode functions, a transition from CDMA to GSM or vice versa is indicated with an updated **CurrentCellularClass**.

Offset	Size	Field	Type	Description
20	4	ProviderIdOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a numeric (0-9) string called **ProviderId** that represents the network provider identity. For GSM-based networks, this string is a concatenation of a three-digit Mobile Country Code (MCC) and a two- or three-digit Mobile Network Code (MNC). GSM-based carriers might have more than one MNC, and hence more than one **ProviderId**. For CDMA-based networks, this string is a five-digit System ID (SID). Generally, a CDMA-based carrier has more than one SID. Typically, a carrier has one SID for each market that is usually divided geographically within a nation by regulations, such as Metropolitan Statistical Areas (MSA) in the United States. CDMA-based devices must specify MBIM_CDMA_DEFAULT_PROVIDER_ID if this information is not available. When processing a query request and the registration state is in automatic register mode, this member contains the provider ID with which the device is currently associated (if applicable). When the registration state is in manual register mode, this member contains the provider ID to which the device is requested to register (even if the provider is unavailable). When processing a set request and the registration state is in manual mode, this contains the provider ID selected by the host with which to register the device. When the registration state is in automatic register mode, this parameter is ignored. CDMA 1xRTT providers must be set to MBIM_CDMA_DEFAULT_PROVIDER_ID if the provider ID is not available.
24	4	ProviderIdSize	SIZE(0..12)	The size, in bytes, for **ProviderId**.
28	4	ProviderNameOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a string called **ProviderName** that represents the network provider's name. This member is limited to, at most, MBIM_PROVIDERNAME_LEN characters. For GSM-based networks, if the Preferred Presentation of Country Initials and Mobile Network Name (PCCI&N) is longer than twenty characters, the device should abbreviate the network name. This member is ignored when the host sets the preferred provider list. Devices should specify a NULL string for devices that do not have this information.
32	4	ProviderNameSize	SIZE(0..40)	The size, in bytes, for **ProviderName**.

Offset	Size	Field	Type	Description
36	4	RoamingTextOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to a string called **RoamingText** to inform a user that the device is roaming. This member is limited to, at most, 63 characters. This text should provide additional information to the user when the registration state is either MBIMRegisterStatePartner or MBIMRegisterStateRoaming. This member is optional.
40	4	RoamingTextSize	SIZE(0..126)	The size, in bytes, for **RoamingText**.
44	4	RegistrationFlag	MBIM_REGISTRATION_FLAGS	Flags set per Table 10-48 in the MBIM specification revision 1.0.
48	4	PreferredDataClass	UINT32	A bitmap of the values in MBIM_DATA_CLASS that represent the enabled data classes on the device. The device can only operate using the data classes that are enabled.
Dynamic	4	DataBuffer	DATABUFFER	The data buffer that contains **ProviderId**, **ProviderName**, and **RoamingText**.

Unsolicited Events

Notifications contain an MBIM_REGISTRATION_STATE_INFO_V2 structure.

Status Codes

This CID only uses generic status codes defined in Section 9.4.5 of the MBIM specification revision 1.0.

MBIM_CID_PACKET_SERVICE

This command is an extension for the existing MBIM_CID_PACKET_SERVICE defined in the MBIM specification revision 1.0.

This extension adds a new member called **FrequencyRange** for the response structure and renamed the **HighestAvailableDataClass** member to **CurrentDataClass** to clarify its purpose.

The **CurrentDataClass** indicates the Radio Access Technology (RAT) with which the device is currently registered. It contains a single value from MBIM_DATA_CLASS.

The **FrequencyRange** indicates the frequency range that the device is currently using. This is valid only if the **CurrentDataClass** field indicates that the MBIMDataClass5G_NSA or MBIMDataClass5G_SA bit is set.

Parameters

Operation	Set	Query	Notification
Command	MBIM_SET_PACKET_SERVICE	Empty	Not applicable
Response	MBIM_PACKET_SERVICE_INFO_V2	MBIM_PACKET_SERVICE_INFO_V2	MBIM_PACKET_SERVICE_INFO_V2

Query

The InformationBuffer is null and the InformationBufferLength is zero.

Set

Information for set commands is described in the MBIM specification revision 1.0.

Response

The InformationBuffer in MBIM_COMMAND_DONE contains an MBIM_PACKET_SERVICE_INFO_V2 structure. Compared with the MBIM_PACKET_SERVICE_INFO structure defined in Section 10.5.10.6 of the MBIM specification revision 1.0, this new structure has the **CurrentDataClass** and **FrequencyRange** fields. Unless stated here, the field descriptions in Table 10-55 of the MBIM specification revision 1.0 apply here.

MBIM_PACKET_SERVICE_INFO_V2

Offset	Size	Field	Type	Description
0	4	NwError	UINT32	A network-specific error. Table 10-44 in the MBIM specification revision 1.0 documents the cause codes for NwError.
4	4	PacketServiceState	MBIM_PACKET_SERVICE_STATE	See Table 10-53 in the MBIM specification revision 1.0.
8	4	CurrentDataClass	MBIM_DATA_CLASS	The current data class in the current cell, specified according to MBIM_DATA_CLASS. Functions must set this member to MBIMDataClassNone if the function is not in the attached packet service state. Except for HSPA (in other words, HSUPA and HSDPA) and 5G DC, the function sets this member to a single MBIM_DATA_CLASS value. For HSPA data services, functions specify a bitwise OR of MBIMDataClass HSDPA and MBIMDataClassHSUPA. For cells that support HSDPA but not HSUPA, only HSDPA is indicated (implying UMTS data class for uplink data). Whenever the current data class changes, functions send a notification indicating the new value of **CurrentDataClass**.
12	8	UplinkSpeed	UINT64	Contains the uplink bit rate, in bits per second.
20	8	DownlinkSpeed	UINT64	Contains the downlink bit rate, in bits per second.
38	4	FrequencyRange	MBIM_FREQUENCY_RANGE	A bitmask of values in MBIM_FREQUENCY_RANGE that represents the frequency ranges that the device is currently using. This is only valid if the **CurrentDataClass** is either MBIMDataClass5G_NSA or MBIMDataClass5G_SA.

MBIM_FREQUENCY_RANGE

The following enumeration is used as a value in the preceding MBIM_PACKET_SERVICE_INFO_V2 structure.

Type	Value	Description

Type	Value	Description
MBIMFrequencyRangeUnknown	0	If the system type is not 5G.
MBIMFrequencyRange1	1	Frequency range 1 (FR1) in 3GPP TS 38.101-1 (Sub-6G).
MBIMFrequencyRange2	2	FR2 in 3GPP TS 38.101-2 (mmWave).
MBIMFrequencyRange1AndRange2	3	If both FR1 and FR2 carriers are connected.

Unsolicited Events

Notifications contain an MBIM_PACKET_SERVICE_INFO_V2 structure.

Status Codes

This CID only uses generic status codes defined in Section 9.4.5 of the MBIM specification revision 1.0.

MBIM_CID_SIGNAL_STATE

This CID is an extension to MBIM_CID_SIGNAL_STATE, introducing RSRP and SNR for signal state criteria. This new extension is only valid if the device indicates support of MBIM Extensions version 2.0. This extension is mandatory if the modem supports MBIMDataClass5G_(N)SA data classes.

The RSRP and SNR fields are only valid if the corresponding SystemType is either MGBIMDataClassLTE or MBIMDataClass5G_(N)SA. IF the modem reports RSRP and/or SNR, then the RSSI field shall be set to a value of **99**.

If the corresponding SystemType is MBIMDataClass5G_(N)SA, the RSRP field is mandatory and the SNR field is optional. If the corresponding SystemType is MBIMDataClassLTE, the RSRP and SNR fields are optional and the RSSI field can be used instead. In this case, the RSRP and SNR fields can be omitted by setting a zero (**0**) value for both **RsrpSnrOffset** and **RsrpSnrSize** members.

Parameters

Operation	Set	Query	Notification
Command	MBIM_SET_SIGNAL_STATE	Empty	Not applicable
Response	MBIM_SIGNAL_STATE_INFO_V2	MBIM_SIGNAL_STATE_INFO_V2	MBIM_SIGNAL_STATE_INFO_V2

Query

The InformationBuffer is null and the InformationBufferLength is zero.

Set

Information for set commands is described in the MBIM specification revision 1.0.

Response

The InformationBuffer in MBIM_COMMAND_DONE contains the following MBIM_SIGNAL_STATE_INFO_V2 structure.

MBIM_SIGNAL_STATE_INFO_V2

Offset	Size	Field	Type	Description
0	4	Rssi	UINT32	See Table 10.58 in the MBIM specification revision 1.0 .
4	4	ErrorRate	UINT32	See Table 10.58 in the MBIM specification revision 1.0 .
8	4	SignalStrengthInterval	UINT32	The reporting interval, in seconds.
12	4	RssiThreshold	UINT32	The difference in RSSI coded values that triggers a report. Use 0xFFFFFFFF if this does not matter.
16	4	ErrorRateThreshold	UINT32	The difference in ErrorRate coded values that trigger a report. Use 0xFFFFFFFF if this does not matter.
20	4	RsrpSnrOffset	OFFSET	The offset in bytes, calculated from the beginning of this structure, to the buffer containing RSRP and SNR signaling info. This member can be **NULL** when no RSRP and SNR signaling info is available.
24	4	RsrpSnrSize	SIZE	The size, in bytes, of the buffer containing the RSRP and SNR signaling info in the format of a MBIM_RSRP_SNR_INFO structure.
	4	DataBuffer	DATABUFFER	An MBIM_RSRP_SNR structure.

MBIM_RSRP_SNR

The following MBIM_RSRP_SNR structure is used in the **DataBuffer** of an MBIM_SIGNAL_STATE_INFO_V2 structure.

Offset	Size	Field	Type	Description
0	4	ElementCount	UINT32	The count of RSRP_SNR entries that follow this element.
4	4	DataBuffer	DATABUFFER	An array of RSRP_SNR records, each specified as an MBIM_RSRP_SNR_INFO structure.

MBIM_RSRP_SNR_INFO

An array of the following MBIM_RSRP_SNR_INFO structures is used in the **DataBuffer** of an MBIM_RSRP_SNR structure.

<table>
<tr><th>Offset</th><th>Size&gt;</th><th>Field</th><th>Type</th><th>Description</th></tr>
<tr><td>0</td><td>4</td><td>RSRP</td><td>UINT32</td><td>
<table>
<tr><th>RSRP value in dBm</th><th>Coded value (min = 0, max = 126)</th></tr>
<tr><td>Less than -156</td><td>0</td></tr>
<tr><td>Less than -155</td><td>1</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Less than -138</td><td>18</td></tr>
<tr><td>...</td><td>...</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td></td><td></td><td></td><td><table>
<tr><td>Less than -45</td><td>111</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Less than -31</td><td>125</td></tr>
<tr><td>-31 or greater</td><td>126</td></tr>
<tr><td>Unknown or undetectable</td><td>127</td></tr>
</table></td></tr>
<tr><td>4</td><td>4</td><td>SNR</td><td>UINT32</td><td><table>
<tr><th>SNR value in dB</th><th>Coded value (min = 0, max = 127)</th></tr>
<tr><td>Less than -23</td><td>0</td></tr>
<tr><td>Less than -22.5</td><td>1</td></tr>
<tr><td>Less than -22</td><td>2</td></tr>
<tr><td>Less than -21.5</td><td>3</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Less than 39.5</td><td>125</td></tr>
<tr><td>Less than 40</td><td>126</td></tr>
<tr><td>40 or greater</td><td>127</td></tr>
<tr><td>Unknown or undetectable</td><td>128</td></tr>
</table></td></tr>
<tr><td>8</td><td>4</td><td>RSRPThreshold</td><td>UINT32</td><td>Defines the threshold between the old (cached) RSRP value and the newly calculated RSRP value. If the absolute difference is larger than the threshold value, the device triggers an unsolicited event. The unit is 1 dBm. If set to zero, use the default behavior in the device function. If set to 0xFFFFFFFF, don't use this to trigger the event. If the given threshold value is not supported by the device, it returns the max threshold value that it supports.</td></tr>
<tr><td>12</td><td>4</td><td>SNRThreshold</td><td>UINT32</td><td>Defines the threshold between the old (cached) SNR value and the newly calculated SNR value. If the absolute difference is larger than the threshold value, the device triggers an unsolicited event. The unit is 1 dB. If set to zero, use the default behavior in the device function. If set to 0xFFFFFFFF, don't use this to trigger the event. If the given threshold is not supported by the device, it returns the max threshold value that it supports.</td></tr>
<tr><td>16</td><td>4</td><td>SystemType</td><td>MBIM_DATA_CLASS</td><td>Indicates the system type for which signal state information is valid. This member is a bitmask of one type as defined in MBIM_DATA_CLASS.</td></tr>
</table>

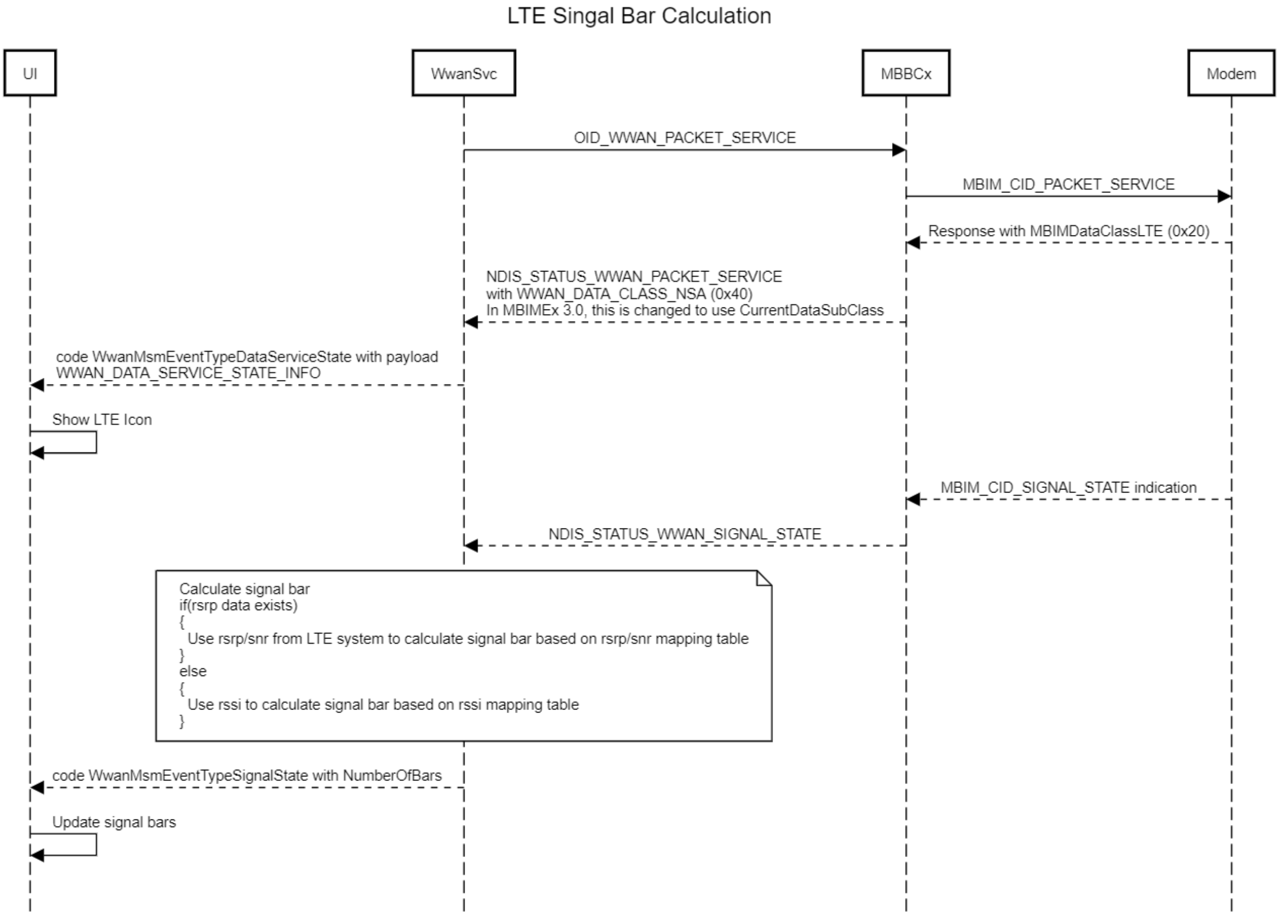
Unsolicited Events

Notifications contain an MBIM_SIGNAL_STATE_INFO_V2 structure.

Status Codes

This CID only uses generic status codes defined in Section 9.4.5 of the MBIM specification revision 1.0 .

LTE signal bar calculation


LTE Singal Bar Calculation
UI
WwanSvc
MBBCx
Modem
OID_WWAN_PACKET_SERVICE
MBIM_CID_PACKET_SERVICE
Response with MBIMDataClassLTE (0x20)
NDIS_STATUS_WWAN_PACKET_SERVICE
with WWAN_DATA_CLASS_NSA (0x40)
In MBIMEx 3.0, this is changed to use CurrentDataSubClass
code WwanMsmEventTypeDataServiceState with payload
WWAN_DATA_SERVICE_STATE_INFO
Show LTE Icon
MBIM_CID_SIGNAL_STATE indication
NDIS_STATUS_WWAN_SIGNAL_STATE
Calculate signal bar
if(rsrp data exists)
{
Use rsrp/snr from LTE system to calculate signal bar based on rsrp/snr mapping table
}
else
{
Use rssi to calculate signal bar based on rssi mapping table
}
code WwanMsmEventTypeSignalState with NumberOfBars
Update signal bars

The OS shall process the registry settings for signal strength calculations in the following order:

Dataclass is CDMA (or its variant) or TDSCDMA

1. If a legacy signal bar mapping table exists under “per_iccid”, use this setting.
2. Else, if a legacy signal bar mapping table exists under “per_device”, use this setting.
3. Else, use the default signal bar mapping table in code.

Dataclass is GSM or WCDMA

1. If a GSM or WCDMA technology specific signal bar mapping table exists under “per_iccid”, use this setting.
2. Else, if a GSM or WCDMA technology specific signal bar mapping table exists under “per_device”, use this setting.
3. Else, if a legacy signal bar mapping table exists under “per_iccid”, use this setting.
4. Else, if a legacy signal bar mapping table exists under “per_device”, use this setting.
5. Else, use the default signal bar mapping table in code.
   a. RSSI >= 17; 5 bars
   b. RSSI >= 12; 4 bars
   c. RSSI >= 7; 3 bars
   d. RSSI >= 4; 2 bars
   e. RSSI >= 2; 1 bars
   f. else; 0 bars

Dataclass is LTE and RSRP is reported by the modem

1. If a LTE technology specific signal bar mapping table for RSRP exists under “per_iccid”, use this setting.
2. Else, if a LTE technology specific signal bar mapping table for RSRP exists under “per_device”, use this setting.
3. Else, use the default LTE RSRP signal bar mapping table in code.

ⓘ **Note**

If EnableLTESnrReporting is enabled but there is no SNR table for LTE or SNR reporting by the modem, only RSRP is used. Otherwise, the better of RSRP or SNR is converted to signal bars.

Dataclass is LTE and RSSI is reported by the modem

1. If a LTE technology specific signal bar mapping table exists under “per_iccid”, use this setting.
2. Else, if a LTE technology specific signal bar mapping table exists under “per_device”, use this setting.
3. Else, if a legacy signal bar mapping table exists under “per_iccid”, use this setting.
4. Else, if a legacy signal bar mapping table exists under “per_device”, use this setting.
5. Else, use the default signal bar mapping table in code.

Dataclass is NR

1. If a NR technology specific signal bar mapping table for RSRP exists under “per_iccid”, use this setting.
2. Else, if a NR technology specific signal bar mapping table for RSRP exists under “per_device”, use this setting.
3. Else, use the default NR RSRP signal bar mapping table in code.

ⓘ **Note**

If EnableNRSnrReporting is enabled, but no there is no SNR table for NR or SNR reporting by the modem, only RSRP is used. Otherwise, the better of RSRP or SNR is converted to signal bars.

Dataclass is NSA

1. If EnableLTEReportingOnNSA is not set or is set to **0**:
   a. Follow the Dataclass NR flow.
2. If EnableLTEReportingOnNSA is set to **1**:
   a. Follow the Dataclass LTE flows (RSRP or RSSI).
3. If EnableLTEReportingOnNSA is set to **2**:
   a. If FrequencyRange is FR1, follow the Dataclass LTE flows (RSRP or RSSI).
   b. If FrequencyRange is <> FR1, follow the Dataclass NR flow.
4. If EnableLTEReportingOnNSA is set to **3**:
   a. If FrequencyRange is FR2, follow the Dataclass LTE flows (RSRP or RSSI).
   b. If FrequencyRange is <> FR2, follow the Dataclass NR flow.
5. If EnableLTEReportingOnNSA is set to **4**:
   a. Calculate signal bar using the LTE and NR flows.
   b. Select the strongest.

ⓘ **Note**

If the LTE signal is not reported by the modem in 1-5, the NR signal is used. If the NR signal is not used, the LTE signal is applied.

COSA customizations for SignalBar calculation

EnableLTEReportingOnNSA:

0 = "Use 5G signal"

1 = "Use LTE signal"

2 = "Use LTE signal if camped on 5G frequency range 1"

3 = "Use LTE signal if camped on 5G frequency range 2"

4 = "Use the strongest signal of LTE and 5G"

EnableNRSnrReporting:

0 = "Use only RSRP"

1 = "Use both RSRP and SNR"

EnableLTESnrReporting:

0 = "Use only RSRP"

1 = "Use both RSRP and SNR"

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown. Technology specific settings take precedence. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/GERAN/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown when device is camped on GSM. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/WCDMA/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown when device is camped on WCDMA. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/LTE/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown when device is camped on LTE. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/LTERSRP/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown, when device is camped on LTE. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/LTERSSNR/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown when device is camped on LTE. Used when EnableLTESnrReporting is set to 1. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/NRRSRP/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown when device is camped on 5G. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

Cellular/PerDevice/SignalBarMappingTable/SignalForBars/NRRSSNR/<SignalBar>

Modify the minimum signal strength value corresponding to the number of bars to be shown when device is camped on 5G. Used when EnableNRSnrReporting is set to 1. Each number of bars needs to have a valid signal strength mapping for this setting to take effect.

<SignalBar> can be 1-5 values.

If the OEM/MO fails to properly configure the mapping table for RSSI or it's incomplete, use the default mapping:

RSSI	Bars Displayed
[0,1]	0
[2,3]	1
[4,6]	2
[7,11]	3
[12,16]	4
[17,31]	5

If the OEM/MO fails to properly configure the mapping table for RSRP or it's incomplete, use the default mapping:

RSRP	Bars Displayed
[0,16]	0
[17,41]	1
[42,51]	2
[52,61]	3
[62,71]	4
[72,126]	5

If the OEM/MO fails to properly configure the mapping table for SNR or it's incomplete, use the default mapping:

SNR	Bars Displayed
[0,18]	0
[19,38]	1
[39,46]	2
[47,53]	3
[54,72]	4
[73,127]	5

MBIMEx 3.0 – 5G SA Phase 1 support

Article • 03/14/2023

Starting with Windows 11, Windows OS supports the first phase of features for 5G systems with the next generation core network. These 5G systems are commonly referred to as 5G SA networks. They contain the new 5G Core Network (5GC) and are independent of Enhanced Packet Core (EPC) as used in 5G NSA networks and 4G networks.

The first phase of features for 5G SA enables MBB functionality parity for Windows MBIM MBB devices on 5G SA networks compared to 5G NSA. These features include registration with 5GC and basic PDU sessions on default eMBB network slice. Note that this phase does not support more advanced 5G SA features such as multiple concurrent network slices and URSP rules.

A new MBIM Extensions Release number 3.0 introduces changes in the MBIM interface to support the first phase of features for 5G SA. This Extensions Release number is commonly referred to as MBIMEx 3.0. Download the MBIMEx 3.0 specification here.

An errata for MBIMEx 3.0 introduced in May 2022 updates and clarifies certain aspects of the original MBIMEx 3.0 specification published in April 2020. Download the MBIMEx 3.0 errata here. The errata is in section 7.1 “Errata for MBIMEx 3.0”.

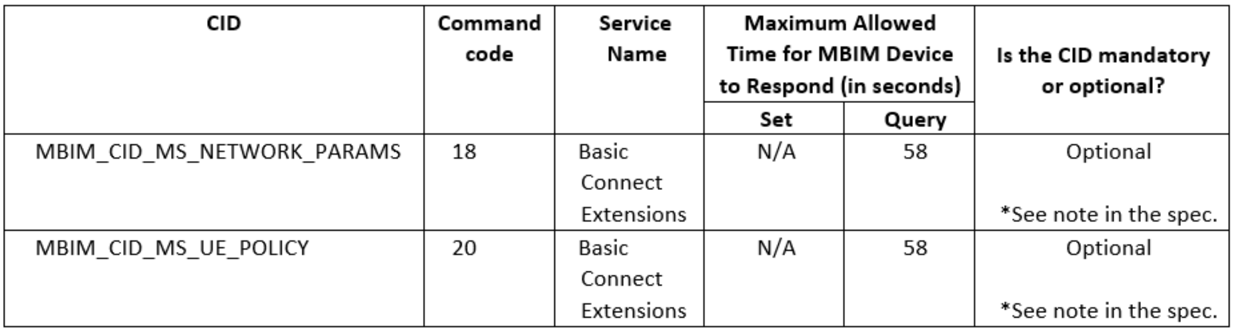
MBIMEx 4.0 – 5G SA Phase 2 support

Article • 03/14/2023

Windows 11, version 22H2 previews the 5G SA Phase 2 feature set. Additionally, it supports all Windows 11 cellular features such as 5G SA Phase 1. The 5G SA Phase 2 feature set includes support for end-to-end URSP handling and multiple concurrent eMBB network slices.

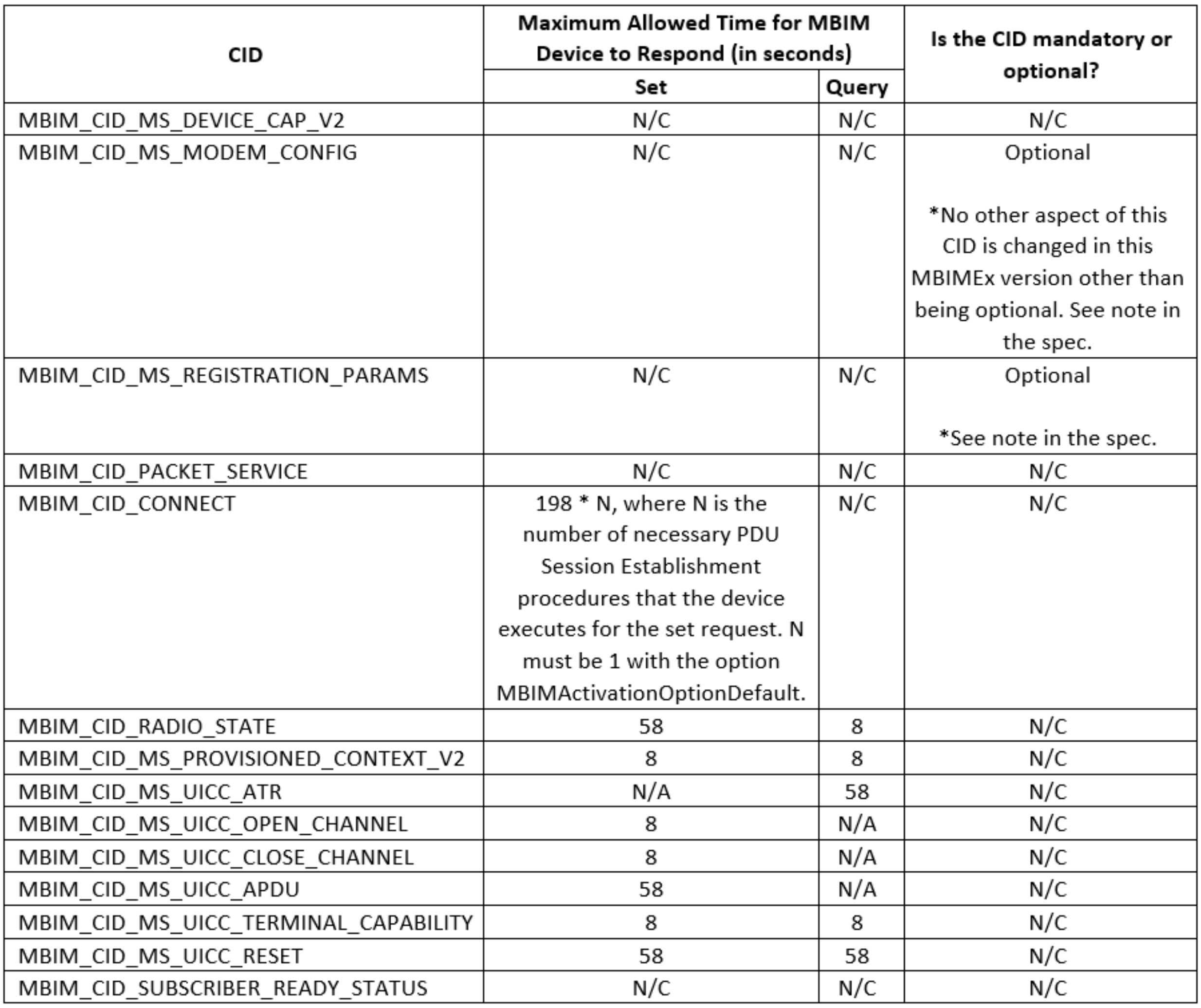
MBIM Extensions Release number 4.0 introduces support for 5G SA Phase 2 features. This Extensions Release number is commonly referred to as MBIMEx 4.0. URSP handling/usage and multiple concurrent eMBB network slices are the main additions in MBIMEx 4.0. All valid slice types (SST) are supported at the MBIM interface level in MBIMEx 4.0, but non-eMBB slice functionality is not implied in Windows 11, version 22H2 and is subject to additional host and device-level support and features. Download the MBIMEx 4.0 specification here. Section 4 “MBIM Interface Extensions for 5G NGC – Phase 2” contains the MBIMEx 4.0 specification.

MBIMEx 4.0 adds the following new CIDs:



CID	Command code	Service Name	Maximum Allowed Time for MBIM Device to Respond (in seconds)		Is the CID mandatory or optional?
			Set	Query	
MBIM_CID_MS_NETWORK_PARAMS	18	Basic Connect Extensions	N/A	58	Optional *See note in the spec.
MBIM_CID_MS_UE_POLICY	20	Basic Connect Extensions	N/A	58	Optional *See note in the spec.

MBIMEx 4.0 modifies the following existing CIDs:



CID	Maximum Allowed Time for MBIM Device to Respond (in seconds)		Is the CID mandatory or optional?
	Set	Query	
MBIM_CID_MS_DEVICE_CAP_V2	N/C	N/C	N/C
MBIM_CID_MS_MODEM_CONFIG	N/C	N/C	Optional *No other aspect of this CID is changed in this MBIMEx version other than being optional. See note in the spec.
MBIM_CID_MS_REGISTRATION_PARAMS	N/C	N/C	Optional *See note in the spec.
MBIM_CID_PACKET_SERVICE	N/C	N/C	N/C
MBIM_CID_CONNECT	198 * N, where N is the number of necessary PDU Session Establishment procedures that the device executes for the set request. N must be 1 with the option MBIMActivationOptionDefault.	N/C	N/C
MBIM_CID_RADIO_STATE	58	8	N/C
MBIM_CID_MS_PROVISIONED_CONTEXT_V2	8	8	N/C
MBIM_CID_MS_UICC_ATR	N/A	58	N/C
MBIM_CID_MS_UICC_OPEN_CHANNEL	8	N/A	N/C
MBIM_CID_MS_UICC_CLOSE_CHANNEL	8	N/A	N/C
MBIM_CID_MS_UICC_APDU	58	N/A	N/C
MBIM_CID_MS_UICC_TERMINAL_CAPABILITY	8	8	N/C
MBIM_CID_MS_UICC_RESET	58	58	N/C
MBIM_CID_SUBSCRIBER_READY_STATUS	N/C	N/C	N/C

By default, Windows 11, version 22H2 announces MBIMEx 3.0 as the highest supported MBIMEx version by the host. A special capability is available to change the default to MBIMEx 4.0 for IHVs and driver developers. Contact your TAM for support if a Microsoft engineering partner needs the capability for testing and development.

NDIS_STATUS_WWAN_ATR_INFO

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_ATR_INFO notification to respond to OID query requests of OID_WWAN_UICC_ATR.

This notification uses the **NDIS_WWAN_ATR_INFO** structure.

Requirements

Version: Windows 10, version 1607

Header: Ntddndis.h (include Ndis.h)

See also

OID_WWAN_UICC_ATR

NDIS_WWAN_ATR_INFO

MB low level UICC access

NDIS_STATUS_WWAN_AUTH_RESPONSE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_AUTH_RESPONSE notification to inform the MB Service of a challenge response received from a previous challenge request issued using an OID_WWAN_AUTH_CHALLENGE query request.

Miniport drivers can also send unsolicited events with this notification.

This NDIS status notification uses the NDIS_WWAN_AUTH_RESPONSE structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

OID_WWAN_AUTH_CHALLENGE

NDIS_WWAN_AUTH_RESPONSE

NDIS_STATUS_WWAN_BASE_STATIONS_INFO

Article • 03/14/2023

The NDIS_STATUS_WWAN_BASE_STATIONS_INFO notification is sent by modem miniport drivers in response to an OID_WWAN_BASE_STATIONS_INFO query request to provide the MB host with information about both serving and neighboring base stations.

This notification uses the NDIS_WWAN_BASE_STATIONS_INFO structure.

Requirements

Version: Windows 10, version 1709 **Header**: Ndis.h

See also

OID_WWAN_BASE_STATIONS_INFO

NDIS_WWAN_BASE_STATIONS_INFO

MB base stations information query operations

NDIS_STATUS_WWAN_CONTEXT_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_CONTEXT_STATE notification to send an event notification when the activation state of a particular context changes.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the NDIS_WWAN_CONTEXT_STATE structure.

Remarks

Miniport drivers must also notify the MB Service when context state changes are not caused as a result of a *set* request from the MB Service. For example, miniport drivers must notify the MB Service if the network deactivates a context. Miniport drivers should not implement network initiated context activations.

Miniport drivers must notify Windows directly about all applicable context state changes, such as when processing NDIS_STATUS_WWAN_PACKET_SERVICE or NDIS_STATUS_WWAN_REGISTER_STATE status notifications.

Miniport drivers of MB devices that support separate voice and data connections must follow these guidelines:

- At the time of initialization, the VoiceCallState must be set to WwanVoiceCallStateNone.
- On the start of the voice call, send an event notification with VoiceCallState set to WwanVoiceCallStateInProgress. All the other members must reflect their current state. In case of no active connection during the voice call, the ConnectionId should be set to "0" .
- Once the voice call is completed, send an event notification with VoiceCallState set to WwanVoiceCallStateHangUp. All the other members must reflect their current state. In case of no active connection during the voice call hang up, the ConnectionId should be set to "0". After this event, the VoiceCallState must be set to **WwanVoiceCallStateNone** in the miniport driver.

Requirements

Version	Available in Windows 7 and later versions of

	Windows.
Header	Ndis.h

See also

NDIS_WWAN_CONTEXT_STATE

OID_WWAN_PROVISIONED_CONTEXTS

NDIS_STATUS_WWAN_DEVICE_CAPS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_CAPS notification to respond to OID_WWAN_DEVICE_CAPS query requests.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_DEVICE_CAPS** structure.

Remarks

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_DEVICE_CAPS

NDIS_WWAN_DEVICE_CAPS

NDIS_STATUS_WWAN_DEVICE_CAPS_EX

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_DEVICE_CAPS_EX** notification to inform the MB service about the completion of a previous OID_WWAN_DEVICE_CAPS_EX query request.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_DEVICE_CAPS_EX** structure.

Requirements

Version	Windows 10, version 1703
Header	Ndis.h

See also

OID_WWAN_DEVICE_CAPS_EX

NDIS_WWAN_DEVICE_CAPS_EX

NDIS_STATUS_WWAN_DEVICE_RESET_STATUS

Article • 03/14/2023

The NDIS_STATUS_WWAN_DEVICE_RESET_STATUS notification is sent by a modem miniport driver to inform the MB host of the reset status of the modem device. This notification is sent as an asynchronous response to an OID_WWAN_DEVICE_RESET set request.

This notification uses the NDIS_WWAN_DEVICE_RESET_STATUS structure.

Requirements

Version: Windows 10, version 1709 **Header**: Ndis.h

See also

OID_WWAN_DEVICE_RESET

NDIS_WWAN_DEVICE_RESET_STATUS

MB modem reset operations

NDIS_STATUS_WWAN_DEVICE_SERVICE_EVENT

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_SERVICE_EVENT indication to notify the MB Service of device service changes.

Miniport drivers can only use this notification to send unsolicited events.

This notification uses the NDIS_WWAN_DEVICE_SERVICE_EVENT structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

NDIS_WWAN_DEVICE_SERVICE_EVENT

NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE indication to implement the transaction completion response for OID_WWAN_DEVICE_SERVICE_COMMAND.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_DEVICE_SERVICE_RESPONSE** structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

OID_WWAN_DEVICE_SERVICE_COMMAND

NDIS_WWAN_DEVICE_SERVICE_RESPONSE

NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION indication to report the completion of a device service session state change originated by OID_WWAN_DEVICE_SERVICE_SESSION.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_DEVICE_SERVICE_SESSION_INFO** structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

OID_WWAN_DEVICE_SERVICE_SESSION

NDIS_WWAN_DEVICE_SERVICE_SESSION_INFO

NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION_READ

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION_READ notification to inform the MB Service that data has been received from an open device service session.

Miniport drivers can only use this notification to send unsolicited events.

This notification uses the NDIS_WWAN_DEVICE_SERVICE_SESSION_READ structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

NDIS_WWAN_DEVICE_SERVICE_SESSION_READ

NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION_WRITE_COMPLETE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION_WRITE_COMPLETE notification to report the status of a write operation on a device service session.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the NDIS_WWAN_DEVICE_SERVICE_SESSION_WRITE_COMPLETE structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

NDIS_WWAN_DEVICE_SERVICE_SESSION_WRITE_COMPLETE

NDIS_STATUS_WWAN_DEVICE_SERVICE_SUBSCRIPTION

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_SERVICE_SUBSCRIPTION notification to inform the MB Service about a device service subscription in response to an OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS set request.

Miniport drivers cannot use this notification to send unsolicited events.

This indication uses the **NDIS_WWAN_DEVICE_SERVICE_SUBSCRIPTION** structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS

NDIS_WWAN_DEVICE_SERVICE_SUBSCRIPTION

NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS notification to report the completion of a query of OID_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS** structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

OID_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS

NDIS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS

NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO** notification to inform the MB service about the completion of a previous OID_WWAN_DEVICE_SLOT_MAPPING_INFO query or set request.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_DEVICE_SLOT_MAPPING_INFO** structure.

Requirements

Version	Windows 10, version 1703
Header	Ndis.h

See also

OID_WWAN_DEVICE_SLOT_MAPPING_INFO

NDIS_WWAN_DEVICE_SLOT_MAPPING_INFO

NDIS_STATUS_WWAN_HOME_PROVIDER

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_HOME_PROVIDER notification to inform the MB Service about the completion of OID_WWAN_HOME_PROVIDER query requests.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_HOME_PROVIDER** structure.

Remarks

Miniport drivers must comply with the following rules when responding to OID_WWAN_HOME_PROVIDER query requests:

- GSM-Based Devices: The home provider name can be retrieved from the Subscriber Identity Module (SIM) using several methods, such as from the EFSPN elementary file in the SIM (EFSPN is defined in the 3GPP TS 31.102 under section Service Provider Name), or from the operator-specific extensions when the EFSPN is not provisioned. You should refer to the operator-specific requirements specifications for obtaining the home provider name, extracting MCC-MNC from IMSI, and performing a look up in the GSMA SE.13 database. Contact the operator when retrieving operator-specific home provider names if the EFSPN is not provisioned. If a SIM is not provisioned with a home provider name through EFSPN or any other mechanism, miniport drivers should set the provider name to **NULL**.

  For details about a SIM card's file system, see the 3GPP TS 11.11 specification. If the provider identification is not provisioned in the Subscriber Identity Module (SIM card), miniport drivers should return WWAN_STATUS_READ_FAILURE.

- CDMA-Based Devices: Returning the home provider name is mandatory. It is recommended that IHVs provide this information in their device as part of network personalization. If the provider identity is not available, miniport drivers for CDMA-based providers must set the **Provider.ProviderId** member of the NDIS_WWAN_HOME_PROVIDER structure to WWAN_CDMA_DEFAULT_PROVIDER_ID.

Miniport drivers must return this information when the device ready-state changes to **WwanReadyStateInitialized** and format all the members of the WWAN_PROVIDER structure, as appropriate.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_HOME_PROVIDER

NDIS_WWAN_HOME_PROVIDER

NDIS_STATUS_WWAN_IP_ADDRESS_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_IP_ADDRESS_STATE notification to inform the MB service about changes to the IP configuration for an additional PDP context.

This notification uses the NDIS_WWAN_IP_ADDRESS_STATE structure.

Remarks

This notification must be sent on the NDIS port associated with the additional PDP context session.

Miniport drivers should send this notification after an additional PDP context has been successfully activated and the IP configuration has been acquired for that context. If the device indicates unsolicited IP configuration changes post-context activation, then miniport drivers should send an unsolicited indication with this notification with the updated IP configuration.

Requirements

Version	Available in Windows 8.1 and later versions of Windows.
Header	Ndis.h (include Ndis.h)

See also

NDIS_WWAN_IP_ADDRESS_STATE

NDIS_STATUS_WWAN_LTE_ATTACH_CONFIG

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_LTE_ATTACH_CONFIG notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_LTE_ATTACH_CONFIG Query or Set request.

Unsolicited events are sent if the default LTE attach context is updated by the network either over the air (OTA) or by short message service (SMS). In this case, the miniport driver must update the default LTE attach contexts and send this notification to the host OS with the updated list.

This status notification uses the **NDIS_WWAN_LTE_ATTACH_CONTEXTS** structure.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB LTE Attach Operations

OID_WWAN_LTE_ATTACH_CONFIG

NDIS_WWAN_LTE_ATTACH_CONTEXTS

NDIS_STATUS_WWAN_LTE_ATTACH_STATUS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_LTE_ATTACH_STATUS notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_LTE_ATTACH_STATUS Query request.

Unsolicited events are sent if a context for LTE attach is activated, which could be when a SIM is inserted for example. In this case, the miniport driver should send this notification to the host OS.

This status notification uses the **NDIS_WWAN_LTE_ATTACH_STATUS** structure.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB LTE Attach Operations

OID_WWAN_LTE_ATTACH_STATUS

NDIS_WWAN_LTE_ATTACH_STATUS

NDIS_STATUS_WWAN_MODEM_CONFIG_INFO

Article • 03/14/2023

MBB drivers use the **NDIS_STATUS_WWAN_MODEM_CONFIG_INFO** notification to inform the MB service about the completion of a previous OID_WWAN_MODEM_CONFIG_INFO query request.

MBB drivers must only send an unsolicited **NDIS_STATUS_WWAN_MODEM_CONFIG_INFO** when the configuration state of the modem has changed.

This notification uses the **NDIS_WWAN_MODEM_CONFIG_INFO** structure.

Requirements

Version	Windows 10, version 1709
Header	Ndis.h

See also

OID_WWAN_MODEM_CONFIG_INFO

NDIS_WWAN_MODEM_CONFIG_INFO

NDIS_STATUS_WWAN_MODEM_LOGGING_CONFIG

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_MODEM_LOGGING_CONFIG** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_MODEM_LOGGING_CONFIG Query or Set request.

Miniport drivers send this notification as an unsolicited event in scenarios where the modem needs to inform the OS about internal changes. Currently, in Windows 10, version 1903, these scenarios do not occur.

This notification uses the **NDIS_WWAN_MODEM_LOGGING_CONFIG** structure.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB modem logging with DSS

OID_WWAN_MODEM_LOGGING_CONFIG

NDIS_WWAN_MODEM_LOGGING_CONFIG

NDIS_STATUS_WWAN_MPDP_LIST

Article • 03/14/2023

The **NDIS_STATUS_WWAN_MPDP_LIST** notification is sent by a mobile broadband miniport driver to inform the MB service about the completion of a previous OID_WWAN_MPDP query request.

This notification is not sent as an unsolicited event.

This notification uses the **NDIS_WWAN_MPDP_LIST** structure.

Requirements

Version: Windows 10, version 1809

Header: Ndis.h

NDIS_STATUS_WWAN_MPDP_STATE

Article • 03/14/2023

The **NDIS_STATUS_WWAN_MPDP_STATE** notification is sent by a mobile broadband miniport driver to inform the MB service about the completion of a previous OID_WWAN_MPDP set request.

This notification is not sent as an unsolicited event.

This notification uses the **NDIS_WWAN_MPDP_STATE** structure.

Requirements

Version: Windows 10, version 1809

Header: Ndis.h

NDIS_STATUS_WWAN_NETWORK_BLACKLIST

Article • 03/14/2023

ⓘ Important

Bias-free communication

Microsoft supports a diverse and inclusive environment. This article contains references to terminology that the Microsoft **style guide for bias-free communication** recognizes as exclusionary. The word or phrase is used in this article for consistency because it currently appears in the software. When the software is updated to remove the language, this article will be updated to be in alignment.

Miniport drivers use the **NDIS_STATUS_WWAN_NETWORK_BLACKLIST** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_NETWORK_BLACKLIST Query or Set request.

Unsolicited events are sent if any of the blacklist states have changed from actuated to not actuated, or vice versa. For example, if a SIM is inserted whose provider matches the SIM provider blacklist.

This notification uses the **NDIS_WWAN_NETWORK_BLACKLIST** structure.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB Network Blacklist Operations

OID_WWAN_NETWORK_BLACKLIST

NDIS_WWAN_NETWORK_BLACKLIST

NDIS_STATUS_WWAN_NETWORK_PARAMS_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_NETWORK_PARAMS_STATE notification to inform the MB Service about changes to network configuration data and/or policy information.

Drivers send an NDIS_STATUS_WWAN_NETWORK_PARAMS_STATE notification in response to an OID query request of OID_WWAN_NETWORK_PARAMS.

This notification uses the **NDIS_WWAN_NETWORK_PARAMS_INFO** structure which contains a **WWAN_NETWORK_PARAMS_INFO** structure.

Remarks

For more information see OID_WWAN_NETWORK_PARAMS.

Requirements

Requirement	Value
Minimum supported client	Windows 11
Version	Windows Server 2022. NDIS 6.84 and later.
Header	Ndis.h

See also

OID_WWAN_NETWORK_PARAMS

NDIS_WWAN_NETWORK_PARAMS_INFO

WWAN_NETWORK_PARAMS_INFO

NDIS_STATUS_WWAN_NITZ_INFO

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_NITZ_INFO** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_NITZ Query request.

Miniport drivers send this notification as an unsolicited event to provide the current network time and time zone intformation.

This notification uses the **NDIS_WWAN_NITZ_INFO** structure.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB modem logging with DSS

OID_WWAN_NITZ

NDIS_WWAN_NITZ_INFO

NDIS_STATUS_WWAN_PACKET_SERVICE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_PACKET_SERVICE notification to inform the MB Service when packet service availability changes, including to notify of a change to the type of packet data service currently used.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the NDIS_WWAN_PACKET_SERVICE_STATE structure.

Remarks

CDMA-based miniport drivers can automatically initiate packet-attach service if there is no resource allocation/release is possible and can send the event notification to the MB Service.

Miniport drivers should obey the following guidelines for event notifications:

- Miniport drivers should set **AvailableDataClasses** is set to WWAN_DATA_CLASS_NONE during miniport driver initialization. Thereafter, miniport drivers must notify the MB Service whenever there is any change to **AvailableDataClasses**.

- Miniport drivers should set **CurrentDataClass** to WWAN_DATA_CLASS_NONE during miniport driver initialization. Thereafter, miniport drivers must notify the MB Service whenever there is any change to **CurrentDataClass** . Miniport drivers should send an NDIS_STATUS_LINK_STATE notification if the change to **CurrentDataClass** results in a change of the transmit or receive link speed.

- Miniport drivers must notify the MB Service whenever there is any change in Packet Service attach state.

Miniport drivers should return *query* results according to the following rules:

- Miniport drivers must return WWAN_STATUS_SUCCESS with **WwanPacketServiceStateAttaching** whenever the device attempts to packet-attach.

- Miniport drivers should return WWAN_STATUS_SUCCESS with **WwanPacketServiceStateDetaching** whenever the device attempts to packet-detach.

- When the device is in final state, miniport drivers should return WWAN_STATUS_SUCCESS along with the appropriate current state (**WwanPacketServiceStateAttached** or **WwanPacketServiceStateDetached**)

- Miniport drivers must list all the available data-classes; not just the highest data-class available. This applies to both *query* operations as well as event notifications.

Miniport drivers should return *set* results according to the following rules:

- Return WWAN_STATUS_SUCCESS, if *set* request with **WwanPacketServiceActionAttach**, is issued by the Service and the device is already in the packet-attached state.

- Return WWAN_STATUS_SUCCESS, if *set* request with **WwanPacketServiceActionDetach**, is issued by the Service and the device is already in the packet-detached state.

- Never return transient states for the *set* request. Only the final states **WwanPacketServiceStateAttached** or **WwanPacketServiceStateDetached** must be returned after the successful completion of the packet service operation with WWAN_STATUS_SUCCESS

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_PACKET_SERVICE_STATE

OID_WWAN_PACKET_SERVICE

NDIS_STATUS_WWAN_PCO_STATUS

Article • 03/14/2023

The **NDIS_STATUS_WWAN_PCO_STATUS** notification is sent by a modem miniport driver to inform the OS of the current Protocol Configuration Options (PCO) state in the modem. Modem miniport drivers will send this notification in the following three scenarios:

1. When a new PCO value has arrived on an activated connection.
2. When the modem has PCO value readily available when a connection is activated or bridged by the host.
3. In response to an OID_WWAN_PCO query request from the host.

When a new PCO value has arrived, this notification will be unsolicited and sent with the latest PCO value from the network. The notification will come up with the NDIS port number that corresponds to the activated connection's PDN.

When a connection is activated or bridged from the host, the modem should check whether it has the PCO value cached or not. If it does, it will send up a notification to the host with the NDIS port number that corresponds to the PDN that the host has activated or bridged.

This notification will be used to notify the host that an **OID_WWAN_PCO** query request has been completed, with the PCO value included in the notification. The host expects the modem to pass the complete structure of PCO values on the PDN corresponding to the port number.

If PCO functionality is supported by the modem but no PCO value is received from the network when the host sends an **OID_WWAN_PCO** query request, the modem should return an **NDIS_STATUS_WWAN_PCO_STATUS** notification with an empty WWAN_PCO_VALUE payload.

This notification uses the NDIS_WWAN_PCO_STATUS structure.

> ⓘ **Note**
>
> Currently, in Windows 10, version 1709 and later, some modems are only able to provide operator specific PCO elements. If a PCO data structure is received by modem but there is no applicable operator specific PCO element, to avoid unnecessary device wakeup, the modem should not advertise the PCO notification to the OS.

Requirements

Version: Windows 10, version 1709 **Header**: Ndis.h

See also

OID_WWAN_PCO

NDIS_WWAN_PCO_STATUS

WWAN_PCO_VALUE

MB Protocol Configuration Options (PCO) operations

NDIS_STATUS_WWAN_PIN_INFO

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_PIN_INFO notification to respond to OID query and set requests of OID_WWAN_PIN.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_PIN_INFO** structure.

Remarks

Miniport drivers should return information about the Personal Identity Number (PIN) that the MB device currently expects in response to a query request. Miniport drivers should return the status notification filled in as described in sections below in response to a set request.

Responding to WwanPinOperationEnter Requests

When miniport drivers use the NDIS_STATUS_WWAN_PIN_INFO notification to respond to **WwanPinOperationEnter** requests, they should implement these procedures:

- For successful **WwanPinOperationEnter** query requests, when the MB device no longer requires a PIN, miniport drivers must set **uStatus** to WWAN_STATUS_SUCCESS and **PinType** to **WwanPinTypeNone**.

- For failed **WwanPinOperationEnter** requests, miniport drivers must set **uStatus** to WWAN_STATUS_FAILURE and include applicable data as per the following details:

    - PIN Disabled or PIN Not Expected: For **WwanPinOperationEnter** set requests, when the corresponding PIN is either disabled or currently not expected by the MB device, miniport drivers must set **PinType** to **WwanPinTypeNone**. All other members are ignored.

    - PIN Not Supported: If the given PIN is not supported by the MB device, miniport drivers must set **uStatus** to WWAN_STATUS_NO_DEVICE_SUPPORT.

    - PIN Retrial: In this mode, the MB device requires the PIN to be re-entered as the **AttemptsRemaining** value is still non-zero for this particular type of PIN. Miniport drivers must set **PinType** to the same value as that of **PinType** in NDIS_WWAN_SET_PIN.

  - PIN Blocking: The PIN is blocked when **AttemptsRemaining** is zero. If the PIN unblock operation is not available, miniport drivers must set **uStatus** to WWAN_STATUS_FAILURE and **PinType** to **WwanPinTypeNone**. All the other members are ignored.

    Note If the MB device supports PIN unblock operations, miniport drivers should follow the PIN Unblocking step to respond to the request.

  - PIN Unblocking: The PIN is blocked when **AttemptsRemaining** is zero. To unblock the PIN, the MB device may request a corresponding PIN Unlock Key (PUK), if applicable. In this case, miniport drivers must set **PinType** to the corresponding WwanPinType*Xxx*PUK with the relevant details.

  - Blocked PUK: If the number of failed trials exceeds the preset value for entering the WwanPinType*Xxx*PUK, then the PUK becomes blocked. Miniport drivers must signal this by setting **uStatus** to WWAN_STATUS_FAILURE and **PinType** to **WwanPinTypeNone**. In case PUK1 is blocked, miniport drivers must send an NDIS_STATUS_WWAN_READY_INFO with **ReadyState** set to **WwanReadyStateBadSim**.

Responding to WwanPinOperationEnable, WwanPinOperationDisable, or WwanPinOperationChange Requests

When miniport drivers use the NDIS_STATUS_WWAN_PIN_INFO notification to respond to **WwanPinOperationEnable**, **WwanPinOperationDisable**, and **WwanPinOperationChange**, they should implement the following operations:

- For successful requests, miniport drivers must set **uStatus** to WWAN_STATUS_SUCCESS. For other members in WWAN_PIN_INFO, see the following circumstances.

- Miniport drivers must set **uStatus** to WWAN_STATUS_SUCCESS for PIN-enable and PIN-disable operations when the PIN is already in the requested state. Miniport drivers must set **PinType** to **WwanPinTypeNone**. Other members are ignored.

- When a PIN mode is changed from disabled to enabled, the PIN state should be WwanPinStateNone.

- If PIN1 is enabled, the PIN state shall become WwanPinStateEnter when power is cycled to the MB device.

- For all other PINs, the PIN state can change from WwanPinStateNone to WwanPinStateEnter depending on MB device specific conditions.

- PIN Not Supported: If a PIN operation is not supported by the MB device, miniport drivers must set **uStatus** to WWAN_STATUS_NO_DEVICE_SUPPORT. For example, enabling and disabling PIN2 is not typically supported by MB devices so the above error code must be returned. All other members are ignored.

- PIN Must be Entered: If a PIN operation requires a PIN to be entered, miniport drivers must set **uStatus** to WWAN_STATUS_PIN_REQUIRED and **PinType** to WwanPinType*Xxx*. Other members are ignored.

- PIN Change Operation: If the MB device restricts the change of PIN value only when it is in enabled state, a request to change in disabled state must be returned with WWAN_STATUS_PIN_DISABLED.

- PIN Retrial: On failure, miniport drivers must set **uStatus** to WWAN_STATUS_FAILURE, and **PinType** to the same value as specified in NDIS_WWAN_SET_PIN. Other members are ignored except for **AttemptsRemaining**. This may occur when an incorrect PIN is entered.

- PIN Blocking: The PIN is blocked when the number of **AttemptsRemaining** is zero. If the PIN unblock operation is not available, miniport drivers must set **uStatus** to WWAN_STATUS_FAILURE and **PinType** to **WwanPinTypeNone**. **AttemptsRemaining** should be set to 0 and all the other members are ignored.

  Note If the MB device supports PIN unblock operations, miniport drivers should follow the PIN Unblocking step to respond to the request.

- Unblocking PIN: The PIN is blocked when **AttemptsRemaining** is zero. To unblock the PIN, the MB device may request a corresponding PUK, if applicable. In this case, miniport drivers must set **uStatus** to WWAN_STATUS_FAILURE, **PinType** to the corresponding WwanPinType*Xxx*PUK, **PinState** to **WwanPinStateEnter**, and **AttemptsRemaining** should have the number of attempts allowed to enter a valid PUK.

  If PIN blocking results in the MB device or SIM becomes blocked, miniport drivers must send an event notification with **ReadyState** set to **WwanReadyStateDeviceLocked**.

- If there is an active PDP context at the time of PIN1 blocking, miniport drivers must deactivate the PDP context and send notifications to the operating system about the PDP deactivation and link state change.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_PIN

NDIS_STATUS_WWAN_PIN_INFO

NDIS_STATUS_WWAN_PIN_LIST

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_PIN_LIST notification to respond to OID query requests of OID_WWAN_PIN_LIST.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_PIN_LIST** structure.

Remarks

This INDICATION is a response only notification to OID query requests of OID_WWAN_PIN_LIST. Unsolicited indications are not expected for this INDICATION.

Any change in the PIN-entry mode caused as a result of an OID_WWAN_PIN enable or disable operation will not result in an NDIS_STATUS_WWAN_PIN_LIST INDICATION.

Note that the current PinMode for all of the PINs that the device supports must be updated to reflect the current state by the miniport driver on each query request.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_PIN_LIST

NDIS_WWAN_PIN_LIST

NDIS_STATUS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS notification to respond to a previous OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS*query* request.

Miniport drivers may also use this notification to inform the MB Service about the update as a result of a OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS *set* request from the MB Service. A response to an OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS *set* request must contain zero elements in the **PreferredListHeader** member. Miniport drivers can also send unsolicited events with this notification to inform the MB Service that the Preferred Multi-Carrier Provider List (PMCPL) has changed.

This notification uses the **NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS** structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS

NDIS_STATUS_WWAN_PREFERRED_PROVIDERS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_PREFERRED_PROVIDERS notification to inform the MB Service that the Preferred Provider List (PPL) has changed.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the **NDIS_WWAN_PREFERRED_PROVIDERS** structure.

Remarks

In some cases, the PPL (for GSM-based devices) is updated by the network either Over-The-Air (OTA) or by Short Message Service (SMS). The miniport driver must update the PPL accordingly. Afterwards, miniport drivers must notify the MB Service about the updates using this INDICATION with the updated PPL. For GSM-based networks, the **PreferredListHeader** member of the NDIS_WWAN_PREFERRED_PROVIDERS structure must point to the updated PPL.

Miniport drivers use this INDICATION to inform the MB Service about the update as a result of a OID_WWAN_PREFERRED_PROVIDERS set request from the MB Service. A response to an OID_WWAN_PREFERRED_PROVIDERS set request must contain zero elements in the **PreferredListHeader** member.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_PREFERRED_PROVIDERS

OID_WWAN_PREFERRED_PROVIDERS

NDIS_STATUS_WWAN_PRESHUTDOWN_STATE

Article • 03/14/2023

The NDIS_STATUS_WWAN_PRESHUTDOWN_STATE notification is a one-way notification from the MBB driver to the host. The MBB driver sends up this notification when the modem has finished all operations required before shutdown.

This notification uses the NDIS_WWAN_PRESHUTDOWN_STATE structure.

Requirements

Version	Supported starting with Windows 10, version 1511.
Header	Ndis.h

NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS notification to inform the MB Service about updates to the list of provisioned contexts as a result of a network update.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the NDIS_WWAN_PROVISIONED_CONTEXTS structure.

Remarks

Miniport drivers must set the **ElementType** member of the NDIS_WWAN_PROVISIONED_CONTEXTS structure's **ContextListHeader** to **WwanStructContext**.

In some cases, the list of provisioned contexts is updated by the network either Over-The-Air (OTA) or by Short Message Service (SMS). The miniport driver must update the list of provisioned contexts accordingly. Thereafter, miniport drivers must notify the MB Service about the updates using this INDICATION with the updated list.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_PROVISIONED_CONTEXTS

OID_WWAN_PROVISIONED_CONTEXTS

NDIS_STATUS_WWAN_RADIO_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_RADIO_STATE notification to inform the MB Service when the user changes the hardware radio power, or the device's software-based radio power state changes in response to an OID query or set request of OID_WWAN_RADIO_STATE.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the **NDIS_WWAN_RADIO_STATE** structure.

Remarks

Miniport drivers should return both the current hardware-based and software-based radio power states in response to a query request

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_RADIO_STATE

OID_WWAN_RADIO_STATE

NDIS_STATUS_WWAN_READY_INFO

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_READY_INFO notification to inform the MB Service of device ready-state changes in response to OID_WWAN_READY_INFO query requests.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the **NDIS_WWAN_READY_INFO** structure.

Remarks

Miniport drivers must report all device ready-state changes as an unsolicited event. When the miniport driver initializes the MB device, the miniport driver must set the WWAN_READY_INFO **ReadyState** member to **WwanReadyStateOff**. Thereafter, miniport drivers must report any device ready-state change to the MB Service through this notification. For example, miniport drivers must report a device ready-state change when the **ReadyState** member changes from **WwanReadyStateOff** to **WwanReadyStateDeviceLocked**, or **WwanReadyStateBadSim**, or **WwanReadyStateSimNotInserted**, or any other different device ready-state.

Most device ready-state changes happen when the device initializes the radio stack and the SIM card (if required). A change can also happen during the course of a session between the MB Service and the miniport driver, such as user changing the SIM card. The behavior of the MB Service shall change accordingly based on the new device ready-state.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_READY_INFO

OID_WWAN_READY_INFO

NDIS_STATUS_WWAN_REGISTER_PARAMS_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_REGISTER_PARAMS_STATE notification to inform the MB Service about the 5G-specific registration parameters used by the MB device.

Drivers send an NDIS_STATUS_WWAN_REGISTER_PARAMS_STATE notification in response to an OID query or set request of OID_WWAN_REGISTER_PARAMS.

This notification uses the **NDIS_WWAN_REGISTER_PARAMS_INFO** structure which contains a **WWAN_REGISTRATION_PARAMS_INFO** structure.

Remarks

For more information see OID_WWAN_REGISTER_PARAMS.

Requirements

Requirement	Value
Minimum supported client	Windows 11
Version	Windows Server 2022. NDIS 6.84 and later.
Header	Ndis.h

See also

WWAN_REGISTRATION_PARAMS_INFO

NDIS_WWAN_REGISTER_PARAMS_INFO

OID_WWAN_REGISTER_PARAMS

NDIS_STATUS_WWAN_REGISTER_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_REGISTER_STATE notification to communicate changes to the MB device's registration state to the MB Service.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the NDIS_WWAN_REGISTRATION_STATE structure.

Remarks

As the registration state of the device changes, the miniport driver must send appropriate indications so that the MB Service can reflect the correct state to the user.

Registration state changes due to a number of reasons. It may directly result from *set* requests from the MB Service for OID_WWAN_REGISTER_STATE such as a transient state transition from **WwanRegisterStateSearching** to **WwanRegisterStateHome**. It may also result from automatic operations by the miniport driver in the case of automatic provider selection. Finally, it may be caused by change of network availability, for example, losing network coverage may result in transition from **WwanRegisterStateHome** to **WwanRegisterStateDeregistered**.

Except for the changes caused by MB Service OID_WWAN_REGISTER_STATE requests, the miniport driver shall notify the MB Service whenever the registration state changes regardless of the underlying cause.

CDMA devices do not support the MB Service initiated registration and deregistration. However, a device initiated register state change notifications based on the availability or non-availability of the carrier network must be sent to the MB Service. CDMA devices must do automatic registration.

For devices that do automatic registration on power-up, irrespective of the current registration mode--auto or manual, the miniport driver must send the register state notification on successful registration.

For manual registration, the MB Service shall only initiate registration after the miniport driver indicates that **ReadyState** is **WwanReadyStateInitialized**.

Miniport driver must use the following guidelines while responding to *set* requests:

- Drivers must not respond with transient state for a *set* requests. Transient state for registration is **WwanRegisterStateSearching**.

- When **RegisterAction** is set to **WwanRegisterActionManual**, if the provider is not visible when the miniport driver receives the request, the miniport driver shall return error code WWAN_STATUS_PROVIDER_NOT_VISIBLE. The device must not switch to automatic registration because of a failure in setting the manual mode. If the device was earlier set to manually register to another network, this request should change the device to register to the network specified in the request. The value of **RegisterState** in response to the request should be set to **WwanRegisterStateDeregistered**.

- When **RegisterAction** is set to **WwanRegisterActionManual**, if the miniport driver has already registered with the same network that is been requested, it shall respond with WWAN_STATUS_SUCCESS.

- The driver should attempt to register to the requested data-class in the set OID_WWAN_REGISTER request. If the miniport driver cannot register to the requested data-class, it should register to the best possible data-class. This is also applicable when the device is already registered to a provider (automatic and manual registration mode) with some other data-class. Any change in the data-class should also result in NDIS_STATUS_WWAN_PACKET_SERVICE notification.

- When **RegisterAction** is set to **WwanRegisterActionManual**, and the Radio is OFF, the miniport driver must program the device to manual registration mode and complete the request with the transaction notification. The **RegisterState** should be set to **WwanRegisterStateDeregistered**. The device must attempt a manual registration when the Radio changes to ON state and the event notification must be sent.

- When **RegisterAction** is set to **WwanRegisterActionAutomatic**, and the Radio is OFF, the miniport driver must program the device to automatic registration mode and must complete the request with the transaction notification. The **RegisterState** should be set to **WwanRegisterStateDeregistered**. The device must do an automatic registration when the Radio goes to ON state and the event notification must be sent.

- In case of emergency state registration (**WwanRegisterStateDenied**), the **uStatus** should be set to WWAN_STATUS_SUCCESS and NDIS_STATUS_WWAN_READY_INFO notification must be sent with **EmergencyMode** set to **WwanEmergencyModeOn**.

- For using the state **WwanRegisterStateDeregistered** the miniport driver must use the following guidelines:

  - **WwanRegisterStateDeregistered** is used by the miniport drivers for notifying the MB Service that the Radio is OFF but the request for **RegisterAction** is completed.

  - **WwanRegisterStateDeregistered** is used by the miniport drivers for notifying the MB Service of a network initiated deregistration.

  - **WwanRegisterStateDeregistered** is used by the miniport drivers for notifying the MB Service of the lost connectivity to the network due to no network coverage.

- GSM and CDMA devices must send the register state notification to notify the availability or non-availability of the carrier for a PS connection. When the MB device detects the availability of the carrier network, it must send an event notification with one of the appropriate register states--**WwanRegisterStateHome**, **WwanRegisterStateRoaming**, or **WwanRegisterStatePartner**. On losing the carrier network signal, an event notification with **WwanRegisterStateDeregistered** must be indicated to the MB Service.

The miniport driver returns the query result according to the following rules:

- When the device is trying to lock on to a provider during registration, the miniport driver shall set **RegisterState** as **WwanRegisterStateSearching**. Both the **ProviderName** and **RoamingText** members should be set to **NULL**. In case of Manual register mode, **ProviderId** must be filled in with the ProviderId from the last manual registration set request. **ProviderId** can be set to **NULL** in case of automatic register mode.

- This is a transient state as the miniport driver will eventually move to a stable state at the end of registration, for example, **WwanRegisterStateHome**, **WwanRegisterStatePartner**, or **WwanRegisterStateRoaming** for a successful registration; or **WwanRegisterStateDenied** for an emergency state registration.

- If the device is not registered with any provider, the miniport driver shall return **WwanRegisterStateDeregistered**. Both the **ProviderName** and **RoamingText** members should be set to **NULL**. In case of Manual register mode, **ProviderId** must be filled in with the ProviderId from the last manual registration set request. **ProviderId** can be set to **NULL** in case of automatic register mode.

- If the device is registered with the home provider, the miniport driver shall set **RegisterState** as **WwanRegisterStateHome**. The **ProviderId** member shall be set to the home provider ID. The **ProviderName** must be set to the name of home provider network. The **RoamingText** member should be set to **NULL**.

- If the device is registered with a roaming provider, the miniport driver shall set **RegisterState** as **WwanRegisterStatePartner** if the provider is a preferred roaming partner or just **WwanRegisterStateRoaming** for a roaming partner, respectively. If the miniport driver does not distinguish the two, it shall set the value to **WwanRegisterStateRoaming**. The **ProviderId** member shall be set to the provider ID of the current provider the device is registered with and the **ProviderName** must be filled in with the current registered provider name. The **RoamingText** member should be set to some provider specific string value if exists or to **NULL** otherwise.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_REGISTRATION_STATE

OID_WWAN_REGISTER_STATE

NDIS_STATUS_WWAN_SAR_CONFIG

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_SAR_CONFIG** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_SAR_CONFIG Query or Set request.

Unsolicited events are not applicable.

This notification uses the **NDIS_WWAN_SAR_CONFIG_INFO** structure.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB SAR Platform Support

OID_WWAN_SAR_CONFIG

NDIS_WWAN_SAR_CONFIG_INFO

NDIS_STATUS_WWAN_SAR_TRANSMISSION_STATUS

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_SAR_TRANSMISSION_STATUS** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_SAR_TRANSMISSION_STATUS Query or Set request.

Unsolicited events are sent when there is a change to the active over-the-air (OTA) channels. For example, if a modem started uploading packet data, it would be required to set up uplink channels when it uses the network data channel so that it can upload payloads. This would trigger the notification to be provided to the operating system.

This notification uses the **NDIS_WWAN_SAR_TRANSMISSION_STATUS_INFO** structure.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB SAR Platform Support

OID_WWAN_SAR_TRANSMISSION_STATUS

NDIS_WWAN_SAR_TRANSMISSION_STATUS_INFO

NDIS_STATUS_WWAN_SET_HOME_PROVIDER_COMPLETE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SET_HOME_PROVIDER_COMPLETE notification to inform the MB Service about the completion of OID_WWAN_HOME_PROVIDER set requests.

This notification uses the **NDIS_WWAN_SET_HOME_PROVIDER** structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

NDIS_STATUS_WWAN_SERVICE_ACTIVATION

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SERVICE_ACTIVATION notification to respond to OID set requests of OID_WWAN_SERVICE_ACTIVATION.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the NDIS_WWAN_SERVICE_ACTIVATION_STATUS structure.

Remarks

Miniport drivers must return the service activation status in response to an OID set request of OID_WWAN_SERVICE_ACTIVATION.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

NDIS_STATUS_WWAN_SIGNAL_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SIGNAL_STATE notification to send a signal strength notification when measured signal strength travels outside the threshold within a pre-defined interval.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the NDIS_WWAN_SIGNAL_STATE structure.

Remarks

By default, miniport drivers must notify the MB Service if the Rssi value changes by at least +/-5 decibels from the last reported value, or at a maximum frequency of one indication every 5 seconds. The threshold value is specified in the **SignalState.RssiThreshold** member of the NDIS_WWAN_SIGNAL_STATE structure; while the maximum frequency value is specified in the **SignalState.RssiInterval** member.

The **DeviceCaps.WwanCellularClass** member of the NDIS_WWAN_DEVICE_CAPS structure controls how the Rssi value will be interpreted by the MB Service. If **WwanCellularClass** is **WwanCellularClassGSM**, Rssi is reported as decibels above the device's sensitivity noise floor. If **WwanCellularClass** is **WwanCellularClassCDMA**, Rssi is reported as compensated RSSI (accounts for noise).

Applications should never poll for signal strength. Only in special situations, such as startup, an application might use a *query* request to obtain signal strength.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_SIGNAL_STATE

OID_WWAN_SIGNAL_STATE

NDIS_STATUS_WWAN_SLOT_INFO

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_SLOT_INFO** notification to inform the MB service about the completion of a previous OID_WWAN_SLOT_INFO query request.

Miniport drivers can send a **NDIS_STATUS_WWAN_SLOT_INFO** notification as an unsolicited event when the slot/card state changes.

This notification uses the **NDIS_WWAN_SLOT_INFO** structure.

Requirements

Version	Windows 10, version 1703
Header	Ndis.h

See also

OID_WWAN_SLOT_INFO

NDIS_WWAN_SLOT_INFO

NDIS_STATUS_WWAN_SMS_CONFIGURATION

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SMS_CONFIGURATION notification to inform the MB Service about either the completion of a previous OID_WWAN_SMS_CONFIGURATION query or set request, or an event notification in the case of change in SMS configuration.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the **NDIS_WWAN_SMS_CONFIGURATION** structure.

Remarks

The miniport driver must send this unsolicited indication when the MB device's SMS subsystem is ready for SMS operation.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_SMS_CONFIGURATION

NDIS_WWAN_SMS_CONFIGURATION

NDIS_STATUS_WWAN_SMS_DELETE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SMS_DELETE notification to inform the MB Service about the completion of a previous delete request through OID_WWAN_SMS_DELETE.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_SMS_DELETE_STATUS** structure.

Remarks

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_SMS_DELETE

NDIS_WWAN_SMS_DELETE_STATUS

NDIS_STATUS_WWAN_SMS_RECEIVE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SMS_RECEIVE notification to inform the MB Service about either the completion of a previous read request through a OID_WWAN_SMS_READ query request, or the arrival of a new class-0 (flash/alert) message from the network provider as an event notification.

Miniport drivers can also send unsolicited events with this notification.

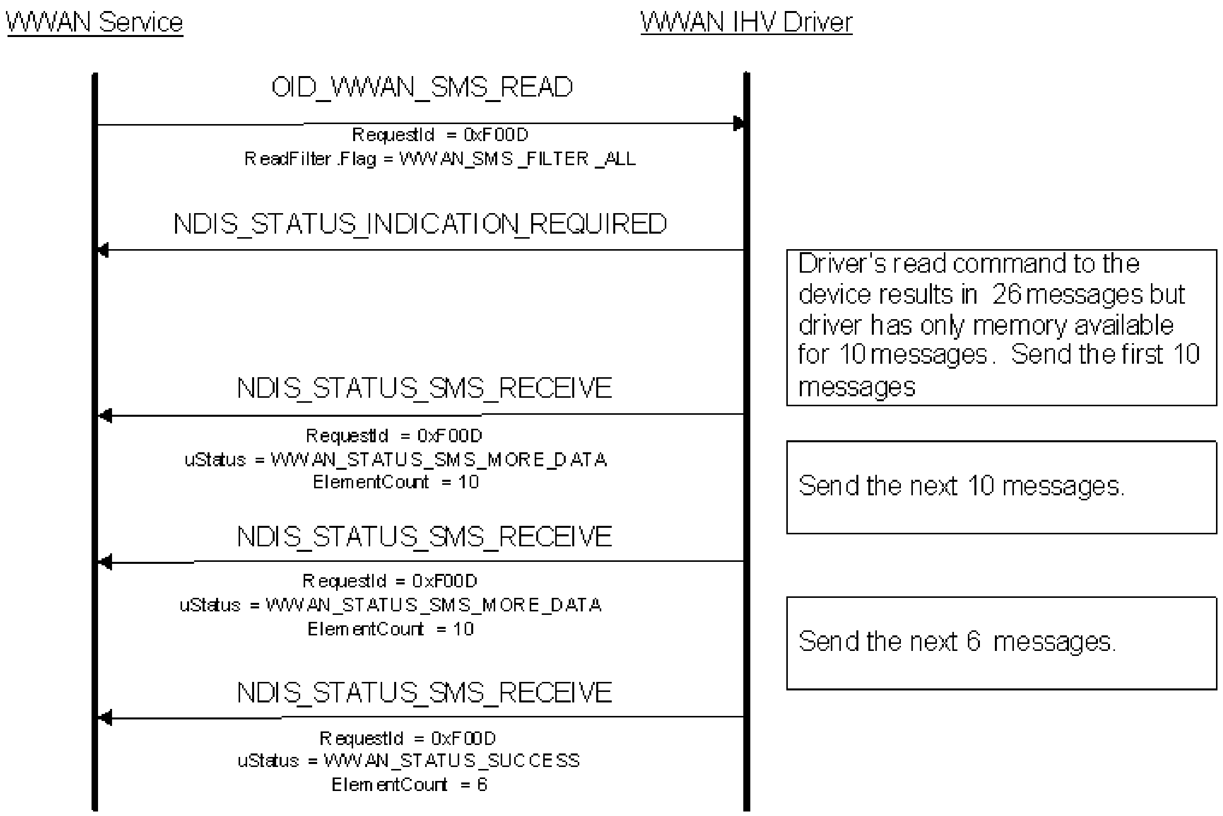
This notification uses the **NDIS_WWAN_SMS_RECEIVE** structure.

Remarks

RequestId is set to "0" by the miniport driver to indicate the arrival of the new class-0 (flash/alert) message. Arrival of new class-0 (flash/alert) messages is dependent on the current network registration state.

If the request for read results in retrieval of large number of SMS records that can't be accommodated in a pre-allocated buffer of miniport driver, then the SMS records can be sent to the MB Service in multiple indications. The uStatus in this case must be set to WWAN_STATUS_SMS_MORE_DATA for the intermediate transactions and the final transaction must end with WWAN_STATUS_SUCCESS.

The following diagram represents the usage of the multiple indication method for large number of SMS record retrieval:



Driver's read command to the device results in 26 messages but driver has only memory available for 10 messages. Send the first 10 messages

Send the next 10 messages.

Send the next 6 messages.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_SMS_READ

NDIS_WWAN_SMS_RECEIVE

NDIS_STATUS_WWAN_SMS_SEND

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SMS_SEND notification to inform the MB Service about the completion of a previous send request through OID_WWAN_SMS_SEND.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_SMS_SEND_STATUS** structure.

Remarks

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_SMS_SEND

NDIS_WWAN_SMS_SEND_STATUS

NDIS_STATUS_WWAN_SMS_STATUS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SMS_STATUS notification to inform the MB Service about the following events:

- The MB device's message store is full.
- A new SMS text message has arrived, with the new message corresponding to *MessageIndex*.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the NDIS_WWAN_SMS_STATUS structure.

Remarks

Miniport drivers must use NDIS_STATUS_WWAN_SMS_STATUS to inform the MB Service about the arrival of all non-class-0 (flash/alert) messages. To inform the MB Service about class-0 (flash/alert) message arrival, miniport drivers must use NDIS_STATUS_WWAN_SMS_RECEIVE.

This indication could be a transactional notification for a *query* request of OID_WWAN_SMS_STATUS or an unsolicited event.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_SMS_STATUS

NDIS_STATUS_WWAN_SMS_RECEIVE

NDIS_STATUS_WWAN_SUPPORTED_DEVICE_SERVICES

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_SUPPORTED_DEVICE_SERVICES notification to inform the MB Service about the completion of OID_WWAN_ENUMERATE_DEVICE_SERVICES query requests.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_SUPPORTED_DEVICE_SERVICES** structure.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

OID_WWAN_ENUMERATE_DEVICE_SERVICES

NDIS_WWAN_SUPPORTED_DEVICE_SERVICES

NDIS_STATUS_WWAN_SYS_CAPS_INFO

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_SYS_CAPS_INFO** notification to inform the MB service about the completion of a previous OID_WWAN_SYS_CAPS_INFO query request.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_SYS_CAPS_INFO** structure.

Requirements

Version	Windows 10, version 1703
Header	Ndis.h

See also

OID_WWAN_SYS_CAPS_INFO

NDIS_WWAN_SYS_CAPS_INFO

NDIS_STATUS_WWAN_UE_POLICY_STATE

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_UE_POLICY_STATE notification to inform the MB Service of device UE policies in response to OID_WWAN_UE_POLICY query requests.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the **NDIS_STATUS_WWAN_UE_POLICY_STATE** structure.

Requirements

Version: Windows 11, version 21H2

Header: Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_UE_POLICY_STATE

OID_WWAN_UE_POLICY

NDIS_STATUS_WWAN_UICC_APP_LIST

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_UICC_APP_LIST** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_UICC_APP_LIST Query request.

Unsolicited events are not applicable.

This notification uses the **NDIS_WWAN_UICC_APP_LIST** structure.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

OID_WWAN_UICC_APP_LIST

NDIS_WWAN_UICC_APP_LIST

NDIS_STATUS_WWAN_UICC_BINARY_RESPONSE

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_UICC_BINARY_RESPONSE** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_UICC_ACCESS_BINARY Query or Set request.

Unsolicited events are not applicable.

This notification uses the **NDIS_WWAN_UICC_RESPONSE** structure.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

OID_WWAN_UICC_ACCESS_BINARY

NDIS_WWAN_UICC_RESPONSE

NDIS_STATUS_WWAN_UICC_FILE_STATUS

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_UICC_FILE_STATUS** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_UICC_FILE_STATUS Query request.

Unsolicited events are not applicable.

This notification uses the **NDIS_WWAN_UICC_FILE_STATUS** structure.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

OID_WWAN_UICC_FILE_STATUS

NDIS_WWAN_UICC_FILE_STATUS

NDIS_STATUS_WWAN_UICC_RECORD_RESPONSE

Article • 03/14/2023

Miniport drivers use the **NDIS_STATUS_WWAN_UICC_RECORD_RESPONSE** notification to inform the mobile broadband (MB) service about the completion of a previous OID_WWAN_UICC_ACCESS_RECORD Query or Set request.

Unsolicited events are not applicable.

This notification uses the **NDIS_WWAN_UICC_RESPONSE** structure.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

OID_WWAN_UICC_ACCESS_RECORD

NDIS_WWAN_UICC_RESPONSE

NDIS_STATUS_WWAN_UICC_RESET_INFO

Article • 03/14/2023

The NDIS_STATUS_WWAN_UICC_RESET_INFO status notification is sent by a modem miniport adapter to inform the MB host of the current passthrough status to a UICC smart card. This notification is sent in the folloiwng two scenarios:

1. After an OID_WWAN_UICC_RESET query request.
2. After UICC reset is complete following an OID_WWAN_UICC_RESET set request, to inform the MB host of the passthrough status of the UICC card post-reset.

This notification uses the NDIS_WWAN_UICC_RESET_INFO structure.

Requirements

Version: Windows 10, version 1709 **Header**: Ndis.h

See also

OID_WWAN_UICC_RESET

NDIS_WWAN_UICC_RESET_INFO

MB low level UICC access

NDIS_STATUS_WWAN_UICC_TERMINAL_CAPABILITY_INFO

Article • 03/14/2023

The NDIS_STATUS_WWAN_UICC_TERMINAL_CAPABILITY_INFO status notification is sent by a modem miniport adapter to inform the MB host of the last terminal capability objects sent to the modem. This notification is sent in response to OID_WWAN_UICC_TERMINAL_CAPABILITY query and set requests.

This notification uses the **NDIS_WWAN_UICC_TERMINAL_CAPABILITY_INFO** structure.

Requirements

Version: Windows 10, version 1607

Header: Ndis.h

See also

OID_WWAN_UICC_TERMINAL_CAPABILITY

NDIS_WWAN_UICC_TERMINAL_CAPABILITY_INFO

MB low level UICC access

NDIS_STATUS_WWAN_USSD

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_USSD notification to implement the transaction completion response for Unstructured Supplementary Service Data (USSD) operations with the NDIS_WWAN_USSD_REQUEST structure.

Miniport drivers can also send unsolicited events with this notification using the NDIS_WWAN_USSD_EVENT structure to describe the nature of the USSD event.

Requirements

Version	Supported starting with Windows 8.
Header	Ndis.h

See also

NDIS_WWAN_USSD_REQUEST

NDIS_WWAN_USSD_EVENT

NDIS_STATUS_WWAN_VENDOR_SPECIFIC

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_VENDOR_SPECIFIC notification to implement the transaction completion response for vendor specific operation or vendor specific change notifications.

Miniport drivers can also send unsolicited events with this notification.

This notification uses the **NDIS_WWAN_VENDOR_SPECIFIC** structure.

Remarks

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

NDIS_WWAN_VENDOR_SPECIFIC

OID_WWAN_VENDOR_SPECIFIC

NDIS_STATUS_WWAN_VISIBLE_PROVIDERS

Article • 03/14/2023

Miniport drivers use the NDIS_STATUS_WWAN_VISIBLE_PROVIDERS notification to inform the MB Service about the completion of OID_WWAN_VISIBLE_PROVIDERS query requests.

Miniport drivers cannot use this notification to send unsolicited events.

This notification uses the **NDIS_WWAN_VISIBLE_PROVIDERS** structure.

Remarks

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ndis.h

See also

OID_WWAN_VISIBLE_PROVIDERS

NDIS_WWAN_VISIBLE_PROVIDERS

Supplemental MB Documentation Topics

Article • 03/14/2023

The following sections provide supplemental information for developers of MB devices.

HOST Shutdown Device Service

IHV Guidance for Implementing Multimode and Multicarrier Capable MB Devices

Mobile Broadband Device Firmware Update

Mobile Broadband Class Driver Logs: Event Trace Log Tracing

Mobile Broadband Implementation Guidelines for USB Devices

Collecting Mobile Broadband Logs

Analyzing Mobile Broadband Logs

Analyzing Mobile Broadband Logs in Wireshark

TextAnalysisTool Filter Files

HOST Shutdown Device Service

Article • 12/15/2021

This topic provides guidelines for Mobile Broadband Interface Model (MBIM)-compliant devices to implement and report the described device service when queried by CID_MBIM_DEVICE_SERVICES.

The information in this topic applies to Windows 8 and later.

Microsoft Host Shutdown

The MBIM-compliant device implements and reports the following device service when queried by CID_MBIM_DEVICE_SERVICES. The existing well-known services are defined in section 10.1 of the USB NCM Mobile Broadband Interface Model (MBIM) V1.0 specification. Microsoft extends this to define the following service.

Service Name = **Microsoft Host Shutdown**

UUID = **UUID_MS_HOSTSHUTDOWN**

UUID Value = **883b7c26-985f-43fa-9804-27d7fb80959c**

Defined CIDs for UUID_MS_HOSTSHUTDOWN device service

CID	Minimum OS version
CID_MBIM_MSHOSTSHUTDOWN	Windows 8
CID_MBIM_MSHOSTPRESHUTDOWN	Windows 10, version 1511

CID_MBIM_MSHOSTSHUTDOWN

This command informs the device that the host is shutting down. The MB device may lose power.

CID: CID_MBIM_MSHOSTSHUTDOWN

Command Code: 1

Set: Yes

Query: No

Event: No

Set InformationBuffer payload: N/A

Query InformationBuffer payload: N/A

Completion InformationBuffer payload: N/A

Set: InformationBuffer on MBIM_COMMAND_MSG not used. InformationBuffer of MBIM_COMMAND_DONE not used.

Query: Unsupported

Unsolicited Event: Unsupported

Remarks

The Mobile Broadband Class Driver sends the host shutdown notification to mobile broadband devices supporting this device service, on each host state transition into S4 and S5 states.

This notification is to provide mobile broadband devices with an early indication to allow them to initiate a mobile network deregister message and initiate SIM electrical de-initialization.

The following information summarizes the list of host sent CIDs/CMDs to the device for various system transitions and device power state transitions:

- MSHOSTSHUTDOWN CID is sent to the device on host state transitioning into S4 and S5.
- MBIM_CMD_CLOSE is sent to the device when host puts the device into D3 mode.

	S0	S1/S2/S3	S4	S5
D0	MBIM_CMD_OPEN	N/A	N/A	N/A
D1	N/A	N/A	N/A	N/A
D2	N/A	N/A	N/A	N/A
D3	N/A	MBIM_CMD_CLOSE	MSHOSTSHUTDOWN	MSHOSTSHUTDOWN

CID_MBIM_MSHOSTPRESHUTDOWN

This command notifies the MBIM modem that the system is undergoing pre-shutdown and it should finish all its operations, deregister from the network, and store necessary information to the host for flashless modem cases. The pre-shutdown notification is sent down when the host is preparing to enter S4 and S5 states and is waiting for all services to shut down appropriately.

CID: CID_MBIM_MSHOSTPRESHUTDOWN

Command Code: 2

Set: Yes

Query: No

Notification: No

Set InformationBuffer payload: N/A

Query InformationBuffer payload: N/A

Completion InformationBuffer payload: N/A

Parameters:

	Set	Query	Notification
Command	CID_MBIM_SET_MSHOSTPRESHUTDOWN	N/A	N/A
Response	Empty	N/A	N/A

For the Set operation, InformationBuffer and InformationBufferLength are empty.

Status Codes:

Status code	Description
MBIM_STATUS_SUCCESS	Pre-shutdown operations completed by the modem.
MBIM_STATUS_NO_DEVICE_SUPPORT	The device does not support pre-shutdown and no pre-shutdown operations are needed.

IHV Guidance for Implementing Multimode and Multicarrier Capable MB Devices

Article • 12/15/2021

This topic provides information about implementing support for multiple Radio Access Technologies (RAT) and multiple operators in Windows. It supplements the USB NCM Mobile Broadband Interface Model (MBIM) V1.0 specification that outlines the CIDs required for supporting the multimode multicarrier scenario.

The information in this topic applies to:

- Windows 8

Supporting multimode devices with firmware switching

This topic provides guidance to IHVs for implementing support for multiple Radio Access Technologies (RAT) and multiple operators in Windows. A short summary of the most relevant sections of the spec is provided below.

A multimode network has multiple RATs or cellular classes. A multicarrier device is able to support multiple network providers within the same device. One of the many networks supported can be a multimode network.

In many places in this document it is required that multicarrier providers reported by a multicarrier device as part of CID processing must be set-able as home providers. This section describes how to determine if a provider is settable as a home provider. The ability to set a provider as a home provider may depend on many factors that are device dependent such as being visible, having firmware support for the provider, and having a SIM or equivalent to register with the provider. There may be additional factors. The device should ensure that the device requirements are met for a multicarrier provider before reporting it as a settable home provider

10.4.1 CID_MBIM_DEVICE_CAPS

The capable device will report multiple supported cellular classes in bmCellularClass of MBIM_DEVICE_CAPS as specified in Table 10-13 of the specification. It will also report support for multiple carriers via the MBIMCtrlCapsMultiCarrier Mask in MBIM_CTRL_CAPS (Table 10-13).

Single-carrier multi-mode devices must behave like GSM devices. Such devices must not set any CDMA related capabilities in their MBIM_DEVICE_CAPS_INFO.

10.4.6 CID_MBIM_HOME_PROVIDER

The home provider can be set as specified in **10.3.6.4.**

10.4.8 CID_MBIM_VISIBLE_PROVIDERS

The visible providers CID contain an Action field that specifies whether the host is expecting:

1. Value 0, where device performs a full scan in the context of current home provider.
2. Value 1, where device is being queried to locate visible multicarrier providers that are settable as home providers. The device may choose to accomplish this via a full scan, partial scan, or static list. The multicarrier preferred providers should be tagged as MBIM_PROVIDER_STATE_PREFERRED_MULTICARRIER in the visible providers result as specified in Table 10-34 of the specification.

When devices report a static list of visible multicarrier providers based on location information, programmed using CID_MBIM_LOCATION_INFO, the list should only contain providers valid for that location. As an extension of the above rule, devices should not report the currently registered provider if the location represented by its MCC (Mobile Country Code) is different than the location currently programmed in the device.

10.4.9 CID_MBIM_REGISTER_STATE

The device indicates the current cellular class using the dwCurrentCellularClass field of MBIM_REGISTRATION_STATE in Table 10-48 of the specification.

10.4.30 CID_MBIM_DEVICE_SERVICES

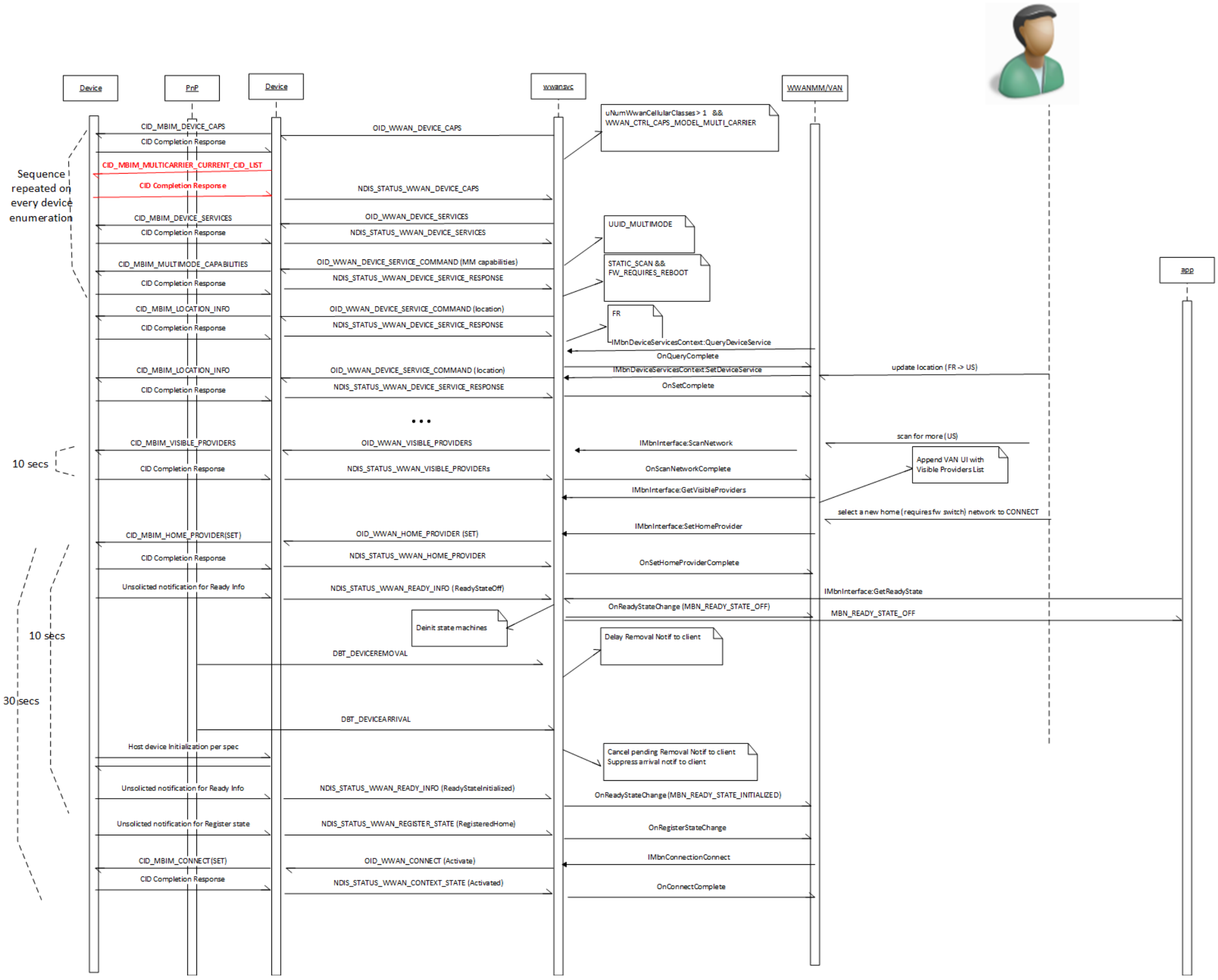
Multicarrier devices are required to report the UUID_MULTICARRIER (described below) device service in response to this CID.

Single-carrier multimode devices are not required to report UUID_MULTICARRIER device service in response to this CID.

10.4.39 CID_MBIM_MULTICARRIER_PROVIDERS

The device uses this CID to report the current and previously added preferred multicarrier providers. This CID is supported when device supports MBIMCtrlCapsMultiCarrier. When the host sets a multicarrier preferred provider, it is not required that the provider is settable as home provider. But when the list is queried by the host, the device should only return multicarrier preferred providers that are settable as home providers.

The following figure provides a sequence diagram of the steps involved in switching a device from its current mode based on a location hint specified by the host. The operation requires the use of a device service described below to get/set additional information from the device. One specific requirement is that the device should ensure that the host has had a chance to recover the pending notification from the device prior to it falling off the bus. The diagram also specifies the time bounds and performance expectations of the various operations.



Device
PnP
Device
wwansvc
WWANMM/VAN
app
Sequence repeated on every device enumeration
CID_MBIM_DEVICE_CAPS
CID Completion Response
CID_MBIM_MULTICARRIER_CURRENT_CID_LIST
CID Completion Response
CID_MBIM_DEVICE_SERVICES
CID Completion Response
CID_MBIM_MULTIMODE_CAPABILITIES
CID Completion Response
CID_MBIM_LOCATION_INFO
CID Completion Response
CID_MBIM_LOCATION_INFO
CID Completion Response
OID_WWAN_DEVICE_CAPS
NDIS_STATUS_WWAN_DEVICE_CAPS
OID_WWAN_DEVICE_SERVICES
NDIS_STATUS_WWAN_DEVICE_SERVICES
OID_WWAN_DEVICE_SERVICE_COMMAND (MM capabilities)
NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE
OID_WWAN_DEVICE_SERVICE_COMMAND (location)
NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE
OID_WWAN_DEVICE_SERVICE_COMMAND (location)
NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE
uNumWwanCellularClasses > 1 &&
WWAN_CTRL_CAPS_MODEL_MULTI_CARRIER
UUID_MULTIMODE
STATIC_SCAN &&
FW_REQUIRES_REBOOT
FR
IMbnDeviceServicesContext:QueryDeviceService
OnQueryComplete
IMbnDeviceServicesContext:SetDeviceService
OnSetComplete
update location (FR -> US)
• • •
10 secs
CID_MBIM_VISIBLE_PROVIDERS
CID Completion Response
OID_WWAN_VISIBLE_PROVIDERS
NDIS_STATUS_WWAN_VISIBLE_PROVIDERs
IMbnInterface:ScanNetwork
OnScanNetworkComplete
IMbnInterface:GetVisibleProviders
scan for more (US)
Append VAN UI with
Visible Providers List
select a new home (requires fw switch) network to CONNECT
CID_MBIM_HOME_PROVIDER(SET)
CID Completion Response
OID_WWAN_HOME_PROVIDER (SET)
NDIS_STATUS_WWAN_HOME_PROVIDER
IMbnInterface:SetHomeProvider
OnSetHomeProviderComplete
Unsolicted notification for Ready Info
NDIS_STATUS_WWAN_READY_INFO (ReadyStateOff)
IMbnInterface:GetReadyState
OnReadyStateChange (MBN_READY_STATE_OFF)
MBN_READY_STATE_OFF
Deinit state machines
Delay Removal Notif to client
10 secs
30 secs
DBT_DEVICEREMOVAL
DBT_DEVICEARRIVAL
Host device Initialization per spec
Cancel pending Removal Notif to client
Suppress arrival notif to client
Unsolicted notification for Ready Info
NDIS_STATUS_WWAN_READY_INFO (ReadyStateInitialized)
OnReadyStateChange (MBN_READY_STATE_INITIALIZED)
Unsolicted notification for Register state
NDIS_STATUS_WWAN_REGISTER_STATE (RegisteredHome)
OnRegisterStateChange
CID_MBIM_CONNECT(SET)
CID Completion Response
OID_WWAN_CONNECT (Activate)
NDIS_STATUS_WWAN_CONTEXT_STATE (Activated)
IMbnConnectionConnect
OnConnectComplete

MULTI-CARRIER DEVICE SERVICE

The IHV device will implement and report the following device service when queried by CID_MBIM_DEVICE_SERVICES. The existing well-known services are defined in the NCM MBIM spec in section 10.1. It is extended it to define the following service.

Service Name	UUID	UUID Value
Multi-carrier	UUID_MULTICARRIER	8b569648- 628d-4653-9b9f- 1025404424e1

Specifically, the following CIDs are defined for UUID_MULTICARRIER device service.

CID	Command Code	Query	Set	Event	Set InformationBuffer payload	Query InformationBuffer payload	Completion InformationBuff
CID_MBIM_MULTICARRIER_CAPABILITIES	1	Y	N	N	N/A	N/A	MBIM_MULTICARRIER_CAPABIL

CID	Command Code	Query	Set	Event	Set InformationBuffer payload	Query InformationBuffer payload	Completion InformationBuff
CID_MBIM_LOCATION_INFO	2	Y	Y	N	MBIM_LOCATION_INFO	N/A	MBIM_LOCATION_INFO
CID_MBIM_MULTICARRIER_CURRENT_CID_LIST	3	Y	N	N	N/A	UUID	MBIM_MULTICARRIERMODE_C

CID_MBIM_MULTICARRIER_CAPABILITIES

The command returns information about a MB device's multi-carrier capabilities. A device that requires a firmware reboot, and correspondingly a device removal/arrival should provide the host with a hint using the appropriate flag to enable the host to provide the appropriate user experience.

Query = **InformationBuffer on MBIM_COMMAND_MSG not used. MBIM_MULTICARRIER_CAPABILITIES returned in InformationBuffer MBIM_COMMAND_DONE**

Set = **Unsupported**

Unsolicited Event = **Unsupported**

MBIM_MC_FLAGS_NONE	0h	No flags set
MBIM_MC_FLAGS_STATIC_SCAN	1h	Indicates that the results reported for visible providers in scan results aren’t obtained from a full network scan. The result may be obtained from a hardcoded list.
MBIM_MC_FLAGS_¬¬FW_REQUIRES_REBOOT	2h	Indicates that the device requires powering cycle and rebooting to switch firmware.

Offset	Size	Field	Description
0	4	dwCapabilities	Returns the capabilities from MBIM_MULTICARRIER_FLAGS

CID_MBIM_LOCATION_INFO

The command is used to set/query the current location information of the host. This is useful to the device if it needs to filter the list of static (no physical scan) visible providers to the ones relevant to the current user location.

Query = **InformationBuffer on MBIM_COMMAND_MSG not used. MBIM_LOCATION_INFO returned in InformationBuffer MBIM_COMMAND_DONE**

Set = **InformationBuffer on MBIM_COMMAND_MSG contains MBIM_LOCATION_INFO**

Unsolicited Event = **Unsupported**

The country code specified by the host will be based on the Geographical Location GEOID available on Windows. For more information, see Table of Geographical Locations (Windows).

Offset	Size	Field	Description
0	4	Country	Geographical Location based on GEOID.

CID_MBIM_MULTICARRIER_CURRENT_CID_LIST

This command is used to query the CIDs currently supposed by a device service.

Query = **InformationBuffer on MBIM_COMMAND_MSG is a UUID for the device service. MBIM_MULTICARRIERMODE_CURRENT_CID_LIST returned in InformationBuffer MBIM_COMMAND_DONE**

Set = **Unsupported**

Unsolicited Event = **Unsupported**

Offset	Size	Field	Type	Description
0	4	CidCount	UINT32	Number of CIDs supported for this device service.
4		DataBuffer	DATABUFFER	CidList: List of CIDs supported for this device service. There must be CID Count number of entries in this list. Each CID is of type UINT32.

Mobile Broadband Device Firmware Update

Article • 09/27/2024

This article provides guidance to Mobile Broadband (MB) module manufacturers intending to support firmware upgrade devices via Windows Update (WU). The devices must be compliant with the USB NCM Mobile Broadband Interface Model (MBIM) V1.0 specification released by the USB-IF Device Working Group.

The information in this article applies to:

- Windows 8/Windows 10/Windows 11

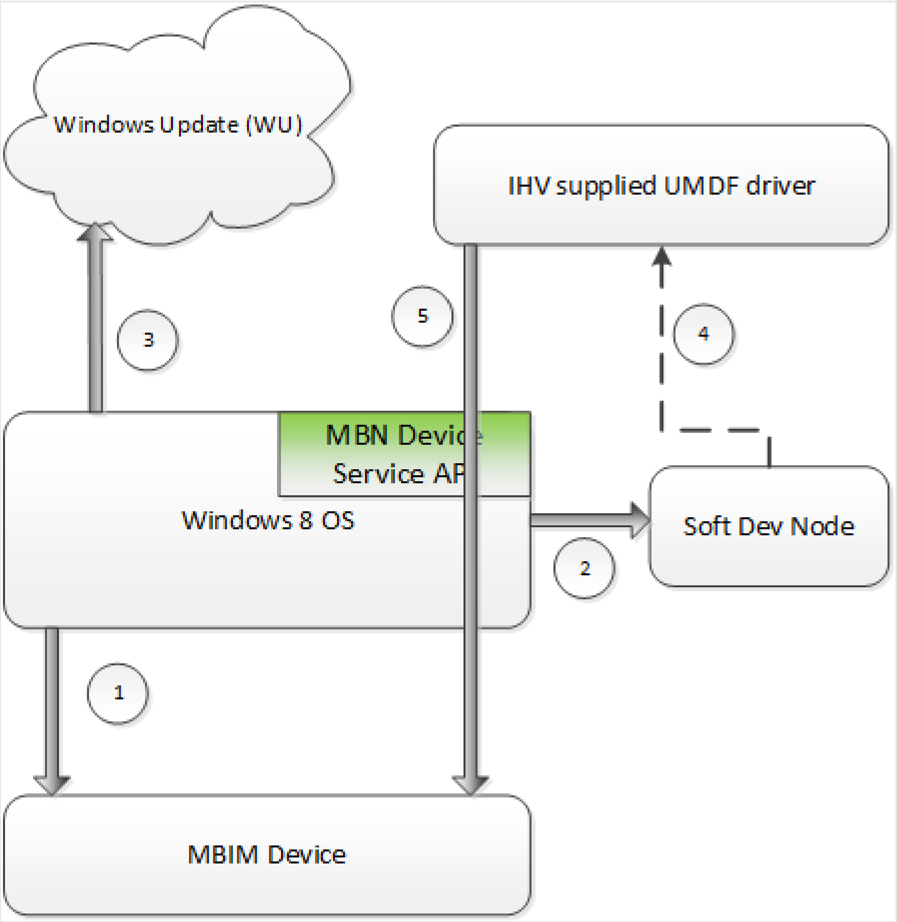
Device Requirements

To support firmware updates on Mobile Broadband using Windows Update, module or device manufacturers need to comply with the following requirements:

- UMDF (User Mode Driver Framework) based driver developed by the module or device manufacturer, packaged along with the INF file and firmware payload. Sample INF file and details are provided in the later part of this document
- Device firmware to implement the following functionalities:
  - Firmware ID Device Service (FID). For more information, see **FID Device Service**.
  - Firmware to support a firmware update device service. This is a device manufacturer specific device service that enables a UMDF driver to call into and execute/download the firmware payload and start the firmware update process.

Operational Overview

The following diagram shows the high level design and interaction between the three components involved: MBIM device, Windows 8 Operating System and IHV supplied firmware upgrade driver.



- When the WWAN Service detects the arrival of new MB device, it checks if device support Firmware ID (FID) Device Service. If it's present, it retrieves the FID, which is defined to be GUID. The Firmware Device Service specification that the IHV needs to the support on the device is described **below**.
- WWAN Service (Windows OS) generates “soft device-node” using the FID obtained above as the device hardware Id. This referred to as “Soft Dev Node” in the diagram above. The creation of the dev-node will kick start PnP subsystem (Windows OS) to find the best matched driver. In Windows 8, PnP system will first attempt to install a driver from the local store, if one is available, and in parallel OS will attempt to fetch a better matched driver from WU. The inbox NULL driver will used be used as default if better match driver isn't available to eliminate “Driver Not Found” issue.
- The IHV WU package, based on the FID match, is pulled down to the machine and installed. It's expected that the FID represents a unique firmware SKU (uniqueness here's defined by combination device VID/PID/REV and MNO). WU package would contain an IHV authored UMDF driver as well as a firmware payload.
- Once the IHV UMDF is loaded on the soft dev-node it's responsible for controlling the firmware update flow. It should be noted that the life time of the soft dev-node is tied to physical presence of the MBIM device. The UMDF driver shall perform the following steps to performing firmware updates
  - It's acceptable for the device to reboot multiple times during the firmware update process, but would cause the UMDF driver to get unloaded/reloaded

  - The entire firmware upgrade process, including reboots, should take place no more than 60 seconds.
  - After the firmware update is completed and device has reverted to MBIM mode, Windows should be notified. This is done by clearing the previously set DEVPKEY_Device_PostInstallInProgress property. The **IWDFUnifiedPropertyStore** interface describes how to set a property on dev-node. A previously set property can be cleared using DEVPROP_TYPE_EMPTY.
  - During OnPrepareHardware UMDF callback, the UMDF driver shall check if the firmware on the device needs to be updated. This is done by comparing the version of the firmware on the device against the one that came in via Windows Update. Additional guidance is provided later in the document regarding placement location of firmware binary. If firmware update is required, the UMDF driver should:
    - Schedule a work-item. The actual firmware upgrade happens in the context of the work-item.
    - Once the work-item is successfully scheduled, notify Windows about the start of firmware update. It's done by setting the DEVPKEY_Device_PostInstallInProgress property on the soft dev-node in the context of OnPrepareHardware UMDF callback.
    - It's important not to block the OnPrepareHardware callback while the firmware update is in progress. It's expected that OnPrepareHardware callback is completed within a second or two at the most.

Sample INF file for the WU Package

This section provides a sample INF that is part of the WU package. The key points to note in INF file are:

- The firmware binaries are independent of the UMDF driver.
- The firmware binaries are located in the driverstore directory, a path determined by the operating system and referenced in the INF using DIRID 13. The binaries can't be executable files containing PE/COFF headers.
- `%13%\<UniqueBinaryName>.bin`
- The INF file stores this location in the registry and the UMDF driver reads the registry value to discover the binary location.
- The following sample INF template highlights items that need to be filled by the IHV.

C++

```
[Version]
Signature       = "$WINDOWS NT$"
Class           = Firmware
```

```
ClassGuid       = {f2e7dd72-6468-4e36-b6f1-6488f42c1b52}
Provider        = %Provider%
DriverVer       = 06/21/2006,6.2.8303.0
CatalogFile     = MBFWDriver.cat
PnpLockdown     = 1

[Manufacturer]
%Mfg%           = Firmware,NTx86

[Firmware.NTx86]
%DeviceDesc%    = Firmware_Install,MBFW\{FirmwareID}    ; From Device
Service
;%DeviceDesc%    = Firmware_Install,MBFW\{2B13DD42-649C-3442-9E08-
D85B26D7825C}

[Firmware_Install.NT]
CopyFiles       = FirmwareDriver_CopyFiles,FirmwareImage_CopyFiles

[Firmware_Install.NT.HW]
AddReg          = Device_AddReg

[Device_AddReg]
HKR,,FirmwareBinary,,"%13%\MBIHVFirmware-XYZ-1.0.bin"

[Firmware_Install.NT.Services]
AddService      = WUDFRd,0x000001fa,WUDFRD_ServiceInstall

[WUDFRD_ServiceInstall]
DisplayName     = %WudfRdDisplayName%
ServiceType     = 1
StartType       = 3
ErrorControl    = 1
ServiceBinary   = %12%\WUDFRd.sys
LoadOrderGroup  = Base

[Firmware_Install.NT.CoInstallers]
CopyFiles       = WudfCoInstaller_CopyFiles

[WudfCoInstaller_AddReg]
HKR,,CoInstallers32,0x00010000,"WUDFCoinstaller.dll"

[Firmware_Install.NT.Wdf]
UmdfService      = MBIHVFirmwareDriver,MBIHVFirmwareDriver_Install
UmdfServiceOrder = MBIHVFirmwareDriver

[MBIHVFirmwareDriver_Install]
UmdfLibraryVersion  = 1.11
ServiceBinary       = %12%\UMDF\MBFWDriver.dll
DriverCLSID         = {<DriverClassGuid>} ; From UMDF driver

[FirmwareImage_CopyFiles]
MBIHVFirmware-XYZ-1.0.bin   ; Firmware Image

[FirmwareDriver_CopyFiles]
MBFWDriver.dll          ; UMDF driver for SoftDevNode
```

```
[DestinationDirs]
FirmwareImage_CopyFiles  = 13      ; Driver Store
FirmwareDriver_CopyFiles = 12,UMDF ;%SystemRoot%\System32\drivers\UMDF

[SourceDisksFiles]
MBIHVFirmware-XYZ-1.0.bin = 1

[SourceDisksNames]
1 = %DiskName%

; ================== Generic ==================================

[Strings]
Provider        = "MBIHV"
Mfg             = "MBIHV"
DeviceDesc      = "MBIHV Mobile Broadband Firmware Device"
DiskName        = "Firmware Driver Installation Media"
```

Firmware Identification Device Service (FID Device Service)

The MBIM compliant device will implement and report the following device service when queried by CID_MBIM_DEVICE_SERVICES. The existing well-known services are defined in the NCM MBIM spec in section 10.1. Microsoft Corporation extends this to define the following service.

Service Name = **Microsoft Firmware ID**

UUID = **UUID_MSFWID UUID**

Value = **e9f7dea2-feaf-4009-93ce-90a3694103b6**

Specifically, the following CID is defined for UUID_MSFWID device service:

CID = **CID_MBIM_MSFWID_FIRMWAREID**

Command Code = **1**

Query = **Yes**

Set = **No**

Event = **No**

Set InformationBuffer payload = **N/A**

Query InformationBuffer payload = **N/A**

Completion InformationBuffer payload = **UUID**

CID_MBIM_MSFWID_FIRMWAREID

The command returns the MNO or IHV assigned Firmware ID for the device. The UUID is encoded based on the guidelines in the MBIM specification.

Query = **InformationBuffer on MBIM_COMMAND_MSG not used. UUID returned in InformationBuffer MBIM_COMMAND_DONE.**

Set = **Unsupported**

Unsolicited Event = **Unsupported**

Code snippets for behavior of UMDF driver

As indicated earlier, the UMDF driver should indicate to the Windows when it starts and completes firmware upgrade. This section provides code snippets that show how the driver should notify Windows of these events.

C++

```
/**
 * This is the IPnpCallbackHardware*:OnPrepareHardware handler
 * in the UMDF driver. This is called every time the firmware
 * update is device is started. Since this handler should be
 * blocked from returning actual the firmware update process
 * should be done in a workitem
 */
HRESULT
CMyDevice::OnPrepareHardware(IWDFDevice* pDevice)
{
    HRESULT hr = S_OK;
    BOOL bFirmwareUpdateInProgress = FALSE;
    BOOL bFirmwareUpdateNeeded = FALSE;
    BOOL bFirmwareUpdateIsDone = FALSE;

    //
    // The snippets below demonstrates the steps for firmware
    // update against a MB device that loads the updated firmware
    // on device boot. So the firmware update driver needs to
    // send the new firmware down to the device and then tell
    // the device to initiate a stop/start. Once the device has
    // reappeared, it would have automatically loaded the
    // new firmware
    //


    //
    // First, determine if firmware update is in progress. This
    // can be based on some registry key that is saved when
    // firmware update is started
    //
```

```
        // Assuming this status is returned in bFirmwareUpdateInProgress
        if (bFirmwareUpdateInProgress)
        {
            //
            // If firmware update is in progress, check if its done. For
            // this it may be necessary to access the MB device. Note that
            // if the MB device (& hence the Firmware update device) needs
            // multiple stop/starts to do the firmware update. In that case
            // it will be marked as done at the end of the process
            //

            // Assuming this status is returned in bFirmwareUpdateIsDone
            if (bFirmwareUpdateIsDone)
            {
                //
                // Signal the completion of the firmware update
                // process.
                //
                SignalFirmwareUpdateComplete(pDevice);
            }
            else
            {
                //
                // Take appropriate steps to get notified when
                // firmware update is done. Call SignalFirmwareUpdateComplete
                // when that notification is received
                //
            }
        }
        else
        {
            //
            // Determine if firmware update is needed. This can be
            // based on checking state in the registry of the last
            // firmware version set on the device to the firmware
            // version associated with this driver
            //

            // Assuming this status is returned in bFirmwareUpdateNeeded
            if (bFirmwareUpdateNeeded)
            {
                //
                // Create and queue a workitem to perform the firmware
                // update process. IWDFWorkItem can be used for this
                //

                // Assuming the creation/enquing status
                // is returned in hr

                if (SUCCEEDED(hr))
                {
                    //
                    // Work item queued. It will do the firmware update
                    // Tell the OS that firmware update is in progress
                    //
                    SignalFirmwareUpdateInProgress(pDevice);
                }
            }
        }

        //
        // If we have a failure, we clear the firmware update
        // in progress state
        //
        if (FAILED(hr))
        {
            SignalFirmwareUpdateComplete(pDevice);
        }
        return S_OK;
    }

    /**
     * This function tells the OS that firmware update is in progress.
     * It should be called from the firmware update UMDF driver's
     * IPnpCallbackHardware*:OnPrepareHardware handler after it has
     * successfully queued a workitem to perform the firmware update
     */
    HRESULT
    CMyDevice::SignalFirmwareUpdateInProgress(
        __in IWDFDevice* pDevice
        )
    {
        HRESULT hr = S_OK;
        IWDFUnifiedPropertyStoreFactory* spPropertyStoreFactory = NULL;
        IWDFUnifiedPropertyStore* spPropStore = NULL;
        WDF_PROPERTY_STORE_ROOT wdfPropRoot = { sizeof(WDF_PROPERTY_STORE_ROOT),
    WdfPropertyStoreRootClassHardwareKey };
        DEVPROP_BOOLEAN boolValue = DEVPROP_TRUE;

        do
        {

            hr = pDevice->QueryInterface(IID_PPV_ARGS(&spPropertyStoreFactory));
            if (FAILED(hr))
            {
                Trace(TRACE_LEVEL_ERROR, "Failed to query for property store
    factory. Error = 0x%x", hr);
                break;

            }

            hr = spPropertyStoreFactory->RetrieveUnifiedDevicePropertyStore(
                &wdfPropRoot,
                &spPropStore
                );
            if (FAILED(hr))
            {
                Trace(TRACE_LEVEL_ERROR, "Failed to query for device property
    store. Error = 0x%x", hr);
```

```
            break;
        }

        // Set the OS flag
        hr = spPropStore->SetPropertyData(
            reinterpret_cast<const DEVPROPKEY*>
(&DEVPKEY_Device_PostInstallInProgress),
            0, // this property is language neutral
            0,
            DEVPROP_TYPE_BOOLEAN,
            sizeof(DEVPROP_BOOLEAN),
            &boolValue
            );
        if (FAILED(hr))
        {
            Trace(TRACE_LEVEL_ERROR, "Failed to set device property for
PostInstallInProgress. Error = 0x%x", hr);
            break;
        }

        //
        // Save some state so that we know we are in the process
        // of firmware update
        //
    } while (FALSE);

    if (spPropStore)
    {
        spPropStore->Release();
    }

    if (spPropertyStoreFactory)
    {
        spPropertyStoreFactory->Release();
    }

    return hr;
}


/**
 * This function tells the OS that firmware update is done
 * It should be called only after the full firmware update process
 * (including any MB device stop/start) has finished
 */
HRESULT
CMyDevice::SignalFirmwareUpdateComplete(
    __in IWDFDevice* pDevice
    )
{
    HRESULT hr = S_OK;
    IWDFUnifiedPropertyStoreFactory* spPropertyStoreFactory = NULL;
    IWDFUnifiedPropertyStore* spPropStore = NULL;
    WDF_PROPERTY_STORE_ROOT wdfPropRoot = { sizeof(WDF_PROPERTY_STORE_ROOT),
WdfPropertyStoreRootClassHardwareKey };
```

```
    do
    {
        hr = pDevice->QueryInterface(IID_PPV_ARGS(&spPropertyStoreFactory));
        if (FAILED(hr))
        {
            Trace(TRACE_LEVEL_ERROR, "Failed to query for property store factory. Error = 0x%x", hr);
            break;

        }

        hr = spPropertyStoreFactory->RetrieveUnifiedDevicePropertyStore(
            &wdfPropRoot,
            &spPropStore
            );
        if (FAILED(hr))
        {
            Trace(TRACE_LEVEL_ERROR, "Failed to query for device property store. Error = 0x%x", hr);
            break;
        }

        hr = spPropStore->SetPropertyData(
            reinterpret_cast<const DEVPROPKEY*>(&DEVPKEY_Device_PostInstallInProgress),
            0, // this property is language neutral
            0,
            DEVPROP_TYPE_BOOLEAN,
            0,
            NULL
            );
        if (FAILED(hr))
        {
            Trace(TRACE_LEVEL_ERROR, "Failed to clear device property for PostInstallInProgress. Error = 0x%x", hr);
            break;
        }

        //
        // Save some state so that we can do quick check on
        // whether firmware update is needed or not
        //

    } while (FALSE);

    if (spPropStore)
    {
        spPropStore->Release();
    }

    if (spPropertyStoreFactory)
    {
        spPropertyStoreFactory->Release();
    }
    return hr;
}
```

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

Mobile Broadband Implementation Guidelines for USB Devices

Article • 12/15/2021

This topic provides specific implementation guidance to help mobile broadband device manufacturers produce compliant USB devices for Windows. It should be used in conjunction with the USB NCM Mobile Broadband Interface Model (MBIM) V1.0 specification released by the USB-IF Device Working Group.

The information in this topic applies to:

- Windows 8
- Windows 8.1

Delaying MBIM Open

MBIM devices may require time to complete initialization when they receive MBIM OPEN message from the host. The device should wait for its initialization to complete before responding to the MBIM OPEN message. The device should not respond to the message with error status like MBIM_STATUS_BUSY and expect the host to poll the device with MBIM OPEN messages. Responding to MBIM OPEN with a status other than MBIM_STATUS_SUCCESS terminates the initialization process on the host.

Multi-carrier\Multi-subscription

Please refer to **IHV Guidance for Implementing Multimode- and Multicarrier- Capable MB Devices** for details.

MBIM_CID_HOME_PROVIDER

MBIM devices must not fail a SET MBIM_CID_HOME_PROVIDER request with the following listed status codes under the mentioned circumstances. The following statuses are valid for the QUERY MBIM_CID_HOME_PROVIDER request but are not applicable for a SET request.

- **MBIM_STATUS_SIM_NOT_INSERTED** - MBIM devices must not fail a SET MBIM_CID_HOME_PROVIDER request with status as MBIM_STATUS_SIM_NOT_INSERTED if the SIM for the new home provider is present but the SIM for the old home provider is not inserted.

- **MBIM_STATUS_BAD_SIM** - MBIM devices must not fail SET MBIM_CID_HOME_PROVIDER request with MBIM_STATUS_BAD_SIM if the SIM for the new home provider is good but the SIM for the old home provider is not bad.
- **MBIM_STATUS_PIN_REQUIRED** - MBIM devices must not fail SET MBIM_CID_HOME_PROVIDER request with MBIM_STATUS_PIN_REQUIRED regardless of whether the old or new SIM is pin locked.

MBIM_CID_VISIBLE_PROVIDERS

- **MBIM_STATUS_SIM_NOT_INSERTED** - When MBIM_VISIBLE_PROVIDERS_ACTION is set to MBIMVisibleProvidersActionRestrictedScan the MBIM device must not fail MBIM_CID_VISIBLE_PROVIDERS request with MBIM_STATUS_SIM_NOT_INSERTED because the SIM for the current home provider is not present.
- **MBIM_STATUS_PIN_REQUIRED** - When MBIM_VISIBLE_PROVIDERS_ACTION is set to MBIMVisibleProvidersActionRestrictedScan the MBIM device must not fail MBIM_CID_VISIBLE_PROVIDERS request with MBIM_STATUS_PIN_REQUIRED because the SIM for the current home provider is PIN locked.

Responding to Pin Operations

MBIM devices must follow these guidelines when responding to **MBIMPinOperationEnter** requests:

- For successful **MBIMPinOperationEnter** requests, when the device no longer requires a PIN, the device must set status to MBIM_STATUS_SUCCESS and **MBIM_PIN_INFO::Pin Type** to **MBIMPinTypeNone.**

- The device must set status to MBIM_STATUS_SUCCESS for PIN-enable and PIN-disable operations when the PIN is already in the requested state. The device must set **MBIM_PIN_INFO::PinType** to **MBIMPinTypeNone**. Other members are ignored.

- When a PIN mode is changed from Disabled to Enabled, the PIN state should be **MBIMPinStateUnlocked**.

- If PIN1 is enabled, the PIN state becomes **MBIMPinStateLocked** when the device is power cycled.

- For all other PINs, the PIN state can change from **MBIMPinStateUnlocked** to **MBIMPinStateLocked** depending on mobile broadband device specific conditions.

- PIN Not Supported: If a PIN operation is not supported by the device, the device must set status to MBIM_STATUS_NO_DEVICE_SUPPORT. For example, enabling and disabling PIN2 is not typically supported by mobile broadband devices so the above error code must be returned. All other members are ignored.

- PIN Must be Entered: If a PIN operation requires a PIN to be entered, the device must set status to MBIM_STATUS_PIN_REQUIRED and **MBIM_PIN_INFO::PinType** to **MBIMPinTypeXxx**. Other members are ignored.

- PIN Change Operation: If the device restricts the change of PIN value only when it is in enabled state, a request to change in disabled state must be returned with MBIM_STATUS_PIN_DISABLED.

- PIN Retrial: On failure, the device must set status to MBIM_STATUS_FAILURE, and **MBIM_PIN_INFO::PinType** to the same value as specified in **MBIM_SET_PIN**. Other members are ignored except for **MBIM_PIN_INFO::RemainingAttempts**. This may occur when an incorrect PIN is entered.

- PIN Blocking: The PIN is blocked when the number of **MBIM_PIN_INFO::RemainingAttempts** is zero. If the PIN unblock operation is not available, the device must set status to MBIM_STATUS_FAILURE and **MBIM_PIN_INFO::PinType** to **MBIMPinTypeNone**. **MBIM_PIN_INFO::RemainingAttempts** should be set to 0 and all the other members are ignored. **Note** If the device supports PIN unblock operations, the device should follow the PIN Unblocking step to respond to the request.

- Unblocking PIN: The PIN is blocked when **MBIM_PIN_INFO::RemainingAttempts** is zero. To unblock the PIN, the device may request a corresponding PUK, if applicable. In this case, the device must set status to MBIM_STATUS_FAILURE, **MBIM_PIN_INFO::PinType** to the corresponding **MBIMPinTypeXxxPUK**, **PinState** to **MBIMPinStateLocked**, and **MBIM_PIN_INFO::RemainingAttempts** should have the number of attempts allowed to enter a valid PUK.

- If PIN blocking results in the device or SIM becomes blocked, the device must send a MBIM_CID_SUBSCRIBER_READY_STATUS notification with **ReadyState** set to **MBIMSubscriberReadyStateDeviceLocked.**

- If there is an active PDP context at the time of PIN1 blocking, the device must deactivate the PDP context and send notifications to the operating system about the PDP deactivation and link state change.

- For successful requests, the device must set status to MBIM_STATUS_SUCCESS. Other members are ignored.

- For failed **MBIMPinOperationEnter** requests, the device must set status to MBIM_STATUS_FAILURE and include applicable data as per the following details:

- PIN Disabled or PIN Not Expected: For MBIMPinOperationEnter set requests, when the corresponding PIN is either disabled or currently not expected by the device, the device must set **MBIM_PIN_INFO::PinType** to **MBIMPinTypeNone**. All other members are ignored.

- PIN Not Supported: If the given PIN is not supported by the device, the device must set status to MBIM_STATUS_NO_DEVICE_SUPPORT.

- PIN Retrial: In this mode, the device requires the PIN to be re-entered as the **MBIM_PIN_INFO::RemainingAttempts** value is still non-zero for this particular type of PIN. The device must set **MBIM_PIN_INFO::PinType** to the same value as that of **MBIM_PIN_INFO::PinType** in **MBIM_SET_PIN**.

- PIN Blocking: The PIN is blocked when **MBIM_PIN_INFO::RemainingAttempts** is zero. If the PIN unblock operation is not available, the device must set status to MBIM_STATUS_FAILURE and **MBIM_PIN_INFO::PinType** to **MBIMPinTypeNone**. All the other members are ignored. **Note** If the device supports PIN unblock operations, the device should follow the PIN Unblocking step to respond to the request.

- PIN Unblocking: The PIN is blocked when **MBIM_PIN_INFO::RemainingAttempts** is zero. To unblock the PIN, the device may request a corresponding PIN Unlock Key (PUK), if applicable. In this case, the device must set **MBIM_PIN_INFO::PinType** to the corresponding **MBIMPinTypeXxxPUK** with the relevant details.

- Blocked PUK: If the number of failed trials exceeds the preset value for entering the **MBIMPinTypeXxxPUK**, then the PUK becomes blocked. The device must signal this by setting status to MBIM_STATUS_FAILURE and **MBIM_PIN_INFO::PinType** to **MBIMPinTypeNone**. In case PUK1 is blocked, the device must send a MBIM_CID_SUBSCRIBER_READY_STATUS with **ReadyState** set to **MBIMSubscriberReadyStateBadSim**.

- MBIM devices must follow these guidelines when responding to **MBIMPinOperationEnable**, **MBIMPinOperationDisable**, or **MBIMPinOperationChange** requests.

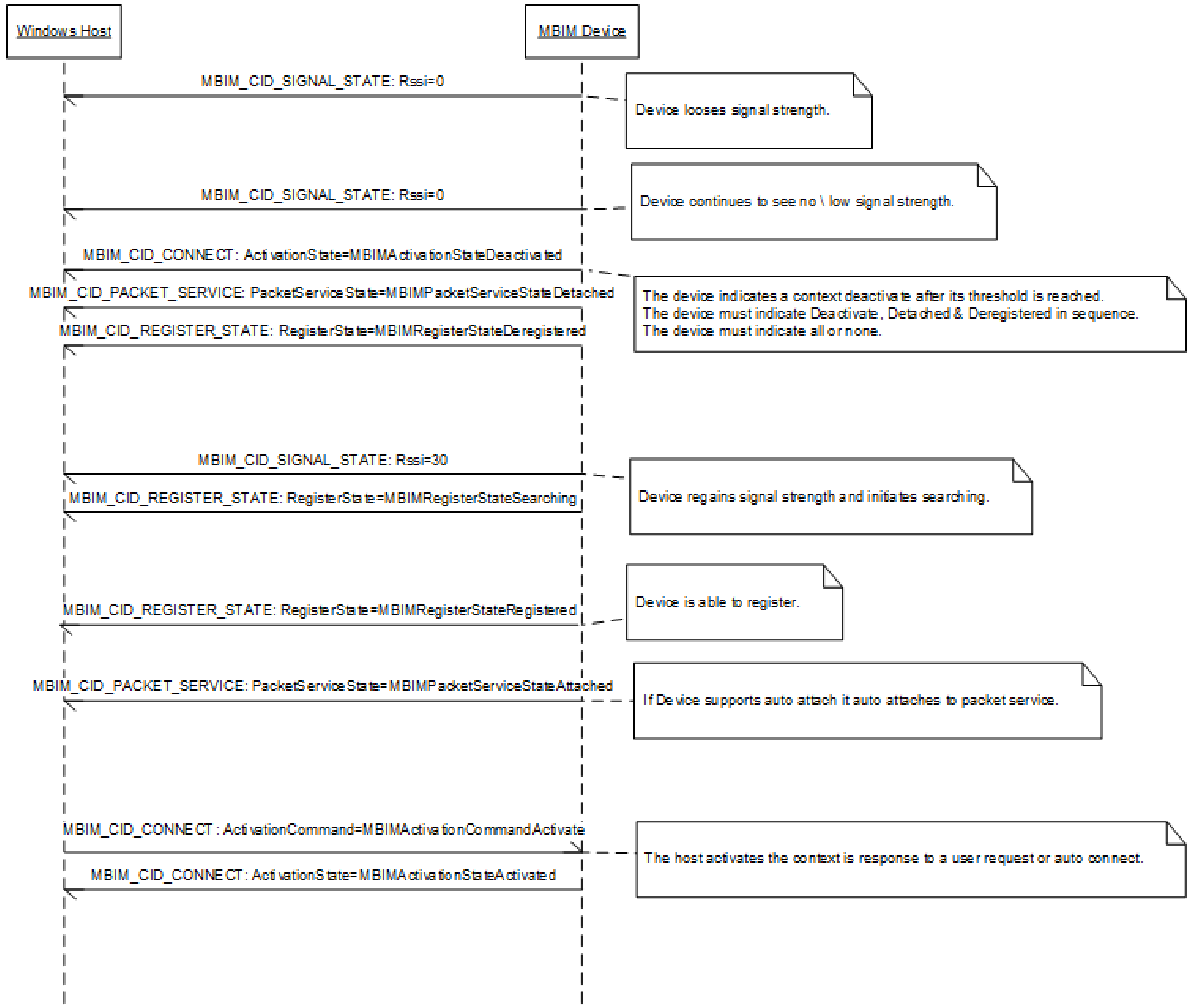
Auto Packet Service Attach

MBIM devices that support Auto Packet Service Attach manage the attachment and detachment of packet service from the mobile network at their discretion. The host may still send an attach request to such a device on user request. When the device receives the attach request from the host it should handle as follows:

- If the device is not attached and not in the middle of an attach operation and is capable of attaching then it should initiate a new attach procedure with the mobile network.
- If the device is not attached but in the middle of an auto attach operation then it should wait for the auto attach operation to complete and complete the attach request from the host with the status of the auto attach operation.
- If the device is already attached then it should complete the attach request from the host successfully.

Signal Strength Loss and Data Connection Loss

When a device loses signal strength the device must indicate **MBIMActivationStateDeactivated** followed by **MBIMPacketServiceStateDetached** followed by **MBIMRegisterStateDeregistered** in that order. If the device loses packet service while it is context activated the device must indicate **MBIMActivationStateDeactivated** followed by **MBIMPacketServiceStateDetached** in that order. The following sequence diagram shows the interaction between the host and the device.


Windows Host
MBIM Device
MBIM_CID_SIGNAL_STATE: Rssi=0
Device looses signal strength.
MBIM_CID_SIGNAL_STATE: Rssi=0
Device continues to see no \ low signal strength.
MBIM_CID_CONNECT: ActivationState=MBIMActivationStateDeactivated
MBIM_CID_PACKET_SERVICE: PacketServiceState=MBIMPacketServiceStateDetached
MBIM_CID_REGISTER_STATE: RegisterState=MBIMRegisterStateDeregistered
The device indicates a context deactivate after its threshold is reached.
The device must indicate Deactivate, Detached & Deregistered in sequence.
The device must indicate all or none.
MBIM_CID_SIGNAL_STATE: Rssi=30
MBIM_CID_REGISTER_STATE: RegisterState=MBIMRegisterStateSearching
Device regains signal strength and initiates searching.
Device is able to register.
MBIM_CID_REGISTER_STATE: RegisterState=MBIMRegisterStateRegistered
MBIM_CID_PACKET_SERVICE: PacketServiceState=MBIMPacketServiceStateAttached
If Device supports auto attach it auto attaches to packet service.
MBIM_CID_CONNECT: ActivationCommand=MBIMActivationCommandActivate
The host activates the context is response to a user request or auto connect.
MBIM_CID_CONNECT: ActivationState=MBIMActivationStateActivated

DNS Server Information

When Basic IP information (as defined in MBIM section 10.5.20.1) is provided via MBIM_CID_IP_CONFIGURATION, DNS server information (as defined in MBIM section 10.5.20.1) can also be provided via MBIM_CID_IP_CONFIGURATION. When DNS server information is updated, MBIM_CID_IP_CONFIGURAITON must have the complete Basic IP information obtained before. DNS server information can be provided solely via MBIM_CID_IP_CONFIGURATION even if the Basic IP information is not provided via MBIM_CID_IP_CONFIGURATION. This applies to both IPv4 and IPv6.

IPv6

For basic IP information (as defined in MBIM section 10.5.20.1), the expected IP Layer configuration mechanism is from router advertisement (RA). For DNS server information (as defined in MBIM section 10.5.20.1), the expected IP Layer configuration mechanism is DHCPv6.

- **Basic IP information from RA** - If a mobile network provides Basic IP information (as defined in MBIM section 10.5.20.1) via RA, then MBIM devices must allow RA packets to be forwarded to the host and must not intercept the RA packets or provide the Basic IP information present in the RA packets via MBIM_CID_IP_CONFIGURATION.
- **DNS server information from RA** - The only IP Layer configuration mechanism for DNS server information (as defined in MBIM section 10.5.20.1) supported by Windows is DHCPv6. MBIM devices must configure DNS server information, even if present in RA, via MBIM_CID_IP_CONFIGURATION.
- **Basic IP information and DNS server information from DHCPv6** - If a mobile network provides basic IP information and DNS server information (as defined in MBIM section 10.5.20.1) from DHCPv6, then MBIM devices must allow DHCPv6 packets to be forwarded to the host and must not intercept the DHCPv6 packets or provide the basic IP information and DNS server information present in the DHCPv6 packets via MBIM_CID_IP_CONFIGURATION.

MBIM_CID_RADIO_STATE

MBIM devices must not fail MBIM_CID_RADIO_STATE operations with status of MBIM_STATUS_SIM_NOT_INSERTED when SIM is not present. Radio operations must not be failed due to SIM absence.

Byte-Ordering Requirements for Authentication CIDs

Data in the byte array fields listed below must be in host-byte order.

MBIM_CID_SIM_AUTH

MBIM_SIM_AUTH_REQ

1. Rand1
2. Rand2
3. Rand3

MBIM_CID_AKA_AUTH

MBIM_AKA_AUTH_REQ

1. Rand
2. Autn

MBIM_AKA_AUTH_INFO

1. Res
2. IK
3. CK
4. Auts

MBIM_CID_AKAP_AUTH

MBIM_AKAP_AUTH_REQ

1. Rand
2. Autn

MBIM_AKAP_AUTH_INFO

1. Res
2. IK
3. CK
4. Auts

Setting Link MTU

Windows supports configuring Link Maximum Transmission Unit (MTU) only during device initialization. Windows does not update the Link MTU based on the MTU reported using MBIM_CID_IP_CONFIGURATION. Devices must communicate the network supported link MTU using the MBIM_FUNCTIONAL_DESCRIPTOR.wMaxSegmentSize. Link MTU values reported in this manner should be at least 1280 and at most 1500.

Mobile Broadband Collecting Logs

Article • 03/14/2023

Follow these steps to collect the logs related to mobile broadband on a Windows Desktop Device:

```
*  Open an Administrator Command Prompt window
*  Run the below command to start tracing
    *  netsh trace start wireless_dbg,provisioning overwrite=yes maxSize=999
level=5
*  <Repro the scenario for which you need to collect logs>
*  Run the below command to stop tracing
    *  netsh trace stop
```

The steps above generate two files:

1. NetTrace.etl - Contains the traces for the run
2. NetTrace.cab - Contains additional details about the system that will be useful for debugging

Collecting logs across a reboot

If the repro scenario includes a reboot update the start tracing command as follows:

```
*  netsh trace start wireless_dbg,provisioning overwrite=yes persist=yes
level=5
```

Please note that PII will be captured in the logs collected using the above method.

Decoding Logs

Run one of these commands to convert the .etl file to a .txt file that can be used for analysis:

`*` tracefmt`<ETL file location>`

`or`

`* netsh trace convert <ETL file location>`

Mobile Broadband Analyzing Logs

Article • 03/14/2023

TextAnalysisTool is an extensive text filtering tool that is useful for complex traces with numerous ETW providers. You can filter logs of interest using the .tat files.

To collect the logs, follow the steps in MobileBroadband Collecting logs.

Use the .tat filters included in the specific feature to filter the logs:

```
*   Copy and paste the lines into a <name>.tat file
*   Open the log of interest in TextAnalysisTool
*   Load the filter file(<name>.tat) into TextAnalysisTool by clicking File > Load Filters
```

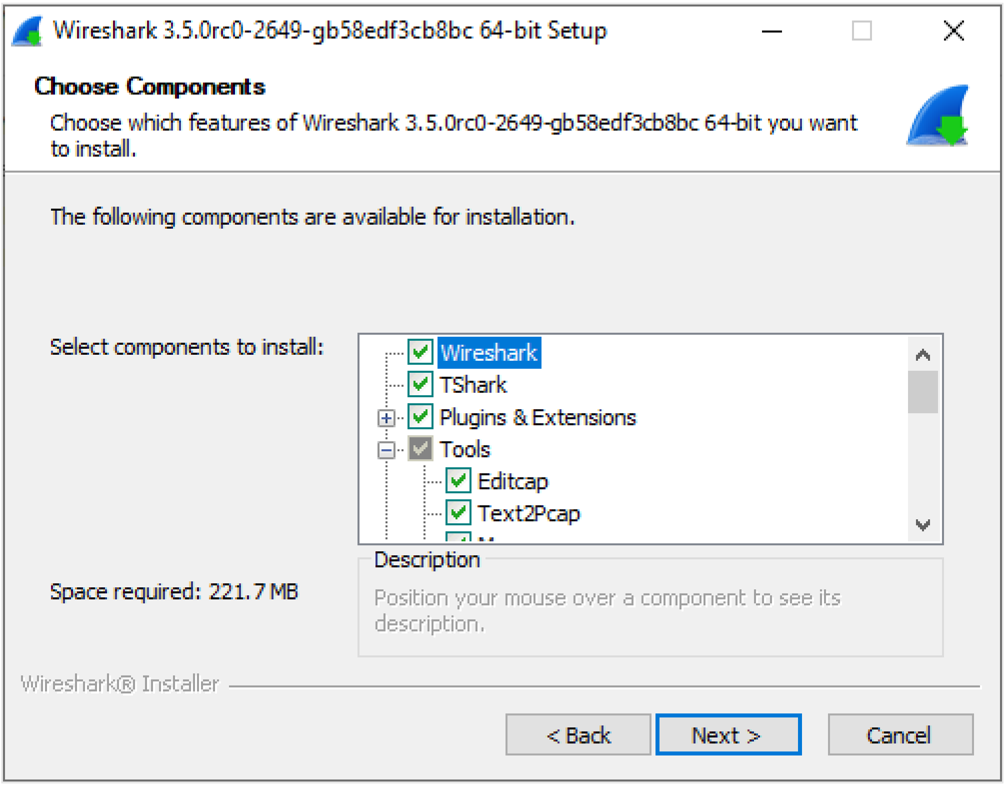
Analyzing Mobile Broadband Logs in Wireshark

Article • 10/06/2023

Follow these steps to diagnose the logs related to mobile broadband using Wireshark:

1. Download the ETW (Event Tracing for Windows) reader. Wireshark packages the ETW reader starting from version 3.5.

2. After you start the Wireshark installer, one of the steps is **Choose Components**.



Wireshark 3.5.0rc0-2649-gb58edf3cb8bc 64-bit Setup

Choose Components

Choose which features of Wireshark 3.5.0rc0-2649-gb58edf3cb8bc 64-bit you want to install.

The following components are available for installation.

Select components to install:

- ☑ Wireshark
- ☑ TShark
- ⊞ ☑ Plugins & Extensions
- ⊟ ☑ Tools
  - ☑ Editcap
  - ☑ Text2Pcap

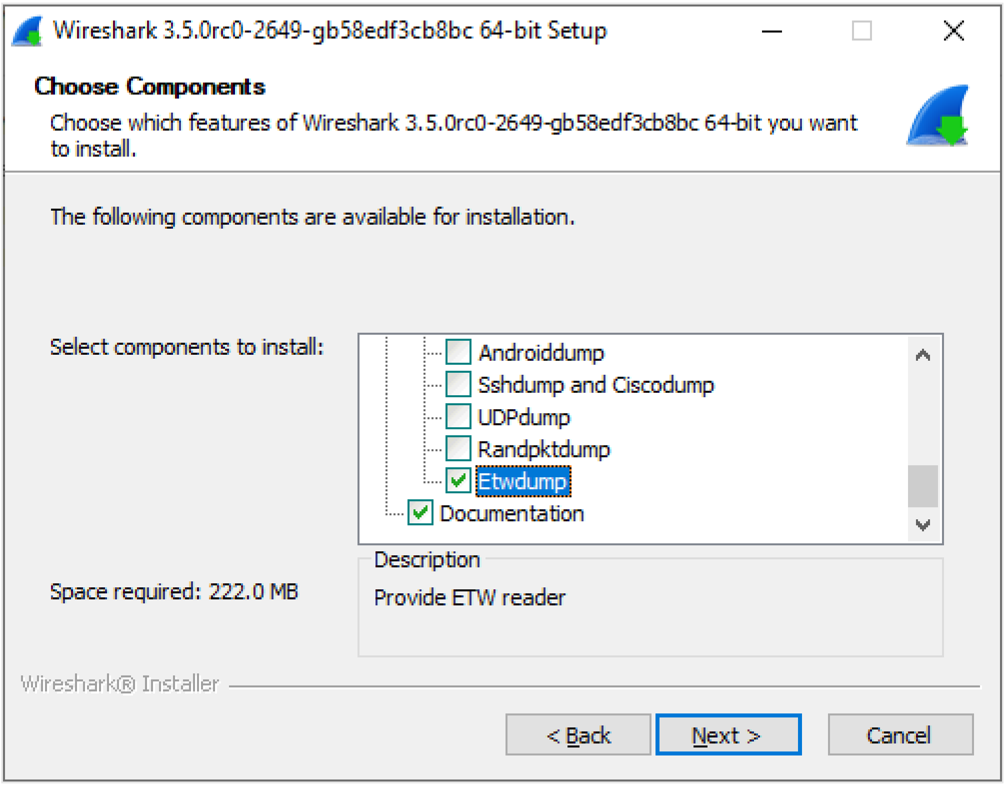
Space required: 221.7 MB

Description

Position your mouse over a component to see its description.

Wireshark® Installer

Expand Tools, scroll down, and select **Etwdump**.



Choose Components

Choose which features of Wireshark 3.5.0rc0-2649-gb58edf3cb8bc 64-bit you want to install.

The following components are available for installation.

Select components to install:

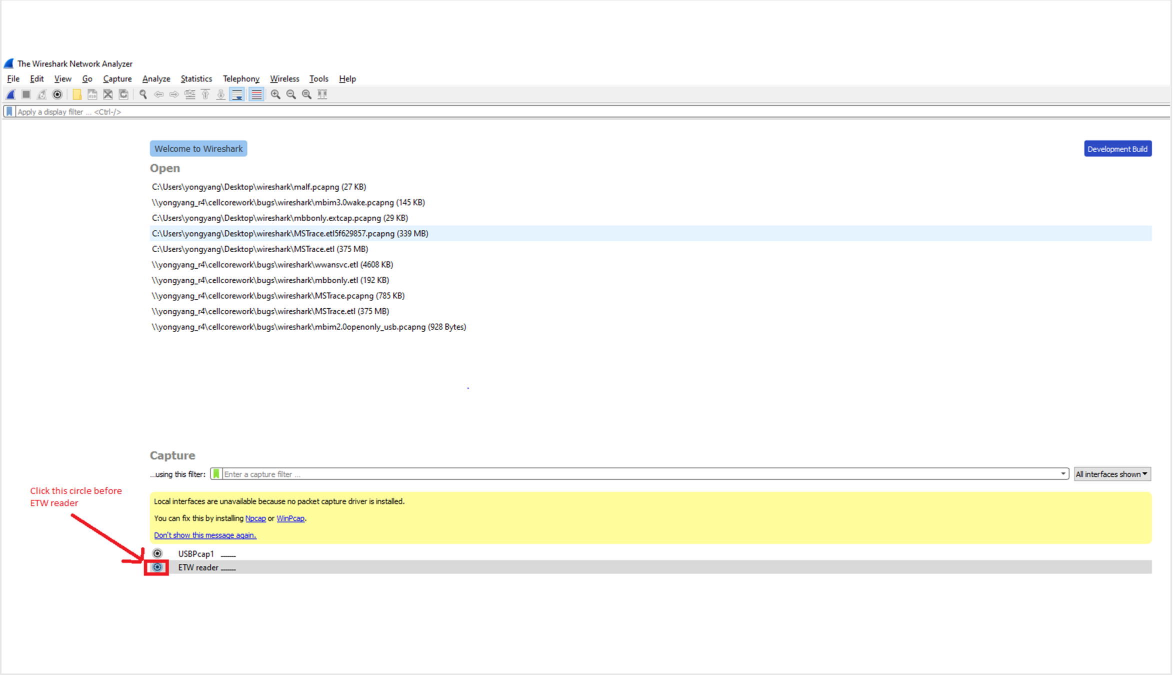
- □ Androiddump
- □ Sshdump and Ciscodump
- □ UDPdump
- □ Randpktdump
- ☑ Etwdump
- ☑ Documentation

Space required: 222.0 MB

Description

Provide ETW reader

3. Launch the ETW reader.



The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Welcome to Wireshark
Development Build

Open

C:\Users\yongyang\Desktop\wireshark\malf.pcapng (27 KB)
\\yongyang_r4\cellcorework\bugs\wireshark\mbim3.0wake.pcapng (145 KB)
C:\Users\yongyang\Desktop\wireshark\mbbonly.extcap.pcapng (29 KB)
C:\Users\yongyang\Desktop\wireshark\MSTrace.etl5f629857.pcapng (339 MB)
C:\Users\yongyang\Desktop\wireshark\MSTrace.etl (375 MB)
\\yongyang_r4\cellcorework\bugs\wireshark\wwansvc.etl (4608 KB)
\\yongyang_r4\cellcorework\bugs\wireshark\mbbonly.etl (192 KB)
\\yongyang_r4\cellcorework\bugs\wireshark\MSTrace.pcapng (785 KB)
\\yongyang_r4\cellcorework\bugs\wireshark\MSTrace.etl (375 MB)
\\yongyang_r4\cellcorework\bugs\wireshark\mbim2.0openonly_usb.pcapng (928 Bytes)

Capture

...using this filter: Enter a capture filter ...

All interfaces shown

Local interfaces are unavailable because no packet capture driver is installed.

You can fix this by installing Npcap or WinPcap.

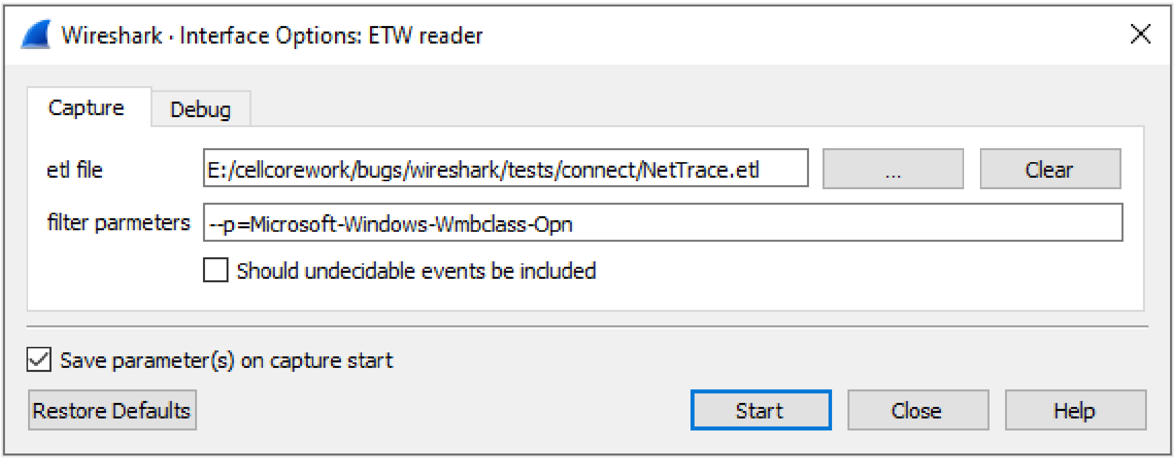
Don't show this message again.

USBPcap1

ETW reader

Click this circle before ETW reader

4. Option A. Click the "..." button to choose an ETL file to decode. You can set filter parameters to only decode events from specific providers. Then click the Start button to decode the file.



Wireshark · Interface Options: ETW reader

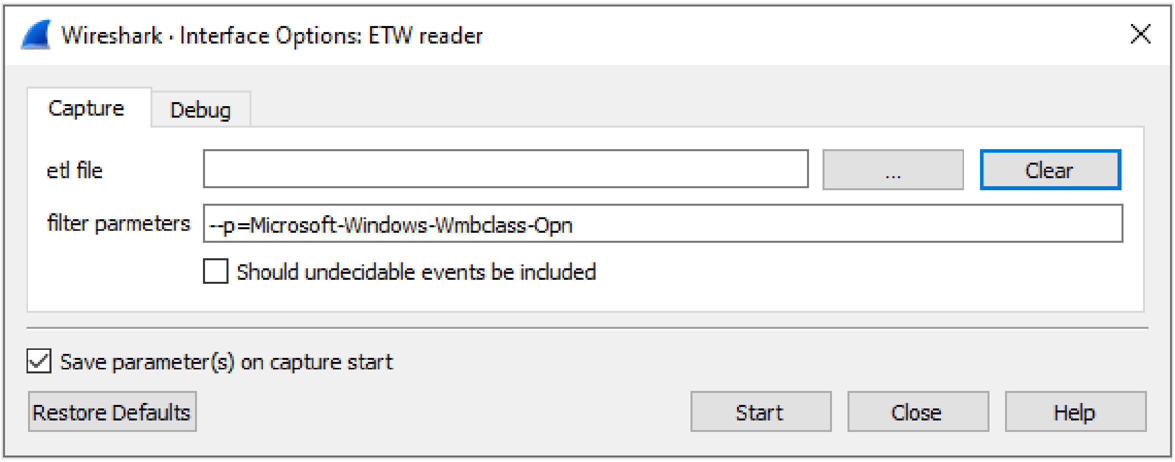
etl file: E:/cellcorework/bugs/wireshark/tests/connect/NetTrace.etl

filter parmeters: --p=Microsoft-Windows-Wmbclass-Opn

□ Should undecidable events be included

☑ Save parameter(s) on capture start

Option B. Start a live session instead of decoding the events from a file. Live sessions require an empty ETL file and you must specify filter parameters. Then click the Start button.



Wireshark · Interface Options: ETW reader

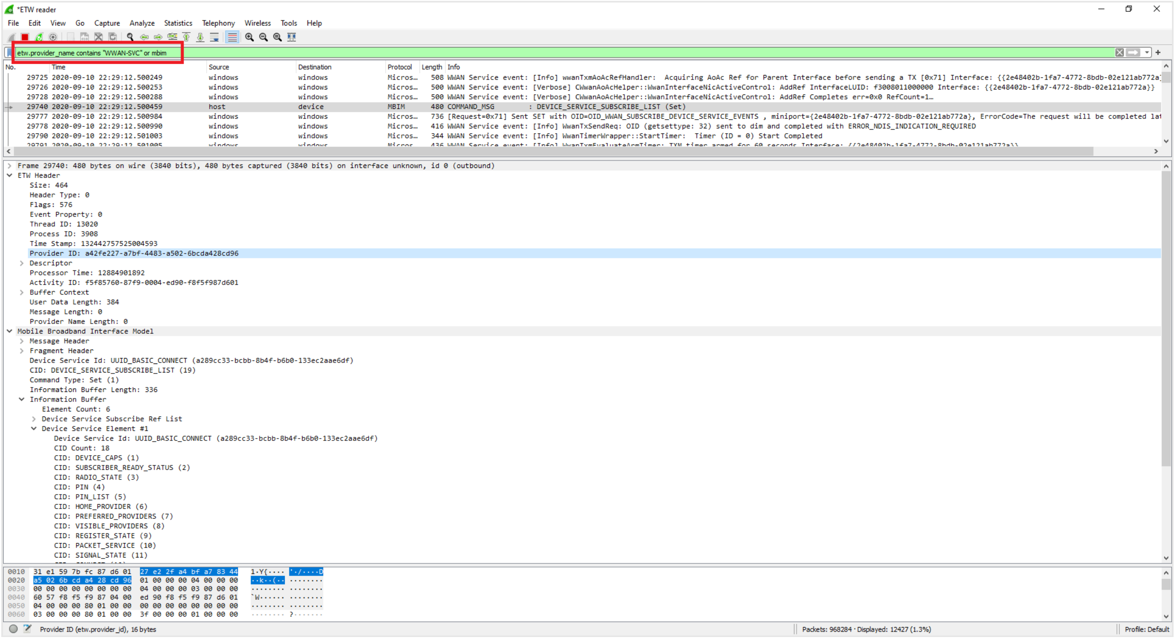
etl file:

filter parmeters: --p=Microsoft-Windows-Wmbclass-Opn

□ Should undecidable events be included

☑ Save parameter(s) on capture start

5. Wireshark will display the decoded ETW messages and MBIM messages from either a file or a live session. You may choose to filter relevant messages. The example below filters out the WWAN-SVC and MBIM messages.



*ETW reader
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

etw.provider_name contains "WWAN-SVC" or mbim

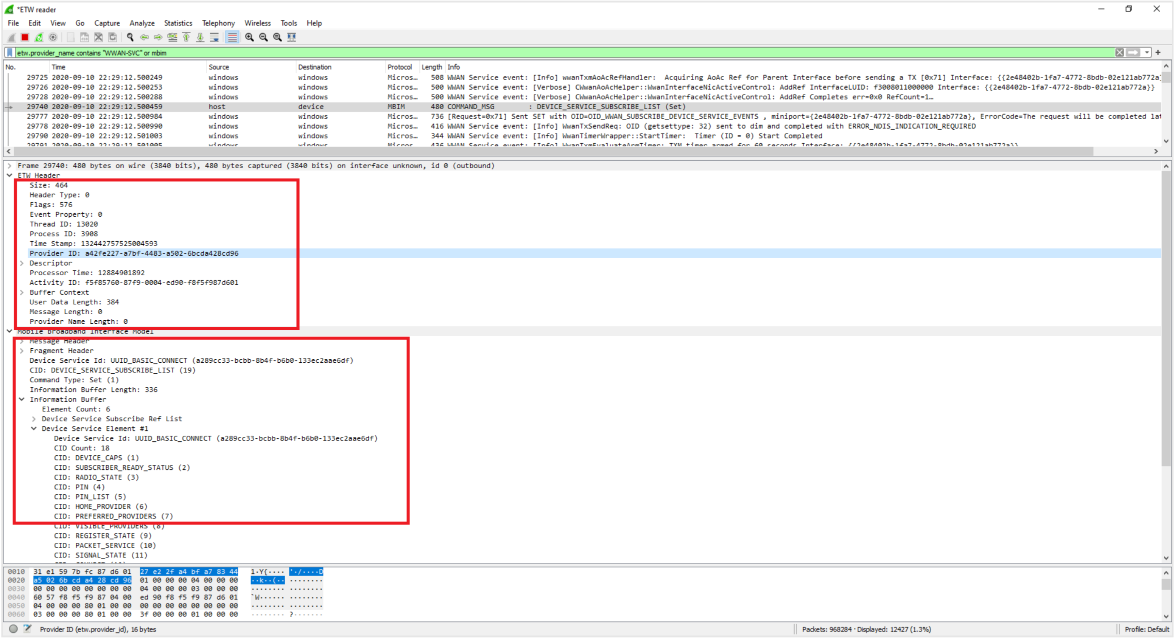
```
No.    Time                        Source    Destination  Protocol  Length Info
29725 2020-09-10 22:29:12.500249  windows   windows      Micros…   508 WWAN Service event: [Info] wwanTxmAoAcRefHandler:  Acquiring AoAc Ref for Parent Interface before sending a TX [0x71] Interface: {{2e48402b-1fa7-4772-8bdb-02e121ab772a
29726 2020-09-10 22:29:12.500253  windows   windows      Micros…   500 WWAN Service event: [Verbose] CWwanAoAcHelper::WwanInterfaceNicActiveControl: AddRef InterfaceLUID: f3008011000000 Interface: {{2e48402b-1fa7-4772-8bdb-02e121ab772a}}
29728 2020-09-10 22:29:12.500288  windows   windows      Micros…   500 WWAN Service event: [Verbose] CWwanAoAcHelper::WwanInterfaceNicActiveControl: AddRef Completes err=0x0 RefCount=1…
29740 2020-09-10 22:29:12.500459  host      device       MBIM      480 COMMAND_MSG        : DEVICE_SERVICE_SUBSCRIBE_LIST (Set)
29777 2020-09-10 22:29:12.500984  windows   windows      Micros…   736 [Request=0x71] Sent SET with OID=OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS , miniport={2e48402b-1fa7-4772-8bdb-02e121ab772a}, ErrorCode=The request will be completed lat
29778 2020-09-10 22:29:12.500990  windows   windows      Micros…   416 WWAN Service event: [Info] WwanTxSendReq: OID (getsettype: 32) sent to dim and completed with ERROR_NDIS_INDICATION_REQUIRED
29790 2020-09-10 22:29:12.501003  windows   windows      Micros…   344 WWAN Service event: [Info] WwanTimerWrapper::StartTimer:  Timer (ID = 0) Start Completed
```

```
> Frame 29740: 480 bytes on wire (3840 bits), 480 bytes captured (3840 bits) on interface unknown, id 0 (outbound)
v ETW Header
    Size: 464
    Header Type: 0
    Flags: 576
    Event Property: 0
    Thread ID: 13020
    Process ID: 3908
    Time Stamp: 132442757525004593
    Provider ID: a42fe227-a7bf-4483-a502-6bcda428cd96
  > Descriptor
    Processor Time: 12884901892
    Activity ID: f5f85760-87f9-0004-ed90-f8f5f987d601
  > Buffer Context
    User Data Length: 384
    Message Length: 0
    Provider Name Length: 0
v Mobile Broadband Interface Model
  > Message Header
  > Fragment Header
    Device Service Id: UUID_BASIC_CONNECT (a289cc33-bcbb-8b4f-b6b0-133ec2aae6df)
    CID: DEVICE_SERVICE_SUBSCRIBE_LIST (19)
    Command Type: Set (1)
    Information Buffer Length: 336
  v Information Buffer
      Element Count: 6
    > Device Service Subscribe Ref List
    v Device Service Element #1
        Device Service Id: UUID_BASIC_CONNECT (a289cc33-bcbb-8b4f-b6b0-133ec2aae6df)
        CID Count: 18
        CID: DEVICE_CAPS (1)
        CID: SUBSCRIBER_READY_STATUS (2)
        CID: RADIO_STATE (3)
        CID: PIN (4)
        CID: PIN_LIST (5)
        CID: HOME_PROVIDER (6)
        CID: PREFERRED_PROVIDERS (7)
        CID: VISIBLE_PROVIDERS (8)
        CID: REGISTER_STATE (9)
        CID: PACKET_SERVICE (10)
        CID: SIGNAL_STATE (11)
```

```
0010  31 e1 59 7b fc 87 d6 01  27 e2 2f a4 bf a7 83 44   1·Y{···· '·/····D
0020  a5 02 6b cd a4 28 cd 96  01 00 00 00 04 00 00 00   ··k··(·· ········
0030  00 00 00 00 00 00 00 00  04 00 00 00 03 00 00 00   ········ ········
0040  60 57 f8 f5 f9 87 04 00  ed 90 f8 f5 f9 87 d6 01   `W······ ········
0050  04 00 00 00 80 01 00 00  00 00 00 00 00 00 00 00   ········ ········
0060  03 00 00 00 80 01 00 00  3f 00 00 00 01 00 00 00   ········ ?·······
```

Provider ID (etw.provider_id), 16 bytes | Packets: 968284 · Displayed: 12427 (1.3%) | Profile: Default

6. Select a specific message to see its details.



*ETW reader
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

etw.provider_name contains "WWAN-SVC" or mbim

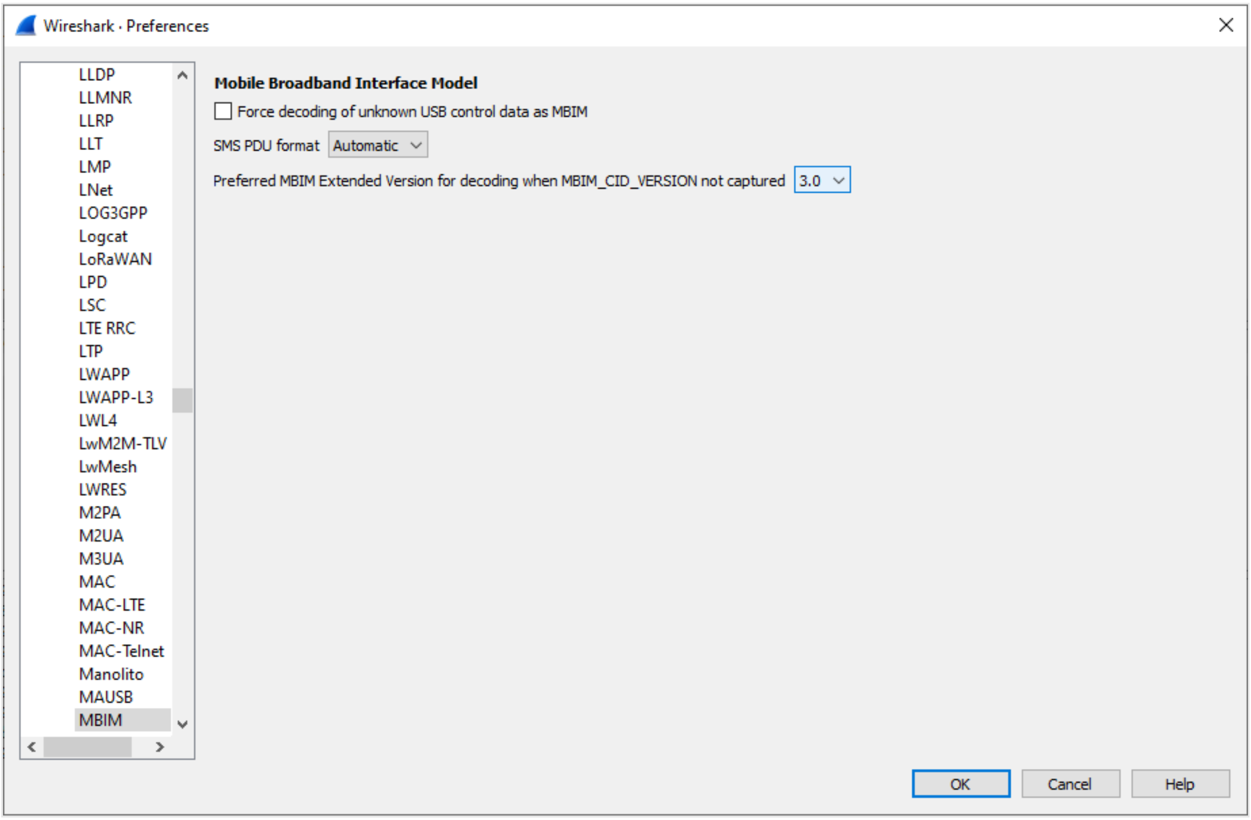
```
No.    Time                          Source    Destination  Protocol  Length Info
 29725 2020-09-10 22:29:12.500249   windows   windows      Micros…   508 WWAN Service event: [Info] wwanTxmAoAcRefHandler:  Acquiring AoAc Ref for Parent Interface before sending a TX [0x71] Interface: {{2e48402b-1fa7-4772-8bdb-02e121ab772a
 29726 2020-09-10 22:29:12.500253   windows   windows      Micros…   500 WWAN Service event: [Verbose] CWwanAoAcHelper::WwanInterfaceNicActiveControl: AddRef InterfaceLUID: f3008011000000 Interface: {{2e48402b-1fa7-4772-8bdb-02e121ab772a}}
 29728 2020-09-10 22:29:12.500288   windows   windows      Micros…   500 WWAN Service event: [Verbose] CWwanAoAcHelper::WwanInterfaceNicActiveControl: AddRef Completes err=0x0 RefCount=1…
 29740 2020-09-10 22:29:12.500459   host      device       MBIM      480 COMMAND_MSG        : DEVICE_SERVICE_SUBSCRIBE_LIST (Set)
 29777 2020-09-10 22:29:12.500984   windows   windows      Micros…   736 [Request=0x71] Sent SET with OID=OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS , miniport={2e48402b-1fa7-4772-8bdb-02e121ab772a}, ErrorCode=The request will be completed lat
 29778 2020-09-10 22:29:12.500990   windows   windows      Micros…   416 WWAN Service event: [Info] WwanTxSendReq: OID (getsettype: 32) sent to dim and completed with ERROR_NDIS_INDICATION_REQUIRED
 29790 2020-09-10 22:29:12.501003   windows   windows      Micros…   344 WWAN Service event: [Info] WwanTimerWrapper::StartTimer:  Timer (ID = 0) Start Completed
 29791 2020-09-10 22:29:12.501005   windows   windows      Micros…   436 WWAN Service event: [Info] WwanTxmEvaluateArmTimer: TXM timer armed for 60 seconds Interface: {{2e48402b-1fa7-4772-8bdb-02e121ab772a}}
```

```
> Frame 29740: 480 bytes on wire (3840 bits), 480 bytes captured (3840 bits) on interface unknown, id 0 (outbound)
v ETW Header
    Size: 464
    Header Type: 0
    Flags: 576
    Event Property: 0
    Thread ID: 13020
    Process ID: 3908
    Time Stamp: 132442757525004593
    Provider ID: a42fe227-a7bf-4483-a502-6bcda428cd96
  > Descriptor
    Processor Time: 12884901892
    Activity ID: f5f85760-87f9-0004-ed90-f8f5f987d601
  > Buffer Context
    User Data Length: 384
    Message Length: 0
    Provider Name Length: 0
v Mobile Broadband Interface Model
  > Message Header
  > Fragment Header
    Device Service Id: UUID_BASIC_CONNECT (a289cc33-bcbb-8b4f-b6b0-133ec2aae6df)
    CID: DEVICE_SERVICE_SUBSCRIBE_LIST (19)
    Command Type: Set (1)
    Information Buffer Length: 336
  v Information Buffer
      Element Count: 6
    > Device Service Subscribe Ref List
    v Device Service Element #1
        Device Service Id: UUID_BASIC_CONNECT (a289cc33-bcbb-8b4f-b6b0-133ec2aae6df)
        CID Count: 18
        CID: DEVICE_CAPS (1)
        CID: SUBSCRIBER_READY_STATUS (2)
        CID: RADIO_STATE (3)
        CID: PIN (4)
        CID: PIN_LIST (5)
        CID: HOME_PROVIDER (6)
        CID: PREFERRED_PROVIDERS (7)
        CID: VISIBLE_PROVIDERS (8)
        CID: REGISTER_STATE (9)
        CID: PACKET_SERVICE (10)
        CID: SIGNAL_STATE (11)
```

```
0010  31 e1 59 7b fc 87 d6 01  27 e2 2f a4 bf a7 83 44   1·Y{···· '·/····D
0020  a5 02 6b cd a4 28 cd 96  01 00 00 00 04 00 00 00   ··k··(·· ········
0030  00 00 00 00 00 00 00 00  04 00 00 00 03 00 00 00   ········ ········
0040  60 57 f8 f5 f9 87 04 00  ed 90 f8 f5 f9 87 d6 01   `W······ ········
0050  04 00 00 00 80 01 00 00  00 00 00 00 00 00 00 00   ········ ········
0060  03 00 00 00 80 01 00 00  3f 00 00 00 01 00 00 00   ········ ?·······
```

Provider ID (etw.provider_id), 16 bytes | Packets: 968284 · Displayed: 12427 (1.3%) | Profile: Default

The MBIM extended version used to decode the MBIM messages will be chosen automatically if MBIM_CID_VERSION is found. If MBIM_CID_VERSION is not found in an ETL file or live session, you can manually choose the MBIM extended version to decode the MBIM messages. Click Edit->Preferences...->Protocols->MBIM->Preferred MBIM Extended Version for decoding when MBIM_CID_VERSION not captured.



Mobile Broadband Interface Model

☐ Force decoding of unknown USB control data as MBIM

SMS PDU format: Automatic

Preferred MBIM Extended Version for decoding when MBIM_CID_VERSION not captured: 3.0

Mobilebroadband TextAnalysisTool Filters

Article • 12/15/2021

TextAnalysisTool is an extensive text filtering tool that is useful for complex traces with numerous ETW providers. You can filter logs of interest using the .tat files.

TextAnalysisTool filters:

- Basic Connectivity Log Filter
- DSSA Log Filter
- eSIM Download and Install Log Filter
- eSIM Profile Operations Log Filter
- LTE Attach Operation Log Filter
- PIN Operations Log Filter

netsh mbn test

Article • 12/15/2021

netsh mbn test is a test kit separate from the normal release version. You need to install the Hardware Lab Kit (HLK) client first to enable this feature.

HLK

Install the HLK client to enable the **netsh mbn test** function in DUT.

Alternatively, you can install **HLK Taef Tool** in the HLK package (path: installer\HLK-TAEF-TOOL-[arch-language]) to enable **netsh mbn test**.

Run the test command on the Device Under Test (DUT) that installed the HLK client.

Example:

```
netsh mbn test feature=connectivity param="AccessString=internet"
```

VHLK

You can also use the VHLK client to enable the **netsh mbn test** function in DUT.

Setup: Install VHLK on a host PC and run the script ConfigureTestSetup.ps1 <DUT's IP> from the location: C:\Program Files (x86)\Windows Kits\10\Hardware Lab Kit\Tests\amd64\Net\logo\Wwan

Then you can run the test command on the DUT.

Example:

```
netsh mbn test feature=connectivity param="AccessString=internet"
```

WLAN feature information

Article • 12/15/2021

This section contains specific WLAN feature information.

Wi-Fi network preference

Wi-Fi network preference is as follows.

1. User-defined networks are used first. These are listed in the Wi-Fi CPL as *Known networks*. A network becomes a *Known network* in the following cases.

   - The user manually taps on the network (including hotspots) in the Wi-Fi CPL.
   - The user adds the network manually from the Wi-Fi CPL.
   - Wireless network profiles that are roamed to the phone from Windows PCs.

   If multiple user-defined networks are found during the network scan, the most recently connected network is used.

   Note Wireless network profiles that are roamed to the phone from Windows PCs do not show up in the *Known networks* list until the phone has connected to the network once.

2. Hotspots (from a Store application or OEM plug-in) are prioritized next. If multiple hotspots are found, the following factors are used to decide preference.

   - Network security. Secure networks (any security type) are preferred over open networks.
   - Signal strength.

Note When there are multiple networks that have the same name or SSID, only the first configured network that uses the name or SSID is accepted.

Windows does not disconnect from a network once it is connected if the signal from the network is still present, even when a more preferred network becomes available. For example, if a user connects to a mobile operator hotspot at a store, and walks home while the signal from the hotspot is still present, the phone will not move to the user's configured home Wi-Fi network.

Also in this section:

- Fast Roaming with 802.11k, 802.11v, and 802.11r

Fast Roaming with 802.11k, 802.11v, and 802.11r

Article • 10/04/2024

Improved WLAN roaming experiences are available to devices running Windows 10. Industry standard implementations that reduce the time needed for a device to roam from one wireless access point (AP) to another are supported.

802.11k (Neighbor Reports)

Wireless Access Points (APs) that support 802.11k are able to provide Neighbor Reports to devices running Windows 10. Neighbor Reports contain information about neighboring access points and allows the device to have a better understanding of its surroundings. Windows 10 takes advantage of this capability by shortening the list of channels that the device needs to scan before finding a neighboring AP to roam to.

802.11v (BSS Transition Management Frames)

APs that support 802.11v can now direct Windows 10 devices to roam to another AP that it deems will provide a better WLAN experience for the device. Windows 10 devices can now accept and respond to these Basic Service Set (BSS) Transition Management frames, leading to improved WLAN quality when connected to a network that supports 802.11v.

802.11r (Fast BSS Transition)

Fast BSS Transition reduces the time needed for a Windows 10 device to transition to an AP that supports 802.11r. This time reduction results from fewer frames being exchanged with the AP prior to data transfer. By decreasing the time before data transfer when the device roams from one AP to another, the connection quality is improved for latency sensitive applications, such as an active Skype call. Windows 10 supports Fast BSS Transitions over networks using 802.1X as the authentication method. Pre-Shared Key (PSK) and Open Networks are currently not supported.

With the combination of 802.11k, 802.11v, and 802.11r, Windows 10 takes advantage of established industry standards to improve the roaming experience for our users. VoIP applications can now take advantage of this improved roaming to deliver better call quality when users are not stationary.

Things to note

Not all Windows 10 devices support 802.11k, 802.11v, and 802.11r. The WLAN Radio driver must support these features to enable them to work on Windows 10. Please check with your device manufacturer to determine whether or not your device supports these features. In addition to device-side support, the network (AP Controllers and APs) must also support the features for the experience to work. Please check with your network administrator to see if these features are supported and have been enabled on the network in question.

Windows 10 continues to support Opportunistic Key Caching (OKC) when 802.11r is not available on the device or the network.

All three features require AP-side support and don't work unless those features are enabled on the APs.

WDI Miniport Driver Design Guide

Article • 09/27/2024

ⓘ **Important**

WiFiCx is the new Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features. The WDI driver model is now in maintenance mode and will only receive high priority fixes.

WLAN Device Driver Interface (WDI) is the new Universal Windows driver model for Wi-Fi drivers, for both Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) and Windows 10 Mobile. The WLAN device manufacturer writes a WDI miniport driver to work with the Windows 10 OS implementation. WDI enables device manufacturers to write less code than the previous Native WLAN driver model. All new WLAN features introduced in Windows 10 require WDI-based drivers.

Vendor-supplied native WLAN drivers continue to work in Windows 10, but functionality is limited to the version of Windows for which they were developed.

The WDI requirements and interface specification are documented in this design guide. The key goals for the new model are:

- Improve the quality and reliability of Windows WLAN drivers.
- Reduce the complexity of the current driver model, which in turn reduces the complexity of the IHV driver and reduces the overall cost of IHV driver development.

The focus of this documentation is to specify the flow and behavior of Wi-Fi operations between Windows and the IHV driver component. It does not cover software interface signature (for example, the device driver interface model) and details about how the IHV component is loaded in Windows.

Design principles

The following principles guided the overall model and design of this protocol.

1. Minimize the chattiness of the traffic between the host component and the IHV component/device. This is particularly important for implementations on buses such as SDIO, which is inherently chatty.
2. Wi-Fi functionality (especially functionality that must be performed with low latency) is expected to be handled by the device.

3. All regulatory related functionality resides in the IHV component and is controlled by the IHV.
4. The Windows experience is controlled by the host component and the Windows operating system.
5. Windows has the ability to resurrect hung devices. It has enough state to reprogram the IHV component and recover within 10 seconds.
6. Operations that require lot of system memory or fast processors and are not vendor specific are handled by the host.

Definitions

Expand table

Term	Description
Device	The entire piece of hardware that connects to the bus. A device can have multiple radios in it (notably Wi-Fi and Bluetooth).
Wi-Fi adapter	The specific part of the device that implements Wi-Fi functionality as described in this specification.
Port	An object that represents a MAC and PHY state for a particular connection.
IHV component	The IHV-developed software component that represents the Wi-Fi Adapter/Device to the host.
Host	The host-side Microsoft/operating system software that interacts with the IHV component using the interfaces described in this specification.
Upper Edge Driver (UE)	UE refers to the WdiWiFi driver, called WDI in this documentation. The UE and the Lower Edge (LE) IHV driver combine into a complete NDIS miniport driver. The UE implements the core Wi-Fi logic.
Lower Edge Driver (LE)	LE refers to the IHV driver at the lower edge. The LE and UE combine into a complete NDIS miniport driver. The LE implements bus and hardware specific functions.
Functional Level Reset (FLR)	Functional Level Reset, as in the PCIe specification. This term refers to the reset of a

Term	Description
	function, versus a reset of the complete device which may have a composite function. The reset of such scope does not impair the other functions on the same device.
Platform Level Reset (PLR)	Platform Level Reset. This reset method impacts all functions on a device. It is very popular to build multiple functions on a device to reduce the cost and footprint. For example, Bluetooth is typically built with Wi-Fi on a chip. However, such a reset method resets all function units on the device.
Reset Recovery (RR)	RR refers to the event sequence of Reset and Recovery. For FLR, this includes: • The request to NDIS, which forwards the request to the bus to reset the Wi-Fi function. • Recovery of firmware context by the driver. • Reconnection to the access point if it was connected before the reset. For PLR, this includes: • The request to NDIS, which forwards the request to the bus. The bus interacts with PnP to surprise-remove the device. • Re-enumeration of the device. • Re-establishing the device stack. • Wi-Fi is restarted and reconnects.
WDI commands	The UE sends WDI OIDs and calls LE callbacks. All of these are called WDI commands.
MAC Address Randomization	In order to improve the privacy of Windows 10 users, configured Wi-Fi MAC addresses are used in some circumstances, such as before connecting to a particular Wi-Fi network or when initiating scans in specific conditions. This only applies to the station port. The system ensures that randomization is used appropriately, so important connectivity scenarios are not broken. The system manages changes of addresses by issuing OID_WDI_TASK_DOT11_RESET commands prior

Term	Description
	to issuing a scan or connect command. The reset command parameters include an optional MAC address argument. If the argument is present, the MAC address is reset to the specified value. If it is absent, the MAC address is left to the current value. When configuring randomized MAC addresses, the operating system uses the "locally administered" format defined for IEEE802 addresses.
ECSA	Extended Channel Switch Announcement.

Related topics

WDI Miniport Driver Reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Communication model, synchronization, and abort

Article • 12/15/2021

At a high level, this documentation defines two types of objects:

1. An adapter, which represents the Wi-Fi device.
2. A port, which represents distinct MAC and PHY entities in the adapter.

For more information about these objects, see Wi-Fi device model and objects. Commands, a set of permissible operations, are defined for each of these objects. Commands are further categorized into Properties and Tasks.

Property commands are simple commands (such as get signal strength, get current BSS list, and set packet filter). They complete in a short duration of time and are not complicated to implement.

Task commands are complex operations that may take several seconds to complete. For example, a Wi-Fi Scan operation would be categorized as a task in this model.

All commands issued to the IHV component can be completed asynchronously.

Sequence of messages

The sequence of messages for each type of command is shown in the following figures.

Figure 1 shows the task command sequence.

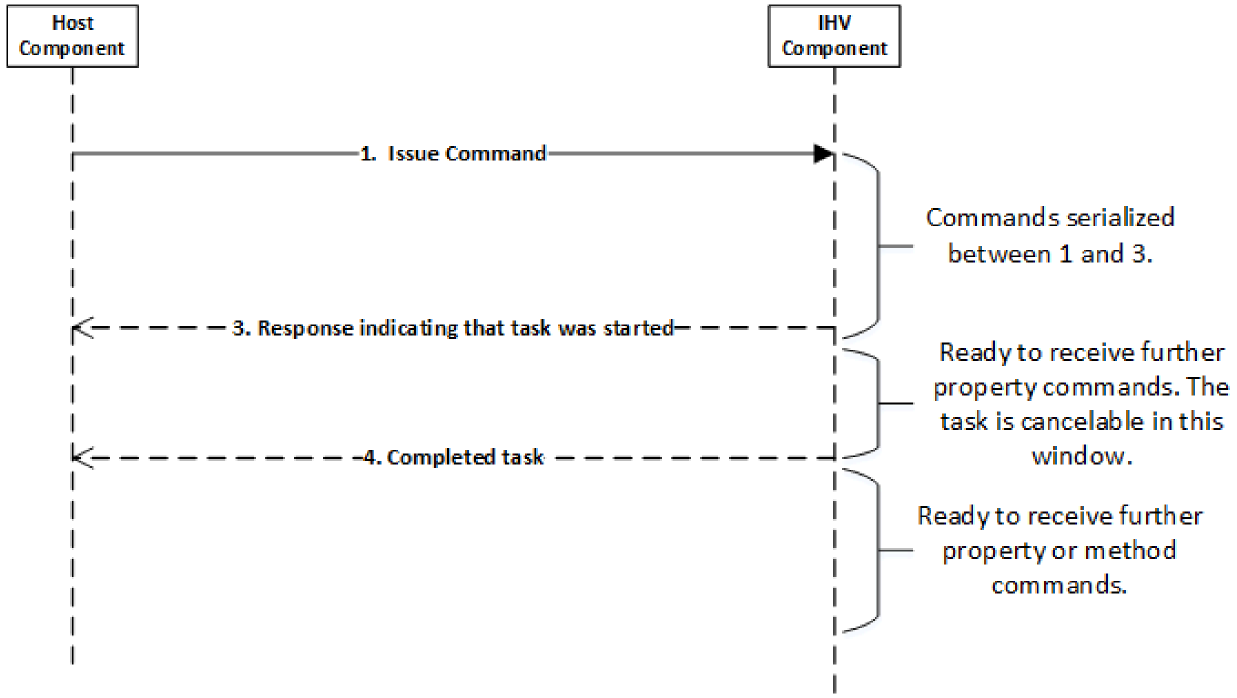

Host Component
IHV Component
1. Issue Command
3. Response indicating that task was started
4. Completed task
Commands serialized between 1 and 3.
Ready to receive further property commands. The task is cancelable in this window.
Ready to receive further property or method commands.

Figure 2 shows the property command flow.

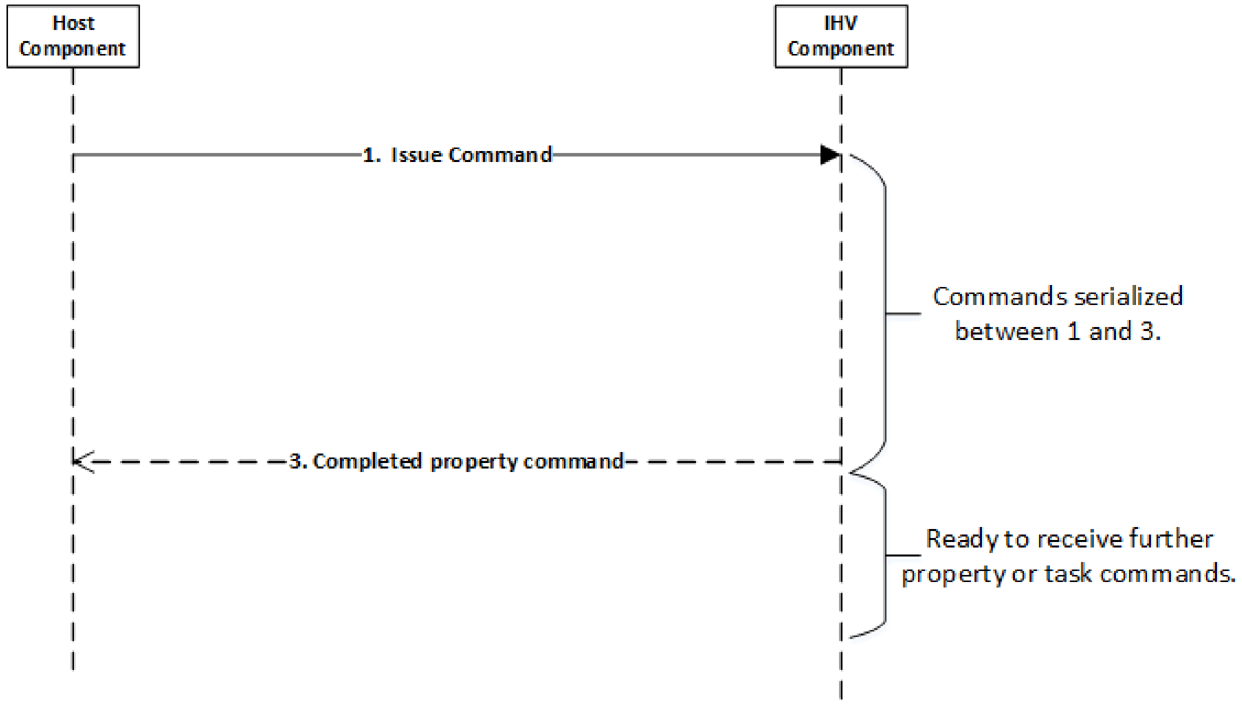
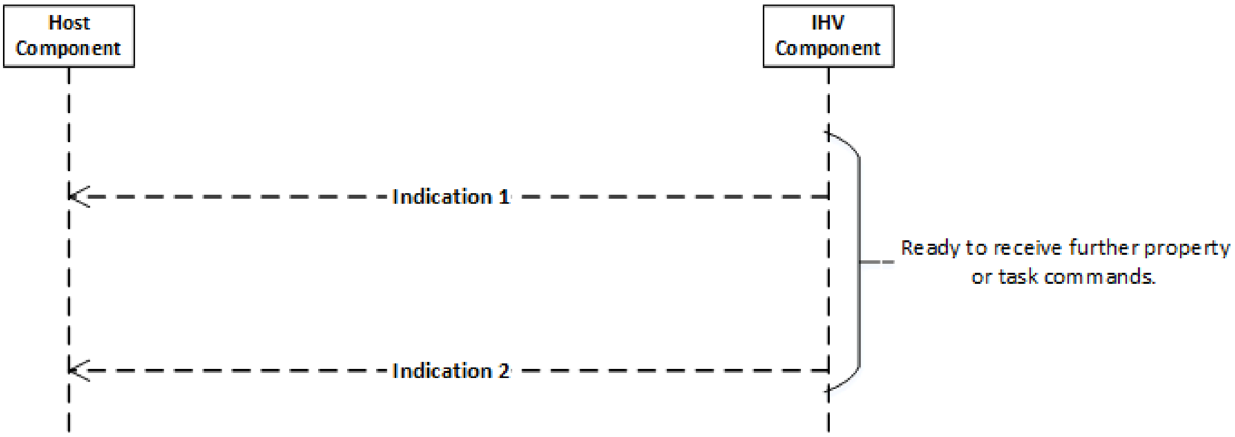

Host Component
IHV Component
1. Issue Command
3. Completed property command
Commands serialized between 1 and 3.
Ready to receive further property or task commands.

Figure 3 shows the flow for indications.



Synchronization

To keep the IHV component implementation simple, the model defines the following synchronization rules:

1. Commands are always serialized between Steps 1 and 3 in Figure 1 and Figure 2. For example, no new commands are issued to the adapter until the indication from the adapter at Step 3. This also implies that all properties are serialized with each other.
2. All task commands are serialized between Steps 1 and 4 in Figure 1. For example, only one task runs on the adapter at a time. However, once a task is started (Step 3 in Figure 1), the adapter may get property command requests. Both Step 3 and Step 4 must finish before the next task command is sent.
3. Property set commands are of two types – those that can be sent after the task has started, and those that must be serialized with pending tasks.
4. Data path is not serialized with the command path, except for specific cases described later in the documentation.
5. The synchronization scope is adapter level scope.
6. A subset of tasks can be aborted after they have been started. This means that if a higher priority task (A) arrives while a lower priority task (B) is outstanding, B can be aborted by the host. Rationalization of prioritization decisions is beyond the scope of this documentation and is dependent on user scenarios.
7. For task commands, Step 4 can come before Step 3 has completed. However, if Step 4 is indicated, Step 3 cannot fail.

Abort

Most tasks can be aborted after they have been started. The purpose of the abort is to trigger the adapter to finish the task quickly by sending the complete indication (Step 4 in Figure 1). Abort is allowed only in the window between Steps 3 and 4 in Figure 1. On receiving the abort, the adapter must complete the task within 50 milliseconds. For most commands, upon receiving the abort, the adapter does not need to roll back to the state before the command was started. Race conditions exist between the abort command being issued and completions arriving to the host component. In this case, if the IHV component receives an abort for a task it has already completed, no further action is needed from the IHV component to process the abort operation. Aborting a task is simply a signal that the IHV component should clean up the task as soon as possible. Command completion semantics are not changed if an abort is issued. Both the completion for the abort property command, and the task completion indication must be appropriately notified in all cases.

Properties are expected to complete in short time so they cannot be aborted.

Task commands have a unique identifier that allows the host to target a specific command for abort.

Hang detection and recovery

Article • 12/15/2021

After a command is issued to the IHV component, the host starts a timer. If the timer expires before the IHV component completes (Step 3 message in the figures in Communication model, synchronization, and abort), the driver assumes that the IHV component is hung, resets the IHV component, and recovers if the precondition is correct.

The precondition is that the system will provide ACPI methods to reset the device, either at a bus or at the device level.

M1-M3 Hang Timeout is 10 seconds.

M3-M4 Task Hang Timeout is 30 seconds, or configurable based on task.

> ⓘ **Note**
>
> Some tasks may be expected to take longer than 30 seconds to complete (for example, Wi-Fi Direct Discover for the selected registrar bit in certain scenarios). In these cases, the host-initiated task timeout is adjusted accordingly to allow for 30 seconds longer than the maximum expected runtime of the task.

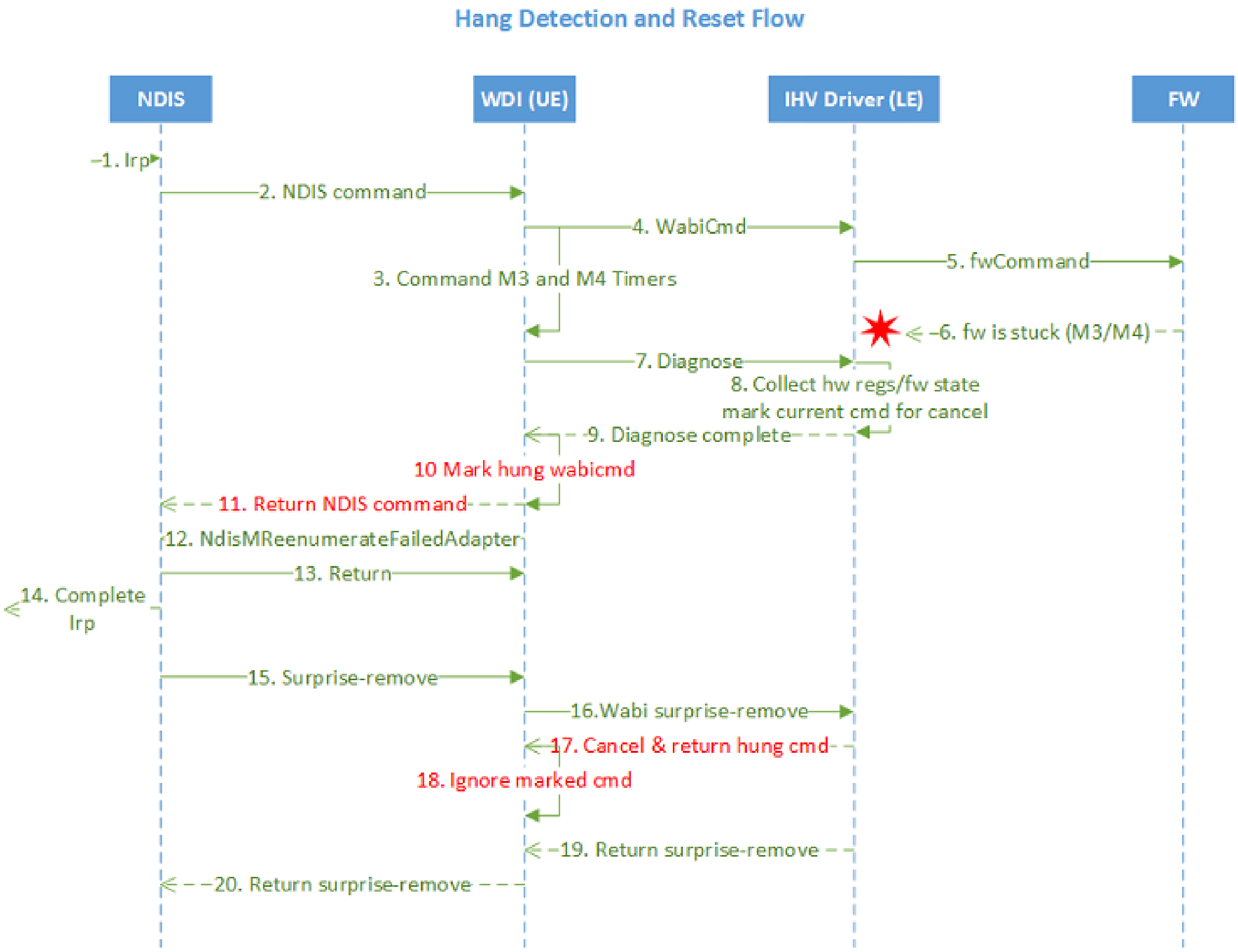
These are maximum upper bounds for the commands and processing that takes longer than this time is considered an error. It is expected that under a normal mode of operation (no CPU stress), most tasks and properties finish significantly sooner than the timeouts specified above. These values are specified with each task/property. The adapter should ensure that it does not have waits that would cause those execution times to be exceeded.

In this section

UE hang detection and recovery flow

Article • 12/15/2021

This diagram shows the UE hang detection and reset flow.



The diagram consists of main phases:

1. Hang detection
2. Log
3. Reset: surprise-remove handling

It is important to keep in mind that this feature addresses firmware hangs. It does not solve IHV driver hangs. IHV driver hangs can only be solved with a hard reboot. The driver must release all handles and other resources before it can be unloaded. If the driver can't be unloaded, the recovery does not work.

This flow diagram applies generically to all NDIS OIDs and callbacks to miniports. There may be exceptional cases where if NDIS or the Bus can't fully support reset recovery, the recovery part of the reset recovery won’t work. Two example cases are during miniport initialization or during a miniport halt operation.

Related topics

UE hang detection: steps 1-14

Reset (surprise remove): steps 15-20

UE hang detection: Steps 1-14

Article • 12/15/2021

Steps 1 through 14 of UE hang detection are described below. The steps correspond to the diagram shown in UE hang detection and recovery flow.

This example uses OID_SET_POWER.

1. NDIS receives a system power IRP or the NIC active references drop to 0.

2. NDIS generates an OID_SET_POWER D3 to the WDI driver.

3. WDI sets a timer for a WDI command (M1), including a task before it sends the WDI OID down to the LE. If the command is a task, an additional timer for the task is also set. Both timers can time out, but at most, only one can trigger a reset recovery.

4. WDI sends the WDI command to the LE. The recommendation for the LE is to remember the WDI OID in the adapter structure if it needs a firmware command to complete the OID. When the firmware completes the job for the WDI OID, the LE completes the OID and removes it from the adapter structure. Since WDI serializes OIDs to the LE, the LE needs only one slot to remember the pending WDI OID. If the firmware command is hung, the LE can return the OID at any time but no later than at surprise-remove (it can be in the context of surprise-remove) at Step 17 when the firmware has been disabled. For any other cases, the LE simply completes the WDI OID when the firmware completes the corresponding job, regardless of if it is before or after a diagnose callback. If the LE needs to remember the WDI OID after Diagnose, it needs another slot to remember it. However, for the OIDs received after Diagnose, the LE should not touch the firmware to avoid cascaded hangs.

5. The LE sends a command to the firmware.

6. If the firmware command timed out, it may be due to a firmware hanging or taking too long.

   - If the firmware is simply taking too long to complete the command after a time out, the LE can complete the WDI command. The UE handles it appropriately.
   - If the firmware is hung, the WDI command is not completed soon. The LE must complete it at surprise-remove at Step 16 when the hardware has been reset, so it is safe to complete without special handling for a potential race condition.

7. The WDI timer fires to generate a WDI Diagnose command. This WDI command is a call to the LE driver, *MiniportWdiAdapterHangDiagnose*, rather than a WDI OID.

8. LE collects hardware control register states, and optionally, the full firmware state.

    - The IHV driver is expected to collect hardware register content which is limited to 1KB, and return it in the function return. Additionally, in the pre-production environment, the LE should also try to dump the firmware context into files so that the IHV can do post-mortem debug thoroughly. The switch can be implemented as a registry key to control the collection of hardware registers and the firmware image.
    - The LE also marks the current command for cancellation. If command completion races to beat the diagnosis, it is acceptable if the LE does nothing for this command.
    - With this command pending, the UE may send further WDI commands as the consequence of Reset. These are in-flight commands or clean-up commands. After the diagnose call, the LE should process them without touching the firmware.

9. WDI receives the control register state.

10. WDI marks the hang WDI command so that it is indicated later by the LE.

11. WDI returns (completes) the NDIS command without the WDI completion. This is safe because WDI double-buffers NDIS commands.

12. WDI calls NDIS to reset and calls **NdisWriteErrorLogEntry** with *Error Code* of **NDIS_ERROR_CODE_HARDWARE_FAILURE** (0xc000138a). This results in an event recorded in the system event log with the module name of LE. The error event ID automatically pops up as (0xc000138a | 0xffff) – 0n5002. If the LE also uses the same error code to write error logs, the first DWORD of the data should contain the high bit set (0x80000000) to easily separate events by the LE. WDI uses 0x00000000 to 0x7fffffff for the first DWORD data.

13. The call returns.

14. NDIS completes the IRP.

After this point, NDIS calls the bus to surprise remove and re-enumerate us. It is important to note that WDI double-buffers NDIS commands so that it does not have to wait for the WDI command to return to complete the NDIS command. This eliminates the need for the LE to do cancel logics, which are notoriously complex to do.

The completion of NDIS OID_SET_POWER is necessary to avoid a deadlock of PnP operations. All PnP and power state changing events are serialized. This means that if a Set power OID does not return, NDIS is not able to return the Set power IRP, which means the Reset Recovery locks up with the Surprise-Remove IRP.

Related topics

MiniportWdiAdapterHangDiagnose

Reset (surprise remove): steps 15-20

UE hang detection and recovery flow

Reset (surprise remove): steps 15-20

Article • 12/15/2021

The steps of reset (surprise-remove), which are Steps 15 through 20, are described below. The steps correspond to the diagram shown in UE hang detection and recovery flow.

Once the Reset Recovery can proceed, the bus causes PnP to generate a surprise-remove IRP. When NDIS receives the surprise-remove IRP, it calls back WDI for a surprise-remove PnP event callback. WDI forwards the surprise-remove as a WDI command to the LE, where the LE returns the hung WDI command. The rest of flow is identical to a real device surprise-remove on a bus (for example, USB).

Cleanup commands flow to the LE to facilitate the return of resources. In this state, the LE should not touch the hardware.

Step	Action
15	NDIS calls back the PnP event for surprise-remove.
16	WDI calls back the LE for surprise-remove.
17	The LE returns the hung WDI command. The LE only needs a slot for outstanding WDI commands because WDI serializes WDI commands to the LE, except Diagnose and Abort.
18	WDI ignores the return of the hung WDI command because it has returned the original NDIS command.
19	The LE returns WDI surprise-remove.
20	WDI returns NDIS PnP callbacks for surprise-remove.

Related topics

UE hang detection: steps 1-14

UE hang detection and recovery flow

Timings for diagnose call

Article • 12/15/2021

The timing requirements of Diagnose to collect debug information are as follows.

At the level of **DiagnoseLevelHardwareRegisters**, the LE is expected to collect device control registers no more than 1KB in the output buffer of the Diagnose call. This is the setting for a normal release product. It is intended for collecting the vital information of device control registers. The time limit to collect such information is 25ms.

At the level of **DiagnoseLevelFirmwareImageDump** or **DiagnoseLevelDriverStateDump**, the LE is expected to collect the device control registers and firmware full dump. If time permits, the LE can also collect the driver state, subject to the time limit. Except the control registers collected in the Diagnose output buffer, the firmware dump and driver state should be saved to files with the choice of names in %windir%\system32\drivers. The time to collect all debug information at either level should be within 25 seconds. These diagnose levels are meant to be used at the self-host phase.

Related topics

eDiagnoseLevel

MiniportWdiAdapterHangDiagnose

LE hang detection

Article • 12/15/2021

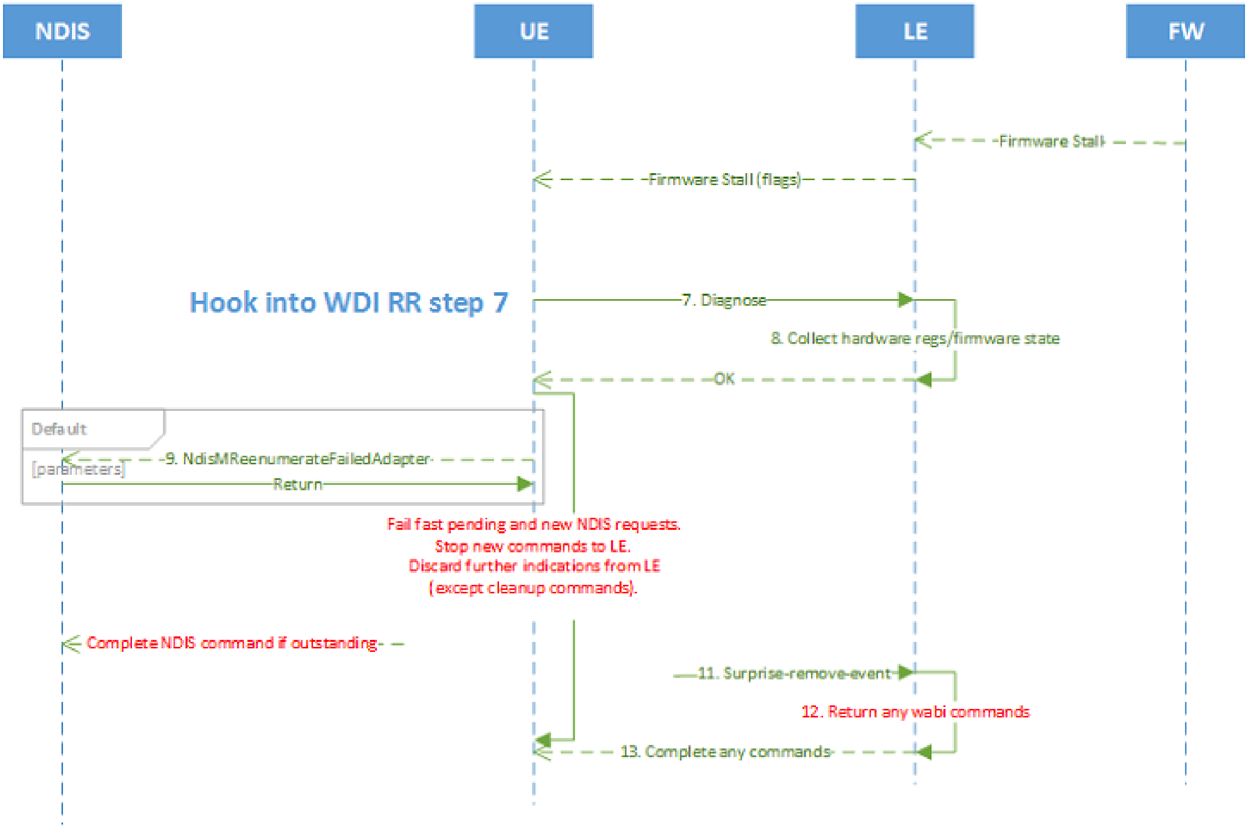
Some firmware have a watchdog timer that can detect firmware hangs. Some IHV drivers (LE) have logic to detect if the firmware is not making forward progress. The UE allows the LE to indicate such conditions.

The indication should be on the Adapter port (for example, portid=0xFFFF). By default, the indications trigger the LE to perform the full reset recovery procedure -- calling diagnose, collecting debug information, and requesting PLDR.

When the LE or firmware watchdog timer detects that the firmware stalled, the expectations from the UE are as follows.

1. If in D0,
   a. The LE indicates NDIS_STATUS_WDI_INDICATION_FIRMWARE_STALLED.
   b. At the return from the indication, the LE returns (if any) the stalled WDI command.
   c. The UE starts the Reset Recovery (RR) procedure.

2. If in Dx, this can only happen with firmware detected stall.
   a. Firmware raises wake interrupt.
   b. On receiving a D0 command, indicates the wake reason of why the firmware stalled.
   c. After returning D0 WDI OID, the LE indicates NDIS_STATUS_WDI_INDICATION_FIRMWARE_STALLED.
   d. Finish the procedure as in D0: 1a, 1b, and 1c.


NDIS
UE
LE
FW
Firmware Stall
Firmware Stall (flags)
Hook into WDI RR step 7
7. Diagnose
8. Collect hardware regs/firmware state
OK
Default
[parameters]
9. NdisMReenumerateFailedAdapter
Return
Fail fast pending and new NDIS requests.
Stop new commands to LE.
Discard further indications from LE
(except cleanup commands).
Complete NDIS command if outstanding
11. Surprise-remove-event
12. Return any wabi commands
13. Complete any commands

Hang detection in Dx

It is possible that the firmware stops progress in Dx. In this case, Dx is D3Hot for PCIe NIC and D2 for USB and SDIO. The NIC is armed to wake and expected to maintain access point association autonomously, or scan NLO if not associated.

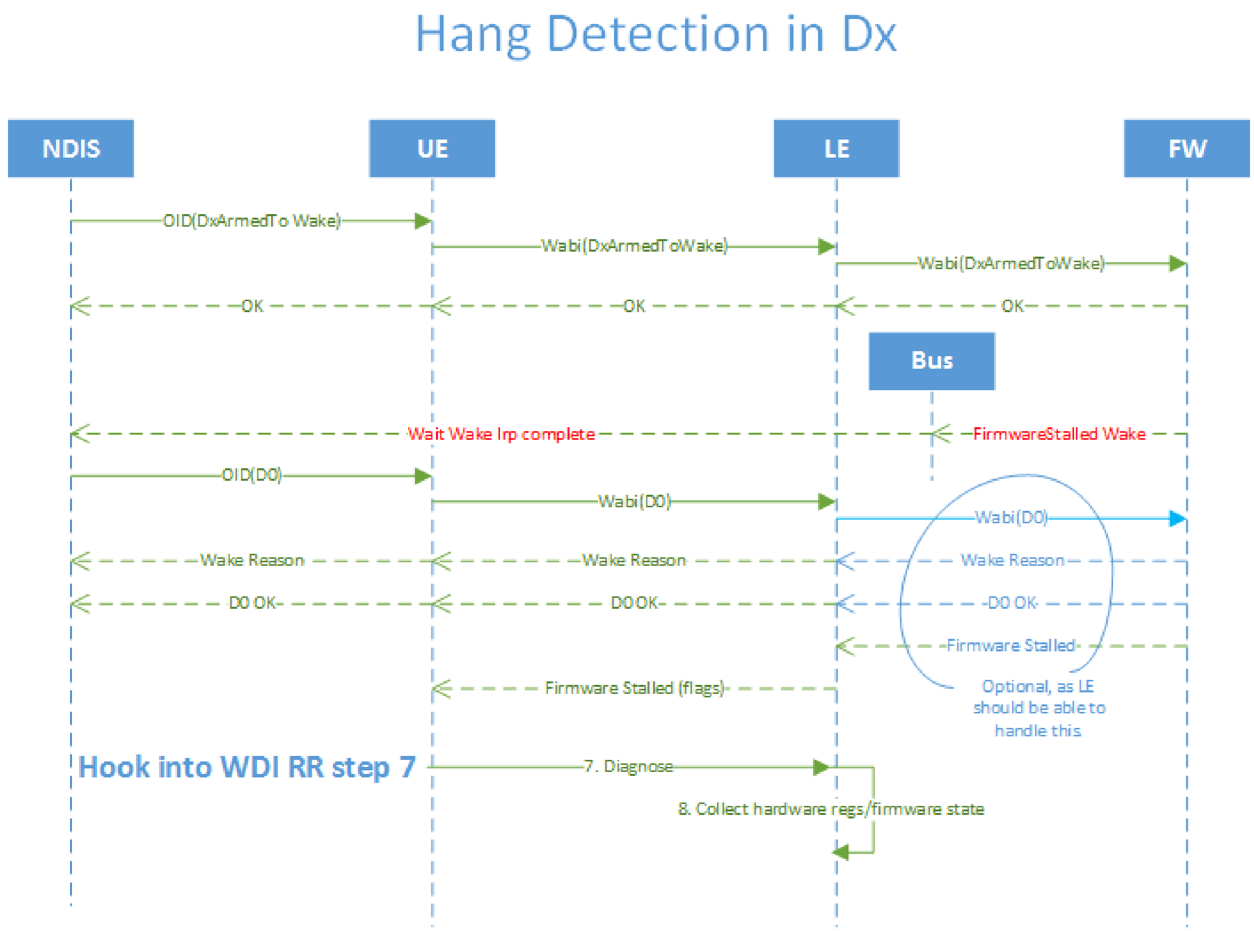
When the NIC is in Dx, the communication to the host is blocked because the bus could be in the power off state. Therefore, the LE is not able to detect stalled firmware. The firmware itself has to detect the condition and raise the wake line (if the wake portion of code is still alive) to bring the stack to D0, indirectly via ACPI or bus completing, NDIS wait_wake_irp. Due to this, NDIS sets D0 to the NIC.

The firmware asserts wake for such a condition. The LE should indicate a wake reason for the firmware stall. The wake reason **WDI_WAKE_REASON_CODE_FIRMWARE_STALLED** is defined as an enum with the other wake reasons.

For Reset Recovery to work in this scenario, at least two portions of the firmware must still function.

1. The hang detection code.
2. The code to assert wake interrupt.

If there is a lack of either one, the Host side does not know if the firmware is stalled and RR does not happen. This scenario is not part of the design goal.



OS module triggered reset recovery

This is informational for IHVs. In addition to UE and LE detected hangs, other OS components may detect hangs and/or trigger the UE to invoke the Reset Recovery procedure. Currently, the user mode wlansvc component in Windows 10 may request a Reset Recovery to UE when it detects a connection with Internet connectivity and subsequently loses the ability to access a DNS server without disassociation for some time. In the future, Microsoft may find additional cases to trigger a Reset Recovery to enhance end user experiences.

Related topics

NDIS_STATUS_WDI_INDICATION_FIRMWARE_STALLED

WDI_TLV_INDICATION_WAKE_REASON

PLDR

Article • 03/14/2023

Recovery of PLDR

After the surprise-remove, the drivers (both UE and LE) must release all resources so that the device object can be removed and re-enumerated (by the bus). If this does not happen, the device is not re-enumerated, and therefore not recovered.

WDI message structure

Article • 03/14/2023

All WDI command messages must start with a **WDI_MESSAGE_HEADER** structure. The command header is followed by zero or more type-length-value (TLV) structures.

The command message IDs defined for messages sent from the host to the Wi-Fi device are documented in WDI Task OIDs, WDI Property OIDs, and WDI Status Indications.

TLVs

The structure of TLVs is defined in the following table. The data in TLVs is in little-endian byte order.

Field	Type	Description
Type	UINT16	The type of the TLV structure. Unrecognized TLV types must be skipped without triggering errors.
Length of the Value buffer	UINT16	The size of the Value buffer in bytes.
Value	BYTE[*]	The payload buffer, which may contain a structure, a list of structures, or other TLVs. If there is more data than expected in a TLV, the additional data should be skipped without triggering errors.

There are two types of TLV groupings: statically sized TLV lists, and multi-TLV groups.

Statically sized TLV lists

Statically-sized TLV lists contain several statically sized members. They are analogous to standard C-style arrays.

In this example, **WDI_TLV_UNICAST_ALGORITHM_LIST** is defined as a list of WDI_ALGO_PAIRS.

Type: WDI_TLV_UNICAST_ALGORITHM_LIST

Length: N * sizeof(WDI_ALGO_PAIRS)

Value: WDI_ALGO_PAIRS[N]

This usage is specified in the TLV reference topics with array notation.

Multi-TLV groups

When the size of a given object is not known ahead of time, multi-TLV groups are used. This usage pattern specifies that N different variably sized TLVs are expected within a given buffer. The number of entries (N) is not known ahead of time, and is inferred by the number of matching TLVs in the given buffer.

In this example, the parent buffer is a WDI_MESSAGE_HEADER, which defines the end of the TLV buffer. Note that WDI_TLV_BSS_ENTRY may be interspersed between other different TLV types in the parent buffer.

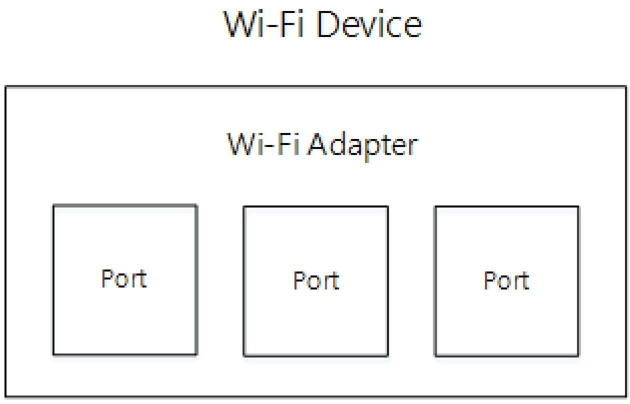
Offset	Field	Type
0	WDI_MESSAGE_HEADER	Message header
sizeof(WDI_MESSAGE_HEADER)	TLV_0 (WDI_TLV_BSS_ENTRY)	WDI_BSS_ENTRY
TLV_0 + L_0 + sizeof(TLV Header)	TLV_1 (WDI_TLV_BSS_ENTRY)	WDI_BSS_ENTRY
TLV_1 + L_1 + sizeof(TLV Header)	TLV_2 (WDI_TLV_BSS_ENTRY)	WDI_BSS_ENTRY
TLV_2 + L_2 + sizeof(TLV Header)	TLV_3 (OTHER_TLV_TYPE)	Some other TLV type
TLV_3 + L_3 + sizeof(TLV Header)	TLV_4 (WDI_TLV_BSS_ENTRY)	WDI_BSS_ENTRY

For TLVs that contain other TLVs, the TLV reference topics have a *Multiple TLV instances allowed* column. If this column is checked, the specified TLV is allowed to appear multiple times. For an example of this, see WDI_TLV_CONNECT_PARAMETERS.

Wi-Fi device model and objects

Article • 12/15/2021

The Wi-Fi device is used by the host in the context of two types of objects: adapter and port.



Adapter

The adapter object represents the Wi-Fi functionality in the Wi-Fi device. Commands and indications on this object are used to indicate state about the Wi-Fi interface. For systems with multiple Wi-Fi devices, each adapter object represents a different instance.

Port

One Wi-Fi adapter can be used simultaneously for multiple connections e.g. Infrastructure client and Wi-Fi Direct group owner. The port object is used to represent the state associated with each such connection. Each port holds the MAC state for the connection and any phy state specific to that connection.

There can be multiple ports in an adapter. Commands issued on a port should only affect the state maintained for that port.

The operating system configures each port with an operation mode, such as 802.11 station, Wi-Fi Direct Client, or Wi-Fi Direct Group Owner. The set commands that the firmware must be prepared to handle on a given port are determined by the operation mode and the state of the port. A port can be in one of two states: INIT and OP. The port is initially in the INIT state and transitions to the OP state only when the operating system issues a command to connect (in the case of infrastructure client) or to start an AP/GO. The port returns to the INIT state when OID_WDI_TASK_DOT11_RESET is sent to the IHV component.

Port availability requirements

Port type	Required count
Station Port	1
Wi-Fi Direct Device	1 (if supported)
Wi-Fi Direct Role (GO or Client)	1 or 2 (if supported)

Port concurrency requirements

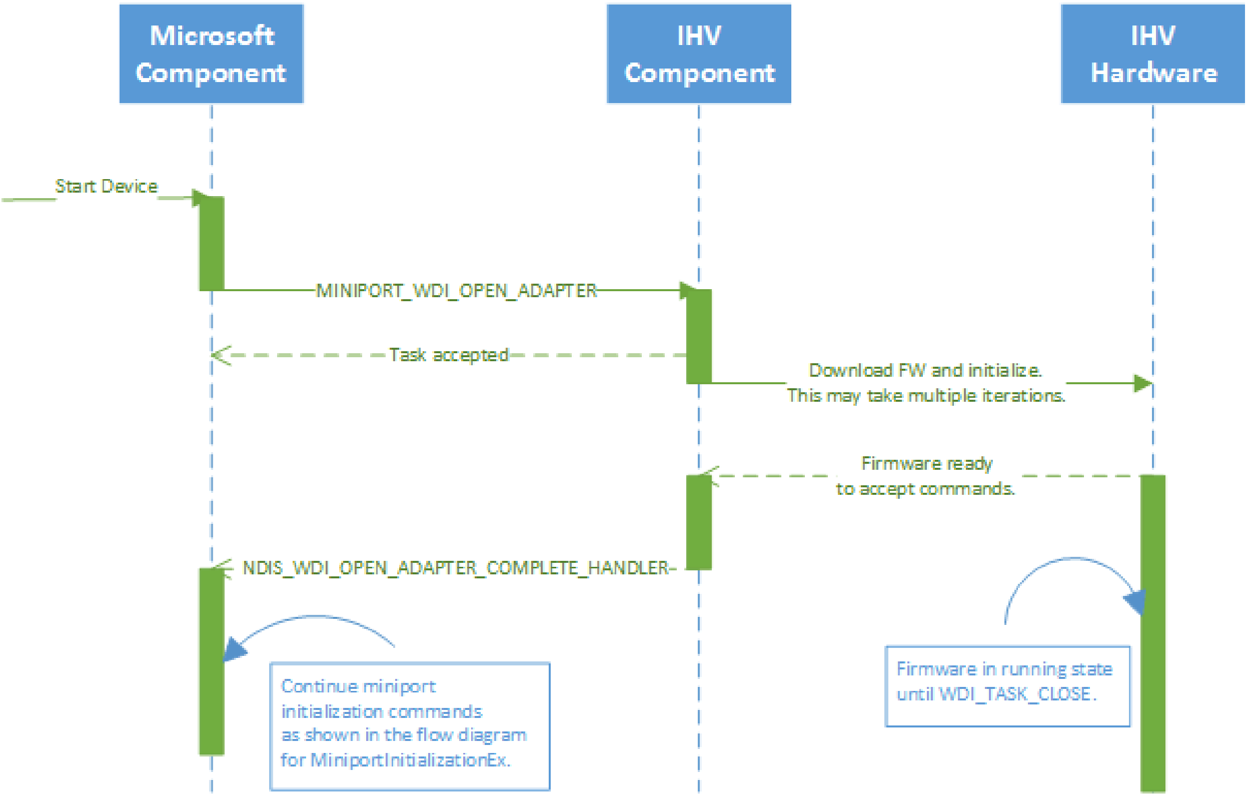
The following concurrency requirements for the different port types are as follows.

1. 1 Station Port is always available.
2. 1 Wi-Fi Direct Device port is always available.
3. 2 Wi-Fi Direct Role ports are available in the following configurations.
    a. 1 GO
    b. 1 Client
    c. 1 GO, 1 Client

Wi-Fi device initialization

Article • 12/15/2021

This topic describes the initialization of the Wi-Fi device after power-on. On power-up, most Wi-Fi devices come up in an uninitialized mode. The Wi-Fi device does not have sufficient ROM to hold the firmware, so the IHV component/driver programs the device with firmware as part of device boot-up. The following diagram shows the initialization sequence in a bus/interconnect independent way.


Microsoft Component
IHV Component
IHV Hardware
Start Device
MINIPORT_WDI_OPEN_ADAPTER
Task accepted
Download FW and initialize.
This may take multiple iterations.
Firmware ready
to accept commands.
NDIS_WDI_OPEN_ADAPTER_COMPLETE_HANDLER
Continue miniport
initialization commands
as shown in the flow diagram
for MiniportInitializationEx.
Firmware in running state
until WDI_TASK_CLOSE.

1. The IHV component is responsible for downloading the firmware to the adapter when the adapter is powered up. The exact mechanism to download the firmware is bus dependent. This operation is done in the context of the *MiniportWdiOpenAdapter* call. This is an asynchronous operation. The host is responsible for ensuring that the adapter is fully initialized and ready to process commands before further commands are sent to it. The exact mechanism is interconnect dependent.
2. Once the adapter is initialized, the host queries the adapter for various Wi-Fi properties, set properties, and creates ports (MACs) as part of miniport initialization.
3. After the ports are created and initialized, the adapter can receive task and property commands.

WDI task command priority and existing state

Article • 03/14/2023

When the adapter is in a particular state, new commands may come down to it that could affect the existing state (for example, a scan that affects existing connections). The table below describes how new commands should be prioritized against the existing state in the adapter. The columns describe how to service the existing state when the new command comes in.

New command Existing state Connection Quality (EAP) - Priority 1 P2P Listen - Priority 2 Connection Quality Latency (Media Streaming) - Priority 3 Existing Connections - Priority 4 Scan/P2P Discovery (forced) Important (delay scan) Pause Pause Throttle Scan/P2P Discovery (not forced) Important (skip scan) Maintain Important (skip scan) Throttle Station Connect, Roam, Disconnect Delay Connect Pause Pause Throttle P2P GO Start, GO Stop Delay Connect Pause Pause Throttle P2P Client Connect, Disconnect Delay Connect Pause Pause Throttle P2P Send Action Response Pause Pause Pause Throttle P2P Send Action Request Delay Send Maintain Pause Throttle

Term	Description
Important	Prioritize the existing state higher than the new request.
Maintain	Prioritize the existing state and the new command equally.
Throttle	Throttle down the servicing of the existing state so that it works, but prioritize the new command higher.
Pause	Stop servicing the existing state and attempt to finish the existing state as soon as possible.

WDI data transfer

Article • 03/14/2023

This section covers WDI data transfer. The following terminology is used in this section.

Term	Description
Target WLAN device (target)	A physical NIC.
Virtual WLAN device (port)	A virtual NIC (for example, P2P role ports).
WDI	A Microsoft WLAN component. It is a target-agnostic component.
Target Interface Layer (TIL)	A target-specific software layer that interfaces with the target through the bus-specific APIs. It creates and manages DMA channels and provides bus abstraction.
RX Manager / RxMgr	RxMgr performs receive processing steps that are not offloaded to the target.
RxEngine	RxEngine sends and receives data-synchronous messages to and from the target, interprets RX descriptor formats, and manages buffers for direct hardware to software RX DMA.
TX Manager / TxMgr	The WDI-compliant driver component that gets TX frames from the operating system, delivers them to the TxEngine at the appropriate time, and returns the completed TX frames back to the operating system.
TxEngine	The target-specific software component that handles the TX data transfer from host to target, and handles TX completion messages from the target.
Target Abstraction Layer (TAL)	The "lower-edge" that has a standardized API to the WDI compliant driver, but has a target-specific internal implementation. The TAL is a container layer for individual target-specific host software components such as TxEngine and RxEngine.

WDI receive operations and offloads

Article • 03/14/2023

These main categories of operation offloads are configurable.

- MSDU-level receive operations
- Frame forwarding (forwarding decision and actuation)
- Protocol/Task offloads

The following is a list of RX operations and offloads.

Function	Description	Ownership	Notes
Decryption	Decrypt the frame contents using the security type and security key specified for the sender.	Target	In host-implemented FIPS mode, the decryption is done within the host software. The target's decryption is bypassed.
A-MPDU deaggregation	Decompose an RX A-MPDU into individual MPDUs.	Target	
A-MSDU deaggregation	Decompose an RX A-MSDU into individual MSDUs.	Target/TAL	Each RX MSDU is placed into a separate buffer.
MSDU Security decap and de-MIC	For security types that involve an MSDU-level MIC, verify the received MIC. Decapsulate the security header and footer.	Target/TAL	The operating system performs countermeasures if needed.
Rx decap	Replace non-initial A-MSDU subframe headers with 802.11 headers, using the 802.11 header fields from the initial A-MSDU subframe as appropriate.	Target/TAL	During A-MSDU deaggregation, the non-initial MSDUs of the A-MSDU need their 802.3 header replaced by a generic 802.11 header. WDI always expects 802.11 headers.

Function	Description	Ownership	Notes
Rx reordering logic	For each RX MPDU, determine which slot of the Rx reordering array it goes to. Determine when a series of in-order frames is present. Determine when to release pending frames even if their preceding frames have not arrived.	Target/TAL	
Rx discard logic	Determine which Rx frames need to be discarded: 1. If it does not match any of the receive packet filters. 2. If the frame is encrypted, discard if: • A cipher key is not available to decrypt the packet. • The packet payload fails to decrypt successfully. • The packet payload fails the MIC verification. • The packet fails the replay mechanism defined for the cipher algorithm (see Rx PN/replay check). • A privacy exemption is defined for the packet's ether type that specifies an **WDI_EXEMPT_ ALWAYS** action. 3. If the frame is unencrypted, discard if: • A cipher key is available to decrypt the packet and a privacy exemption is defined for the packet's Ethertype that specifies a **WDI_EXEMPT_ON_ KEY_UNAVAILABLE** action. • The dot11ExcludeUnencrypted MIB is set to true.	Target/TAL makes all discard decisions.	

Function	Description	Ownership	Notes
Rx PN/replay check	Confirm that each MPDU has a distinct Packet Number that is larger than prior PNs.	This is a mandatory and always enabled offload except for streams associated with a reorder queue and reordering queue management is not fully offloaded to the target.	
Chatter offload	Avoid interrupting the host for each deferrable "noise" Rx frame. Instead, group the Rx noise frames and use a single interrupt to deliver all such frames.	Target	
Defragmentation	Reassemble 802.11 fragments into their original MSDU.	Target/TAL	
Rx reorder queuing	Store out-of-order Rx MPDUs until the missing prior MPDUs from the flow arrive.	Target/TAL	
Rx discard actuation	Discard Rx frames based on the specifications flagged by the Rx discard logic run at the target.	Target/TAL	
Higher-level protocol (task) offloads	Checksum	Checksum: Configurable offload at boot-up if required.	Checksum: The target passes its checksum offload capabilities as part of device caps to WDI during bring-up. For information about capabilities, see NDIS_TCP_IP_CHECKSUM_OFFLOAD.

Receive operations in Host-Implemented FIPS mode

In this mode, the target may indicate the received frame with either an 802.11 header or an 802.3 header. The frame must not be decrypted before indication.

If discard logic is offloaded to the target, it must mark received frames for discard if they meet any of the following criteria.

- Frames that have a bad CRC.
- Duplicate frames.
- Frames that do not match the configured packet filters.

The target must increment the appropriate MAC and PHY statistics for packets that are either received successfully or discarded by the port.

In addition, the target must perform discard actuation if offloaded.

The target should not strip the QoS flag from the 802.11 header on the RX path when operating in Host-implemented FIPS mode. The target should indicate the frame without modifying the QoS header.

For the case of fragmented packets, the payload type reported by the LE for FIPS mode is always **WDI_FRAME_MSDU_FRAGMENT** as the host is doing the defragmentation process. However, in non-FIPS mode, the reported payload type should be **WDI_FRAME_MSDU** as the Target/TAL is performing the defragmentation.

Related topics

NDIS_TCP_IP_CHECKSUM_OFFLOAD

WDI data transfer

WDI_EXEMPTION_ACTION_TYPE

WDI_FRAME_PAYLOAD_TYPE

WDI transmit operations and offloads

Article • 03/14/2023

WDI operates in one of two Tx modes: Port queuing and PeerTID queuing. The target sets the mode with the TargetPriorityQueueing capability (true = WDI Port queuing, false = WDI PeerTID queuing).

The host performs the following operations.

- TX classification (only when **TargetPriorityQueueing** = false)
- TX queuing (either at a Port level or a PeerTID level)
- Transfer scheduling (scheduling frame download to the target)
- Host-Target TX flow control

The following is a list of TX operations and offloads.

Processing step	Description	Owner/Applicable offloads	Notes
High-level protocol (task) offloads	Checksum, LSO.	Checksum is a configurable offload at boot-up. Each frame has flags to specify the applicable checksum operations. WDI handles LSO segmentation transparently from the TAL/Target if applicable.	Checksum: The target passes to WDI its checksum offload capabilities as part of device caps during bringup. For capability information, see NDIS_TCP_IP_CHECKSUM_OFFLOAD. WDI handles LSO segmentation transparently from the TAL/Target if applicable.
TX encap	Update/replace the generic 802.11 header with the appropriate type of 802.11 frame header.	Target/TAL	The first contiguous buffer of the frame has space available at beginning (before the MAC header). This space is determined by the BackfillSize specified in the device parameters. For Non-EAPOL packets, the first buffer contains the MAC and LLC/SNAP headers, but not the payload. The first contiguous buffer for an EAPOL packet may contain part (or all) of the payload.

Processing step	Description	Owner/Applicable offloads	Notes
TX classification	Determine the TID. Determine the recipient based on unicast RA or DA.	WDI performs classification when **TargetPriorityQueueing** is false. WDI does not perform classification if **TargetPriorityQueueing** is true.	
TX queue	Store TX frames in separate queues.	WDI queues TX frames if needed.	WDI queues TX frames by port (**TargetPriorityQueueing** = true) or by recipient and traffic type (PeerId,TID) (**TargetPriorityQueueing** = false).
TX flow control	Prevent the overrunning of the TIL or target buffers with TX frames.	WDI/TIL/Target	See the section on Host-Target flow control.
Transfer scheduling	Select the TX queue from which to transfer frames to the TAL/TIL when there are multiple backlogged queues.	WDI if it needs to queue TX frames.	
A-MSDU aggregation	Determine which frames to group into an A-MSDU aggregate which must be maintained during retransmissions.	TAL/Target	
Fragmentation		TAL/Target	

Processing step	Description	Owner/Applicable offloads	Notes
WLAN scheduling	Determine which recipient to transmit to next, which traffic type to send, and how many frames to send.	Target	
Encryption	Encrypt the frame contents using the security type and security key specified for the recipient (or sender, for multicast frames). Add security encapsulation where applicable.	Target	For systems supporting FIPS, the encryption is done within the host software. The target's encryption is bypassed.
A-MPDU aggregation	Determine which frames to group into an A-MPDU aggregate, and which can be modified during a retransmission.	Target	
Retry	Retransmit MPDUs that are nacked or not acked by the recipient.	Target	

Operation in Host-Implemented FIPS mode

If the host provides FIPS for a given connection (host FIPS mode is set to true in WDI_TLV_CONNECTION_SETTINGS), the host encrypts the packets before they are submitted to the target. The target transmits the packets without additional changes that affect the data integrity of the packets. For example, the target must not perform transmit MSDU aggregation in this mode.

In the more common case where host-FIPS mode is not enabled (target-implemented encryption mode), the header 802.11 header is followed by the unencrypted payload data. If the packet requires encryption before transmission, the target encrypts the packet. It also performs QoS prioritization of packets, and may perform TCP layer offload operations (such as checksum or large send). For this send processing, the target may need to add additional headers into the packet (for example, QoS, HT headers, or IV).

TX Encapsulation: Population of 802.11 TX Frame Headers

The network data is submitted in 802.11 packet format to the port (target device). Each transmitted frame starts with a 802.11 MAC header. The host sets some of the fields of the MAC header, while the target sets other fields. The table below describes which fields of the 802.11 MAC header and cipher headers are populated by the host, and which should be populated by the target device.

Field name	**Subfield name**	**Target-implemented encryption mode**		**Host-implemented FIPS mode**	
		Set by host	**Set by target**	**Set by host**	**Set by target**
Frame Control	Protocol Version	X		X	
Frame Control	Type	X		X	
Frame Control	Subtype	X		X	
Frame Control	To DS	X		X	
Frame Control	From DS	X		X	
Frame Control	More Fragments	X		X	
Frame Control	Retry		X		X

Frame Control	Pwr Mgmt		X		X
Frame Control	More Data		X		X
Frame Control	Protected Frame		X	X	
Frame Control	Order	X		X	
Duration/Id			X		X
Address 1		X		X	
Address 2		X		X	
Address 3		X		X	
Sequence Control	Fragment Number	X			X
Sequence Control	Sequence Number		X		X
Address 4		X		X	
QoS Control			Added/populated by target.		Added/populated by target in the case of 11n QoS association.
HT Control			Added/populated by target.		Added/populated by target.

Related topics

NDIS_TCP_IP_CHECKSUM_OFFLOAD

WDI data transfer

WDI_TLV_CONNECTION_SETTINGS

WDI_TXRX_CAPABILITIES

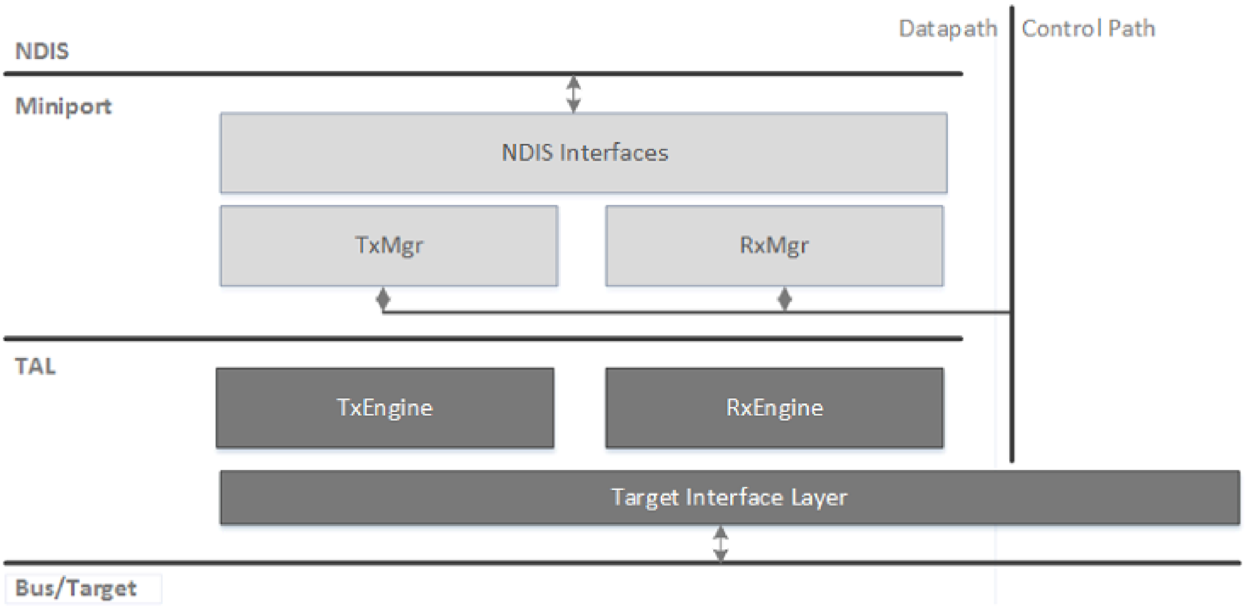
WDI datapath architecture

Article • 03/14/2023

To interoperate with existing WLAN target devices, the current version of the driver interface does not specify a host-controller interface (HCI) for TX/RX. Instead, it specifies a request/indication software interface between WDI and the Target Adaptation Layer (TAL).

The WDI component implements the NDIS datapath interfaces and the target-agnostic TX/RX functions performed by the host. It also maintains the state of individual virtual WLAN devices (ports) and peer-specific state.

The TAL provides the TX/RX WLAN functions for which the implementation depends on the host-controller interface, as well as the controller and bus interface functions.



In addition to the TX/RX function, the TAL provides a Target Interface Layer (TIL) that is used by the control and data paths. The responsibilities of the TIL are listed in the following table.

TIL function	Description
Management of Host-Target communication	Example: Allocate and manage the DMA channels required for control and data paths.
Bus adaptation/abstraction	Provide a standard host/target communication API that abstracts the communication differences between different bus types, software/hardware bus endpoints, and bus DMA engines.

WDI general datapath interfaces

Article • 03/14/2023

802.11 frame handling and frame metadata

802.11 frames are passed between WDI and the TAL in the form of **NET_BUFFER_LIST** (NBL) chains. Each NBL represents one MSDU. Through macros, the NBL structure provides operations on the data buffers and access to metadata, including operating system populated Wi-Fi TX metadata. The structure is extensible through its context and **MiniportReserved** members. **MiniportReserved[0]** points to a buffer of type **WDI_FRAME_METADATA**. This buffer is allocated by WDI in the TX path, and by the TAL in the RX path via the callback *NdisWdiAllocateWiFiFrameMetaData*. The TAL uses MiniportReserved[1] to store any additional frame metadata.

Datapath management requests and indications

For datapath management request and indication function reference, see WDI Datapath Management Functions.

Related topics

NDIS_WDI_DATA_API

NET_BUFFER_LIST

NdisWdiAllocateWiFiFrameMetaData

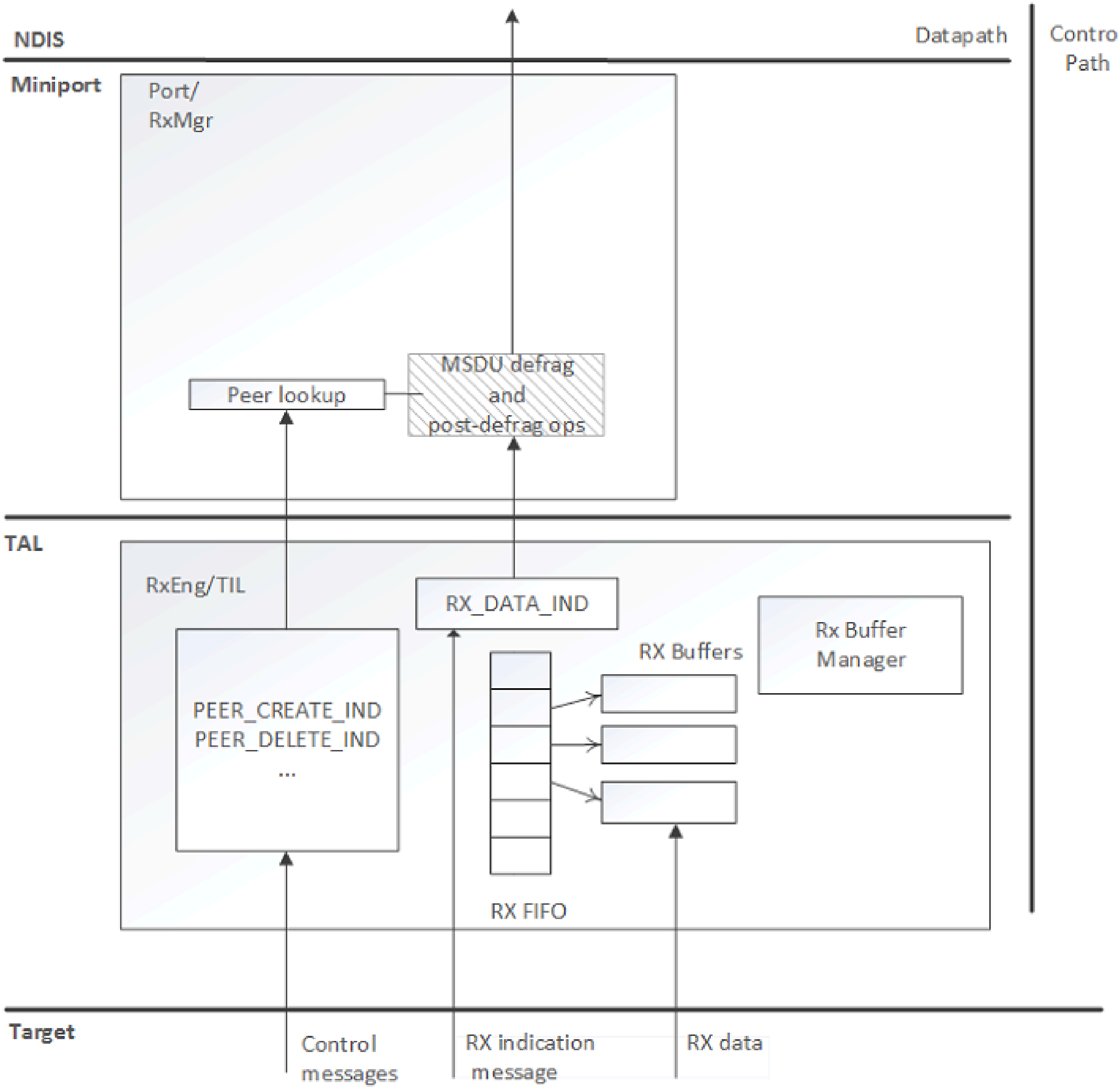
WDI_FRAME_METADATA

WDI RX path

Article • 03/14/2023

RX path components

The following diagram shows the RX path components.



RX Manager (RxMgr)

The RX Manager performs receive processing steps that are not offloaded to the target or performed by the RxEngine.

RX function	Description

RX function	Description
MSDU discard	Discard MSDUs with errors.
Queuing and throttling	Manage the DPC watchdog to prevent a bugcheck from too many indications per DPC, and too long at dispatch level. Provide backpressure to the RxEngine when appropriate to help with throttling.

RxEngine

The RxEngine sends and receives data-synchronous messages to and from the target, interprets RX descriptor formats, and manages buffers for direct hardware to software RX DMAs.

RX function	Description
Host-to-Target message construction	Construct host-to-target data path-related messages.
Target-to-Host message parsing	Parse and process target-to-host data-synchronous messages such as *NdisWdiRxInorderDataIndication*.
Interpretation of target RX descriptors	Provide an interface (functions) for querying RX frame attributes from the target-specific descriptor.
RX FIFO management	Provide a target-accessible FIFO for posting empty RX buffers for the target to fill. Remove buffers from the FIFO during *NdisWdiRxInorderDataIndication* processing, and provide replacement empty buffers.
RX buffer pool management	Maintain a pool of buffers for DMA transfer of receive frames.
MPDU discard	Discard MPDUs with errors. The target indicates the receive frames marked for discard - for example, due to FCS errors or ARQ duplication errors. This is only done if it is not implemented by the target.
MPDU reorder	Store MPDUs in order within an RX reordering array until the missing preceding MPDUs arrive. This is only done if it is not implemented by the target.

RX function	Description
MPDU PN chk	This is only done if it is not offloaded to the target.
MSDU Fragment Reassembly	This is only done if it is not offloaded to the target.

RX path requests and indications

For RX path request and indication function reference, see WDI RX Path Functions.

Related topics

NdisWdiRxInorderDataIndication

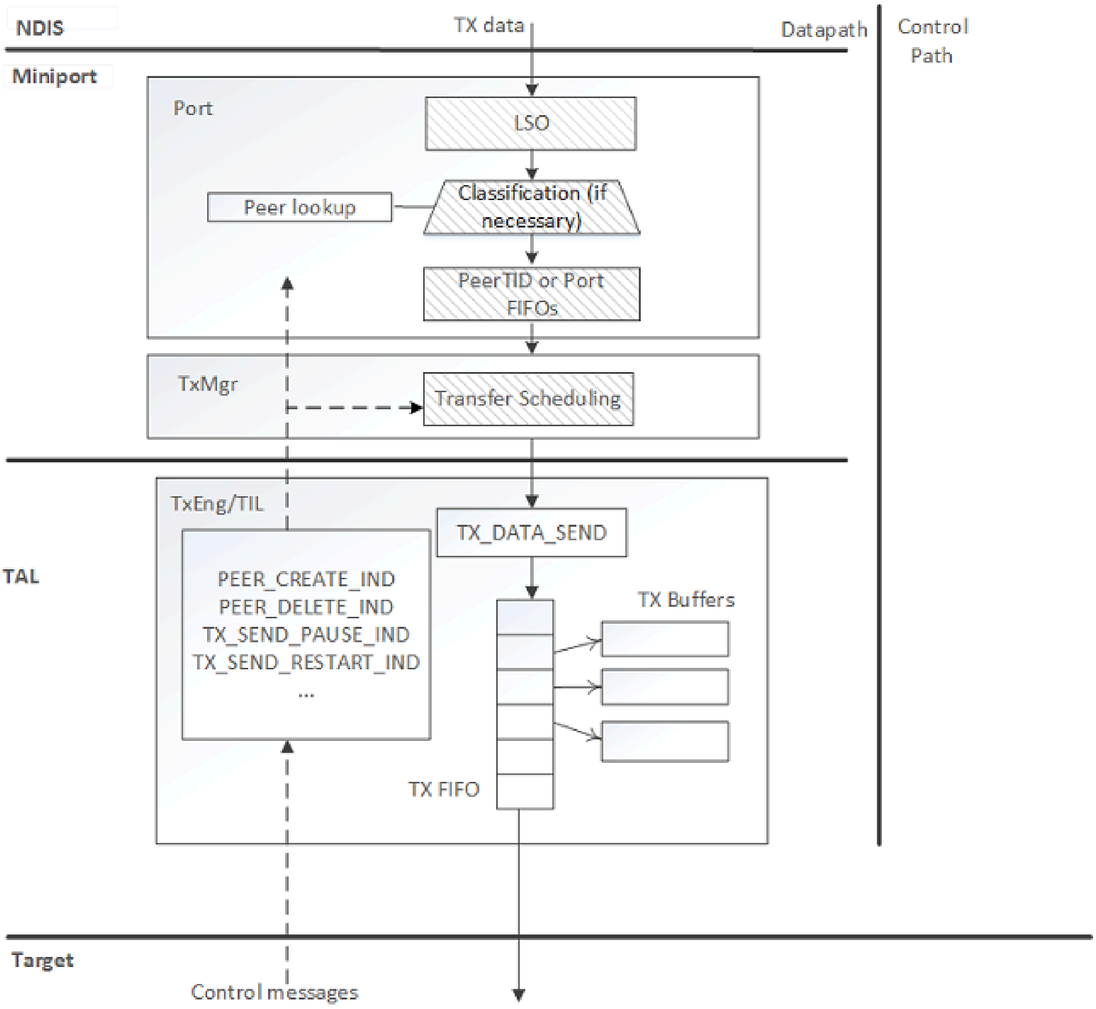
WDI RX Path Functions

WDI TX path

Article • 03/14/2023

TX path components

The following diagram shows the TX path components.



TX descriptors

The TAL uses a Target TX Descriptor (TTD) to inform the target of the size and location of the frame.

Different target WLAN devices may have different definitions of the TTD. Due to this, the TTD programming is done within the TAL, based on information provided by WDI. To program a TTD, WDI specifies a NET_BUFFER_LIST (NBL), through which the frame metadata, such as frame ID, extended TID, applicable task offloads, and encryption exemption action, is accessible.

The TAL transfers the TTD and the TX frame to the target. From the metadata in the TTD and fields within the frame's header, the target can determine the intended recipient of the transmit frame and how to transmit it.

Eventually, the target transmits the frame, and notifies the host when the transfer (and possibly transmission) is done. The target uses a TX completion message that specifies whether the transmission was successful, and the IDs of the frames whose transmission was completed.

Basic operation

Transmitting a data frame involves the following steps within the WLAN host TX software.

1. WDI obtains an NBL from NDIS and performs TX classification (if WDI is operating in PeerTID queuing mode).
2. The NBL is linked to a TTD obtained by querying the TAL. For efficiency, the TAL may preallocate TTDs from a lookaside list.
3. The TxMgr queues the transmit frame based on the PeerTID or Port, depending on the **TargetPriorityQueueing** mode.
4. The TxMgr provides the NBL and the attached TTD to the TxEngine, which in turn passes it to the TIL for transfer to the target. The TxEngine/TIL does not queue frames (for example, prior to making them available for DMA).
5. The TxEngine indicates the updated TX status of frames owned by the TxEngine/target using transfer completion (and transmit completion indication if applicable).
6. When a frame is both Transfer Complete (and if required, TX Complete), the TxMgr looks up the NBL using the frame ID, returns the TTD to the TxEngine's pool, and send-completes the frame to NDIS.

Host - target TX flow control

TX Flow control is necessary to avoid overwhelming the TIL and target resources.

The target-credit scheme and the pause/resume mechanism

The TxMgr queues and transfers TX frames to the target according to a credit-based scheme. The target provides the TX Engine with credit-update indications that specify the resources available for additional frames on the target. The number of credits used up by each frame on the target is determined at the time of TTD programming. The number of frames passed to the TxEngine as part of a send operation from a given queue is limited by the available credits and the cost of the frames at the head of the line in FIFO order.

To the TxMgr, credits have an abstract unit. The Target/TxEngine should use whatever definition of credit is most useful to the specific implementation.

The TAL uses pause/resume indications to stop/resume the flow of TX traffic from a given port, or destined to a particular receiver with a given TID. If the TxEngine gets a send request while the available credit is less than the maximum frame cost, the TxEngine pauses the traffic from the TxMgr (across all ports) until the next credit update from the target.

When WDI is in port queuing mode (**TargetPriorityQueueing** equals TRUE), pause/resume indications are only allowed/defined at a port or adapter level due to the absence of Peer,TID classification, and queuing.

Limiting the maximum frame count for send operations

To avoid the need for temporary queues in the TIL (for example, DMA rate matching queue), the number of frames that TxMgr passes to TxEngine in a send operation is limited by a maximum count specified by the TxEngine. This limit may be specific to the queue the TxMgr is attempting to send from and changes over time as more space is available in the TIL.

Host - target TX transfer scheduling

The TxMgr uses a single TX thread to submit frames to the TxEngine. There is a TX thread actively submitting frames to the TxEngine as long as there are backlogged queues.

The TxMgr schedules queues in the following manner depending on the queuing mode.

For WDI port queuing (**TargetPriorityQueueing** equals TRUE), the TxMgr services queues using Deficit Round Robin (DRR) across all backlogged port queues.

For WDI PeerTID queuing (**TargetPriorityQueueing** equals FALSE), the TxMgr services queues according to AC priority without starving any queues, and ensures that any bottlenecked resources in TIL and target are shared among RA-TID streams in a fair manner. It prevents slow streams from consuming a disproportionate share of such resources.

In general, the scheduler uses DRR to choose the Peer-TID queue to transmit from at any given time. For each queue, DRR associates a quantum parameter that limits the number of octets to send from the queue in each round. The TxEngine updates this parameter in each send operation involving the queue to match the expected size of one or two transmission opportunities.

In general, the DRR scheduler services only the RA-TID queues associated with the backlogged AC of highest priority. To prevent starvation, the scheduler periodically performs DRR across all backlogged queues.

Priority mapping for IHV reserved extended TIDs

Frames injected by the IHV with extended TID in the IHV reserved range map to the following extended ACs for the purposes of priority scheduling. The table is in order of increasing priority.

Extended TID	Extended AC
17	AC_BK
18	AC_BE
19	AC_VI
20	AC_VO
21	AC_PR0
22	AC_PR1
23	AC_PR2
24	AC_PR3

For WDI port queuing, all injected frames are treated equally regardless of the extended TID.

TxMgr-TxEngine interface

Requests to TxEngine

- *MINIPORT_WDI_TX_ABORT*
- *MINIPORT_WDI_TX_DATA_SEND*
- *MINIPORT_WDI_TX_TAL_QUEUE_IN_ORDER*

- *MINIPORT_WDI_TX_TAL_SEND*
- *MINIPORT_WDI_TX_TAL_SEND_COMPLETE*
- *MINIPORT_WDI_TX_TARGET_DESC_DEINIT*
- *MINIPORT_WDI_TX_TARGET_DESC_INIT*

Indications from TxEngine

- *NDIS_WDI_TX_DEQUEUE_IND*
- *NDIS_WDI_TX_TRANSFER_COMPLETE_IND*
- *NDIS_WDI_TX_SEND_COMPLETE_IND*
- *NDIS_WDI_TX_QUERY_RA_TID_STATE*

TX specific control requests

- *MINIPORT_WDI_TX_PEER_BACKLOG*

TX specific control indications

- *NDIS_WDI_TX_SEND_PAUSE_IND*
- *NDIS_WDI_TX_SEND_RESTART_IND*
- *NDIS_WDI_TX_RELEASE_FRAMES_IND*
- *NDIS_WDI_TX_INJECT_FRAME_IND*

Related topics

WDI TX Path Functions

NET_BUFFER_LIST

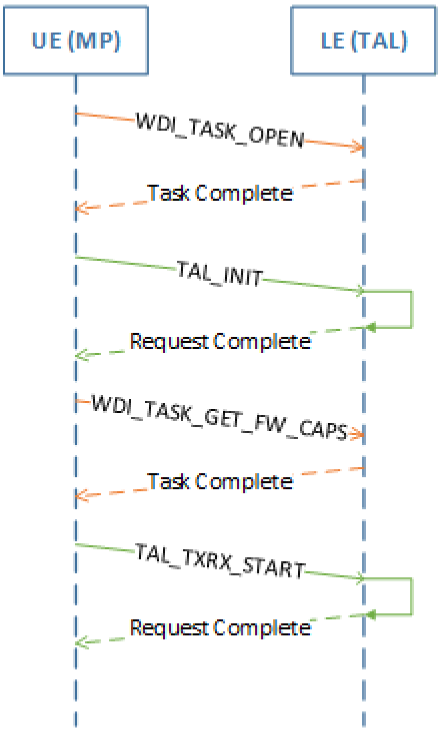
WDI_TXRX_CAPABILITIES

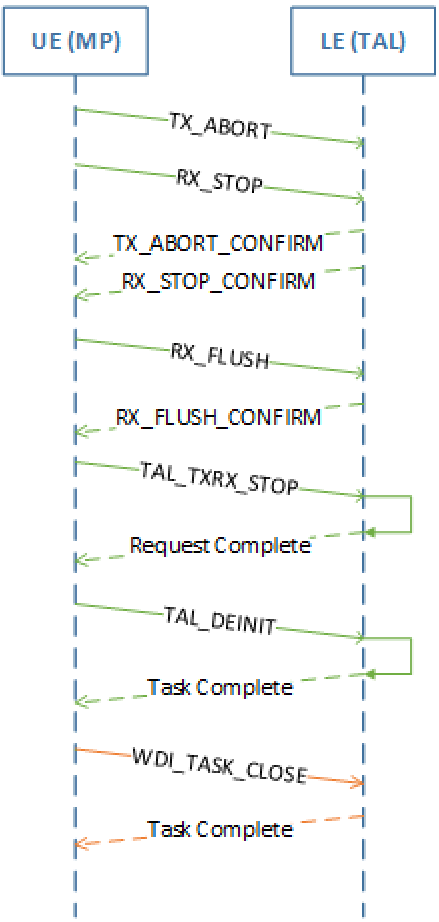
WDI datapath operation sequence diagrams

Article • 03/14/2023

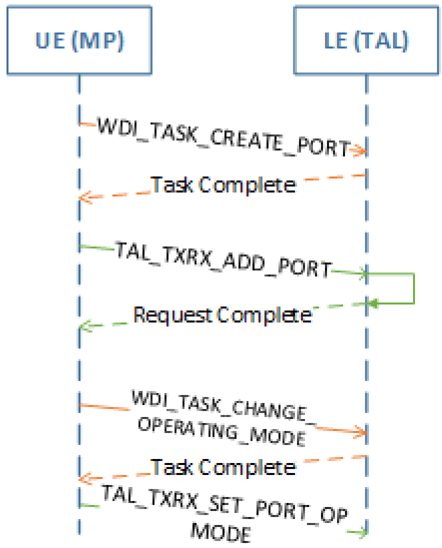
Control path commands are shown in orange.

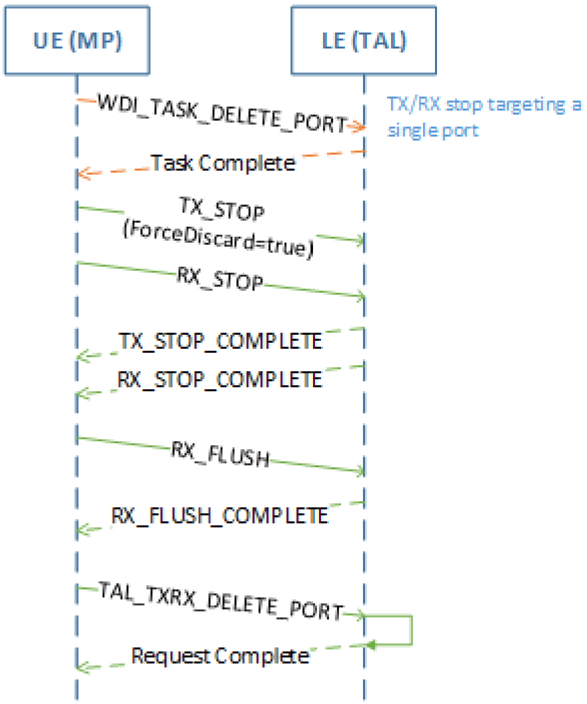
Datapath initialization and deinitialization



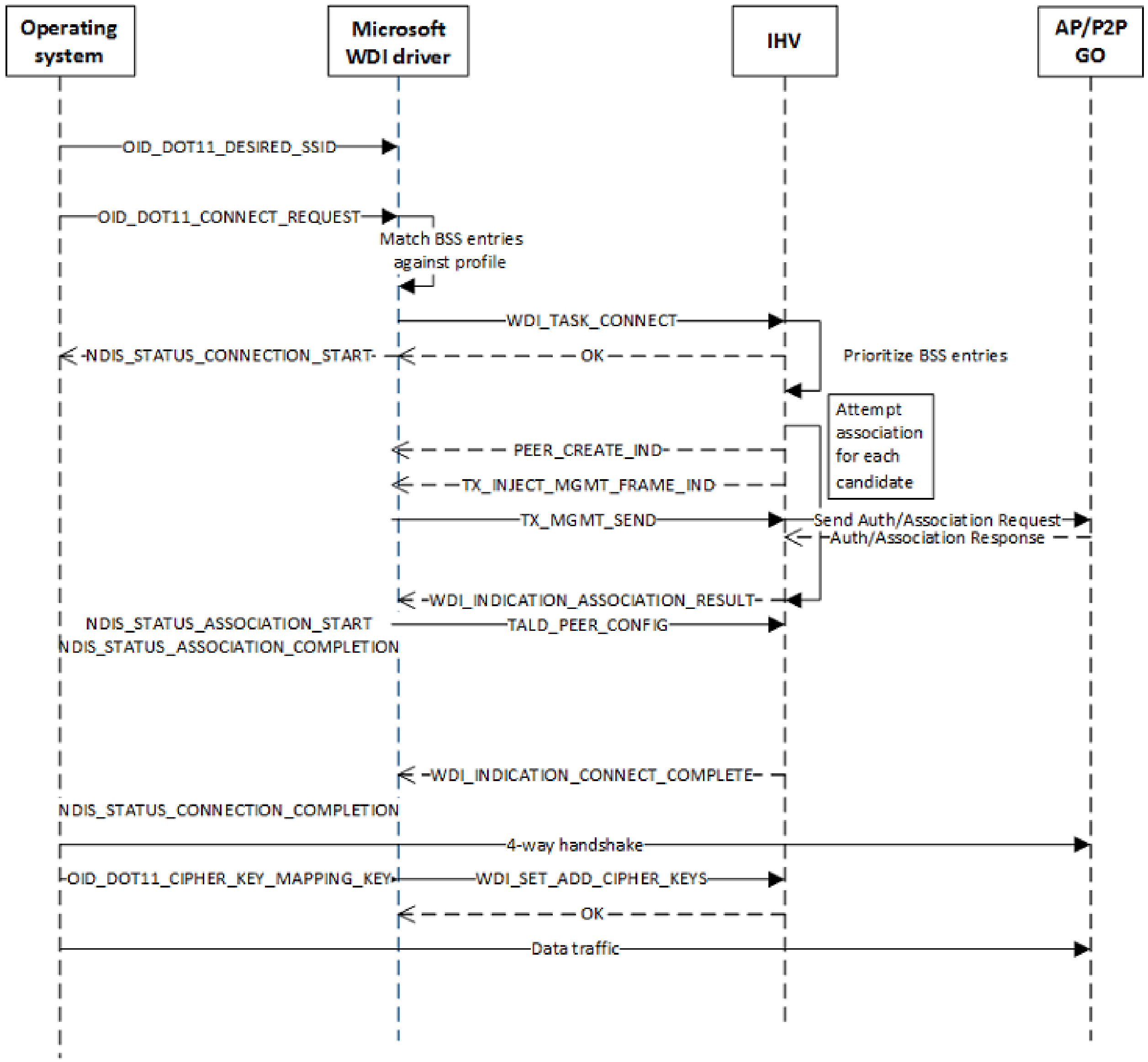


Datapath operations for port creation/deletion

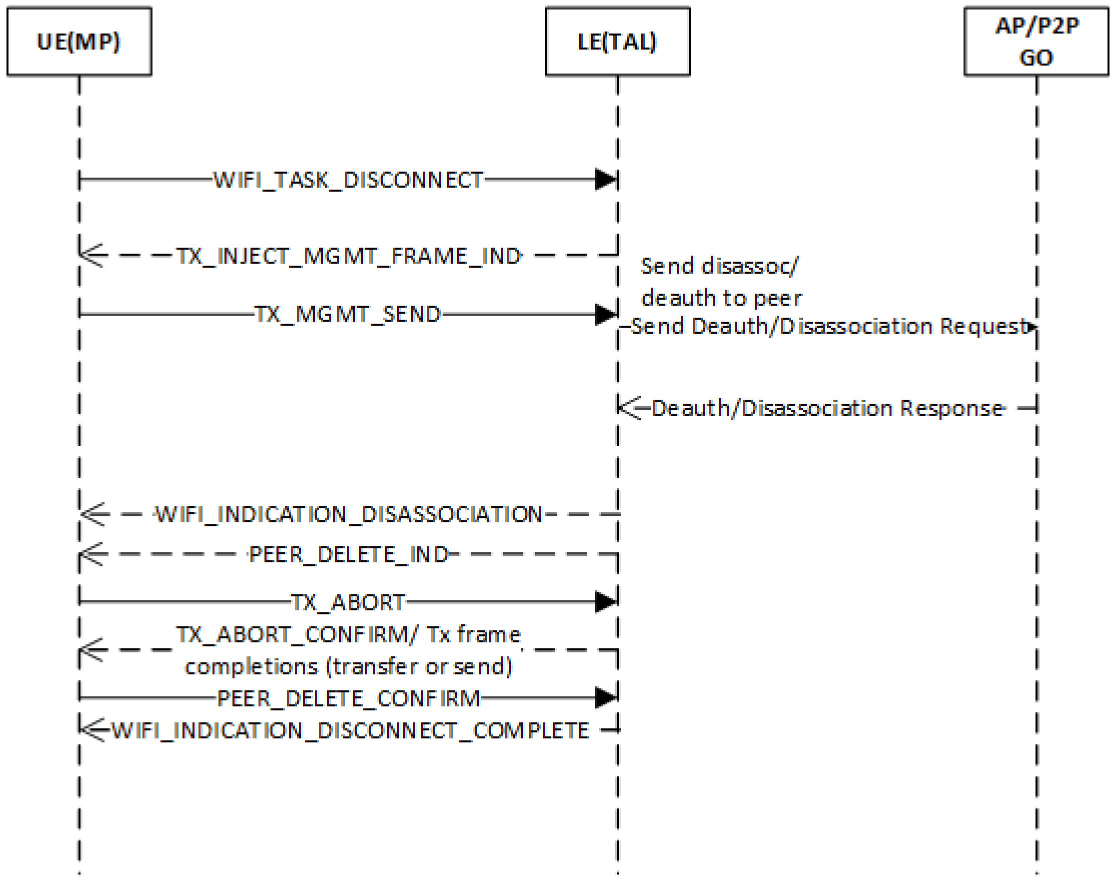




STA/Wi-Fi Direct client connect


Operating system
Microsoft WDI driver
IHV
AP/P2P GO
OID_DOT11_DESIRED_SSID
OID_DOT11_CONNECT_REQUEST
Match BSS entries against profile
WDI_TASK_CONNECT
NDIS_STATUS_CONNECTION_START
OK
Prioritize BSS entries
Attempt association for each candidate
PEER_CREATE_IND
TX_INJECT_MGMT_FRAME_IND
TX_MGMT_SEND
Send Auth/Association Request
Auth/Association Response
WDI_INDICATION_ASSOCIATION_RESULT
NDIS_STATUS_ASSOCIATION_START
TALD_PEER_CONFIG
NDIS_STATUS_ASSOCIATION_COMPLETION
WDI_INDICATION_CONNECT_COMPLETE
NDIS_STATUS_CONNECTION_COMPLETION
4-way handshake
OID_DOT11_CIPHER_KEY_MAPPING_KEY
WDI_SET_ADD_CIPHER_KEYS
OK
Data traffic

STA/Wi-Fi Direct client disconnect


UE(MP)
LE(TAL)
AP/P2P GO
WIFI_TASK_DISCONNECT
TX_INJECT_MGMT_FRAME_IND
Send disassoc/ deauth to peer
TX_MGMT_SEND
Send Deauth/Disassociation Request
Deauth/Disassociation Response
WIFI_INDICATION_DISASSOCIATION
PEER_DELETE_IND
TX_ABORT
TX_ABORT_CONFIRM/ Tx frame completions (transfer or send)
PEER_DELETE_CONFIRM
WIFI_INDICATION_DISCONNECT_COMPLETE

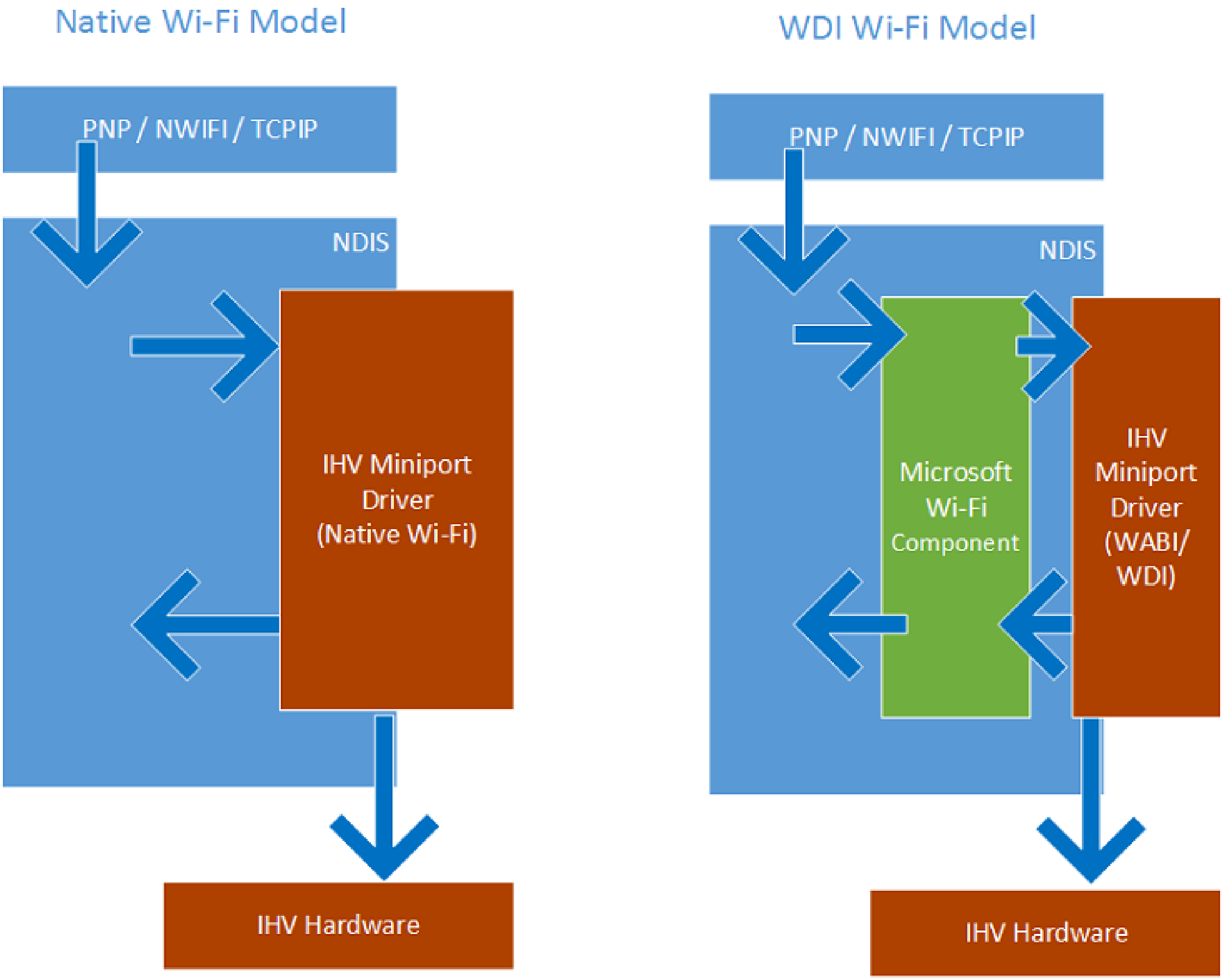
WDI IHV component model

Article • 03/14/2023

This section provides an overview of the NDIS interfaces for the WDI miniport driver and the expectations for those interfaces.

The IHV component in the WDI model is an NDIS miniport. It interfaces with the operating system and its networking stack using existing and new NDIS APIs. A Microsoft WLAN component sits between the WDI IHV miniport driver and the rest of the operating system. It provides a mapping between the WDI interfaces and the existing NDIS/Native WLAN interfaces. WDI commands are packaged as new NDIS OIDs and WDI indications are packaged as new NDIS indications. The data path interacts via NET_BUFFER_LIST structures using new handlers.

The figure below shows the overall architecture layout and a sample flow of messages (PNP actions, OIDs, and sends) from the operating system to the IHV miniport driver for both the old Native WLAN model and the new WDI WLAN model.



Besides assisting with the Native Wi-Fi interface requirements, the Microsoft WLAN component also handles most of the common NDIS requirements. For example, it handles the *MiniportPause* requirements from NDIS and converts them to WDI data and control path messages to ensure that NDIS requirements are met. However, it also provides the IHV miniport driver the ability to do additional work. The driver can register to be notified on *MiniportPause* call to do any additional cleanup it wants to do during *MiniportPause*.

WDI IHV driver interfaces

Article • 03/14/2023

The WDI IHV miniport is like any other NDIS miniport driver and it would follow the development practices and documentation for any NDIS miniport. A Native WLAN Miniport driver's responsibilities for the NDIS handlers are split between the MS Component and the WDI IHV driver. The Microsoft WLAN component takes care of the NDIS requirements that are applicable for all Wi-Fi miniports so that every IHV does not have to redo all that work. The mapping of and behavior changes for the NDIS handlers for the Native WLAN IHV miniport when applied to a WDI IHV miniport are described below.

- Driver installation
- DriverEntry
- MiniportSetOptions
- MiniportInitializeEx
- MiniportHaltEx
- MiniportDriverUnload
- MiniportPause
- MiniportRestart
- MiniportResetEx
- MiniportDevicePnPEventNotify
- MiniportShutdownEx
- MiniportOidRequest
- MiniportCancelOidRequest
- NdisMIndicateStatusEx
- MiniportDirectOidRequest
- MiniportCancelDirectOidRequest
- MiniportSendNetBufferLists
- MiniportCancelSend
- MiniportReturnNetBufferLists
- WDI handler: MiniportWdiOpenAdapter
- WDI handler: MiniportWdiCloseAdapter

Driver installation

There are no changes to the way the WDI IHV miniport driver is loaded and installed on the system. The INF and install process is similar to that of an IHV Native WLAN miniport driver. Like existing NDIS drivers, when the IHV driver needs to be loaded to work with the IHV's WLAN adapter, the operating system calls the IHV miniport driver's DriverEntry routine.

DriverEntry

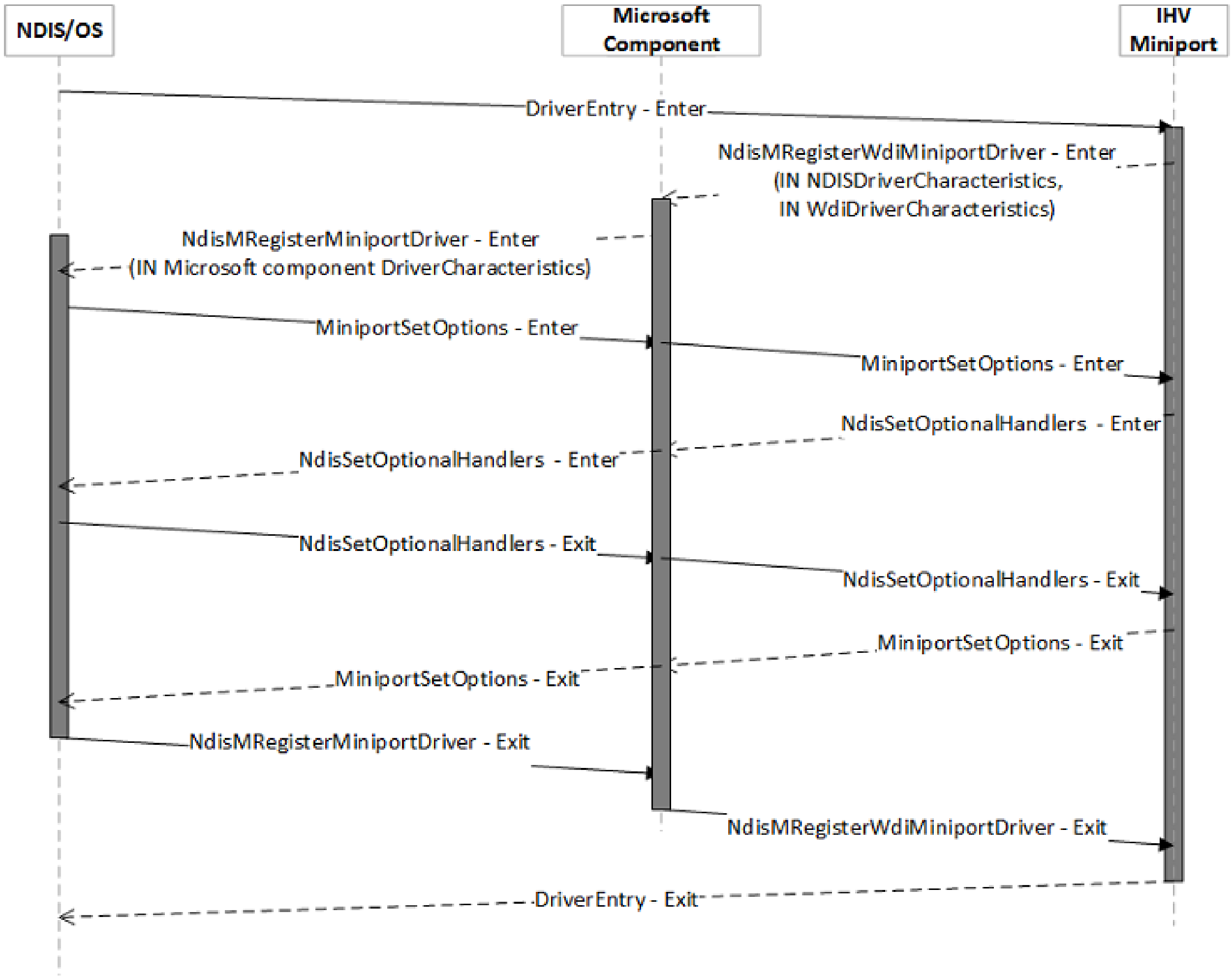
The operating system directly calls the WDI IHV miniport driver's DriverEntry routine. The IHV miniport follows most of the guidelines of a regular NDIS miniport's DriverEntry routine. The one exception is that instead of calling NdisMRegisterMiniportDriver, the IHV miniport calls NdisMRegisterWdiMiniportDriver to tell the operating system to enable the Microsoft WLAN component.

The following are the key parameters of NdisMRegisterWdiMiniportDriver.

- NDIS_MINIPORT_DRIVER_CHARACTERISTICS: This is the original NDIS structure that a Native Wi-Fi miniport uses to register with NDIS. For a WDI model, most of the handler parameters are optional. The only required handlers are **MINIPORT_OID_REQUEST_HANDLER** and **MINIPORT_DRIVER_UNLOAD**. **MINIPORT_OID_REQUEST_HANDLER** is used to pass WDI messages to the IHV driver. If any other handler is specified, the Microsoft WLAN component generally calls the handler after it has performed its own processing for the handler.
- NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS: This is the new set of handlers that a WDI miniport driver must implement. It is used by the IHV driver to register additional handlers for the control path, and the full set of handlers for the data path.

When the IHV miniport calls NdisMRegisterWdiMiniportDriver, the Microsoft WLAN component updates the handlers of NDIS_MINIPORT_DRIVER_CHARACTERISTICS and call NDIS's NdisMRegisterMiniportDriver. The updates are done so that the Microsoft WLAN component can intercept the handlers for which it can provide assistance/simplification to the WDI IHV miniport driver.

Below is the typical flow of the DriverEntry process for the WDI IHV miniport driver


NDIS/OS
Microsoft Component
IHV Miniport
DriverEntry - Enter
NdisMRegisterWdiMiniportDriver - Enter
(IN NDISDriverCharacteristics,
IN WdiDriverCharacteristics)
NdisMRegisterMiniportDriver - Enter
(IN Microsoft component DriverCharacteristics)
MiniportSetOptions - Enter
MiniportSetOptions - Enter
NdisSetOptionalHandlers - Enter
NdisSetOptionalHandlers - Enter
NdisSetOptionalHandlers - Exit
NdisSetOptionalHandlers - Exit
MiniportSetOptions - Exit
MiniportSetOptions - Exit
NdisMRegisterMiniportDriver - Exit
NdisMRegisterWdiMiniportDriver - Exit
DriverEntry - Exit

For more information about DriverEntry, see DriverEntry of NDIS Miniport Drivers.

MiniportSetOptions

As shown in the above DriverEntry diagram, if the WDI IHV miniport has registered the *MiniportSetOptions* handler, the operating system calls that function in the context of the miniport driver calling NdisMRegisterWdiMiniportDriver.

If the IHV miniport driver registers any option handlers using NdisSetOptionalHandlers, those handlers may not be serialized through the WDI layer by the Microsoft component. Therefore, the IHV component is responsible for handling any synchronization requirements for those handlers.

MiniportInitializeEx

The WDI model splits the *MiniportInitializeEx* behavior into multiple WDI interface calls.

1. Call *MiniportWdiAllocateAdapter*.

   When the operating system finds an instance of the IHV hardware, this is the first call into the WDI IHV miniport driver. In this call, the WDI miniport performs the actions that are required to create a software representation (**MiniportAdapterContext**) of the device. It also determines information about the device to fill in the NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES structure. The actual initialization of the device and the Wi-Fi stack is done later when the Microsoft component sends WDI commands down to perform specific initializations.

Using data obtained from the WDI IHV miniport driver, the Microsoft component calls **NdisMSetMiniportAttributes** and sets the NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES on NDIS. Most fields of **NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES** are filled with defaults by the Microsoft component. The IHV driver must populate the **MiniportAdapterContext** and **InterfaceType** fields.

Once this call returns from the IHV miniport driver, it starts receiving WDI commands via its *MiniportOidRequest* handler. During this call, the Microsoft component may not be able to perform reset/recovery operations, so any activity performed here should be quick and reliable.

2. Call *MiniportWdiOpenAdapter*.

   After *MiniportWdiAllocateAdapter*, the Microsoft component calls *MiniportWdiOpenAdapter* to load the firmware and initialize the hardware.

3. Multiple WDI commands using *MiniportOidRequest*.

   After *MiniportWdiOpenAdapter*, the Microsoft component sends the following tasks/properties/calls to the IHV miniport.

   a. Call *MiniportWdiTalTxRxInitialize* to initialize the data path and exchange handlers.
   b. Call OID_WDI_GET_ADAPTER_CAPABILITIES to get the adapter's capabilities.
   c. Call OID_WDI_SET_ADAPTER_CONFIGURATION to configure the adapter.
   d. Call OID_WDI_TASK_SET_RADIO_STATE to set the initial radio state if it is not already in the expected state.
   e. Call *MiniportWdiTalTxRxStart* to set up the data path.
   f. Call OID_WDI_TASK_CREATE_PORT to create the initial port.

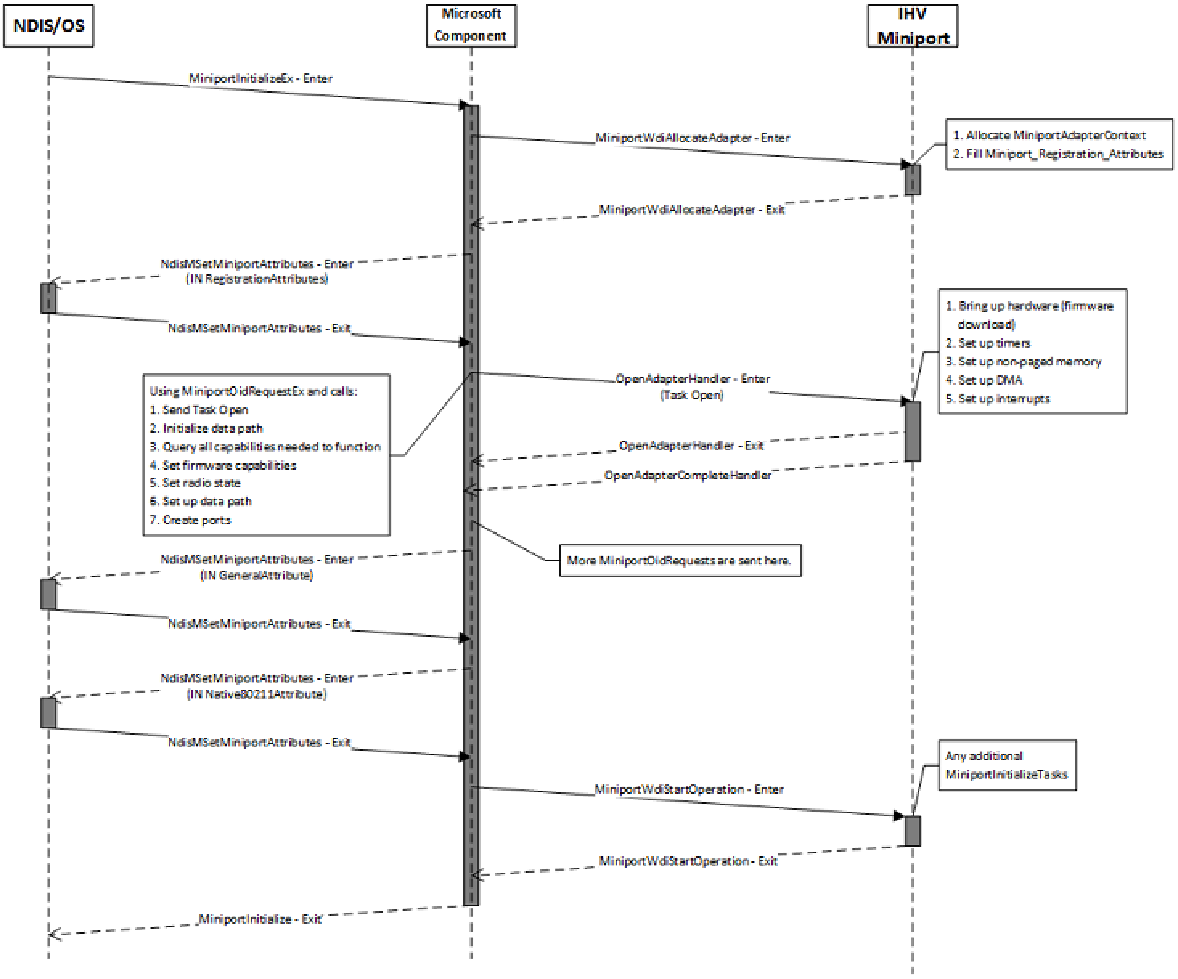
   Other commands may also be sent down to the IHV component as part of the MiniportInitializeEx processing of the Microsoft Component. However, until *MiniportWdiStartOperation* is called, the Microsoft component does not send any tasks down that need over-the-air communication. Except for OID_WDI_TASK_OPEN always being sent first, the order of the other commands/calls may change.

Using data obtained from the WDI IHV miniport driver, the Microsoft component calls NdisMSetMiniportAttributes and sets NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES and NDIS_MINIPORT_ADAPTER_NATIVE_802_11_ATTRIBUTES on NDIS.

4. Call *MiniportWdiStartOperation*.

   This is an optional WDI miniport handler inside NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS that the IHV driver can use to perform any additional MiniportInitializeEx tasks. It can also be used by the IHV miniport as a hint that the Microsoft component has finished initializing the miniport and the miniport can start any needed background activities.

   The diagram below shows the flow of MiniportInitializeEx.


NDIS/OS
Microsoft Component
IHV Miniport
MiniportInitializeEx - Enter
MiniportWdiAllocateAdapter - Enter
1. Allocate MiniportAdapterContext
2. Fill Miniport_Registration_Attributes
MiniportWdiAllocateAdapter - Exit
NdisMSetMiniportAttributes - Enter
(IN RegistrationAttributes)
NdisMSetMiniportAttributes - Exit
1. Bring up hardware (firmware download)
2. Set up timers
3. Set up non-paged memory
4. Set up DMA
5. Set up interrupts
Using MiniportOidRequestEx and calls:
1. Send Task Open
2. Initialize data path
3. Query all capabilities needed to function
4. Set firmware capabilities
5. Set radio state
6. Set up data path
7. Create ports
OpenAdapterHandler - Enter
(Task Open)
OpenAdapterHandler - Exit
OpenAdapterCompleteHandler
More MiniportOidRequests are sent here.
NdisMSetMiniportAttributes - Enter
(IN GeneralAttribute)
NdisMSetMiniportAttributes - Exit
NdisMSetMiniportAttributes - Enter
(IN Native80211Attribute)
NdisMSetMiniportAttributes - Exit
Any additional MiniportInitializeTasks
MiniportWdiStartOperation - Enter
MiniportWdiStartOperation - Exit
MiniportInitialize - Exit

If an intermediate operation fails, the Microsoft component undoes the previous operations and fails the miniport bring up. For example, if OID_WDI_TASK_CREATE_PORT fails, the data path is cleaned up, OID_WDI_TASK_CLOSE is sent, and the miniport fails.

MiniportHaltEx

In a Native Wi-Fi miniport, *MiniportHaltEx* is used to tell the miniport to stop operations and clean up the adapter instance. In the WDI model, the Microsoft component handles the original *MiniportHaltEx* call and splits it into multiple WDI interface calls.

1. Call *MiniportWdiStopOperation*.

   This is an optional WDI miniport handler inside **NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS** that the IHV driver can use to undo the operations it performed in *MiniportWdiStartOperation*.

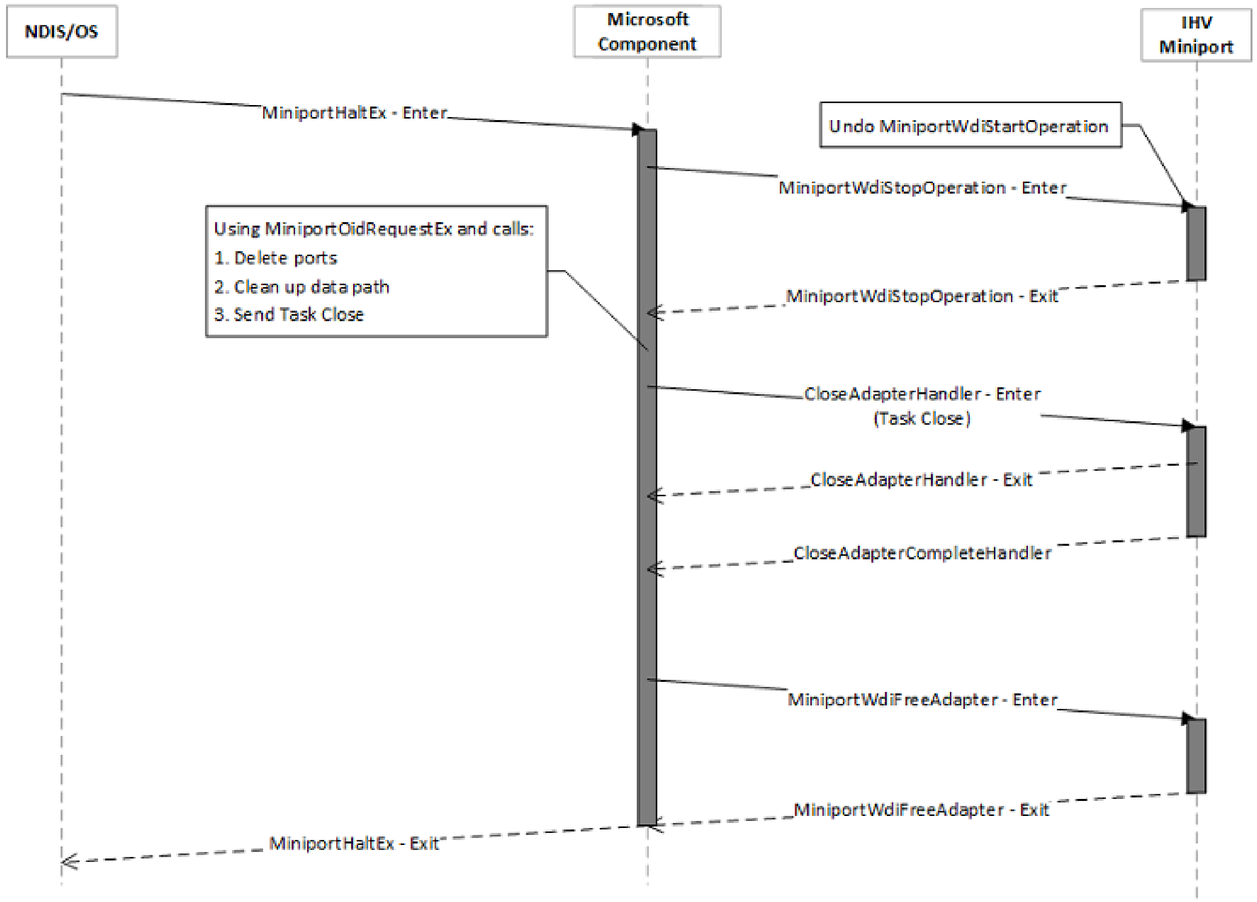
2. Multiple WDI Commands using *MiniportOidRequest*.

   After *MiniportWdiStopOperation*, the Microsoft component sends tasks/properties to the IHV miniport to clean up the current state of the IHV driver. This cleanup may include the following.

   a. Call OID_WDI_TASK_DISCONNECT/OID_WDI_TASK_STOP_AP to tear down any existing connections.
   b. Call OID_WDI_TASK_DELETE_PORT to delete all created ports.
   c. Call *MiniportWdiTalTxRxStop* to stop the data path.
   d. Call *MiniportWdiTalTxRxDeinitialize* to deinitialize the data path.
   e. Call to clean up the hardware state. This is sent to the IHV using the *MiniportWdiCloseAdapter* that has been registered by the IHV driver.

3. Once all of the above commands are called, the Microsoft component calls *MiniportWdiFreeAdapter* to have the IHV driver delete any software state it may have.

The diagram below shows the flow of MiniportHaltEx.

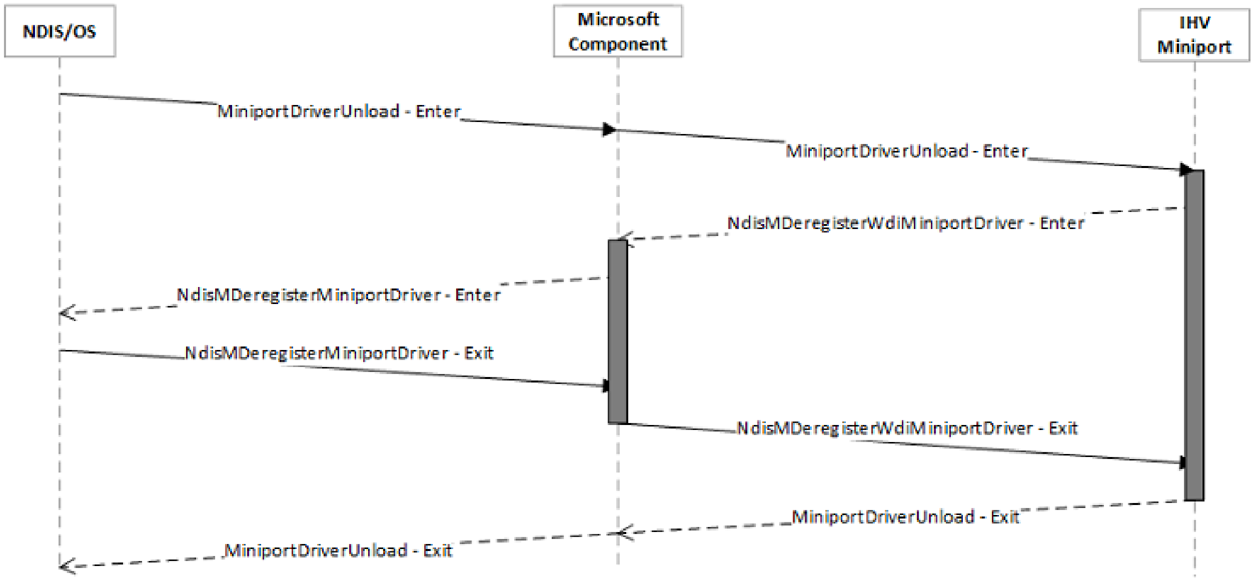

NDIS/OS
Microsoft Component
IHV Miniport
MiniportHaltEx - Enter
Undo MiniportWdiStartOperation
MiniportWdiStopOperation - Enter
Using MiniportOidRequestEx and calls:
1. Delete ports
2. Clean up data path
3. Send Task Close
MiniportWdiStopOperation - Exit
CloseAdapterHandler - Enter
(Task Close)
CloseAdapterHandler - Exit
CloseAdapterCompleteHandler
MiniportWdiFreeAdapter - Enter
MiniportWdiFreeAdapter - Exit
MiniportHaltEx - Exit

The MiniportHaltEx processing is not performed if the device is surprise removed or if the system is being powered off. For surprise removal, refer to the MiniportDevicePnPEventNotify handler behavior. For system shutdown, refer to the MiniportShutdownEx handler behavior.

MiniportDriverUnload

MiniportDriverUnload is the handler that is called before the WDI IHV miniport is unloaded. The WDI IHV miniport driver calls the Microsoft component to deregister itself. The Microsoft component calls **NdisMDeregisterMiniportDriver**.

The diagram below shows the flow of MiniportDriverUnload.

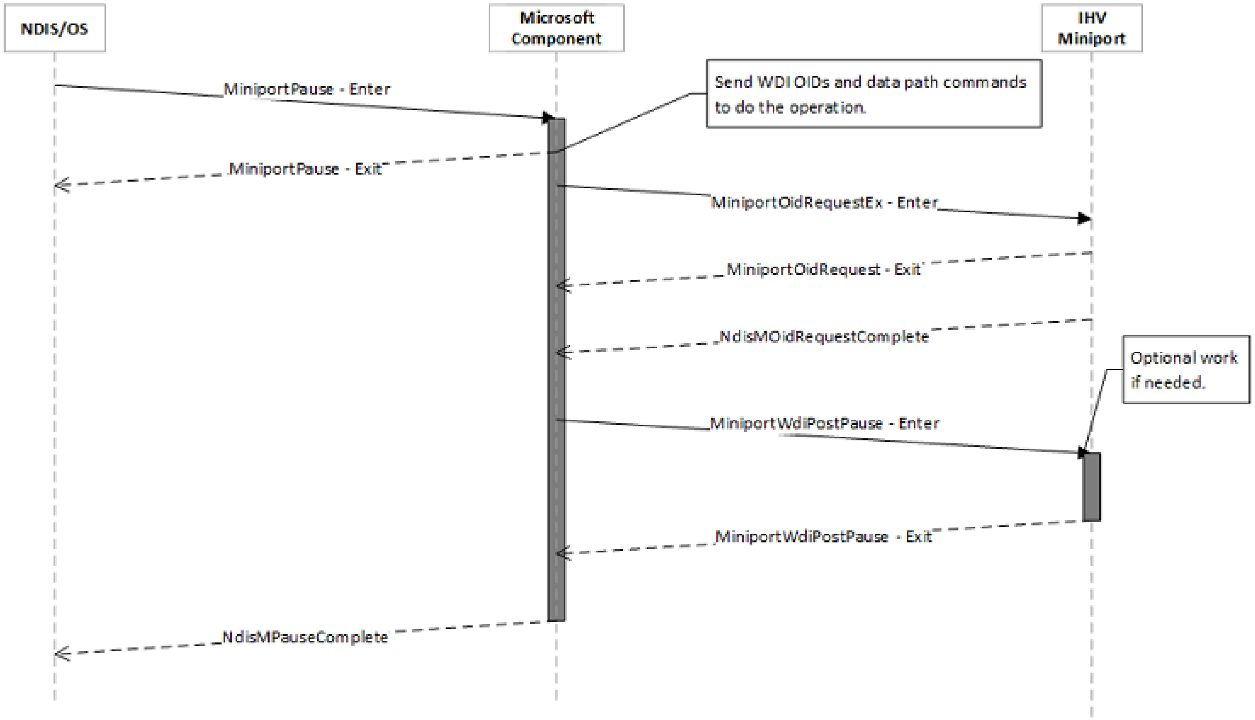


NDIS/OS
Microsoft Component
IHV Miniport
MiniportDriverUnload - Enter
MiniportDriverUnload - Enter
NdisMDeregisterWdiMiniportDriver - Enter
NdisMDeregisterMiniportDriver - Enter
NdisMDeregisterMiniportDriver - Exit
NdisMDeregisterWdiMiniportDriver - Exit
MiniportDriverUnload - Exit
MiniportDriverUnload - Exit

MiniportPause

The NDIS *MiniportPause* requirements are handled by the Microsoft component. As part of MiniportPause, the Microsoft component stops the data path and waits for it to clean up. The WDI IHV miniport can optionally register for a *MiniportWdiPostAdapterPause* callback that is called by the Microsoft component after it finishes the data path cleanup.

The diagram below shows the flow of MiniportPause.

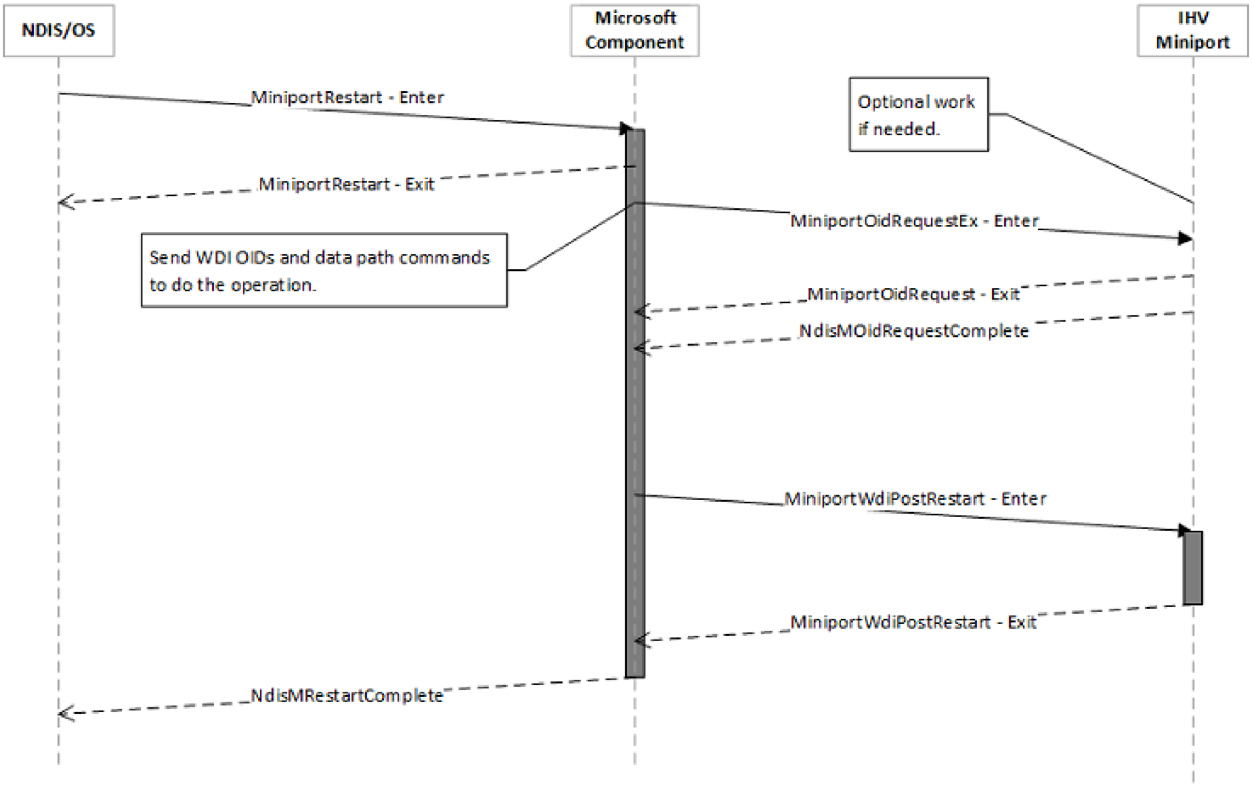


NDIS/OS
Microsoft Component
IHV Miniport
MiniportPause - Enter
Send WDI OIDs and data path commands to do the operation.
MiniportPause - Exit
MiniportOidRequestEx - Enter
MiniportOidRequest - Exit
NdisMOidRequestComplete
Optional work if needed.
MiniportWdiPostPause - Enter
MiniportWdiPostPause - Exit
NdisMPauseComplete

MiniportRestart

The NDIS *MiniportRestart* requirements are handled by the Microsoft component. As part of MiniportRestart, the Microsoft component undoes the data path pause work that it performed as part of MiniportPause. The WDI IHV miniport can optionally register for a *MiniportWdiPostAdapterRestart* callback that is called by the Microsoft component after it finishes restarting the data path.

The diagram below shows the flow of MiniportRestart.


NDIS/OS
Microsoft Component
IHV Miniport
MiniportRestart - Enter
Optional work if needed.
MiniportRestart - Exit
MiniportOidRequestEx - Enter
Send WDI OIDs and data path commands to do the operation.
MiniportOidRequest - Exit
NdisMOidRequestComplete
MiniportWdiPostRestart - Enter
MiniportWdiPostRestart - Exit
NdisMRestartComplete

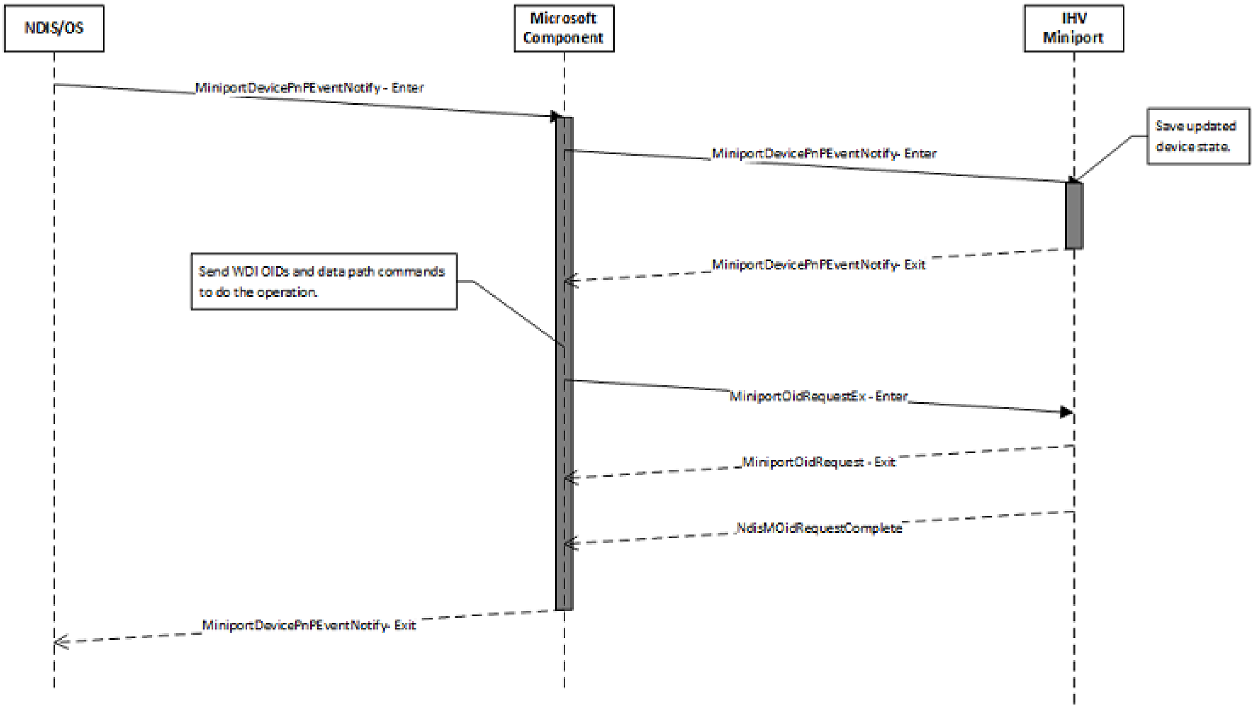
MiniportResetEx

MiniportResetEx is not handled by the Microsoft component. The WDI IHV miniport can optionally register for a *MiniportResetEx* callback that is called by the Microsoft component.

MiniportDevicePnPEventNotify

MiniportDevicePnPEventNotify is used to notify an NDIS driver of PNP events such as a device's surprise removal. When NDIS sends this notification, it is first forwarded to the WDI IHV miniport for processing. After the IHV component has finished processing it, the Microsoft component performs the appropriate processing for this event. The call that is forwarded into the IHV component is not serialized with other tasks and callbacks.

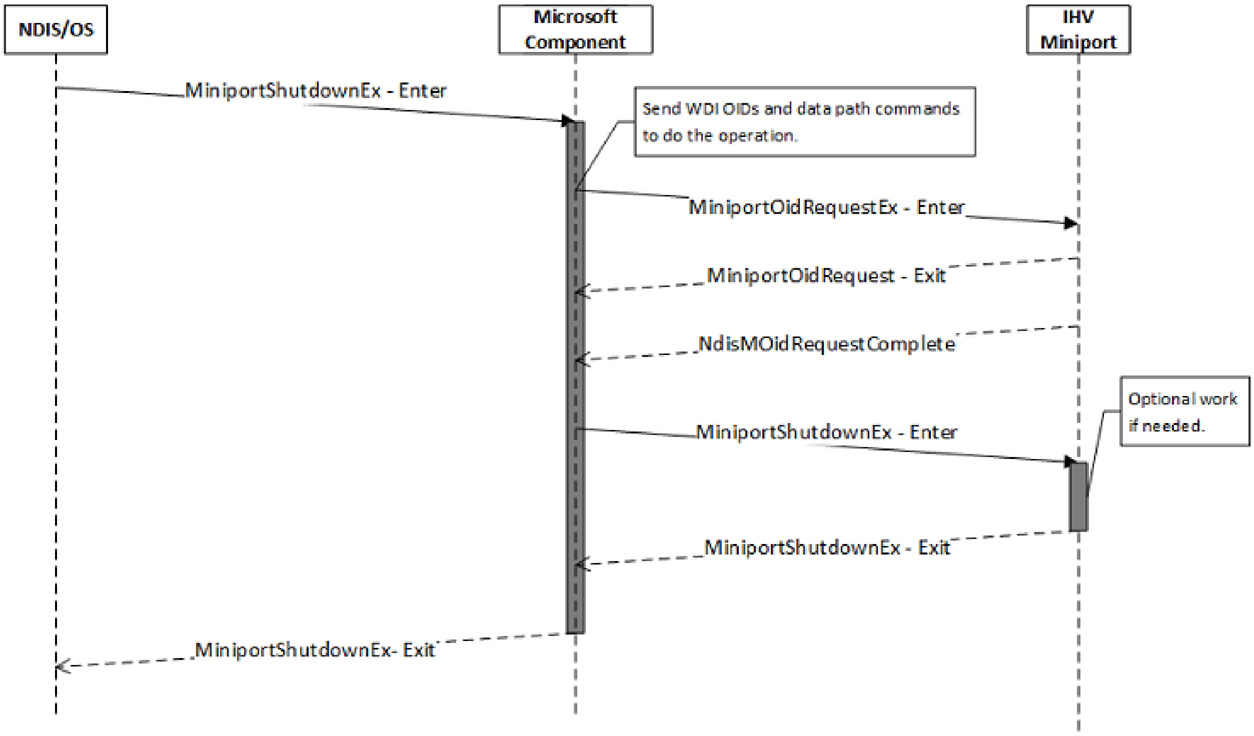
The diagram below shows the flow of MiniportDevicePnPEventNotify.


NDIS/OS
Microsoft Component
IHV Miniport
MiniportDevicePnPEventNotify - Enter
MiniportDevicePnPEventNotify- Enter
Save updated device state.
MiniportDevicePnPEventNotify- Exit
Send WDI OIDs and data path commands to do the operation.
MiniportOidRequestEx - Enter
MiniportOidRequest - Exit
NdisMOidRequestComplete
MiniportDevicePnPEventNotify- Exit

MiniportShutdownEx

MiniportShutdownEx is used to notify an NDIS driver about system shutdown events. When NDIS sends this notification, it is first handled by the Microsoft component. After the Microsoft component finishes processing it, it passes the event to the WDI IHV miniport for processing.

The diagram below shows the flow of MiniportShutdownEx.


NDIS/OS
Microsoft Component
IHV Miniport
MiniportShutdownEx - Enter
Send WDI OIDs and data path commands to do the operation.
MiniportOidRequestEx - Enter
MiniportOidRequest - Exit
NdisMOidRequestComplete
Optional work if needed.
MiniportShutdownEx - Enter
MiniportShutdownEx - Exit
MiniportShutdownEx- Exit

MiniportOidRequest

The *MiniportOidRequest* handler is a required handler that the WDI IHV miniport must implement. It is used by the Microsoft component to submit WDI commands to the IHV miniport. It is also used to forward OIDs that the Microsoft component does not handle to the IHV miniport.

The *MiniportOidRequest* call into the WDI IHV miniport should be considered as the M1 message for a WDI command. The completion of the OID (either via **NdisMOidRequestComplete** or via a return non-PENDING from *MiniportOidRequest*) should be considered as the M3 message for a WDI task/command.

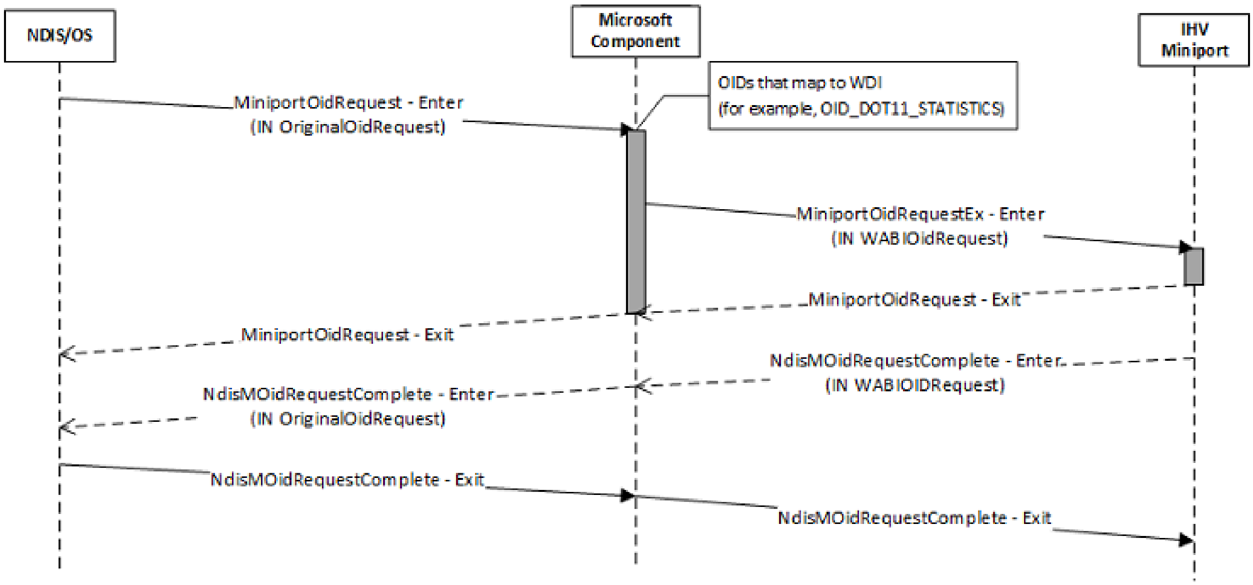
For every WDI command, there are two potential fields where an NDIS_STATUS code can be returned for the operation -- the status code from the *MiniportOidRequest* call (or **NdisMOidRequestComplete**), and the status code in the **WDI_MESSAGE_HEADER** field (either on the OID completion or via **NdisMIndicateStatusEx**). The Microsoft component always looks at the NDIS_STATUS from the OID completion before it looks at the **WDI_MESSAGE_HEADERStatus** field. The expectations of the IHV component for WDI OID processing are as follows.

1. WDI OIDs are submitted to the IHV component using an **NDIS_OID_REQUESTRequestType** of **NdisRequestMethod**, and the corresponding message and message length are in the **DATA.METHOD_INFORMATION.InformationBuffer** and **DATA.METHOD_INFORMATION.InputBufferLength** fields respectively.
2. The IHV component reports an error in the OID completion if there is an error while processing the command, and sets the Status field of the **WDI_MESSAGE_HEADER** to non-success if it has a Wi-Fi level failure.
3. For tasks and properties, the port number for the request is in the **WDI_MESSAGE_HEADERPortId** field. The **PortNumber** in the **NDIS_OID_REQUEST** is always set to 0.
4. For completion of the OID, it is acceptable for the *MiniportOidRequest* to return NDIS_STATUS_PENDING and complete the OID later (synchronously or asynchronously) with **NdisMOidRequestComplete**.
5. If the IHV component completes the OID with NDIS_STATUS_SUCCESS, it must populate the **BytesWritten** field of the OID request with the appropriate number of bytes, including space for the **WDI_MESSAGE_HEADER**.
6. If the IHV component does not have enough space in the **DATA.METHOD_INFORMATION.OutputBufferLength** field to fill the response, it completes the OID with NDIS_STATUS_BUFFER_TOO_SHORT and populates the **DATA.METHOD_INFORMATION.BytesNeeded** field. The Microsoft component may

attempt to allocate a buffer of the requested size and submit a new request to the IHV.

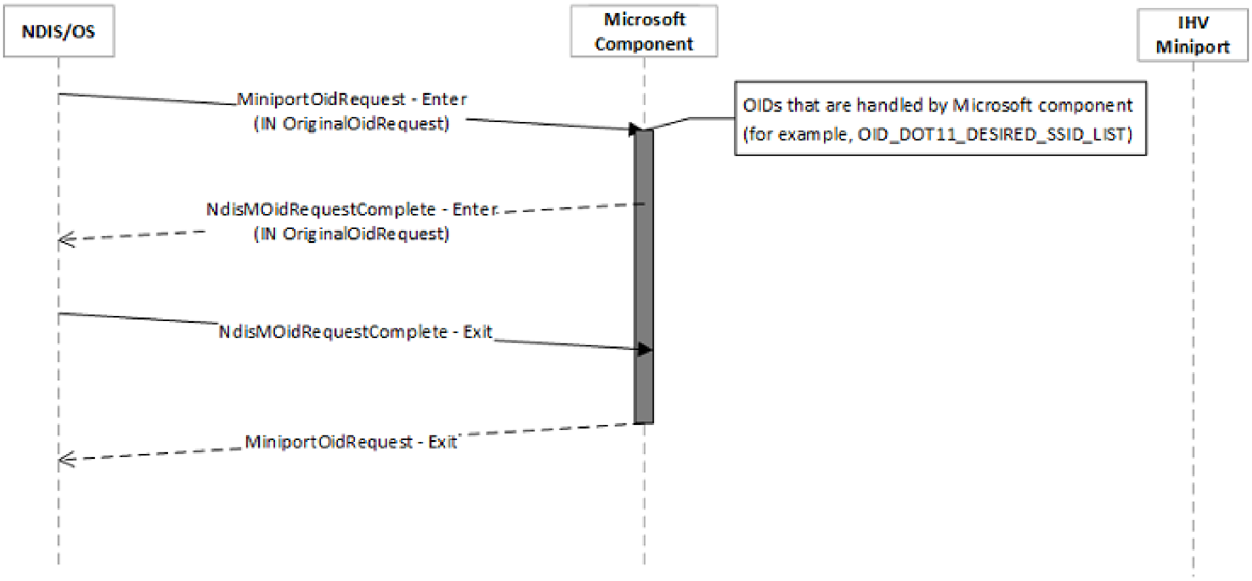
7. If it is a task, the task's M4 (NdisMIndicateStatusEx) must only be indicated if the task was reported as started successfully -- OID completion is successful and the **Status** in the WDI_MESSAGE_HEADER in OID completion was success.

The diagram below shows an example of an NDIS OID request that maps to a single WDI command. When the OID request is submitted by the operating system, the Microsoft component converts it to a WDI OID request and submits the WDI OID request to the IHV miniport. When the IHV miniport completes the OID, the Microsoft component appropriately completes the original OID request.

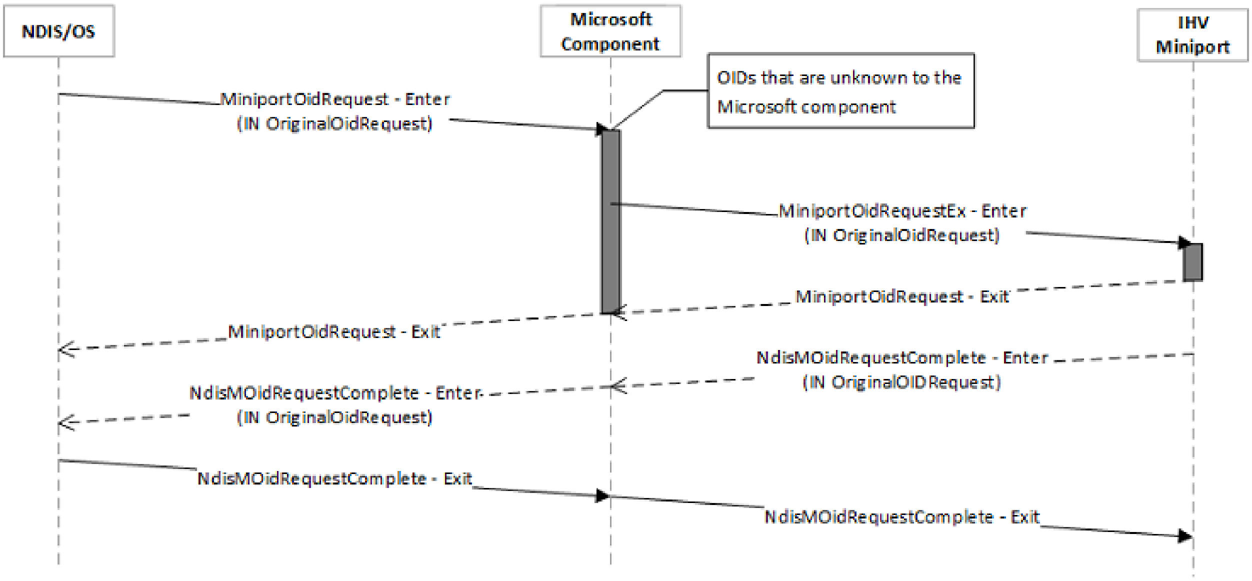

NDIS/OS
Microsoft Component
IHV Miniport
OIDs that map to WDI
(for example, OID_DOT11_STATISTICS)
MiniportOidRequest - Enter
(IN OriginalOidRequest)
MiniportOidRequestEx - Enter
(IN WABIOidRequest)
MiniportOidRequest - Exit
MiniportOidRequest - Exit
NdisMOidRequestComplete - Enter
(IN WABIOIDRequest)
NdisMOidRequestComplete - Enter
(IN OriginalOidRequest)
NdisMOidRequestComplete - Exit
NdisMOidRequestComplete - Exit

If the OriginalOidRequest maps to multiple WDI OidRequests and one of the WDI requests fails, the OriginalOidRequest also fails. If a subset of the intermediate operations already finished, the Microsoft component attempts to undo the operations that support clean up.

The diagram below shows an example of an NDIS OID request that is handled completed by the Microsoft component. When the OID request is submitted by the operating system, the Microsoft component processes and completes the OID. This OID is not passed to the WDI IHV miniport.


NDIS/OS
Microsoft Component
IHV Miniport
OIDs that are handled by Microsoft component (for example, OID_DOT11_DESIRED_SSID_LIST)
MiniportOidRequest - Enter (IN OriginalOidRequest)
NdisMOidRequestComplete - Enter (IN OriginalOidRequest)
NdisMOidRequestComplete - Exit
MiniportOidRequest - Exit

OIDs that are not understood by the Microsoft component are forwarded directly to the IHV component for processing.


NDIS/OS
Microsoft Component
IHV Miniport
OIDs that are unknown to the Microsoft component
MiniportOidRequest - Enter (IN OriginalOidRequest)
MiniportOidRequestEx - Enter (IN OriginalOidRequest)
MiniportOidRequest - Exit
MiniportOidRequest - Exit
NdisMOidRequestComplete - Enter (IN OriginalOIDRequest)
NdisMOidRequestComplete - Enter (IN OriginalOidRequest)
NdisMOidRequestComplete - Exit
NdisMOidRequestComplete - Exit

The behavior of MiniportOidRequest is unchanged for the WDI IHV miniport driver (as compared to a Native Wi-Fi miniport). The calls are serialized and the IHV miniport can either complete it synchronously or asynchronously with a call to NdisMOidRequestComplete.

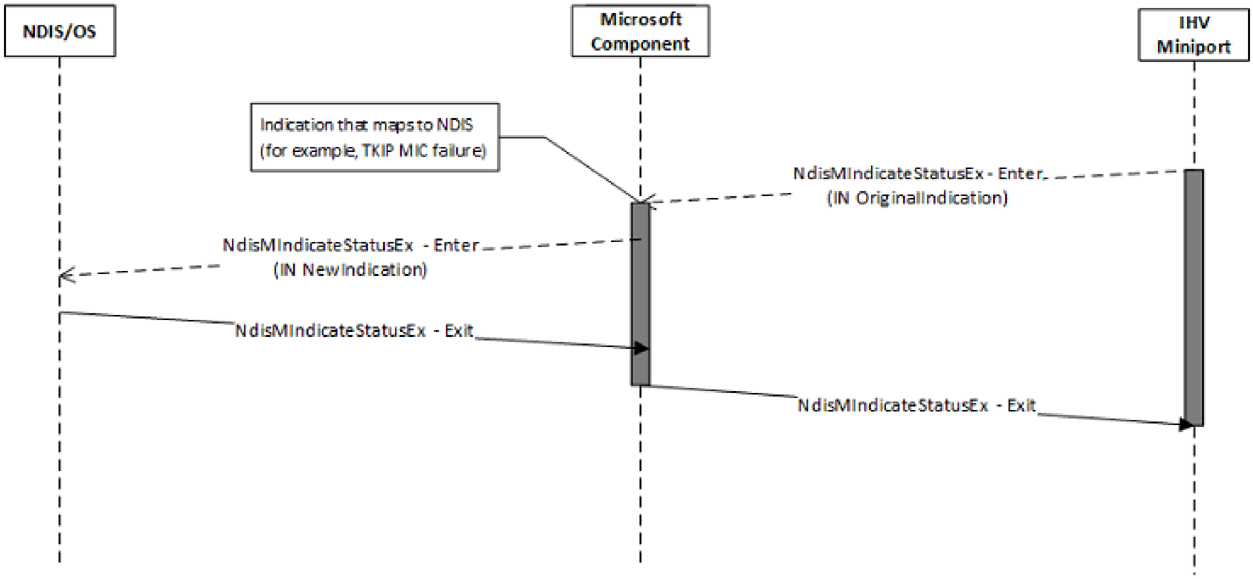
MiniportCancelOidRequest

This is an optional handler that is used by a WDI IHV miniport that needs to handle OIDs that are not mapped to WDI messages. This handler is not used for any WDI OIDs. WDI OIDs must complete quickly and there is no need for the IHV miniport driver to attempt to cancel a pending OID. Cancellation of WDI tasks is handled using the appropriate cancel task OID request. For unmapped OIDs, the expected behavior is defined by NDIS.

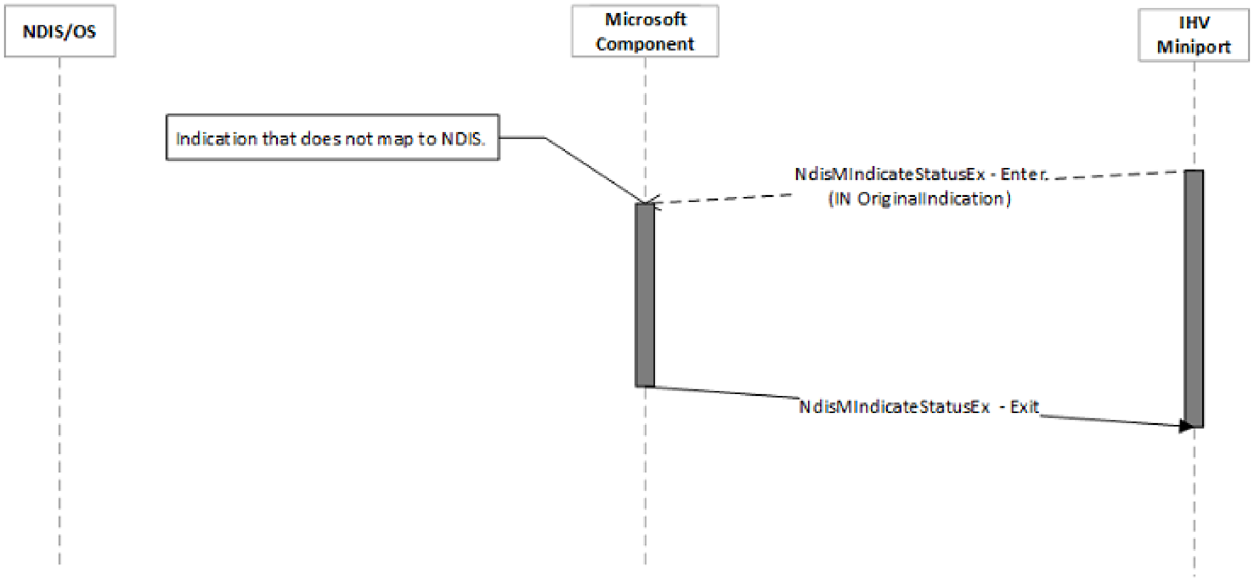
NdisMIndicateStatusEx

NdisMIndicateStatusEx is used by the WDI IHV miniport to send indications to the Microsoft component. The indications may be unsolicited indications such as TKIP MIC failures, or solicited indications for the completion (M4) for a task.

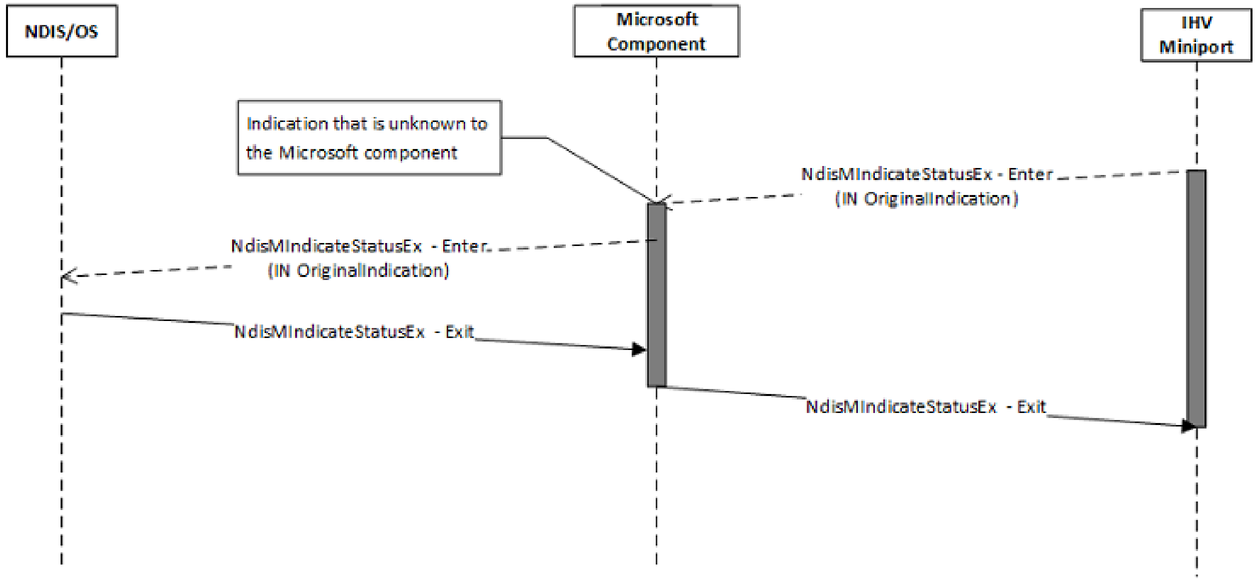
The diagram below shows an example of a WDI indication that has a corresponding NDIS/Native Wi-Fi indication. When the indication is submitted by the IHV miniport to the Microsoft component, the Microsoft component converts it to an existing indication and forwards it to the operating system.


NDIS/OS
Microsoft Component
IHV Miniport
Indication that maps to NDIS (for example, TKIP MIC failure)
NdisMIndicateStatusEx - Enter (IN OriginalIndication)
NdisMIndicateStatusEx - Enter (IN NewIndication)
NdisMIndicateStatusEx - Exit
NdisMIndicateStatusEx - Exit

The diagram below shows an example of a WDI indication that has no corresponding NDIS/Native Wi-Fi indication. This is handled by the Microsoft component.


NDIS/OS
Microsoft Component
IHV Miniport
Indication that does not map to NDIS.
NdisMIndicateStatusEx - Enter (IN OriginalIndication)
NdisMIndicateStatusEx - Exit

The diagram below shows an indication that is not recognized by the Microsoft component. The indication is forwarded as-is to the operating system.


NDIS/OS
Microsoft Component
IHV Miniport
Indication that is unknown to the Microsoft component
NdisMIndicateStatusEx - Enter
(IN OriginalIndication)
NdisMIndicateStatusEx - Enter
(IN OriginalIndication)
NdisMIndicateStatusEx - Exit
NdisMIndicateStatusEx - Exit

The behavior of NdisMIndicateStatusEx is unchanged for the WDI IHV miniport driver (as compared to a Native Wi-Fi miniport).

MiniportDirectOidRequest

This is an optional handler that is registered by a WDI IHV miniport driver if it needs to handle Direct OIDs that are not mapped to WDI messages. All existing Direct OIDs for Wi-Fi Direct are mapped to WDI messages, so this handler is not required to support that functionality. Unsupported Direct OIDs are not serialized by the Microsoft component.

MiniportCancelDirectOidRequest

This is an optional handler that is used by a WDI IHV miniport that needs to handle Direct OIDs that are not mapped to WDI messages. For unmapped OIDs, the expected behavior is defined by NDIS.

MiniportSendNetBufferLists

This handler is not used in a WDI IHV miniport driver and should not be provided. The Microsoft component uses the data path handlers registered through NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS to submit send packets to the IHV miniport.

MiniportCancelSend

This handler is not used in a WDI IHV miniport driver and should not be provided.

MiniportReturnNetBufferLists

This handler is not used in a WDI IHV Miniport driver and should not be provided. The Microsoft component uses the data path handlers registered through **NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS** to return received packets to the IHV miniport.

WDI handler: MiniportWdiOpenAdapter

The *MiniportWdiOpenAdapter* handler is used by the Microsoft component to initiate the Open Task operation on the IHV driver. This call must complete quickly and if the open operation has been successfully started, the IHV must return NDIS_STATUS_SUCCESS on this call and call the **OpenAdapterComplete** handler that is passed into the **NDIS_WDI_INIT_PARAMETERS** parameter of *MiniportWdiAllocateAdapter*.

WDI handler: MiniportWdiCloseAdapter

The *MiniportWdiCloseAdapter* handler is used by the Microsoft component to initiate the Close Task operation on the IHV driver. This call must complete quickly and if the open operation has been successfully started, the IHV must return NDIS_STATUS_SUCCESS on this call and call the **CloseAdapterComplete** handler that is passed into the **NDIS_WDI_INIT_PARAMETERS** parameter of the *MiniportWdiAllocateAdapter*.

WDI NDIS interface restrictions

Article • 03/14/2023

The WDI IHV miniport driver has access to all of the functionality provided by NDIS and KMDF. It is recommended that the IHV driver use the KMDF primitives for talking to the rest of the operating system whenever possible. While the model does not restrict the IHV miniport from calling any of the NDIS APIs, some NDIS APIs are called by the Microsoft WLAN component so the IHV driver should not call them.

The WDI IHV miniport driver needs to be aware of the following restrictions on NDIS interfaces.

Function	Restrictions	Alternative
NdisMRegisterMiniportDriver	Disallowed	NdisMRegisterWdiMiniportDriver
NdisMDeregisterMiniportDriver	Disallowed	NdisMDeregisterWdiMiniportDriver
NdisMSetMiniportAttributes	Disallowed with **MiniportAttributes** types: NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES NDIS_MINIPORT_ADAPTER_NATIVE_802_11_ATTRIBUTES	None. These are queried using WDI commands.
NdisMIndicateReceiveNetBufferLists	Disallowed	The WDI data path receive handler to indicate received packets.
NdisMSendNetBufferListsComplete	Disallowed	The WDI data path send handler to complete sent packets.

Native WLAN to WDI OID mapping

Article • 03/14/2023

The following table contains the mappings between Native 802.11 WLAN OIDs to WDI commands. This is not a static mapping and, depending on the current state, additional or fewer WDI commands may be sent down.

Note

Due to long DDI names, you may need to scroll the table horizontally to see all content. There is a scrollbar at the bottom of the page.

DDI	Query	Set/Method/Function
MIBS		
OID_DOT11_BEACON_PERIOD	Handled by Microsoft	Saved and set with OID_WDI_TASK_START_AP
OID_DOT11_COUNTRY_STRING	Unsupported	Not applicable
OID_DOT11_CURRENT_CHANNEL	Handled by Microsoft	Unsupported
OID_DOT11_CURRENT_FREQUENCY	Handled by Microsoft	Unsupported
OID_DOT11_CURRENT_REG_DOMAIN	Unsupported	Not applicable
OID_DOT11_DTIM_PERIOD	Handled by Microsoft	Saved and set with OID_WDI_TASK_START_AP
OID_DOT11_FRAGMENTATION_THRESHOLD	Unsupported	Unsupported
OID_DOT11_LONG_RETRY_LIMIT	Unsupported	Unsupported
OID_DOT11_MULTI_DOMAIN_CAPABILITY_ENABLED	Unsupported	Unsupported
OID_DOT11_OPERATIONAL_RATE_SET	Handled by Microsoft	Unsupported
OID_DOT11_REG_DOMAINS_SUPPORT_VALUE	Unsupported	Not applicable
OID_DOT11_RTS_THRESHOLD	Unsupported	Not applicable
OID_DOT11_SHORT_RETRY_LIMIT	Unsupported	Unsupported
Operational		
OID_DOT11_CURRENT_OPERATION_MODE	Handled by Microsoft	OID_WDI_TASK_CHANGE_OPERATION_MODE
OID_DOT11_MULTICAST_LIST	Handled by Microsoft	OID_WDI_SET_MULTICAST_LIST
OID_DOT11_NIC_POWER_STATE	Handled by Microsoft	OID_WDI_TASK_DOT11_RESET, OID_WDI_TASK_SET_RADIO_STATE
OID_DOT11_NIC_SPECIFIC_EXTENSION	Not applicable	OID_WDI_IHV_REQUEST
OID_DOT11_RESET_REQUEST	Not applicable	OID_WDI_TASK_DOT11_RESET
OID_DOT11_SCAN_REQUEST	Not applicable	OID_WDI_TASK_SCAN
Virtual Wi-Fi		
OID_DOT11_CREATE_MAC	Not applicable	OID_WDI_TASK_CREATE_PORT, OID_WDI_TASK_DOT11_RESET
OID_DOT11_DELETE_MAC	Not applicable	OID_WDI_TASK_DOT11_RESET, OID_WDI_TASK_DELETE_PORT
OID_DOT11_VIRTUAL_STATION_CAPABILITY	**No planned support**	
ExtSTA/ExtAP		
OID_DOT11_ACTIVE_PHY_LIST	Handled by Microsoft based on **WDI_TLV_ASSOCIATION_RESULT**	
OID_DOT11_ASSOCIATION_PARAMS	Not applicable	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_AUTO_CONFIG_ENABLED	Handled by Microsoft	Handled by Microsoft

DDI	Query	Set/Method/Function
OID_DOT11_CIPHER_DEFAULT_KEY	Not applicable	OID_WDI_SET_ADD_CIPHER_KEYS
OID_DOT11_CIPHER_DEFAULT_KEY_ID	Handled by Microsoft	OID_WDI_SET_DEFAULT_KEY_ID
OID_DOT11_CIPHER_KEY_MAPPING_KEY	Not applicable	OID_WDI_SET_ADD_CIPHER_KEYS
OID_DOT11_CONNECT_REQUEST	Not applicable	OID_WDI_SET_PRIVACY_EXEMPTION_LIST, OID_WDI_TASK_CONNECT
OID_DOT11_CURRENT_PHY_ID	Handled by Microsoft	Handled by Microsoft
OID_DOT11_DESIRED_BSS_TYPE	Handled by Microsoft	Handled by Microsoft
OID_DOT11_DESIRED_BSSID_LIST	Handled by Microsoft	Handled by Microsoft
OID_DOT11_DESIRED_COUNTRY_OR_REGION_STRING	Unsupported	Unsupported
OID_DOT11_DESIRED_PHY_LIST	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_DESIRED_SSID_LIST	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_DISCONNECT_REQUEST	Not applicable	OID_WDI_TASK_DISCONNECT
OID_DOT11_ENABLED_AUTHENTICATION_ALGORITHM	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_ENABLED_MULTICAST_CIPHER_ALGORITHM	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_ENABLED_UNICAST_CIPHER_ALGORITHM	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_ENUM_ASSOCIATION_INFO	Handled by Microsoft	Not applicable
OID_DOT11_ENUM_BSS_LIST	Not applicable	Handled by Microsoft
OID_DOT11_EXCLUDE_UNENCRYPTED	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_EXCLUDED_MAC_ADDRESS_LIST	Handled by Microsoft	Handled by Microsoft
OID_DOT11_EXTSTA_CAPABILITY	Handled by Microsoft	Not applicable
OID_DOT11_FLUSH_BSS_LIST	Not applicable	Handled by Microsoft
OID_DOT11_HARDWARE_PHY_STATE	Handled by Microsoft	Not applicable
OID_DOT11_HIDDEN_NETWORK_ENABLED	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_IBSS_PARAMS	**No planned support**	
OID_DOT11_MEDIA_STREAMING_ENABLED	Handled by Microsoft	OID_WDI_SET_CONNECTION_QUALITY
OID_DOT11_PMKID_LIST	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_PORT_STATE_NOTIFICATION	Not applicable	Unsupported
OID_DOT11_POWER_MGMT_REQUEST	Handled by Microsoft	OID_WDI_SET_CONNECTION_QUALITY
OID_DOT11_PRIVACY_EXEMPTION_LIST	Not applicable	OID_WDI_SET_PRIVACY_EXEMPTION_LIST
OID_DOT11_QOS_PARAMS	Handled by Microsoft	Handled by Microsoft
OID_DOT11_SAFE_MODE_ENABLED	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_STATISTICS	OID_WDI_GET_STATISTICS	Not applicable
OID_DOT11_SUPPORTED_COUNTRY_OR_REGION_STRING	Unsupported	Not applicable
OID_DOT11_SUPPORTED_MULTICAST_ALGORITHM_PAIR	Handled by Microsoft	
OID_DOT11_SUPPORTED_UNICAST_ALGORITHM_PAIR	Handled by Microsoft	
OID_DOT11_UNICAST_USE_GROUP_ENABLED	Handled by Microsoft	Saved and set with OID_WDI_TASK_CONNECT
OID_DOT11_UNREACHABLE_DETECTION_THRESHOLD	Unsupported	Unsupported
ExtAP		

DDI	Query	Set/Method/Function
OID_DOT11_ADDITIONAL_IE	**No planned support**	
OID_DOT11_AVAILABLE_CHANNEL_LIST	**No planned support**	
OID_DOT11_AVAILABLE_FREQUENCY_LIST	**No planned support**	
OID_DOT11_DISASSOCIATE_PEER_REQUEST	Not applicable	OID_WDI_TASK_DISCONNECT
OID_DOT11_ENUM_PEER_INFO	**No planned support**	
OID_DOT11_INCOMING_ASSOCIATION_DECISION	Not applicable	OID_WDI_TASK_SEND_AP_ASSOCIATION_RESPONSE
OID_DOT11_START_AP_REQUEST	Not applicable	OID_WDI_TASK_START_AP
OID_DOT11_WPS_ENABLED	Handled by Microsoft	Handled by Microsoft
Wi-Fi Direct		
OID_DOT11_WFD_ADDITIONAL_IE	Not applicable	OID_WDI_SET_ADVERTISEMENT_INFORMATION
OID_DOT11_WFD_CONNECT_TO_GROUP_REQUEST	Not applicable	OID_WDI_SET_PRIVACY_EXEMPTION_LIST, OID_WDI_TASK_CONNECT
OID_DOT11_WFD_DESIRED_GROUP_ID	Not applicable	Saved and set with OID_WDI_TASK_CONNECT / OID_WDI_TASK_START_AP
OID_DOT11_WFD_DEVICE_CAPABILITY	Not applicable	OID_WDI_SET_ADVERTISEMENT_INFORMATION
OID_DOT11_WFD_DEVICE_INFO	Not applicable	OID_WDI_SET_ADVERTISEMENT_INFORMATION
OID_DOT11_WFD_DEVICE_LISTEN_CHANNEL	Not applicable	Unsupported
OID_DOT11_WFD_DISCONNECT_FROM_GROUP_REQUEST	Not applicable	OID_WDI_TASK_DISCONNECT
OID_DOT11_WFD_DISCOVER_REQUEST	Not applicable	OID_WDI_TASK_P2P_DISCOVER
OID_DOT11_WFD_ENUM_DEVICE_LIST	Not applicable	Handled by Microsoft
OID_DOT11_WFD_FLUSH_DEVICE_LIST	Not applicable	Handled by Microsoft
OID_DOT11_WFD_GET_DIALOG_TOKEN	Not applicable	Handled by Microsoft
OID_DOT11_WFD_GROUP_JOIN_PARAMETERS	Not applicable	Saved and set with OID_WDI_TASK_CONNECT / OID_WDI_TASK_START_AP
OID_DOT11_WFD_GROUP_OWNER_CAPABILITY	Not applicable	OID_WDI_SET_ADVERTISEMENT_INFORMATION
OID_DOT11_WFD_GROUP_START_PARAMETERS	Not applicable	Saved and set with OID_WDI_TASK_CONNECT / OID_WDI_TASK_START_AP
OID_DOT11_WFD_LISTEN_STATE_DISCOVERABILITY	Not applicable	OID_WDI_SET_P2P_LISTEN_STATE
OID_DOT11_WFD_SECONDARY_DEVICE_TYPE_LIST	Not applicable	OID_WDI_SET_ADVERTISEMENT_INFORMATION
OID_DOT11_WFD_SEND_GO_NEGOTIATION_CONFIRMATION	Not applicable	OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME
OID_DOT11_WFD_SEND_GO_NEGOTIATION_REQUEST	Not applicable	OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME
OID_DOT11_WFD_SEND_GO_NEGOTIATION_RESPONSE	Not applicable	OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME
OID_DOT11_WFD_SEND_INVITATION_REQUEST	Not applicable	OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME
OID_DOT11_WFD_SEND_INVITATION_RESPONSE	Not applicable	OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME
OID_DOT11_WFD_SEND_PROVISION_DISCOVERY_REQUEST	Not applicable	OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME
OID_DOT11_WFD_SEND_PROVISION_DISCOVERY_RESPONSE	Not applicable	OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME
OID_DOT11_WFD_START_GO_REQUEST	Not applicable	OID_WDI_TASK_START_AP
OID_DOT11_WFD_STOP_DISCOVERY	Not applicable	Converts to OID_WDI_ABORT_TASK
Wi-Fi Power		

DDI	Query	Set/Method/Function
OID_DOT11_POWER_MGMT_MODE_AUTO_ENABLED	Not applicable	OID_WDI_SET_CONNECTION_QUALITY
OID_DOT11_POWER_MGMT_MODE_STATUS	WDI_GET_AUTO_POWER_SAVE	Not applicable
OID_DOT11_OFFLOAD_NETWORK_LIST	Not applicable	OID_WDI_SET_NETWORK_LIST_OFFLOAD
NDIS		
OID_GEN_CURRENT_LOOKAHEAD	Not applicable	Unsupported
OID_GEN_CURRENT_PACKET_FILTER	Not applicable	OID_WDI_SET_RECEIVE_PACKET_FILTER
OID_GEN_INTERRUPT_MODERATION	Handled by Microsoft	Unsupported
OID_GEN_LINK_PARAMETERS	Not applicable	Unsupported
OID_GEN_MAXIMUM_TOTAL_SIZE	Handled by Microsoft	Not applicable
OID_GEN_RCV_OK	Handled by Microsoft	Not applicable
OID_GEN_RECEIVE_BLOCK_SIZE	Handled by Microsoft	Not applicable
OID_GEN_RECEIVE_BUFFER_SPACE	Handled by Microsoft	Not applicable
OID_GEN_STATISTICS	Handled by Microsoft	Not applicable
OID_GEN_TRANSMIT_BLOCK_SIZE	Handled by Microsoft	Not applicable
OID_GEN_TRANSMIT_BUFFER_SPACE	Handled by Microsoft	Not applicable
OID_GEN_VENDOR_DESCRIPTION	Handled by Microsoft	Not applicable
OID_GEN_VENDOR_DRIVER_VERSION	Handled by Microsoft	Not applicable
OID_GEN_VENDOR_ID	Handled by Microsoft	Not applicable
OID_GEN_XMIT_OK	Handled by Microsoft	Not applicable
OID_GEN_PORT_AUTHENTICATION_PARAMETERS	Handled by Microsoft	Not applicable
Ethernet		
OID_802_3_ADD_MULTICAST_ADDRESS	Unsupported	Unsupported
OID_802_3_DELETE_MULTICAST_ADDRESS	Unsupported	Unsupported
OID_802_3_PERMANENT_ADDRESS	Handled by Microsoft	Not applicable
OID_802_3_CURRENT_ADDRESS	Handled by Microsoft	Not applicable
OID_802_3_MULTICAST_LIST	Handled by Microsoft	OID_WDI_SET_MULTICAST_LIST
OID_802_3_MAXIMUM_LIST_SIZE	Handled by Microsoft	Not applicable
OID_802_3_MAC_OPTIONS	Handled by Microsoft	Not applicable
OID_802_3_RCV_ERROR_ALIGNMENT	Handled by Microsoft	Not applicable
OID_802_3_XMIT_ONE_COLLISION	Handled by Microsoft	Not applicable
OID_802_3_XMIT_MORE_COLLISIONS	Handled by Microsoft	Not applicable
NDIS Power		
OID_PNP_QUERY_POWER	Handled by Microsoft	Not applicable
OID_PNP_SET_POWER	Not applicable	OID_WDI_SET_POWER_STATE, OID_WDI_SET_NETWORK_LIST_OFFLOAD, OID_WDI_TASK_OPEN (hibernate), OID_WDI_TASK_CLOSE (hibernate)
OID_PM_ADD_PROTOCOL_OFFLOAD	Not applicable	OID_WDI_SET_ADD_PM_PROTOCOL_OFFLOAD
OID_PM_ADD_WOL_PATTERN	Not applicable	OID_WDI_SET_ADD_WOL_PATTERN

DDI	Query	Set/Method/Function
OID_PM_CURRENT_CAPABILITIES	Handled by Microsoft	
OID_PM_GET_PROTOCOL_OFFLOAD	Not applicable	OID_WDI_GET_PM_PROTOCOL_OFFLOAD
OID_PM_HARDWARE_CAPABILITIES	Handled by Microsoft	Not applicable
OID_PM_PARAMETERS	Handled by Microsoft	Handled by Microsoft
OID_PM_PROTOCOL_OFFLOAD_LIST	Handled by Microsoft	Not applicable
OID_PM_REMOVE_PROTOCOL_OFFLOAD	Not applicable	OID_WDI_SET_REMOVE_PM_PROTOCOL_OFFLOAD
OID_PM_REMOVE_WOL_PATTERN	Not applicable	OID_WDI_SET_REMOVE_WOL_PATTERN
OID_PM_WOL_PATTERN_LIST	Handled by Microsoft	Not applicable
OID_PACKET_COALESCING_FILTER_MATCH_COUNT	WDI_GET_RECEIVE_ COALESCING_MATCH_COUNT	Not applicable
OID_RECEIVE_FILTER_SET_FILTER	Not applicable	OID_WDI_SET_RECEIVE_COALESCING
OID_RECEIVE_FILTER_CLEAR_FILTER	Not applicable	OID_WDI_SET_CLEAR_RECEIVE_COALESCING
TCP Task Offload		
OID_TCP_TASK_OFFLOAD	Unsupported	Unsupported

WDI tracing with WDILib

Article • 03/14/2023

The WDILib component currently supports tracing using WPP. The trace provider's GUID is {21ba7b61-05f8-41f1-9048-c09493dcfe38}. The following instructions can be used to collect and view the traces.

Start tracing

To collect traces, run the following command line from an Administrator command prompt.

PowerShell

```
netsh trace start wireless_dbg provider={21ba7b61-05f8-41f1-9048-
c09493dcfe38} level=0xff keywords=0xff
```

You can create a batch cmd file that contains the netsh trace command line. This allows you to expand the command to include IHV WPP events if desired.

PowerShell

```
@echo off
Setlocal
Rem
Rem Example trace script enabling both WDI and IHV WPP events
Rem

Rem
Rem replace IHV_WPP_GUID with your IHV driver GUID and desired trace flags
Rem
Set IHV_WPP_GUID={0160d072-248f-11e2-be71-082e5f28d97c} 0xff
Set WDI_WPP_GUID={21ba7b61-05f8-41f1-9048-c09493dcfe38} level=0xff
keywords=0xff

Rem
Rem Start the trace
Rem
netsh trace start wireless_dbg provider=%WDI_WPP_GUID%
provider=%IHV_WPP_GUID% globallevel=0xff
```

You can use 'wireless_dbg' to enable the rest of the operating system-side traces. Additional useful options are 'capture=yes' to enable packet traces and 'persistent=yes' to keep tracing enabled across reboots until it is stopped. This means you do not have to remember to enable tracing across reboots.

Alternatively, the trace provider GUID can be incorporated into the trace commands used for collecting the IHV component tracing.

Stop tracing

Once a repro is obtained, the tracing is stopped and traces saved with the following command.

PowerShell

```
netsh trace stop
```

This saves the traces in %TEMP%\NetTraces\NetTrace.etl.

Convert WPP traces to text

Traces are converted to text with any WPP to Text conversion tools. One option is to use Netsh.

PowerShell

```
netsh trace convert NetTrace.etl tmfpath=C:\TMF
```

In this example, C:\TMF contains the TMFs (generated from the wdiwifi.PDB by running tracepdb.exe). If the TMFs are not generated, the converted traces do not show up correctly.

Analyze traces

The converted text file (NetTrace.txt) can be viewed using any text viewer. If the viewer has the ability to add filters, using '[ERROR]', '[Wdi', and '[MSG ' filters may help scope the traces down to interesting lines.

Tracing on phones

To trace on phones, save the following text as wdiguids.txt, including the trace providers you want to trace.

Text

```
0C5A3172-2248-44FD-B9A6-8389CB1DC56A        wlansvc
D905AC1D-65E7-4242-99EA-FE66A8355DF8        WlanAPI
abe47285-c002-46d1-95e4-c4aec3c78f50        WFD WPS Provider
9CC9BEB7-9D24-47C7-8F9D-CCC9DCAC29EB        WFD WPS Provider
7D7180B3-A452-4FFF-8D1F-7B32B248AB70        DAF WFD Provider
19e464a4-7408-49bd-b960-53446ae47820        DAS
e176aa66-5cc8-4321-9624-f9c1d2b7bf06        UPnP
71975d00-46bb-4236-bd4f-41a8c72fadfc        DAF UPnP Provider
2B175479-BBE4-4262-AA71-19AFB22A46F5        UPnP
4D946A46-275B-4C9D-B835-0B2160559256        WPS
C100BECE-D33A-4A4B-BF23-BBEF4663D017        WCN
20644520-D1C2-4024-B6F6-311F99AA51ED        MSMSEC
ED092A80-0125-4403-92AC-4C06632420F8        MSMSEC
253F4CD1-9475-4642-88E0-6790D7A86CDE        MSMSEC
7076BF7A-DB99-4A63-8AFE-0BB2AB92997A        1X
5F31090B-D990-4E91-B16D-46121D0255AA        EAPHOST
D905AC1C-65E7-4242-99EA-FE66A8355DF8        NWifi
e49b27dd-b2f0-4571-8c6a-3271a3a3a6b9        VNE wlanvmp
11111111-0000-0000-0000-000000000000        VNE UserMode
c02edc8d-d627-46c9-abd9-c8b78f88c223        VWifiBus
914598a6-28f0-42ac-bf3d-a29c6047a739        VWifiFlt
ad8fe36a-0581-4571-a143-5a3f93e30160        Bluewire devicepairing.dll
14E44EEB-E037-45a5-9CA6-472258724563        Bluewire devicepairingfolder.dll
05516000-0670-11e0-8e5c-f4ce462d9187        ProximityCommon.dll
ProximityService.dll
05516002-0670-11e0-8e5c-f4ce462d9187        ProximityUxHost
3475aabe-44ba-49eb-bfd8-de32e88b4f35
Windows.Networking.NearFieldProximity.dll
8C3E3B78-4E27-48ea-B52C-7BCDC9CC2DA9        WMP
C9C074D2-FF9B-410F-8AC6-81C7B8E60D0F        MediaEngineCtrlGuid
3496b396-5c43-45e7-b38e-d509b79ae721        WFDPAL
c7491fe4-66f4-4421-9954-b55f03db3186        WiFiDisplay
F9A92EFE-77C3-4D19-8B00-1EE9E7CBE8C1        UX
802ec45b-1e99-4b83-9920-87c98277ba9d        Miradisp
f860141e-94e0-418e-a8a6-2321623c3018        Vlib
0160d072-248f-11e2-be71-082e5f28d97c        WIFI Driver
07C9AAA5-FF6B-44CF-A417-C3BE5A719C0B        WIFI Driver
21ba7b61-05f8-41f1-9048-c09493dcfe38        WDI Driver
D710D46C-235D-4798-AC20-9F83E1DCD557        EapMethod Ttls
E21E2366-917F-4CCC-BFE4-0FD23CB31209        TTLS WPP
7076bf7a-db99-4a63-8afe-0bb2ab92997a        OneX WPP
3A9B0DE9-2A69-413E-9074-444C0E3A81D9        EAP SIM WPP
5F31090B-D990-4E91-B16D-46121D0255AA        Eaphost WPP
6EB8DB94-FE96-443F-A366-5FE0CEE7FB1C        Eaphost
E5C16D49-2464-4382-BB20-97A4B5465DB9        WifiNetworkmanager
5CA18737-22AC-4050-85BC-B8DBB9F7D986        WifiNetworkmanager wpp
CC3DF8E3-4111-48d0-9B21-7631021F7CA6        Dhcpv4 Client
07a29c3d-26a4-41e2-856a-095b3eb8b6ef        Dhcpv6 Client
```

Using TShell, connect to the device, copy the files over, and begin tracing.

PowerShell

```
open-device 127.0.0.1
cmdd "mkdir \data\test\wlan"
putd wdiguids.txt \data\test\wlan
cmdd tracelog.exe '-start wdiwpp -f \data\test\wlan\wdiwpp.etl -cir 256 -rt
-ls -ft 1'
cmdd tracelog.exe '-enable wdiwpp -guid \data\test\wlan\wdiguids.txt -level
0x7fffffff -flag 0x7fffffff'
```

Run your scenario. To stop tracing and collect the logs, reconnect to TShell (if needed) and run the following commands.

PowerShell

```
cmdd tracelog.exe '-flush wdiwpp'
cmdd tracelog.exe '-stop wdiwpp'
getd \data\test\wlan\wdiwpp.etl
cmdd 'tracelog -flush WiFiDriver -f C:\data\systemdata\etw\wifidriver.etl'
getd 'C:\data\systemdata\etw\wifidriver.etl'
cmdd 'tracelog -flush WiFiSession'
getd 'C:\Data\SystemData\ETW\WiFi.etl.001'
getd 'C:\Data\SystemData\ETW\WiFi.etl.002'
getd 'C:\Data\SystemData\ETW\WiFi.etl.003'
```

WDI information collection for bugs

Article • 03/14/2023

Bugs in any non-trivial software are inevitable. In the driver development phase, bugs and debugging activities are expected to be a non-trivial part of the endeavor. Bugs may require joint company efforts, as they can be in the operating system, WDI UE, or WDI LE. It is crucial to collect relevant information to quickly narrow in on root causes. The information to collect varies widely depending on the nature of the bugs. Iterations of reproduction of a bug to collect further information are sometimes necessary, but it is critical to reduce these iterations as much as possible. Here are some rules to start with.

OS crash without kernel debugger attached

The operating system generates a crash dump. There are different types of crash dumps, such as mini-dumps and full dumps. While a mini-dump is small, it is often only good for triage. In order to root cause an issue, a full dump is almost always necessary. Enabling full dumps during the driver development and self-hosting phases is recommended. To enable full dumps:

1. From the desktop, right-click on **This PC** and choose **Properties**.
2. On the **Advanced** tab, go to the **Startup and Recovery** section and click on the **Settings...** button.
3. In the **Write debugging information** section, choose **Kernel memory dump** (rather than **Automatic memory dump**).

When an operating system crash occurs, a memory dump file is generated at %windir%\memory.dmp.

OS crash with kernel debugger attached

Developers or QA should have kernel debuggers attached if possible. A kernel debugger can quickly tell what is wrong and which direction to investigate further. The kd command '!analyze –v' is useful as the first command to run after a bug check. This command points to the location inside a module where the crash occurred and the reason (bug check code) for the crash.

When Reset Recovery is invoked

The Reset Recovery feature of WDI builds in the ability to collect live kernel dumps when Reset Recovery is invoked. The kernel dump enables developers to investigate the root causes post-mortem. Live kernel dumps are collected under %windir%\LiveKernelDumps.

Reset Recovery triggers

The current Reset Recovery triggers are listed below. More triggers may be added in the future.

- The UE detects a timeout of a WDI command (M3) sent to the LE.
- The UE detects a timeout of a WDI task (M4) sent to the LE.
- The LE detects and indicates a firmware hang.
- User mode requests a Reset Recovery. This is currently only for loss of internet connectivity. When the NIC is connected and has internet connectivity, wlansvc starts the internet loss state machine. If internet connectivity is lost and not regained within 35 seconds, wlansvc requests that WDI initiates a Reset Recovery. The 35 second timeout is subject to change in the future.

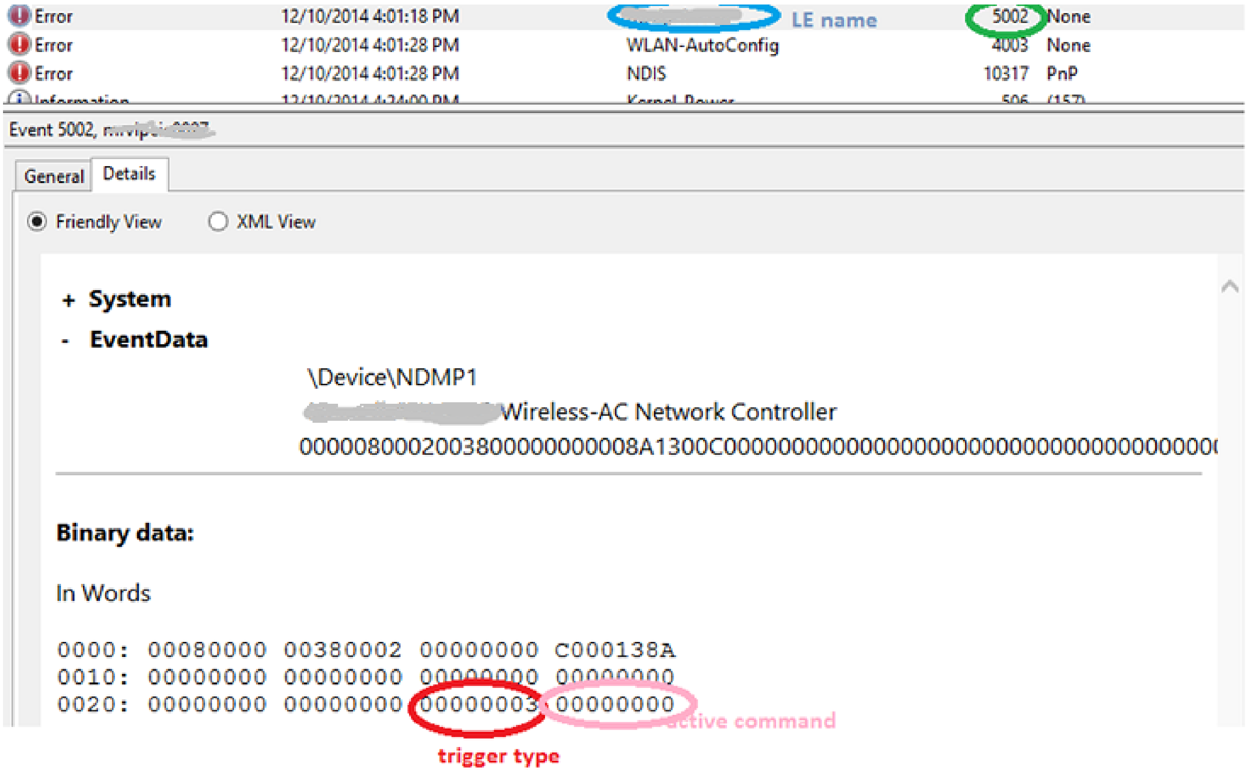
Events for Reset Recovery triggers

WDI calls NDIS to log an Error event to the system event log when it receives a Reset Recovery trigger. The event is in the LE name and the ID is 5002. The last two DWORDs are [TriggerType, ActiveWdiCommand]. The current trigger types are listed below.

- COMMAND_TIMER_ELAPSED (1)
- TASK_TIMER_ELAPSED (2)
- RESET_RECOVERY_OID_REQUEST (3)
- RESET_RECOVERY_FIRMWARE_STALLED (4)

ActiveWdiCommand may be 0 (no active command) if the trigger type is RESET_RECOVERY_OID_REQUEST or RESET_RECOVERY_FIRMWARE_STALLED.

This screenshot is an example view of eventvwr showing system.evtx. The trigger type is 3 and there is no active command.



```
Error        12/10/2014 4:01:18 PM   [redacted]        5002  None
Error        12/10/2014 4:01:28 PM   WLAN-AutoConfig   4003  None
Error        12/10/2014 4:01:28 PM   NDIS              10317 PnP
Information  12/10/2014 4:24:00 PM   Kernel-Power      506   (157)
```

Event 5002, [redacted]

General | Details

● Friendly View ○ XML View

+ System

- EventData

```
\Device\NDMP1
[redacted]Wireless-AC Network Controller
0000080002003800000000008A1300C000000000000000000000000000000000000000
```

Binary data:

In Words

```
0000: 00080000 00380002 00000000 C000138A
0010: 00000000 00000000 00000000 00000000
0020: 00000000 00000000 00000003 00000000
```

When Wi-Fi malfunctions

If there is no crash but Wi-Fi does not function as expected, collect and analyze a trace log. To see if the issue is confined to the WDI UE and LE, a wlan_dbg trace should be included to see the operating system events for the context. Wlan_dbg contains WPP events that require operating system private symbols. The original etl trace should be reserved and included in communication.

Connected Standby issues

Sometimes, the Wi-Fi NIC does not go to low power (device_power_state_Dx). Other times, the device wakes up frequently. A SleepStudy report is helpful in first level triage. SleepStudy events are always on, but only collected if CS sessions are longer than 10 minutes. The events are also persistent (for example, you can inspect the study postmortem). To generate a SleepStudy, run the following command line in an Administrator command prompt.

CMD

```
Powercfg /SleepStudy
```

A report file named SleepStudy-report.html is generated. It should be opened outside %windiir%\system. The report breaks down what modules are keeping the system out of very low power state (DRIPS). It can also further break down which components are keeping the Wi-Fi NIC up (out of Dx).

WDI non-TLV versioning

Article • 03/14/2023

Data structures that are passed between WDI and the IHV miniport and contain a NDIS_OBJECT_HEADER (such as NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS) follow the standard NDIS versioning model. The miniport must check the **Revision** and **Size** fields to ensure the fields it cares about are present, and ignore any extra fields or data without error. Ensure that newer revisions or larger sizes of such structures are not excluded.

All data structures without an NDIS_OBJECT_HEADER (such as WDI_FRAME_METADATA) follow the TLV versioning model, where WDI and the miniport use the size/revision determined by the lowest **WdiVersion** value from NDIS_WDI_INIT_PARAMETERS and NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS.

For example, if WDI sets **WdiVersion** in NDIS_WDI_INIT_PARAMETERS to **WDI_VERSION_1_0**, and the miniport sets **WdiVersion** in NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS to **WDI_VERSION_2_0**, then both WDI and the miniport should use the structure sizes and definitions compatible with **WDI_VERSION_1_0** for all structures without NDIS_OBJECT_HEADER fields. However, in the same situation but with structures that have an **NDIS_OBJECT_HEADER** field, WDI and the miniport may use a larger or newer structure as long as the **Revision** and **Size** fields are correctly set.

User-initiated feedback with IHV trace logging overview

Article • 12/15/2021

This topics in this section outline the steps required for collecting verbose IHV trace logs during user-initiated feedback (UIF) reports submitted via the Feedback tool. There are two separate scenarios in which the Feedback tool will collect logs. The first scenario is a snapshot of the system at the time when the user initiates the feedback. During this time, Windows collects the WMI auto-loggers and some other snapshot data. The second scenario involves the user reproducing the issue. During this feedback, Windows starts loggers with more verbose logging and larger file sizes to capture as much data as possible for the repro. This section describes the expectations for IHVs for each of these feedback scenarios.

In this section:

- Logging scenarios
- User-initiated feedback - normal mode
- User-initiated feedback - repro mode

Logging scenarios

Article • 12/15/2021

The events saved in IHV log files, for both repro mode and the auto-logger, should be appropriately throttled via flags/level/keywords to ensure that at least the past 30 minutes of the log events are always saved. Practically, you should target the 30 minute time period to cover one scan/WFD discovery, one connect/roam event, one disconnect event, several power transitions, and 10 minutes' worth of send and receive data. Because the IHV repro mode log is much larger than the normal IHV auto-logger, more verbose logging is expected.

The following scenarios might help you when logging:

- Getting connected
- Power transitions
- Radio management
- Roaming
- Hang/recovery
- Transition from Connected to Limited

Related links

User-initiated feedback with IHV trace logging

User-initiated feedback - normal mode

Article • 12/15/2021

In the normal user-initiated feedback (UIF) scenario, a user experiences a problem with Wi-Fi and submits a feedback report. This report collects a snapshot of the Wi-Fi subsystem, including Wi-Fi WMI auto-loggers, network statistics, etc. To collect IHV-specific logs, Microsoft provides a WMI auto-logger session with no initial ETW providers. Each IHV adds their set of ETW providers under the Microsoft-provided WMI auto-logger session registry entry. When the UIF report is submitted, the IHV auto-logger ETL is collected and sent to Microsoft for analysis. This log file is implemented using a circular buffer with a somewhat limited size (<= 1MB). The events saved in this log file should be appropriately throttled via flags/level/keywords to ensure that at least the past 30 minutes of the log events are always saved.

Microsoft-provided WMI auto-logger session

Microsoft provides a WMI auto-logger session with no initial ETW providers. When the IHV's drivers are installed, they must add the required WMI provider registry keys under the Microsoft-provided WMI auto-logger session key. The IHV should not change any of the auto-logger session registry values. However, all ETW provider options are available to the IHV including enable level, match any, match all, etc. This logging session always runs and has a limited circular buffer, so IHVs should set the provider EnableLevels appropriately.

The WMI auto-logger session is added to the HKLM registry hive with the following path:

`HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession`

The resulting ETL log file is located here:

`%SystemDrive%\System32\LogFiles\WMI\WifiDriverIHVSession.etl`

IHV driver INF changes

IHVs need to update their driver INF files to add the following registry key values so they can get verbose IHV logs during UIF normal mode. The following snippets provide a template for adding a single ETW provider to the auto-logger session. An IHV may add as many providers as they see fit. In addition, the enable level values are IHV-specific per

ETW provider, so they don't have to necessarily be the same as the Microsoft-defined values (TRACE_LEVEL_CRITICAL, TRACE_LEVEL_ERROR, etc.).

Enable the IHV auto-logger session

Because the IHV auto-logger session is initialized with no ETW providers, it is disabled by default. IHVs are required to enable this session by updating the "Start" value to **1** in their driver's INF file, as shown in this example:

INF

```
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession,Start,%REG_DWORD%,1
```

Add IHV ETW providers

The following snippet shows how to add IHV ETW providers in the INF file:

INF

```
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,Enabled,%REG_DWORD%,1
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,,EnableLevel,%REG_DWORD%,<IHV_LogEnableLevelValue>
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,MatchAnyKeyword,%REG_QWORD%,<IHV_MatchAnyKewordValue>

[The following is optional]
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,MatchAllKeyword,%REG_QWORD%,<IHV_MatchAllKewordValue>

[Strings]
REG_DWORD = 0x00010001
REG_QWORD = 0x000B0001
```

Example values

This example illustrates a Native Wi-Fi custom level setting (enable all) with all Native Wi-Fi keywords:

INF

```
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
{0BD3506A-9030-4f76-9B88-3E8FE1F7CFB6},Enabled,%REG_DWORD%,1
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
{0BD3506A-9030-4f76-9B88-3E8FE1F7CFB6},,EnableLevel,%REG_DWORD%,0x04
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
{0BD3506A-9030-4f76-9B88-
3E8FE1F7CFB6},MatchAnyKeyword,%REG_QWORD%,0x000FFFFF

Standard EnableLevel values:
0x5 - Verbose
0x4 - Informational
0x3 - Warning
0x2 - Error
0x1 - Critical
0x0 – LogAlways
```

Related links

User-initiated feedback with IHV trace logging

User-initiated feedback - repro mode

User-initiated feedback - repro mode

Article • 12/15/2021

User-initiated feedback (UIF) repro mode permits the system to collect more verbose logging while the user reproduces the bug. Like UIF normal mode, this is also accomplished by creating a new WMI logging session with IHV-defined ETW providers. After the repro mode is complete, the verbose logs are gathered and sent to Microsoft for analysis. There are IHV extension points for enabling or disabling verbose firmware logs. Repro logs are intended to be more verbose to be able to track down the reason the customer is having a problem. Therefore, the log file size for the repro mode log is set at a maximum size of 10MB. The IHV should use more verbose settings for ETW provider flags/level/keywords values.

The current UIF repro mode model requires that Microsoft is notified of all provider GUIDs, levels, and flags before the IHV feedback logs are included. Adding the providers to the registry, as indicated in this topic, enables an IHV to test the logs for appropriate levels.

Microsoft-provided WMI auto-logger session

Microsoft provides a WMI auto-logger session with no initial ETW providers. When the IHV's drivers are installed, they must add the required WMI provider registry keys under the Microsoft-provided WMI auto-logger session key. The IHV should not change any of the auto-logger session registry values. However, all ETW provider options are available to the IHV including enable level, match any, match all, etc.

> ⓘ **Important**
>
> The auto-logger is never enabled as an auto-logger. These values are used to validate the repro mode IHV logging described in Testing the repro mode logs. In addition, we might ask users to submit these logs manually by using the `netsh` tool. The provider GUIDs, level, and flags must also be submitted to Microsoft, along with a sample of the logs, so they will be included in repro mode UIFs (see Submitting IHV providers to Microsoft.

The WMI auto-logger session is added to the HKLM registry hive with the following path:

`HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSessionRepro`

The resulting ETL log file is located here:

`%SystemDrive%\System32\LogFiles\WMI\WifiDriverIHVSessionRepro.etl`

IHV driver INF changes

IHVs need to update their driver INF files to add the following registry key values so they can get verbose IHV logs during UIF normal mode. The following snippets provide a template for adding a single ETW provider to the auto-logger session. An IHV may add as many providers as they see fit. In addition, the enable level values are IHV-specific per ETW provider, so they don't have to necessarily be the same as the Microsoft-defined values (TRACE_LEVEL_CRITICAL, TRACE_LEVEL_ERROR, etc.).

Add IHV ETW providers

The following snippet shows how to add IHV ETW providers in the INF file:

INF

```
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,Enabled,%REG_DWORD%,1
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,,EnableLevel,%REG_DWORD%,<IHV_LogEnableLevelValue>
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,MatchAnyKeyword,%REG_QWORD%,<IHV_MatchAnyKewordValue>

[The following is optional]
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
<IHVProviderGUID_1>,MatchAllKeyword,%REG_QWORD%,<IHV_MatchAllKewordValue>

[Strings]
REG_DWORD = 0x00010001
REG_QWORD = 0x000B0001
```

Example values

This example shows WDI UE informational level setting with all keywords:

INF

```
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
{21ba7b61-05f8-41f1-9048-c09493dcfe38},Enabled,%REG_DWORD%,1
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
{21ba7b61-05f8-41f1-9048-c09493dcfe38},,EnableLevel,%REG_DWORD%,0xFF
HKLM,SYSTEM\CurrentControlSet\Control\WMI\Autologger\WifiDriverIHVSession\
{21ba7b61-05f8-41f1-9048-
c09493dcfe38},MatchAnyKeyword,%REG_QWORD%,0x000FFFFF

Standard EnableLevel values:
0x5 - Verbose
0x4 - Informational
0x3 - Warning
0x2 - Error
0x1 - Critical
0x0 – LogAlways
```

ETW control callback

An IHV can register for the ETW control callback in its ETW logging code. This enables the IHV to be notified when ETW providers are enabled, disabled, or a capture control is initiated. This way, the IHV can turn on or off verbose firmware logs for repro mode.

> ⓘ **Note**
>
> If the ETW providers are shared between normal and repro mode, the IHV should key off the IHV-defined EnableLevel, defined in the INF file, to start/stop verbose firmware logs.

ETW callback function

The following snippet shows how to register for the ETW callback. This is only important if the IHV needs to take special action during the start and end of the UIF repro mode (like starting or stopping verbose firmware logging). If multiple ETW providers are used, IHVs might consider only implementing one callback to initiate firmware logging. All firmware logging must be routed to the IHV's ETW trace provider. The diagnostic tools for UIF will only collect traces for the IHV's ETW provider.

There are two ways to enable the ETW callback, depending on how you implemented ETW logging.

1. Manifested ETWs with autogenerated code via `MC.exe`. See Writing an Instrumentation Manifest for more details.
   a. The header in the following snippet (etwtracingevents.h) is an autogenerated ETW event header that was created via `MC.exe`. It is assumed that the ETW events have already been generated, so this topic will not focus on this part.
   b. MCGEN_PRIVATE_ENABLE_CALLBACK_V2 must be defined before including the autogenerated ETW header. Otherwise, the callback will not be called.
2. Registering for the ETW callback via the **EventRegister** API.

a. The ETW callback provider must be passed to the **EventRegister** function when registering the trace provider.

This snippet shows the prototype for the ETW callback function.

```C++
#include <evntprov.h>
extern
VOID
EtwEventControlCallback(
    _In_ LPCGUID SourceId,
    _In_ ULONG ControlCode,
    _In_ UCHAR Level,
    _In_ ULONGLONG MatchAnyKeyword,
    _In_ ULONGLONG MatchAllKeyword,
    _In_opt_ PEVENT_FILTER_DESCRIPTOR FilterData,
    _Inout_opt_ PVOID CallbackContext
    );
```

The following code is only required if you used autogenerated ETW events using the `MC.exe` tool.

```C++
#define MCGEN_PRIVATE_ENABLE_CALLBACK_V2 EtwEventControlCallback

#include "etwtracingEvents.h" // Generated from manifest - This must come
                              // after MCGEN_PRIVATE_ENABLE_CALLBACK_V2 is
                              // defined
```

The *ControlCode* parameter of the ETW callback indicates when the provider is enabled or disabled. The values are defined in `<evntrace.h>` and have the following values:

```C++
#define EVENT_CONTROL_CODE_DISABLE_PROVIDER 0
#define EVENT_CONTROL_CODE_ENABLE_PROVIDER  1
#define EVENT_CONTROL_CODE_CAPTURE_STATE    2
```

EVENT_CONTROL_CODE_ENABLE_PROVIDER

This flag enables the ETW provider and indicates that the UIF repro mode session has started. This should be used to initiate verbose firmware logging and/or packet logging.

EVENT_CONTROL_CODE_DISABLE_PROVIDER

This flag disables the ETW provider and indicates that the UIF repro mode session has ended. The IHV's implementation should flush and reset firmware logs at this point if the *Level* parameter matches the IHV-specified UIF repro mode level in the INF file (0xFF in the following section's sample).

EVENT_CONTROL_CODE_CAPTURE_STATE

This flag requests that the provider logs its state information. This is generally called to flush the in-memory logs to disk. The IHV's implementation should flush and reset firmware logs at this point if the *Level* parameter matches the IHV-specified UIF repro mode level in the INF file (0xFF in the following section's sample).

Sample code

The following is a sample ETW callback implementation that can be used as a template to enable verbose driver and firmware logging for UIF repro mode scenarios.

> ⓘ Note
>
> IHVs need to flush any pending firmware logs for both the **EVENT_CONTROL_CODE_CAPTURE_STATE** and **EVENT_CONTROL_CODE_DISABLE_PROVIDER** control codes.

After the **EVENT_CONTROL_CODE_CAPTURE_STATE** is invoked, the UIF diagnostics tool invokes the ETW callback two more times with the **EVENT_CONTROL_CODE_ENABLE_PROVIDER** control code. Therefore, to avoid re-enabling the firmware logging, the state machine moves from the *ReproModeStateCaptured* state to the *ReproModeStateFinal* state before moving back to the *ReproModeStateNotStarted* state. The **EVENT_CONTROL_CODE_DISABLE_PROVIDER** control code is only used to disable the provider. This is not part of the UIF process, but still needs to be honored.

IHVs should change the **IHV_ETW_REPRO_MODE_LEVEL** value in the following example to match the repro mode level set in the INF file.

C++

```cpp
#define IHV_ETW_REPRO_MODE_LEVEL 0xFF // This value must match the repro
mode
                                      // EnableLevel INF value
```

```
typedef enum _EtwReproModeState
{
    ReproModeStateNotStarted = 0,
    ReproModeStateStarted,
    ReproModeStateCaptured,
    ReproModeStateFinal
} EtwReproModeState;

static EtwReproModeState g_eReproModeLoggingEnabled =
ReproModeStateNotStarted;

VOID
EtwEventControlCallback(
    _In_ LPCGUID SourceId,
    _In_ ULONG ControlCode,
    _In_ UCHAR Level,
    _In_ ULONGLONG MatchAnyKeyword,
    _In_ ULONGLONG MatchAllKeyword,
    _In_opt_ PEVENT_FILTER_DESCRIPTOR FilterData,
    _Inout_opt_ PVOID CallbackContext
    )
{
    UNREFERENCED_PARAMETER(SourceId);
    UNREFERENCED_PARAMETER(MatchAnyKeyword);
    UNREFERENCED_PARAMETER(MatchAllKeyword);
    UNREFERENCED_PARAMETER(FilterData);
    UNREFERENCED_PARAMETER(CallbackContext);

    switch(ControlCode)
    {
        case EVENT_CONTROL_CODE_ENABLE_PROVIDER:
            if (Level == IHV_ETW_REPRO_MODE_LEVEL)
            {
                switch(g_eReproModeLoggingEnabled)
                {
                    case ReproModeStateNotStarted:
                        //
                        // Enable verbose Firmware logs.
                        //
                        g_eReproModeLoggingEnabled = ReproModeStateStarted;
                        break;

                    case ReproModeStateCaptured:
                        //
                        // The diagnostic tools will invoke the callback
after
                        // the capture with
EVENT_CONTROL_CODE_ENABLE_PROVIDER
                        // twice.
                        //
                        g_eReproModeLoggingEnabled = ReproModeStateFinal;
                        break;

                    case ReproModeStateFinal:
                        //
                        // The state machine is now complete, reset the
state.
                        //
                        g_eReproModeLoggingEnabled =
ReproModeStateNotStarted;
                        break;

                    case ReproModeStateStarted:
                    default:
                        break;
                }
            }
            break;

        case EVENT_CONTROL_CODE_DISABLE_PROVIDER:
            if (g_eReproModeLoggingEnabled == ReproModeStateStarted)
            {
                //
                // Merge verbose firmware logs into ETW log (if not done
already).
                // Disable verbose firmware logs
                //
                g_eReproModeLoggingEnabled = ReproModeStateNotStarted;
            }
            break;

        case EVENT_CONTROL_CODE_CAPTURE_STATE:
            if (Level == IHV_ETW_REPRO_MODE_LEVEL &&
                g_eReproModeLoggingEnabled == ReproModeStateStarted)
            {
                //
                // Merge verbose firmware logs into ETW log (if not done
already).
                // Disable verbose firmware logs
                //
                g_eReproModeLoggingEnabled = ReproModeStateCaptured;
            }
            break;
    }
}
```

Testing the repro mode logs

To test the IHV repro mode logs, the following commands can be used to start and stop capture.

> ⓘ Note
>
> The resulting ETL file will contain some OS logs.

- netsh wlan IHV startlogging
- netsh wlan IHV stoplogging

These commands are also used by customers to manually collect logs from a device.

Submitting IHV providers to Microsoft

The final step for IHVs to submit repro mode user-initiated feedback is to contact Microsoft and supply the requested provider GUIDs, levels, and flags along with sample log data for review. Once the logging is approved, the providers will be added to the user-initiated feedback system.

> ⓘ **Note**
>
> Any modifications to the provider GUIDs, levels, or flags after submission will have no effect on the UIF logs.

Related links

User-initiated feedback with IHV trace logging

User-initiated feedback - normal mode

WDI TLV generator/parser topics

Article • 03/14/2023

The TLV generator and parser shared library allows IHV drivers to correctly parse TLVs into strongly typed C/C++ structures, or conversely generate a TLV byte blob from the structures. It also handles the versioning semantics so the IHV does not need to.

In this section:

TLV parser interface overview

TLV generator interface overview

TLV generator/parser memory interface

TLV generator/parser special members

Adding the WDI TLV generator/parser to your driver

TLV generator/parser XML semantics and syntax

TLV versioning

TLV dumpers

WDI TLV parser interface overview

Article • 03/14/2023

Callee allocation model

An entry point within the driver receives a message or indication that contains TLVs. After the code extracts the message ID and determines if it is an ID that it wants to handle, it calls the generic parse routine and passes the TLV blob (after advancing past the WDI_MESSAGE_HEADER) to parse the TLVs into a C-structure.

```C
ndisStatus = Parse(
    cbBufferLength,
    pvBuffer,
    messageId,
    &Context,
    &pParsed);
```

After checking the return for errors, the code can cast the output buffer (*pParsed*) into a concrete type, such as in the below example.

```C
((WDI_INDICATION_BSS_ENTRY_LIST_PARAMETERS*)pParsed)
```

After the caller is finished with the parsed data, the caller must return the memory back to the parser. The parser needs to know the original message ID used to allocate so it frees the correct data.

```C
FreeParsed(messageId, pParsed);
pParsed = NULL;
```

Caller allocation model

In this model, the caller has already determined the correct specific TLV to parse and is possibly using a stack local to avoid allocations on the heap. The caller creates the local and calls a specific parse routine. The API does not need the message ID, and the parameter is strongly typed with one less level of indirection.

C

```
WDI_GET_ADAPTER_CAPABILITIES_PARAMETERS adapterCapabilitiesParsed;

ndisStatus = ParseWdiGetAdapterCapabilities(
    cbBufferLength,
    pvBuffer,
    &Context
    &adapterCapabilitiesParsed);
```

After the caller is finished using the structure, the caller should give the parser a chance to clean up any allocation it made during parsing, and wipe the structure so it is ready to be reused. The parameter is strongly typed, so the callee does not need any additional parameters.

C

```
CleanupParsedWdiGetAdapterCapabilities(&adapterCapabilitiesParsed);
```

After calling the CleanupParse API, all data in the structure is invalid.

Some messages do not have any associated data. For completeness of the API, appropriately named Parse methods are provided. These methods validate that the byte stream is empty. Typedefs are provided for the parameter type, but callers can also pass NULL for the out parameter if they use the Caller Allocation Model. In all cases, the Parser avoids any allocations by returning a constant empty parse structure. Callers should never write into this returned empty structure (hence the only field is named **_Reserved**). These messages are documented as "No additional data. The data in the header is sufficient".

Message direction

Most messages have a different format for their M1 versus their M0, M3, or M4. To accommodate for this, such messages have different parse and generate APIs. For M1 messages, the APIs follow the naming convention of Parse<*MessageName*>ToIhv or Generate<*MessageName*>ToIhv. For M0, M3, or M4 messages, the APIs follow the naming convention of Parse<*MessageName*>FromIhv or Generate<*MessageName*>FromIhv. However, to simplify code in the IHV miniport, defines are added to alias Parse<*MessageName*> to Parse<*MessageName*>ToIhv and Generate<*MessageName*> to Generate<*MessageName*>FromIhv. IHV code only needs to be aware of this aliasing if it needs to parse its own M3, or generate an M1.

Error codes

The TLV parser generator can return several different NDIS_STATUS codes. For more information, look at the WPP trace logs. The logs should always indicate the root cause. Here is a list of the most common error codes and what they mean.

NDIS_STATUS_INVALID_DATA	When parsing, this indicates that a fixed sized TLV is of the incorrect size. For lists, this means the overall size is not an even multiple of the individual element size, or there are more elements than there should be. This could also mean a list contained 0 elements, when 1 or more is required. If 0 elements is desired, then *Optional_IsPresent* should be set to false (the TLV header should not be in the byte stream).
NDIS_STATUS_BUFFER_OVERFLOW	When generating, this indicates that due to the number of elements in an array (list), it overflows the 2 byte **Length** field within the TLV header. You should reduce the number of elements. This can also occur when an outer TLV has too many (or too large of) inner TLVs, again overflowing the 2 byte **Length** field of the header. When parsing, this indicates a TLV header's **Length** field is larger than the outer TLV or the byte stream.
NDIS_STATUS_FILE_NOT_FOUND	When parsing, this indicates that a required TLV is not present in the byte stream. It is usually a bug with the generator of the byte stream.
NDIS_STATUS_RESOURCES	When generating, this indicates that the allocator failed.
NDIS_STATUS_UNSUPPORTED_REVISION	When parsing or generating, the **Context** parameter is NULL, or the **PeerVersion** is less than **WDI_VERSION_MIN_SUPPORTED**.

WDI TLV generator interface overview

Article • 03/14/2023

C++ overloaded function model

In this model, there is only one function call to generate a TLV byte array from your data structure.

```C++
WDI_INDICATION_BSS_ENTRY_LIST_PARAMETERS BssEntryList = ...;
BYTE* pOutput = NULL;
ULONG length = 0;
NDIS_STATUS ndisStatus = NDIS_STATUS_SUCCESS;

ndisStatus = Generate(
    &BssEntryList,
    cbHeaderLength,
    &Context,
    &length,
    &pOutput);
```

The second parameter can be very helpful. Sometimes, the TLV buffer is packed into a bigger data structure, and this parameter allows you to pre-reserve space at the beginning of the buffer for that header. The correct value for *cbHeaderLength* is often `sizeof(WDI_MESSAGE_HEADER)`.

For messages that have no associated data, there are still overloaded Generate APIs, but the first parameter is optional and may simply be passed in as `(EmptyMessageStructureType*)NULL`.

When you are done with the TLV data contained in *pOutput*, you must call back into the library to release the buffer.

```C++
    FreeGenerated(pOutput);
    pOutput = NULL;
```

C-style function model

In this model, there is a specific Generate routine for each top-level message or structure because C does not support overloaded functions. Otherwise, it behaves the same as the C++ model.

C

```
ndisStatus = GenerateWdiGetAdapterCapabilities(
    &adapterCapabilities,
    (ULONG)sizeof(WFC_COMMAND_HEADER),
    &Context,
    &length,
    &pOutput);
```

When you are done with the TLV byte array, call back to release the memory in the same way as the C++ model.

C

```
    FreeGenerated(pOutput);
    pOutput = NULL;
```

WDI TLV generator/parser memory interface

Article • 03/14/2023

The parser and generator internally use C++ with new/delete. This simplifies several implementation details. This means that consumers of the library must provide overloaded operator implementations of these APIs when linking to the library. This is the only C++ dependency that your code must take.

All APIs that do any allocations take a parameter *Context* typed as PCTLV_CONTEXT which has 2 fields: a ULONG_PTR named **AllocationContext** and a ULONG named **PeerVersion**. The **AllocationContext** field is passed through to the overloaded `new` operator. This allows consumers of the APIs to customize the allocation in various ways. For more information about the TLV_CONTEXT parameter, see WDI TLV versioning.

Warning Although you may be tempted to skip calling the library's cleanup routines (such as FreeParsed, CleanupParsed, and FreeGenerated), do not skip calling them! It might work on some code paths, but will lead to hard-to-diagnose memory leaks.

Here is a sample overloaded operator.

C++

```
/*++
Module Name:
    sample.cpp
Abstract:
    Contains sample code to override C++ new/delete for use with TLV
parser/generator library
Environment:
    Kernel mode
--*/
#include "precomp.h"

#define TLV_POOL_TAG (ULONG) '_VLT'

void* __cdecl operator new(size_t Size, ULONG_PTR AllocationContext)
/*++
  Override C++ allocation operator.
--*/
{
    PVOID pData = ExAllocatePoolWithTag(NonPagedPoolNx, Size, TLV_POOL_TAG);
    UNREFERENCED_PARAMETER(AllocationContext);
    if (pData != NULL)
    {
        RtlZeroMemory( pData, Size);
    }
    return pData;
}

void __cdecl operator delete(void* pData)
/*++
  Override C++ delete operator.
--*/
{
    if (pData != NULL)
    {
        ExFreePoolWithTag(pData, TLV_POOL_TAG);
    }
}
```

WDI TLV generator/parser special members

Article • 03/14/2023

Optional members

For any TLV that has optional child TLV members, the parent has one field named **Optional**. Within that field, there is one Boolean field for each optional child named **<*child_name*>_IsPresent**, which is set to TRUE by the parser if the child is present, and FALSE otherwise. Similarly, the generation APIs expect the field to be TRUE if it should be present in the TLV byte stream, and FALSE otherwise.

```C++
WDI_SET_FIRMWARE_CONFIGURATION_PARAMETERS fwConfig = { 0 };
NDIS_STATUS status;
status = ParseWdiSetAdapterConfiguration(
    pNdisRequest->DATA.METHOD_INFORMATION.InputBufferLength -
        sizeof(WDI_MESSAGE_HEADER),
    (PUINT8)pNdisRequest->DATA.METHOD_INFORMATION.InformationBuffer +
        sizeof(WDI_MESSAGE_HEADER),
    0,
    &fwConfig);

if (status == NDIS_STATUS_SUCCESS)
{
    if (fwConfig.Optional.MacAddress_IsPresent)
    {
        // Safe to use fwConfig.MacAddress
        fwConfig.MacAddress;
    }
}
```

Array members

When multiple children of the same type appear within the same parent (for example, <container />'s *isCollection* attribute), the parser and generator use a special structure to represent the array: ArrayOfElements. For C++ clients, this is a strongly typed template structure with clean up on destruction semantics. For C clients, explicitly named structures are created (for example, ArrayOfElementsOfUINT8). However, these structures are not automatically cleaned up because C does not support destructors, so users of the C APIs must be careful not to introduce memory leaks (or double-frees).

There are two important fields within ArrayOfElements: **ElementCount** and **pElements**. **ElementCount** is the count of elements within the array. **pElements** is a C-Style array of the elements. The elements can be iterated over as shown in this sample.

C++

```
for (UINT32 i = 0;
    i < pConnectTaskParameters->ConnectParameters.
            MulticastCipherAlgorithms.ElementCount;
    i++)
{
    // Safe to use pElements[i]
    pConnectTaskParameters->ConnectParameters.MulticastCipherAlgorithms.
        pElements[i];
}
```

The third field, **MemoryInternallyAllocated**, is used internally by the parser/generator. It should not be modified by the IHV.

Adding the WDI TLV generator/parser to your driver

Article • 03/14/2023

To add the WDI TLV generator/parser to your driver, follow these steps.

1. Add this include after dot11wdi.h and wditypes.hpp.

   `#include "TlvGeneratorParser.hpp"`

2. Add this library to the linker.

   `TLVGeneratorParser.lib`

3. Define, create, and write your memory APIs (overloaded operator new/delete).

4. Start calling the APIs.

WDI TLV generator/parser XML semantics

Article • 03/14/2023

TLV (Type-Length-Value) is a protocol design where each bit of data is contained in a stream of bytes that has a standard Type and Length header.

The TLV generator/parser XML file is a list of messages, containers (TLVs), and property groups (structs). This topic covers the XML syntax.

`<message />`

Describes a single top-level WDI message. There are only parser/generator functions for these message entries.

Attributes

- `commandId` - Symbolic constant that must be defined in dot11wdi.h.
- `type` - Type name to be exposed to the code (you use this type when calling into parser/generator functions).
- `description` - Description of the command.
- `direction` – Indicates whether this message describes the TLV stream as it goes from WDI to the IHV miniport as part of an M1 (called "ToIhv"), describes the TLV stream as it goes from the IHV miniport to WDI as an M0, M3, or M4 (called "FromIhv"), or it goes in both directions (called "Both"). See *Message Direction* in WDI TLV parser interface overview.

Content

List of container references (`<containerRef />`). These are the different TLVs that make up the message. They are references to types defined in the `<containers />` section.

Example

XML

```xml
<message commandId="WDI_SET_P2P_LISTEN_STATE"
         type="WDI_SET_P2P_LISTEN_STATE_PARAMETERS"
         description="Parameters to set listen state."
         direction="ToIhv">
  <containerRef id="WDI_TLV_P2P_CHANNEL_NUMBER"
                name="ListenChannel"
                optional="true"
                type="WFDChannelContainer" />
  <containerRef id="WDI_TLV_P2P_LISTEN_STATE"
                name="ListenState"
                type="P2PListenStateContainer" />
</message>
```

`<containerRef />`

Reference to a `<container />` defined in the `<containers />` section.

`<containerRef />` Attributes

- `id` - TLV ID that must be defined in wditypes.h.
- `name` - The name of the variable in the parent structure.
- `optional` - Specifies whether or not it is an optional field. False by default. Generated code enforces "optional-ness".
- `multiContainer` – Specifies whether or not the generated code should expect multiple TLVs of the same type. False by default. If false, generated code enforces that only one is present.
- `type` - Reference to a specific element’s "name" attribute in the `<containers />` section.
- `versionAdded` - Part of versioning. Indicates that this TLV container should not appear in byte streams to/from peers with a version less than the one indicated in this attribute.
- `versionRemoved` - Part of versioning. Indicates that this TLV container should not appear in byte streams to/from peers with a version greater than or equal to the one indicated in this attribute.

`<containerRef />` Content

None.

`<containerRef />` Example

XML

```xml
<containerRef id="WDI_TLV_P2P_CHANNEL_NUMBER"
              name="ListenChannel"
              optional="true"
              type="WFDChannelContainer"/>
```

`<containers />`

Describes all containers/TLVs used in WDI messages. Containers can be considered TLV buckets. There are 2 types: `<container />` and `<aggregateContainer />`.

`<container />`

TLV Container for a single structure reference or named type. It is statically sized, but may be a C-style array as long as it is statically sized.

`<container />` Attributes

- `name` - ID that is referenced by WDI messages/other containers.
- `description` - Friendly description of what the container is for.
- `type` – Type name to be exposed to the code.
- `isCollection` - Specifies whether or not generated code should expect many of the same size element within the same TLV (C-style array). The default is false (only expect one element of the given type).
- `isZeroValid` - Only valid when `isCollection` is true. Determines whether a zero element array is allowed. This is useful when the TLV stream needs to distinguish between an optional TLV that is not present versus one that is present but has zero length (like SSIDs). Since this distinction is rare, the default is false.

`<container />` Contents

One of `<groupRef />` or `<namedType />`.

`<container />` Example

XML

```xml
<container name="P2PListenStateContainer"
           description="Container for P2P Listen State setting."
           type="WDI_P2P_LISTEN_STATE_CONTAINER">
  <namedType name="ListenState"
             type="WDI_P2P_LISTEN_STATE"
             description="P2P Listen State."/>
</container>
```

`<groupRef />`

Reference to a property group (structure) defined in the `<propertyGroups />` section.

`<groupRef />` Attributes

- `name` - Name of the structure in the parent structure.
- `ref` - Reference to a named structure in a `<propertyGroups />` section.
- `description` – Friendly descriptor of what the structure is used for.

`<groupRef />` Content

None.

`<groupRef />` Examples

XML

```xml
<container name="WFDChannelContainer"
           description="Container for a Wi-Fi Direct channel."
           type="WDI_P2P_CHANNEL_CONTAINER">
  <groupRef name="Channel"
            ref="WFDChannelStruct"
            description="Wi-Fi Direct Channel." />
</container>
```

`<namedType />`

Reference to a raw type exposed by wditypes.hpp or dot11wdi.h. Uses default serializer (memcpy), so use at your own risk because of padding issues.

`<namedType />` Attributes

- `name` - Name of the structure in the parent structure.
- `type` - Type name to use in the actual code.
- `description` – Friendly description of what the structure is used for.

`<namedType />` Content

None.

`<namedType />` Example

XML

```xml
<container name="P2PListenStateContainer"
           description="Container for P2P Listen State setting."
           type="WDI_P2P_LISTEN_STATE_CONTAINER">
  <namedType name="ListenState"
             type="WDI_P2P_LISTEN_STATE"
             description="P2P Listen State."/>
</container>
```

`<aggregateContainer />`

TLV Container for many different containers. This is used for handling nested TLVs.

`<aggregateContainer />` Attributes

- `name` - ID that is referenced by WDI messages/other containers.
- `description` – Friendly description of what the container is for.
- `type` - Type name to be exposed to the code.

`<aggregateContainer />` Content

List of `<containerRef />`.

`<aggregateContainer />` Example

XML

```xml
<aggregateContainer
    name="P2PInvitationRequestInfoContainer"
    type="WDI_P2P_INVITATION_REQUEST_INFO_CONTAINER"
    description="Generic container for Invitation Request-related
containers.">
  <containerRef
    id="WDI_TLV_P2P_INVITATION_REQUEST_PARAMETERS"
    type="P2PInvitationRequestParamsContainer"
    name="RequestParams" />
  <containerRef
    id="WDI_TLV_P2P_GROUP_BSSID"
    type="MacAddressContainer"
    name="GroupBSSID"
    optional="true" />
  <containerRef
    id="WDI_TLV_P2P_CHANNEL_NUMBER"
    type="WFDChannelContainer"
    name="OperatingChannel"
    optional="true" />
  <containerRef
    id="WDI_TLV_P2P_GROUP_ID"
    type="P2PGroupIDContainer"
    name="GroupID" />
</aggregateContainer>
```

`<propertyGroups />`

Describes all structures used in all containers. Structures can either be used by a `<container />`, or referenced by another `<propertyGroup />` (nested structures). They are defined independently of TLVs containers so they can be re-used. They do not have a TLV header.

These definitions are necessary as they help to solve padding issues with structures and gives the code generator instructions on how to interpret the data.

> ⓘ Note
>
> Order matters here. All data offsets are implied based on the property group description, and data is written/parsed in the order it is defined here. These structures have to be defined here.

Primitive Field Types (`<bool/> <uint8/> <uint16/> <uint32/> <int8/> <int16/> <int32/>`)

These are the available primitive types, and are converted/marshalled appropriately by the generated code.

Attributes for Primitive Field Types

- `name` - Field name in the parent structure.
- `description` – Friendly description of what the property is for.
- `count` - How many of the given property there are. Default is one. Values greater than one make this property into a statically sized array in the code.

Contents for Primitive Field Types

None

`<propertyGroup />`

An individual structure.

`<propertyGroup />` Attributes

- `name` - ID that is referenced by WDI messages/other containers.
- `description` – Friendly description of what the property group is for.
- `type` - Type name to be exposed to the code.

`<propertyGroup />` Contents

There are several possible property types (struct fields).

- `<bool/> <uint8/> <uint16/> <uint32/> <int8/> <int16/> <int32/>`
- `<groupRef />`
- `<namedType />`

`<propertyGroup />` Example

XML

```xml
<propertyGroup name="P2PDiscoverModeStruct"
               type="WDI_P2P_DISCOVER_MODE"
               description="Structure definition for P2P Discover Mode
Parameters">
  <namedType name="DiscoveryType"
             type="WDI_P2P_DISCOVER_TYPE"
             description="Type of discovery to be performed by the port."/>
  <bool name="ForcedDiscovery"
        description="A flag indicating that a complete device discovery is
required. If this flag is not set, a partial discovery may be performed." />
  <namedType name="ScanType"
             type="WDI_P2P_SCAN_TYPE"
             description="Type of scan to be performed by port in scan
phase." />
  <bool name="ScanRepeatCount"
        description="How many times the full scan procedure should be
repeated. If set to 0, scan should be repeated until the task is aborted by
the host."/>
</propertyGroup>
<propertyGroup name="P2PDeviceInfoParametersStruct"
               type="WDI_P2P_DEVICE_INFO_PARAMETERS"
               description="Structure definition for P2P Device Information
Parameters.">
```

```
  <uint8 count="6"
         name="DeviceAddress"
         description="Peer's device address." />
  <uint16 name="ConfigurationMethods"
          description="Configuration Methods supported by this device." />
  <groupRef name="DeviceType"
            description="Primary Device Type."
            ref="WFDDeviceType" />
</propertyGroup>
```

WDI TLV versioning

Article • 03/14/2023

To maintain backwards compatibility, both WDI and the miniport use the TLV stream as a versioning boundary. The producer of the TLV byte stream must always generate a backwards compatible TLV and not include any newly added fields. This is accomplished by adding a **PeerVersion** to the *Context* parameter. This field should be initialized by the caller to the *WdiVersion* received during initialization.

Here is the type definition of the *Context* parameter, which is passed into every Parse and Generate API.

C++

```
typedef struct _TLV_CONTEXT
{
    ULONG_PTR   AllocationContext;
    ULONG       PeerVersion;
} TLV_CONTEXT, *PTLV_CONTEXT;
typedef const TLV_CONTEXT * PCTLV_CONTEXT;
```

AllocationContext is unmodified by the Parse and Generate APIs and continues to be passed through to the miniport-provided operator `new` callback. For more information, see WDI TLV generator/parser memory interface.

If a WDI-based single-binary driver runs against an older version of WDI, the generator in the miniport uses the **PeerVersion** to generate the older byte stream. Conversely, the parser consumes the older byte stream based on the **PeerVersion** and converts it into the new data structures.

If a miniport driver does not use the TLV parser generator library and instead writes their own TLV parser and generator, and the desire is to have a single binary running only older OS versions (and thus old versions of WDI), they must include this capability also. Their parser must accept the TLV grammar produced by older WDI, and their generator must only generate TLVs according to the older grammar.

The XML has been augmented to support this versioning with two attributes allowed on containerRefs: *versionAdded* and *versionRemoved*. This is what drives the parser and generator to adjust the byte stream according to the peer version.

Note The parser and generator assume that they are always linked with WDI_VERSION_LATEST. The miniport should always pass WDI_VERSION_LATEST for NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS::**WdiVersion** when calling

NdisMRegisterWdiMiniportDriver rather than using a specific version, like WDI_VERSION_1_0, as they will become out of date and cause problems with the TLV parser generator because the other end might send a byte stream that is unexpected.

WDI TLV dumpers

Article • 03/14/2023

The parser generator library has routines to decode TLV byte arrays into trace statements.

C++

```
    typedef _Function_class_( TlvDumperCallback ) void( __stdcall
*TlvDumperCallback )(_In_ UINT_PTR Context, _In_z_ _Printf_format_string_
PCSTR Format, ...);

    void __stdcall TraceUnknownTlvByteStream(
        _In_ ULONG PeerVersion,
        _In_ ULONG BufferLength,
        _In_reads_bytes_( BufferLength ) UINT8 const * pBuffer );

    void __stdcall TraceMessageTlvByteStream(
        _In_ ULONG MessageId,
        _In_ BOOLEAN fToIhv,
        _In_ ULONG PeerVersion,
        _In_ ULONG BufferLength,
        _In_reads_bytes_( BufferLength ) UINT8 const * pBuffer );

    void __stdcall DumpUnknownTlvByteStream(
        _In_ ULONG PeerVersion,
        _In_ ULONG BufferLength,
        _In_reads_bytes_( BufferLength ) UINT8 const * pBuffer,
        _In_opt_ ULONG_PTR Context,
        _In_opt_ TlvDumperCallback pCallback );

    void __stdcall DumpMessageTlvByteStream(
        _In_ ULONG MessageId,
        _In_ BOOLEAN fToIhv,
        _In_ ULONG PeerVersion,
        _In_ ULONG BufferLength,
        _In_reads_bytes_( BufferLength ) UINT8 const * pBuffer,
        _In_opt_ ULONG_PTR Context,
        _In_opt_ TlvDumperCallback pCallback );
```

If you only need WPP tracing, use the Trace APIs as they are optimized to have the smallest impact to code size as well as log size (fewer strings in the ETL file). If you need a more general purpose dumper, use the Dump APIs as they include WPP tracing and also include a callback routine. The stub driver has an example of using this callback routine to redirect the output to the kernel debugger via DebugPrint APIs.

Unlike the Parse and Generate APIs, the dumper is very permissive. It attempts to make sense of the TLV bytes as best as it can, regardless of the canonical form for a given message or TLV. This means the dumper might correctly decode and dump something that the parser rejects.

Warning If the dumper successfully decodes the bytes into a human readable format, it does not mean the bytes are a well-formed TLV.

Like the Parse APIs, the *pBuffer* pointer and *BufferLength* parameters should exclude any headers and point directly at the first TLV.

The Message variants of the APIs include the message ID and the message direction to better disambiguate the TLV. This is helpful because the same TLV ID can be decoded in different ways depending upon context. For example, **WDI_TLV_BSSID** can directly contain a **WDI_MAC_ADDRESS** when part of OID_WDI_TASK_SCAN, or it can contain a list of **WDI_MAC_ADDRESS** when part of **WDI_TLV_P2P_ATTRIBUTES**.

WDI_TLV_ACCESS_NETWORK_TYPE

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ACCESS_NETWORK_TYPE is a TLV that contains an Access Network Type.

TLV Type

0x100

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The Access Network Type to be used in probe requests for the network being connected to.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ACTION_FRAME_BODY

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ACTION_FRAME_BODY is a TLV that contains the body of an Action Frame.

TLV Type

0xBE

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the body of an Action Frame.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ACTION_FRAME_DEVICE_CONTEXT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ACTION_FRAME_DEVICE_CONTEXT is a TLV that contains an Action Frame device context.

TLV Type

0xAC

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains an Action Frame device context.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ADAPTER_NLO_SCAN_MODE

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ADAPTER_NLO_SCAN_MODE is a TLV that indicates whether scans should be performed in active or passive mode.

TLV Type

0x125

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	WDI_SCAN_TYPE value that indicates whether scans should be performed in active or passive mode.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ADAPTER_RESUME_REQUIRED

Article • 03/14/2023

WDI_TLV_ADAPTER_RESUME_REQUIRED is a TLV that specifies if adapter resume is required.

TLV Type

0xB7

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	Specifies if adapter resume is required. Valid values are 0 (not required) and 1 (required). If set to 1, the firmware requires OS assistance to resume its context.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ADDITIONAL_BEACON_IES

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ADDITIONAL_BEACON_IES is a TLV that contains additional beacon IEs.

TLV Type

0x98

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The array of beacon IEs. The Wi-Fi Direct port must add these additional IEs to the beacon packets when it is acting as a Group Owner. These are ignored when the Wi-Fi Direct port is operating in device or client mode.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ADDITIONAL_IES

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ADDITIONAL_IES is a TLV that contains additional Information Element (IE) settings.

TLV Type

0x8A

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ADDITIONAL_BEACON_IES		X	An array of beacon IEs. The Wi-Fi Direct port must add these additional IEs to the beacon packets when it is acting as a Group Owner. This is ignored when the Wi-Fi Direct port is operating in device or client mode.
WDI_TLV_ADDITIONAL_PROBE_RESPONSE_IES		X	An array of probe response IEs. The Wi-Fi Direct port must add these additional IEs to the probe response packets when it is acting as a Wi-Fi Direct device or Group Owner. This is ignored when the Wi-Fi Direct port is operating in client mode.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ADDITIONAL_PROBE_REQUEST_DEFAULT_IES		X	An array of additional probe request IEs. This offset is relative to the start of the buffer that contains this structure. The Wi-Fi Direct port must add these additional IEs to the probe request packets that it transmits. Note that a Wi-Fi Direct discover request may override the default probe request IEs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ADDITIONAL_PROBE_REQUEST_DEFAULT_IES

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ADDITIONAL_PROBE_REQUEST_DEFAULT_IES is a TLV that contains additional probe request IEs.

TLV Type

0x70

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of probe request IEs. The Wi-Fi Direct port must add these additional IEs to transmitted probe request packets. **Note** A Wi-Fi Direct Discover Request may override the default probe request IEs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ADDITIONAL_PROBE_RESPONSE_IES

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ADDITIONAL_PROBE_RESPONSE_IES is a TLV that contains probe response IEs.

TLV Type

0x93

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The array of probe response IEs. The Wi-Fi Direct port must add these additional IEs to the probe response packets when it is acting as a Wi-Fi Direct device or Group Owner. This member is ignored when the Wi-Fi Direct port is operating in client mode.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ALLOWED_BSSIDS_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ALLOWED_BSSIDS_LIST is a TLV that contains a list of BSSIDs that are allowed to be used for association.

TLV Type

0xC2

Length

The size (in bytes) of the array of WDI_MAC_ADDRESS structures. The array must contain 1 or more structures.

Values

Type	Description
WDI_MAC_ADDRESS[]	A list of BSSIDs that are allowed to be used for association.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ANQP_ELEMENTS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ANQP_ELEMENTS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ANQP_QUERY_PARAMETERS

Article • 03/14/2023

WDI_TLV_ANQP_QUERY_PARAMETERS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ANQP_QUERY_STATUS

Article • 03/14/2023

WDI_TLV_ANQP_QUERY_STATUS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_AP_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_AP_ATTRIBUTES is a TLV that contains the attributes of an access point.

TLV Type

0x23

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_AP_CAPABILITIES			The access point capabilities.
WDI_TLV_UNICAST_ALGORITHM_LIST			The supported unicast algorithms.
WDI_TLV_MULTICAST_DATA_ALGORITHM_LIST			The supported multicast data algorithms.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_AP_BAND_CHANNEL

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_AP_BAND_CHANNEL is a TLV that specifies access point band and channel information.

Note This TLV was added in Windows 10, version 1511, WDI version 1.0.10.

TLV Type

0x127

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BANDID			Specifies the identifier for the band.
WDI_TLV_CHANNEL_INFO_LIST		X	Specifies a list of channels to start the access point on.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

See also

OID_WDI_TASK_START_AP

WDI_TLV_AP_CAPABILITIES

Article • 03/14/2023

WDI_TLV_AP_CAPABILITIES is a TLV that contains the capabilities of an access point.

TLV Type

0x16

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	The scan SSID list size.
UINT32	The desired SSID list size.
UINT32	The privacy exemption list size.
UINT32	The association table size.
UINT32	The key mapping table size.
UINT32	The default key table size.
UINT32	The maximum length of the WEP key value.
UINT8	Specifies whether the AP supports radar detection. Valid values are 0 (not supported) and 1 (supported).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_PARAMETERS_REQUESTED_TYPE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_PARAMETERS_REQUESTED_TYPE is a TLV that contains the requested Association Parameter TLV types.

TLV Type

0xBB

Length

The size (in bytes) of the array of UINT16 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT16[]	The list of Association Parameters TLV types that are requested. Valid TLV types are WDI_TLV_PMKID (0x9F) and WDI_TLV_EXTRA_ASSOCIATION_REQUEST_IES (0x40).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_REQUEST_DEVICE_CONTEXT

Article • 07/16/2024

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_REQUEST_DEVICE_CONTEXT is a TLV that contains vendor-specific information that is passed down to the port if the host decides to send a response to an incoming association request.

TLV Type

0x72

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Expand table

Type	Description
UINT8[]	Vendor-specific information that is passed down to the port if the host decides to send a response to an incoming association request.

Requirements

Expand table

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

WDI_TLV_ASSOCIATION_REQUEST_FRAME

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_REQUEST_FRAME is a TLV that contains the association request that was used for the association.

TLV Type

0x2E

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the association request that was used for the association. This does not include the 802.11 MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_REQUEST_IES

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_REQUEST_IES is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_RESPONSE_FRAME

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_RESPONSE_FRAME is a TLV that contains the received association response.

TLV Type

0x2F

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the received association response. This does not include the 802.11 MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_RESPONSE_IES

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_RESPONSE_IES is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_RESPONSE_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_RESPONSE_PARAMETERS is a TLV that contains association response parameters for OID_WDI_TASK_SEND_AP_ASSOCIATION_RESPONSE.

TLV Type

0x97

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Specifies whether or not to accept the association request. Valid values are 0 (do not accept) and 1 (accept).
UINT16	Specifies the reason code. If accept request is set to 0, this field provides a reason code to send back to the peer adapter.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_RESPONSE_RESULT_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_RESPONSE_RESULT_PARAMETERS is a TLV that contains association response result parameters.

TLV Type

0x76

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	The MAC address of the peer adapter.
UINT8	A bit value that indicates whether the request from the peer station is a reassociation request. Valid values are 0 and 1. A value of 1 indicates that it is a reassociation request.
UINT8	A bit value that indicates whether the response from the peer station is a reassociation response. Valid values are 0 and 1. A value of 1 indicates that it is a reassociation response.

Type	Description
WDI_AUTH_ALGORITHM	The authentication algorithm for the association.
WDI_CIPHER_ALGORITHM	The unicast cipher algorithm for the association.
WDI_CIPHER_ALGORITHM	The multicast cipher algorithm for the association.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_RESULT

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_RESULT is a TLV that contains the results of an association.

TLV Type

0x35

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID			The BSSID of the BSS.
WDI_TLV_ASSOCIATION_RESULT_PARAMETERS			The association result parameters.
WDI_TLV_ASSOCIATION_REQUEST_FRAME		X	The association request that was used for association. This does not include the 802.11 MAC header.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ASSOCIATION_RESPONSE_FRAME		X	The association response that was received. This does not include the 802.11 MAC header.
WDI_TLV_AUTHENTICATION_RESPONSE_FRAME		X	The authentication response that was received with a failure code. This does not include the 802.11 MAC header. It should only be included if the connection attempt failed during authentication exchange.
WDI_TLV_BEACON_PROBE_RESPONSE		X	The latest beacon or probe response frame received by the port. This does not include the 802.11 MAC header.
WDI_TLV_ETHERTYPE_ENCAP_TABLE		X	The Ethertype encapsulations for the association.
WDI_TLV_PHY_TYPE_LIST			The list of PHY identifiers that the 802.11 station uses to send or receive packets on the BSS network connection.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_ASSOCIATION_RESULT_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_RESULT_PARAMETERS is a TLV that contains parameters for an association result.

TLV Type

0x2D

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the completion status of the association attempt as defined in WDI_ASSOC_STATUS.
UINT32	The 802.11 status code sent by the peer in response to an authentication or association request from this port.
UINT8	Specifies whether the port sent an 802.11 association or an 802.11 reassociation request to the AP. This value should be set to 1 if a reassociation request was used.
WDI_AUTH_ALGORITHM	The authentication algorithm that the port negotiated with the peer during association.

Type	Description
WDI_CIPHER_ALGORITHM	The unicast cipher algorithm that the port negotiated with the peer during association.
WDI_CIPHER_ALGORITHM	The multicast data cipher algorithm that the port negotiated with the peer during association.
WDI_CIPHER_ALGORITHM	The multicast management cipher algorithm that the port negotiated with the peer during association.
UINT8	Specifies if the port has associated with a peer that supports distribution system (DS) services for ISO Layer 2 bridging on any station in the BSS network, including mobile stations and APs. This value should be set to 1 if this is supported.
UINT8	Specifies whether the port has performed port authorization during the association operation.
UINT8	Specifies whether 802.11 WMM QoS protocol has been negotiated for this association. This value should be set to 1 if it has been negotiated.
WDI_DS_INFO	Specifies whether the port is connected to the same DS as its previous association.
UINT32	When a (re)association fails with an 802.11 reason code of 30, this value indicates the value of the association comeback time requested by the peer.
WDI_BAND_ID (UINT32)	The band ID on which the association is established.
UINT32	The IHV association status. If the association failed, this can contain an IHV-defined status code. This is only used for debugging purpose.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_AUTH_ALGO_LIST

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_AUTH_ALGO_LIST is a TLV that contains a list of authentication algorithms.

TLV Type

0x3C

Length

The size (in bytes) of the array of WDI_AUTH_ALGORITHM structures. The array must contain 1 or more elements.

Values

Type	**Description**
WDI_AUTH_ALGORITHM[]	An array of authentication algorithms.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_AUTHENTICATION_RESPONSE_FRAME

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ASSOCIATION_RESPONSE_FRAME is a TLV that contains an authentication response frame.

TLV Type

0x124

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the authentication response that was received with a failure code. This does not include the 802.11 MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BAND_CAPABILITIES

Article • 03/14/2023

WDI_TLV_BAND_CAPABILITIES is a TLV that contains the capabilities of a band.

TLV Type

0x1A

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	The identifier for the band.
UINT8	Specifies whether the band is enabled or not.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BAND_CHANNEL

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BAND_CHANNEL is a TLV that contains the channels to scan for a specified band.

TLV Type

0x2C

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BANDID			Specifies the identifier for the band.
WDI_TLV_CHANNEL_INFO_LIST			Specifies a list of channels to scan. If the list is empty, the port must scan on all channels.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_BAND_ID_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BAND_ID_LIST is a TLV that contains a list of band IDs.

TLV Type

0xB6

Length

The size (in bytes) of the array of WDI_BAND_ID (UINT32) elements. The array must contain 1 or more elements.

Values

Type	Description
WDI_BAND_ID[]	An array of band IDs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BAND_INFO

Article • 03/14/2023

WDI_TLV_BAND_INFO is a TLV that contains band information.

TLV Type

0x27

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BAND_CAPABILITIES			The capabilities of the band.
WDI_TLV_PHY_TYPE_LIST			A list of valid PHY types in this band.
WDI_TLV_CHANNEL_LIST			A list of valid channel numbers in this band.
WDI_TLV_CHANNEL_WIDTH_LIST			A list of channel widths in MHz

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BANDID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BANDID is a TLV that contains a band ID.

TLV Type

0x39

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The identifier for the band.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BEACON_FRAME

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BEACON_FRAME is a TLV that contains a beacon frame.

TLV Type

0xA

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the beacon frame.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BEACON_IES

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BEACON_IES is a TLV that contains beacon IEs from an association.

TLV Type

0x78

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The beacon IEs from an association.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BEACON_PROBE_RESPONSE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BEACON_PROBE_RESPONSE is a TLV that contains the latest beacon or probe response frame received by the port.

TLV Type

0x30

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the latest beacon or probe response frame received by the port. This does not include the 802.11 MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BITMAP_PATTERN

Article • 03/14/2023

WDI_TLV_BITMAP_PATTERN is a TLV that contains the byte array of a pattern.

TLV Type

0x68

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the byte array of the pattern. Length = (Pattern length + 7)/8.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BITMAP_PATTERN_AND_MASK

Article • 03/14/2023

WDI_TLV_BITMAP_PATTERN_AND_MASK is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BITMAP_PATTERN_MASK

Article • 03/14/2023

WDI_TLV_BITMAP_PATTERN_MASK is a TLV that contains the bitmap pattern mask.

TLV Type

0xE4

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the byte array of the pattern mask. The mask must have 1 bit per pattern byte, therefore the mask length should equal (pattern length + 7) / 8.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSS_ENTRY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSS_ENTRY is a TLV that contains BSS entry information.

TLV Type

0x8

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID			The BSSID of the BSS.
WDI_TLV_PROBE_RESPONSE_FRAME		X	The probe response frame. If no probe response frame has been received, this is empty.
WDI_TLV_BEACON_FRAME		X	The beacon frame. If no beacon has been received, this is empty.
WDI_TLV_BSS_ENTRY_SIGNAL_INFO			The signal information (received signal strength and link quality) of the BSS.
WDI_TLV_BSS_ENTRY_CHANNEL_INFO			The logical channel number and band ID for the BSS entry.
WDI_TLV_BSS_ENTRY_DEVICE_CONTEXT		X	Device context about the peer. This context is provided from the IHV component and can be used to store per-BSS entry state that the IHV component wants to maintain. To avoid lifetime management issues, the IHV component must not use pointers in this field.
WDI_TLV_BSS_ENTRY_AGE_INFO		X (Note: This TLV is mandatory if the BSS list is maintained by the IHV component.)	The age information for this BSS entry, including the timestamp of when this entry was most recently discovered.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_DISCOVERED_SERVICE_ENTRY	X	X	The list of services found on the remote device, including the service information retrieved with a GAS query if the discovery request specified WDI_P2P_SERVICE_DISCOVERY_TYPE_SERVICE_INFORMATION as the discovery type.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSS_ENTRY_AGE_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSS_ENTRY_AGE_INFO is a TLV that contains age information for a BSS entry.

TLV Type

0xBA

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT64	Timestamp of when this BSS entry was most recently discovered. The timestamp should be obtained with NdisGetCurrentSystemTime or KeQuerySystemTime.
UINT8	Specifies whether this information is live (found during a currently running scan) or is coming from the IHV component's BSS list cache. Valid values are 0 (live) or 1 (cached).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_BSS_ENTRY_CHANNEL_INFO

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSS_ENTRY_CHANNEL_INFO is a TLV that contains BSS entry channel information.

TLV Type

0x3A

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_CHANNEL_NUMBER (UINT32)	The logical channel number on which the peer was discovered.
UINT32	The band ID for the BSS entry.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSS_ENTRY_DEVICE_CONTEXT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSS_ENTRY_DEVICE_CONTEXT is a TLV that contains device context for the BSS entry.

TLV Type

0xD

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the context data. This context is provided by the IHV component and can be used to store per-BSS entry state that the IHV component wants to maintain. To avoid lifetime management issues, the IHV component must not use pointers in this TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSS_ENTRY_PHY_INFO

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSS_ENTRY_PHY_INFO is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSS_ENTRY_SIGNAL_INFO

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSS_ENTRY_SIGNAL_INFO is a TLV that contains signal information for a BSS entry.

TLV Type

0xB

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
INT32	The received signal strength indicator (RSSI) value of the beacon or probe response from the peer. This value is specified in units of decibels referenced to 1.0 milliwatts (dBm)
UINT32	The link quality specified by a value from 0 to 100. A value of 100 specifies the highest link quality.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSS_SELECTION_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSS_SELECTION_PARAMETERS is a TLV that contains WDI_BSS_SELECTION_FLAGS that are used by host for BSS selection.

TLV Type

0x10F

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	WDI_BSS_SELECTION_FLAGS that are used by the host for BSS selection.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSSID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSSID is a TLV that contains the BSSID of a BSS.

TLV Type

0x2

Length

The size (in bytes) of a WDI_MAC_ADDRESS structure.

Values

Type	Description
WDI_MAC_ADDRESS	A Wi-Fi MAC address that specifies a BSSID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_BSSID_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_BSSID_INFO is a TLV that contains BSSID information.

TLV Type

0x120

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	AP reachability. Valid values are 1 (not reachable), 2 (unknown), and 3 (reachable).
UINT8	Security. If this is set to 1, it indicates that the AP identified by this BSSID supports the same security provisioning as used by the STA in its current association. If this is set to 0, it indicates that either this AP does not support the same security provisioning, or the security information is not available at this time.
UINT8	Key scope bit. If it is set to 1, it indicates the AP indicated by this BSSID has the same authenticator as the AP sending the report. If this bit is set to 0, it indicates a distinct authenticator or the information is not available.
UINT8	This is set to 1 if dot11SpectrumManagementRequired is true.
UINT8	This is set to 1 if dot11QosOptionImplemented is true.
UINT8	An AP sets the APSD subfield to 1 within the Capability Information field when dot11APSDOptionImplemented is true, and sets it to 0 otherwise.

Type	Description
UINT8	This is set to 1 if dot11RadioMeasurementActivated is true.
UINT8	This is set to 1 if dot11DelayedBlockAckOptionImplemented is true.
UINT8	This is set to 1 if dot11ImmediateBlockAckOptionImplemented is true.
UINT8	This is set to 1 if the AP represented by this BSSID includes an MDE in its Beacon frames.
UINT8	This is set to 1 to indicate that the AP represented by this BSSID is an HT AP, including the HT Capabilities element in its Beacon.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CANCEL_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CANCEL_PARAMETERS is a TLV that contains parameters for OID_WDI_ABORT_TASK.

TLV Type

0x2B

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
NDIS_OID	Specifies the OID from the original task being aborted.
UINT32	Specifies the transaction ID from the original task.
WDI_PORT_ID (UINT16)	Specifies the port ID from the original task.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHANNEL_INFO_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CHANNEL_INFO_LIST is a TLV that contains a list of channels.

TLV Type

0x41

Length

The size (in bytes) of the array of WDI_CHANNEL_NUMBER (UINT32) structures. The array must contain 1 or more elements.

Values

Type	Description
UINT32[]	An array of Wi-Fi channels.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHANNEL_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CHANNEL_LIST is a TLV that contains one or more channel numbers.

TLV Type

0x4

Length

The size (in bytes) of the array of WDI_CHANNEL_MAPPING_ENTRY structures. The array must contain 1 or more structures.

Values

Type	Description
WDI_CHANNEL_MAPPING_ENTRY[]	An array of channel mapping entries.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHANNEL_NUMBER

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CHANNEL_NUMBER is a TLV that contains a channel number.

TLV Type

0x121

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The channel number.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHANNEL_WIDTH_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CHANNEL_WIDTH_LIST is a TLV that contains a list of channel widths.

TLV Type

0xF5

Length

The size (in bytes) of the array of UINT32 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT32[]	A list of channel widths, specified in MHz.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHECKSUM_OFFLOAD_CAPABILITIES

Article • 03/14/2023

WDI_TLV_CHECKSUM_OFFLOAD_CAPABILITIES is a TLV that contains checksum offload capabilities for IPv4 and IPv6.

TLV Type

0xCB

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_IPV4_CHECKSUM_OFFLOAD			Parameters for IPv4 checksum offload.
WDI_TLV_IPV6_CHECKSUM_OFFLOAD			Parameters for IPv6 checksum offload.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHECKSUM_OFFLOAD_V4_RX_PARAMETERS (0xD2)

Article • 03/14/2023

WDI_TLV_CHECKSUM_OFFLOAD_V4_RX_PARAMETERS is a TLV that contains parameters for Rx checksum offload for IPv4.

Capability values are reported as documented in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**. Use NDIS_OFFLOAD_NOT_SUPPORTED and NDIS_OFFLOAD_SUPPORTED when indicating capabilities through OID_WDI_GET_ADAPTER_CAPABILITIES.

TLV Type

0xD2

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Encapsulation settings. Valid values are: • WDI_ENCAPSULATION_IEEE_802_11
UINT32	Specifies if offload of checksum with IP options is supported.
UINT32	Specifies if offload of checksum with TCP options is supported.
UINT32	Specifies if TCP checksum offload is enabled.
UINT32	Specifies if UDP offload is enabled.
UINT32	Specifies if IP checksum is enabled.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHECKSUM_OFFLOAD_V4_TX_PARAMETERS (0xD1)

Article • 03/14/2023

WDI_TLV_CHECKSUM_OFFLOAD_V4_TX_PARAMETERS is a TLV that contains parameters for Tx checksum offload for IPv4.

Capability values are reported as documented in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**. Use NDIS_OFFLOAD_NOT_SUPPORTED and NDIS_OFFLOAD_SUPPORTED when indicating capabilities through OID_WDI_GET_ADAPTER_CAPABILITIES.

TLV Type

0xD1

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Encapsulation settings. Valid values are: • WDI_ENCAPSULATION_IEEE_802_11
UINT32	Specifies if offload of checksum with IP options is supported.
UINT32	Specifies if offload of checksum with TCP options is supported.
UINT32	Specifies if TCP checksum offload is enabled.
UINT32	Specifies if UDP offload is enabled.
UINT32	Specifies if IP checksum is enabled.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHECKSUM_OFFLOAD_V6_RX_PARAMETERS (0xDD)

Article • 03/14/2023

WDI_TLV_CHECKSUM_OFFLOAD_V6_RX_PARAMETERS is a TLV that contains for Rx checksum offload for IPv6.

Capability values are reported as documented in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**. Use NDIS_OFFLOAD_NOT_SUPPORTED and NDIS_OFFLOAD_SUPPORTED when indicating capabilities through OID_WDI_GET_ADAPTER_CAPABILITIES.

TLV Type

0xDD

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	**Description**
UINT32	Encapsulation type. Valid values are: • WDI_ENCAPSULATION_IEEE_802_11
UINT32	Specifies if offload of checksum of packets with IP extension headers is supported.
UINT32	Specifies if offload of checksum with TCP options is supported.
UINT32	Specifies if TCP checksum offload is enabled.
UINT32	Specifies if UDP offload is enabled.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CHECKSUM_OFFLOAD_V6_TX_PARAMETERS (0xDC)

Article • 03/14/2023

WDI_TLV_CHECKSUM_OFFLOAD_V6_TX_PARAMETERS is a TLV that contains for Tx checksum offload for IPv6.

Capability values are reported as documented in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**. Use NDIS_OFFLOAD_NOT_SUPPORTED and NDIS_OFFLOAD_SUPPORTED when indicating capabilities through OID_WDI_GET_ADAPTER_CAPABILITIES.

TLV Type

0xDC

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Encapsulation type. Valid values are: • WDI_ENCAPSULATION_IEEE_802_11
UINT32	Specifies if offload of checksum of packets with IP extension headers is supported.
UINT32	Specifies if offload of checksum with TCP options is supported.
UINT32	Specifies if TCP checksum offload is enabled.
UINT32	Specifies if UDP offload is enabled.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_BIP_KEY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_BIP_KEY is a TLV that contains BIP cipher algorithm key data for OID_WDI_SET_ADD_CIPHER_KEYS.

TLV Type

0x51

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Specifies BIP cipher algorithm key data.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_BIP_GMAC_256_KEY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_BIP_GMAC_256_KEY is a TLV that contains BIP GMAC 256 cipher algorithm key data for OID_WDI_SET_ADD_CIPHER_KEYS.

TLV Type

0x165

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Specifies BIP GMAC 256 cipher algorithm key data.

Requirements

Minimum supported client	Windows 10, version 2004
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_CCMP_KEY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_CCMP_KEY is a TLV that contains CCMP cipher algorithm key data for OID_WDI_SET_ADD_CIPHER_KEY.

TLV Type

0x50

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Specifies CCMP cipher algorithm key data.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_GCMP_KEY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_GCMP_KEY (0x12F) is an unused TLV.

TLV Type

0x12F

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_GCMP_256_KEY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_GCMP_256_KEY is a TLV that contains GCMP 256 cipher algorithm key data for OID_WDI_SET_ADD_CIPHER_KEYS.

TLV Type

0x164

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Specifies GCMP 256 cipher algorithm key data.

Requirements

Minimum supported client	Windows 10, Version 2004
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_ID

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_ID is a TLV that contains a cipher key ID for OID_WDI_SET_ADD_CIPHER_KEYS and OID_WDI_SET_DELETE_CIPHER_KEYS.

TLV Type

0x4D

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the cipher key ID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_IHV_KEY

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_IHV_KEY is a TLV that contains an IHV key.

TLV Type

0x118

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the IHV key.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_RECEIVE_SEQUENCE_COUNT

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_RECEIVE_SEQUENCE_COUNT is a TLV that contains the receive sequence count.

TLV Type

0x4F

Length

The size (in bytes) of the array of UINT8 elements.

Values

Type	Description
UINT8[6]	Specifies the initial 48-bit value of Packet Number (PN), which is used for replay protection.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_TKIP_INFO

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_TKIP_INFO is a TLV that contains TKIP information.

TLV Type

0x4B

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CIPHER_KEY_TKIP_KEY			Specifies the TKIP key material.
WDI_TLV_CIPHER_KEY_TKIP_MIC			Specifies the TKIP MIC material.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_TKIP_KEY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_TKIP_KEY is a TLV that contains TKIP key material.

TLV Type

0x49

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the TKIP key material.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_TKIP_MIC

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_TKIP_MIC is a TLV that contains the TKIP MIC material.

TLV Type

0x4A

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the TKIP MIC material.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_TYPE_INFO

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_TYPE_INFO is a TLV that contains cipher key type information for OID_WDI_SET_ADD_CIPHER_KEYS and OID_WDI_SET_DELETE_CIPHER_KEYS.

TLV Type

0x4E

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_CIPHER_ALGORITHM	Specifies the cipher algorithm that uses the key.
WDI_CIPHER_KEY_DIRECTION	Specifies whether the key should be used for transmit only, receive only, or both.
UINT8	Specifies whether the port should delete the key on a roam. If this value is set to 0, the key must be deleted when the port disconnects from the BSS network or connects to the BSS network. If this value is set to 1, the key should be deleted on an explicit delete or on a reset request.
WDI_CIPHER_KEY_TYPE	Specifies the type of key being published.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CIPHER_KEY_WEP_KEY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CIPHER_KEY_WEP_KEY is a TLV that contains a WEP key.

TLV Type

0x58

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the WEP key.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_COALESCING_FILTER_MATCH_COUNT

Article • 03/14/2023

WDI_TLV_COALESCING_FILTER_MATCH_COUNT is a TLV that contains the number of packets that have matched receive filters on the network port.

TLV Type

0x66

Length

The size (in bytes) of the a UINT64.

Values

Type	Description
UINT64	The number of packets that have matched receive filters on the network port.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_COMMUNICATION_CAPABILITIES

Article • 03/14/2023

WDI_TLV_COMMUNICATION_CAPABILITIES is a TLV that specifies the communication capabilities.

TLV Type

0xE

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The maximum command size in bytes.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_COMMUNICATION_CONFIGURATION_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_COMMUNICATION_CONFIGURATION_ATTRIBUTES is a TLV that contains the host-adapter communication protocol configuration attributes.

TLV Type

0x20

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_COMMUNICATION_CAPABILITIES		X	The communication capabilities.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CONFIGURED_CIPHER_KEY

Article • 03/14/2023

WDI_TLV_CONFIGURED_CIPHER_KEY is a TLV that contains a list of configured ciphers to be set in OID_WDI_GET_PM_PROTOCOL_OFFLOAD. Drivers must return any GTK or iGTK keys that are currently configured. This TLV is a value of the WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY TLV.

TLV Type

0x147

Length

The size (in bytes) of the following values.

Values

Type	Description
WDI_CIPHER_KEY_TYPE	The type of key being returned.
WDI_CIPHER_ALGORITHM	Specifies the cipher algorithm that uses this key.
WDI_TLV_CIPHER_KEY_GCMP_256_KEY	Contains GCMP_256 cipher algorithm key data. This is only present if the cipher algorithm is WDI_CIPHER_ALGO_GCMP_256.
UINT8[6]	The initial 48-bit value of the Packet Number (PN), which is used for replay protection. Optional if **CipherAlgorithm** is **WDI_CIPHER_ALGO_WEP40**, **WDI_CIPHER_ALGO_WEP104**, or **WDI_CIPHER_ALGO_WEP**.
TLV<LIST<UINT8>>	Present if and only if **CipherAlgorithm** is **WDI_CIPHER_ALGO_CCMP**. Contains CCMP cipher algorithm key data.
TLV<LIST<UINT8>>	Present if and only if **CipherAlgorithm** is **WDI_CIPHER_ALGO_GCMP**. Contains GCMP cipher algorithm key data.
WDI_TLV_CIPHER_KEY_TKIP_INFO	Present if and only if **CipherAlgorithm** is **WDI_CIPHER_ALGO_TKIP**.

Type	Description
WDI_TLV_CIPHER_KEY_BIP_GMAC_256_KEY	Present ony if cipher algorithm is WDI_CIPHER_ALGO_BIP_GMAC_256.
TLV<LIST<UINT8>>	Present if and only if **CipherAlgorithm** is **WDI_CIPHER_ALGO_BIP**.
TLV<LIST<UINT8>>	Present if and only if **CipherAlgorithm** is **WDI_CIPHER_ALGO_WEP40**, **WDI_CIPHER_ALGO_WEP104**, or **WDI_CIPHER_ALGO_WEP**.
TLV<LIST<UINT8>>	Present if and only if **CipherAlgorithm** is in the range of **WDI_CIPHER_ALGO_IHV_START** to **WDI_CIPHER_ALGO_IHV_END**.

Requirements

Minimum supported client	Windows 10, version 2004
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CONFIGURED_MAC_ADDRESS

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CONFIGURED_MAC_ADDRESS is a TLV that contains a custom MAC address.

TLV Type

0x99

Length

The size (in bytes) of a WDI_MAC_ADDRESS structure.

Values

Type	Description
WDI_MAC_ADDRESS	The MAC address that should be used for the port.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CONNECT_BSS_ENTRY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CONNECT_BSS_ENTRY is a TLV that contains a list of candidate connect BSS entries.

TLV Type

0x34

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID			The BSSID of the BSS.
WDI_TLV_PROBE_RESPONSE_FRAME		X	The probe response frame. If no probe response has been received, this would be empty.
WDI_TLV_BEACON_FRAME		X	The beacon frame. If no beacon has been received, this would be empty.
WDI_TLV_BSS_ENTRY_SIGNAL_INFO			The signal information for this BSS entry.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSS_ENTRY_CHANNEL_INFO			The channel information for this BSS entry.
WDI_TLV_BSS_ENTRY_DEVICE_CONTEXT		X	The IHV provided context data about this peer.
WDI_TLV_PMKID		X	The 16 byte PMKID value for this BSS entry.
WDI_TLV_EXTRA_ASSOCIATION_REQUEST_IES		X	The IE to be included in the (re)association request frame for this BSSID. If present, this should be included in addition to the common IE.
WDI_TLV_FT_INITIAL_ASSOC_PARAMETERS		X	The initial Mobility Domain association parameters.
WDI_TLV_FT_REASSOC_PARAMETERS		X	The fast transition parameters (MDIE, R0KH-ID, PMKR0Name, SNonce). This is only present for Fast Transition (not during initial mobility domain association).
WDI_TLV_BSS_SELECTION_PARAMETERS		X	WDI_BSS_SELECTION_FLAGS that provide information used by the host for BSS selection.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CONNECT_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CONNECT_PARAMETERS is a TLV that contains parameters for OID_WDI_TASK_CONNECT and OID_WDI_TASK_ROAM.

TLV Type

0x33

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CONNECTION_SETTINGS			The settings for the connection.
WDI_TLV_SSID	X		List of SSIDs that the port is allowed to connect to.
WDI_TLV_HESSID_INFO		X	List of HESSIDs that the port is allowed to connect to. This is an additional requirement to the SSID list.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_AUTH_ALGO_LIST			The list of authentication algorithms that the connection can use.
WDI_TLV_MULTICAST_CIPHER_ALGO_LIST			The list of multicast cipher algorithms that the connection can use.
WDI_TLV_UNICAST_CIPHER_ALGO_LIST			The list of unicast cipher algorithms that the connection can use.
WDI_TLV_EXTRA_ASSOCIATION_REQUEST_IES		X	The IE blobs that must be included in the association requests sent by the port. This is applicable to any BSSID that the device would associate with. It should be used in addition to the BSSID specific IEs.
WDI_TLV_PHY_TYPE_LIST		X	The list of PHYs that are allowed to be used for the association. If not specified, any supported PHY can be used. If specified, the device must only use the listed PHYs.
WDI_TLV_DISALLOWED_BSSIDS_LIST		X	The list of BSSIDs that are not allowed to be used for association. If specified, the adapter must not associate to any AP that is in this list.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ALLOWED_BSSIDS_LIST		X	The list of BSSIDs that are allowed to be used for association. If WDI specifies 255.255.255.255 then all BSSIDs are allowed.
WDI_TLV_OWE_DH_IE		X	Diffie-Hellman Extension IE blob that must be included in the association request sent by the station when auth type is OWE. This is applicable to any BSSID that the device would associate with and should be included in addition to the other associated req vendor IEs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CONNECTION_QUALITY_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CONNECTION_QUALITY_PARAMETERS is a TLV that contains the desired Wi-Fi Connection Quality Hint.

TLV Type

0xA3

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The desired Wi-Fi Connection Quality Hint, as defined in WDI_CONNECTION_QUALITY_HINT.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CONNECTION_SETTINGS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_CONNECTION_SETTINGS is a TLV that contains connection settings for OID_WDI_TASK_CONNECT.

TLV Type

0x3F

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Specifies if this is a first-time connection request (value of 0) or a roaming connection (value of 1).
UINT8	Specifies if this is a connection to a network with hidden/non-broadcast SSIDs. This value is 1 when connecting to a hidden network.
UINT8	This sets the dot11ExcludeUnencrypted MIB. When this value is false (0) and the cipher algorithm is WEP, the port must connect to APs that do not set the privacy field in management frames.
UINT8	Specifies if MFP is enabled (1) or disabled (0). The station must advertise its 802.11w capabilities in the association request if and only if this value is set to 1 (enabled).
UINT8	Specifies if host-FIPS mode is enabled (1) or disabled (0).

Type	Description
WDI_ASSOC_STATUS (UINT32)	Specifies the roaming needed reason. If this is triggered due to NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED, this contains the reason from the roam indication.
WDI_ROAM_TRIGGER (UINT32)	Specifies whether this roam is a critical roam because the AP has set the Disassociation Imminent bit in its BSS Transition Request action frame.
UINT8	Specifies if 802.11v BSS transition is supported. If this bit is set to 1, the Station must set the BSS Transition field of the Extended capabilities element (Bit 19) to 1 in the association request.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_COUNTRY_REGION_LIST

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_COUNTRY_REGION_LIST is a TLV that contains a list of country or region codes.

TLV Type

0x12

Length

The size (in bytes) of the array of WDI_COUNTRY_REGION_LIST elements. The array must contain 1 or more elements.

Note WDI_COUNTRY_REGION_LIST is not a WDI structure. It is defined in the WDI TLV parser generator, and is used for documentation purposes only.

Values

Type	Description
WDI_COUNTRY_REGION_LIST[]	An array of country or region codes.

WDI_COUNTRY_REGION_LIST consists of the following elements.

Type	Description
UINT8[3]	A country or region code.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CREATE_PORT_MAC_ADDRESS

Article • 03/14/2023

WDI_TLV_CREATE_PORT_MAC_ADDRESS is a TLV that contains a MAC address for OID_WDI_TASK_CREATE_PORT.

TLV Type

0xD9

Length

The size (in bytes) of a **WDI_MAC_ADDRESS** structure.

Values

Type	Description
WDI_MAC_ADDRESS	The MAC address to be used for port creation.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CREATE_PORT_PARAMETERS

Article • 03/14/2023

WDI_TLV_CREATE_PORT_PARAMETERS is a TLV that contains parameters for OID_WDI_TASK_CREATE_PORT.

TLV Type

0x28

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT16	A bitwise OR value of the operation modes the host may configure on the port being created. The operation modes are defined in **WDI_OPERATION_MODE**.
UINT32	The NDIS_PORT_NUMBER that will be associated with the created port. Unless the adapter wants to handle non-WDI OIDs, it does not need to do anything with this field.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_CURRENT_CHANNEL_PARAMETERS

Article • 03/14/2023

WDI_TLV_CURRENT_CHANNEL_PARAMETERS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DATAPATH_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_DATAPATH_ATTRIBUTES is a TLV that contains datapath attributes.

TLV Type

0xB8

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DATAPATH_CAPABILITIES		X	The datapath capabilities.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DATAPATH_CAPABILITIES

Article • 03/14/2023

WDI_TLV_DATAPATH_CAPABILITIES is a TLV that contains datapath capabilities.

TLV Type

0xB9

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_INTERCONNECT_TYPE (UINT32)	Interconnect type.
UINT8	Maximum number of peers.
UINT8	Specifies transmit capability: Target priority queuing. Valid values are 0 and 1. If set to 0, WDI classifies Tx frames by Peer and TID and utilizes the full scheduler to select TX queues to transfer. It is recommended that this is set to false unless the target is capable of classification and Peer-TID queueing. If set to 1, WDI classifies Tx frames by Peer and TID and only provides queuing at a port level. WDI schedules backlogged port queues using a global DRR.

Type	Description
UINT16	Specifies transmit capability: Maximum number of Scatter Gather elements in frame. WDI coalesces frames as necessary such that the IHV miniport does not receive a frame that requires more scatter gather elements than specified by this capability. For best performance, it is suggested that this capability is set higher than the typical frame as the coalescing requires a memory copy. If this capability is not greater than max frame size divided by page size, WDI may be unable to successfully coalesce the frame and it may be dropped.
UINT8	Specifies transmit capability: Explicit Send Complete flag required. Valid values are 0 and 1. If set to 0, the target/TAL generates a TX send complete for all frames. If set to 1, the target/TAL generates TX send completion indication only for frames that have this flag set in the frame's metadata.
UINT16	Specifies transmit capability: Minimum effective frame size. When dequeuing frames, the TxMgr treats frames smaller than this value as having an effective size of this value.
UINT16	Specifies transmit capability: Frame size granularity. This value is equal to the granularity of memory allocation per frame. For the purposes of dequeuing, the TxMgr treats a frame as having an effective size equal to the frame size plus the least amount of padding such that the effective size is an integer multiple of this value. This value must be set to a power of two.
UINT8	Specifies transmit capability: Rx Tx forwarding. Valid values are 0 and 1. If set to 1, the target is capable of forwarding received frames.
UINT32	Specifies transmit capability: Maximum throughput, in units of 0.5 Mbps. This value is used for the allocation of descriptors and buffers.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DEFAULT_TX_KEY_ID_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DEFAULT_TX_KEY_ID_PARAMETERS is a TLV that contains the default key ID for packet transmission on a port for OID_WDI_SET_DEFAULT_KEY_ID.

TLV Type

0x54

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the default key ID for packet transmission on a port.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DELETE_CIPHER_KEY_INFO

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DELETE_CIPHER_KEY_INFO is a TLV that contains information to identify a single cipher key to remove with OID_WDI_SET_DELETE_CIPHER_KEYS.

TLV Type

0x53

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PEER_MAC_ADDRESS		X	Specifies the peer MAC address. At least one of WDI_TLV_PEER_MAC_ADDRESS or WDI_TLV_CIPHER_KEY_ID must be present.
WDI_TLV_CIPHER_KEY_ID		X	Specifies the cipher key ID. At least one of WDI_TLV_PEER_MAC_ADDRESS or WDI_TLV_CIPHER_KEY_ID must be present.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CIPHER_KEY_TYPE_INFO			Specifies the cipher key type information.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DELETE_PEER_STATE_PARAMETERS

Article • 03/14/2023

WDI_TLV_DELETE_PEER_STATE_PARAMETERS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DELETE_PORT_PARAMETERS

Article • 03/14/2023

WDI_TLV_DELETE_PORT_PARAMETERS is a TLV that contains parameters for OID_WDI_TASK_DELETE_PORT.

TLV Type

0x2A

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT16	Specifies the port number to delete.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DEVICE_SERVICE_PARAMS_DATA_BLOB

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DEVICE_SERVICE_PARAMS_DATA_BLOB is a TLV that contains information about a device service received from the IHV driver. This TLV is used in the NDIS_STATUS_WDI_INDICATION_DEVICE_SERVICE_EVENT status indication.

TLV Type

0x141

Length

The size, in bytes, of a `(UINT8 * (the number of elements in the list))`.

Values

Type	Description
TLV<List<UINT8>>	[Optional] The information received from the IHV driver.

Requirements

Minimum supported client: Windows 10, version 1809

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_TLV_DEVICE_SERVICE_PARAMS_GUID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DEVICE_SERVICE_PARAMS_GUID is a TLV that contains a GUID that identifies the device service to which this status indication belongs. This TLV is used in the NDIS_STATUS_WDI_INDICATION_DEVICE_SERVICE_EVENT status indication.

TLV Type

0x140

Length

The size, in bytes, of a GUID.

Values

Type	Description
GUID	The GUID that identifies the device service to which this status indication belongs (as defined by the IHV/OEM).

Requirements

Minimum supported client: Windows 10, version 1809

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_TLV_DEVICE_SERVICE_PARAMS_OPCODE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DEVICE_SERVICE_PARAMS_OPCODE is a TLV that contains the opcode specific to the device service. This TLV is used in the NDIS_STATUS_WDI_INDICATION_DEVICE_SERVICE_EVENT status indication.

TLV Type

0x13F

Length

The size, in bytes, of a UINT8.

Values

Type	Description
TLV<UINT8>	The opcode specific to the device service.

Requirements

Minimum supported client: Windows 10, version 1809

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_TLV_DISALLOWED_BSSIDS_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DISALLOWED_BSSIDS_LIST is a TLV that contains a list of BSSIDs that are not allowed to be used for association.

TLV Type

0xC3

Length

The size (in bytes) of the array of WDI_MAC_ADDRESS structures. The array must contain 1 or more structures.

Values

Type	Description
WDI_MAC_ADDRESS[]	A list of BSSIDs that are not allowed to be used for association. If this is specified, the adapter must not associate to any AP that is not in this list

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DISASSOCIATION_INDICATION_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DISASSOCIATION_INDICATION_PARAMETERS is a TLV that contains disassociation indication parameters for NDIS_STATUS_WDI_INDICATION_DISASSOCIATION.

TLV Type

0xBC

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	The MAC address of the peer associated with the disassociation indication.
WDI_ASSOC_STATUS (UINT32)	The trigger for the disassociation indication.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_DISASSOCIATION_PARAMETERS

Article • 03/14/2023

WDI_TLV_DISASSOCIATION_PARAMETERS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DISCONNECT_DEAUTH_FRAME

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DISCONNECT_DEAUTH_FRAME is a TLV that contains the received deauthentication frame.

TLV Type

0x37

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the received deauthentication frame.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DISCONNECT_DISASSOCIATION_FRAME

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DISCONNECT_DISASSOCIATION_FRAME is a TLV that contains the received disassociation frame.

TLV Type

0x38

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the received disassociation frame. This does not include the 802.11 MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DISCONNECT_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DISCONNECT_PARAMETERS is a TLV that contains parameters for OID_WDI_TASK_DISCONNECT.

TLV Type

0x36

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	The MAC address of the peer to disassociate.
UINT16	The reason for the host-triggered disassociation. This value is provided in little endian byte order and should be appropriately copied into the reason code of the outgoing frame.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_DOT11_RESET_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_DOT11_RESET_PARAMETERS is a TLV that contains parameters for OID_WDI_TASK_DOT11_RESET.

TLV Type

0xA2

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	If (and only if) this is set to 1, all MIB attributes for the port being reset are set to their default values.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ENABLE_WAKE_EVENTS

Article • 03/14/2023

WDI_TLV_ENABLE_WAKE_EVENTS is a TLV that contains the enabled wake events.

TLV Type

0x60

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the enabled wake-on-LAN packet patterns using the flags as documented in NDIS_PM_PARAMETERS.EnabledWoLPacketPatterns.
UINT32	Specifies the enabled protocol offloads using the flags as documented in NDIS_PM_PARAMETERS.EnabledProtocolOffloads.
UINT32	Specifies the wake-up flags using the flags as documented in NDIS_PM_PARAMETERS.WakeUpFlags.
UINT32	Specifies the media-specific wake up events using the flags as documented in NDIS_PM_PARAMETERS.MediaSpecificWakeUpEvents.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_ETHERTYPE_ENCAP_TABLE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ETHERTYPE_ENCAP_TABLE is a TLV that contains the Ethertype encapsulations for the association.

TLV Type

0x31

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_ETHERTYPE_ENCAPSULATION_ENTRY[]	An array of WDI_ETHERTYPE_ENCAPSULATION_ENTRY elements that specifies the Ethertype encapsulations for the association.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_EXTRA_ASSOCIATION_REQUEST_IES

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_EXTRA_ASSOCIATION_REQUEST_IES is a TLV that contains Information Elements (IEs) that must be included in association requests sent by the port.

TLV Type

0x40

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the IEs that must be included in association requests sent by the port. These are applicable to any BSSID that the device associates with. They should be used in addition to the common and BSSID specific IEs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FIRMWARE_VERSION

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FIRMWARE_VERSION is a TLV that contains the firmware version.

TLV Type

0xF4

Length

The size (in bytes) of the array of char elements. The array must contain 1 or more elements.

Values

Type	Description
char[]	The firmware version, stored as a null-terminated ASCII string.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_AUTH_REQUEST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_AUTH_REQUEST is a TLV that contains the Fast Transition authentication request byte blob.

TLV Type

0x119

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the Fast Transition authentication request byte blob.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_AUTH_RESPONSE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_AUTH_RESPONSE is a TLV that contains the Fast Transition authentication response byte blob.

TLV Type

0x10E

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the Fast Transition authentication response byte blob.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_FTE

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_FTE is a TLV that contains a Fast Transition Element.

TLV Type

0x10B

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	A Fast Transition Element that contains the R0KHID and SNonce.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_INITIAL_ASSOC_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_INITIAL_ASSOC_PARAMETERS is a TLV that contains initial association parameters for Fast Transition.

TLV Type

0x105

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_TLV_FT_MDE	The MDIE of the BSS entry.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_MDE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_MDE is a TLV that contains the MDIE of a BSS entry.

TLV Type

0x10D

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The MDIE of a BSS entry.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_PMKR0NAME

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_PMKR0NAME is a TLV that contains a PMKR0Name or PMKR1Name (802.11r).

TLV Type

0x107

Length

The size (in bytes) of a WDI_TYPE_PMK_NAME structure.

Values

Type	Description
WDI_TYPE_PMK_NAME	A PMKR0Name or PMKR1Name (802.11r).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_R0KHID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_R0KHID is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_R1KHID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_R1KHID is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_REASSOC_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_REASSOC_PARAMETERS is a TLV that contains reassociation parameters for Fast Transition.

TLV Type

0x106

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_TLV_FT_MDE	The MDIE of the BSS entry.
WDI_TLV_FT_PMKR0NAME	The PMKR0Name. This is needed during Fast Transition. The STA needs to send the PMKR0Name during the authentication request to the AP.
WDI_TLV_FT_FTE	The Fast Transition Element that contains the R0KHID and SNonce.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_RSNIE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_RSNIE is a TLV that contains the Fast Transition RSN IE byte blob.

TLV Type

0x10C

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the Fast Transition RSN IE byte blob.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FT_SNONCE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FT_SNONCE is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_FTM_NUMBER_OF_MEASUREMENTS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FTM_NUMBER_OF_MEASUREMENTS is a TLV that contains the number of measurements used to provide the round trip time (RTT) for a Fine Timing Measurement (FTM) request.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x15B

Length

The size (in bytes) of a UINT16.

Values

Type	Description
UINT16	The number of measurements used to provide the RTT.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_FTM_REQUEST_TIMEOUT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FTM_REQUEST_TIMEOUT is a TLV that contains the maximum time, in milliseconds, to complete a Fine Timing Measurement (FTM).

This TLV is used in the task parameters of OID_WDI_TASK_REQUEST_FTM.

TLV Type

0x161

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The maximum time, in milliseconds, to complete the FTM. The timeout is set to 150 ms multiplied by the number of targets.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_FTM_RESPONSE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FTM_RESPONSE is a TLV that contains Fine Timing Measurement (FTM) response information from a BSS target.

This TLV is used in the payload data of an NDIS_STATUS_WDI_INDICATION_REQUEST_FTM_COMPLETE task completion indication.

TLV type

0x163

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID	**WDI_MAC_ADDRESS**			The BSSID of the target to which this FTM response belongs.
WDI_TLV_FTM_RESPONSE_STATUS	**WDI_FTM_RESPONSE_STATUS**			The FTM response status. If success, the rest of the fields in this TLV are present.
WDI_TLV_RETRY_AFTER	UINT16			A duration, in seconds, that should pass before trying to request a new FTM from this target.
WDI_TLV_FTM_NUMBER_OF_MEASUREMENTS	UINT16			The number of measurements used to provide the round trip time (RTT). If the FTM response status was a success, this field is mandatory.

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSS_ENTRY_SIGNAL_INFO	INT32			The received signal strength indicator (RSSI) from the FTM target. This is in units of decibels referenced to 1.0 milliwatts (dBm). If the FTM response status was a success, this field is mandatory.
Same as row above	UINT32			The link quality value of the FTM target, ranging from 0 through 100. A value of 100 specifies the highest link quality. If the FTM response status was a success, this field is mandatory.
WDI_TLV_RTT	UINT32			The measured roundtrip time (RTT), in picoseconds. If the FTM response status was a success, this field is mandatory.
WDI_TLV_RTT_ACCURACY	UINT32			The accuracy, or expected degree of closeness, of the provided RTT measurement to the true value. The unit is in picoseconds. For more information, see the WDI_TLV_RTT_ACCURACY.
WDI_TLV_RTT_VARIANCE	UINT64			If more than one measurement was used to calculate the RTT, this field provides the statistical variance of the measurements used.
WDI_TLV_LCI_REPORT_STATUS	**WDI_LCI_REPORT_STATUS**			If an LCI report was requested, this field provides the status result. If successful, the following fields are present and mandatory.

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_LCI_REPORT_BODY	TLV<LIST<UINT8>>			The Location Configuration Information (LCI) report, as defined in Section 9.4.2.22.10 of the 802-11-2016 standard, including the LCI subelement and other Optional subelements available. In other words, this is the measurement report section of the Measurement Report element (as per Section 9.4.2.22 from the 802-11-2016 standard).

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_FTM_RESPONSE_STATUS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FTM_RESPONSE_STATUS is a TLV that contains the Fine Timing Measurement (FTM) response status from a target BSS.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x159

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_FTM_RESPONSE_STATUS	The FTM response status.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_FTM_TARGET_BSS_ENTRY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_FTM_TARGET_BSS_ENTRY is a TLV that contains information for a BSS target with which Fine Timing Measurement (FTM) procedures should be completed.

This TLV is used in the task parameters for OID_WDI_TASK_REQUEST_FTM.

TLV type

0x162

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID	**WDI_MAC_ADDRESS**			The BSSID of the target BSS.
WDI_TLV_PROBE_RESPONSE_FRAME	TLV<LIST<UINT8>>		X	The probe response frame. If no probe response has been received, this field is empty.

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BEACON_FRAME	TLV<LIST<UINT8>>		X	The beacon frame. If no beacon has been received, this field is empty.
WDI_TLV_BSS_ENTRY_SIGNAL_INFO	INT32			The received signal strength indicator (RSSI) value of the beacon or probe response from the peer. This is in units of decibels referenced to 1.0 milliwatts (dBm).
	UINT32			The link quality value ranging from 0 through 100. A value of 100 specifies the highest link quality.
WDI_TLV_BSS_ENTRY_CHANNEL_INFO	UINT32			The logical channel number of the target BSS.
	UINT32			The Band ID of the target BSS.

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSS_ENTRY_DEVICE_CONTEXT	TLV<LIST<UINT8>>			IHV component-provided context data about this peer. This can be usd to store per-BSS entry state that the IHV component wants to maintain. To avoid lifetime management issues, the IHV component must not use pointers in this field.
WDI_TLV_REQUEST_LCI_REPORT	UINT8			Possible values: • 0: LCI report not needed. • 1: LCI report should be requested.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_GET_AUTO_POWER_SAVE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_GET_AUTO_POWER_SAVE is a TLV that contains auto power save information for OID_WDI_GET_AUTO_POWER_SAVE.

TLV Type

0xB3

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The firmware current AutoPSM state.
UINT8	Reserved.
UINT16	Reserved.
UINT16	The beacon interval in milliseconds.
UINT8	The listen interval, in the unit of the beacon interval (for example, 1).
UINT8	The listen interval in the last low power state (for example, 5). If there is no last low power state, set to 255.
WDI_POWER_SAVE_LEVEL (UINT32)	The power mode.
WDI_POWER_SAVE_LEVEL (UINT32)	The power mode in Dx.

Type	Description
WDI_POWER_MODE_REASON_CODE (UINT32)	The reason for entering the Power Save state and listen interval.
UINT64	Milliseconds since start.
UINT64	Milliseconds in power save mode.
UINT64	Number of received multicast packets, including broadcast.
UINT64	Number of sent multicast packets, including broadcast.
UINT64	Number of received unicast packets.
UINT64	Number of sent unicast packets.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_HESSID

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_HESSID is a TLV that contains a list of HESSIDs.

TLV Type

0xC8

Length

The size (in bytes) of the array of WDI_MAC_ADDRESS structures. The array must contain 1 or more structures.

Values

Type	Description
WDI_MAC_ADDRESS[]	A list of HESSIDs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_HESSID_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_HESSID_INFO is a TLV that contains HESSID information, which includes a list of HESSIDs, the Access Network Type, and Hotspot Indication Element.

TLV Type

0xFF

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ACCESS_NETWORK_TYPE			The Access Network Type to be used in probe requests for the network being connected to.
WDI_TLV_HESSID			The list of HESSIDs that the port is allowed to connect to.
WDI_TLV_HOTSPOT_INDICATION_ELEMENT			The Hotspot Indication Element to be used in the Association Request.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_HOTSPOT_DOMAIN_PARTNER

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_HOTSPOT_DOMAIN_PARTNER is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_HOTSPOT_INDICATION_ELEMENT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_HOTSPOT_INDICATION_ELEMENT is a TLV that contains a Hotspot Indication Element that is used in a Association Request.

TLV Type

0x101

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	A Hotspot Indication Element that is used in a Association Request.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_IHV_DATA

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_IHV_DATA is a TLV that contains IHV-specific information that is used by the IHV extensibility module.

TLV Type

0xBD

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	IHV specific information that is used by the IHV extensibility module.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_IHV_NON_WDI_OIDS_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_IHV_NON_WDI_OIDS_LIST is a TLV that contains a list of non-WDI OIDs that the adapter wants to advertise to the operating system.

TLV Type

0x104

Length

The size (in bytes) of the array of UINT32 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT32[]	A list of non-WDI OIDs that the adapter wants to advertise to the operating system. The adapter should not assume that the operating system has already filtered non-WDI OIDs to match this list.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_IHV_TASK_DEVICE_CONTEXT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_IHV_TASK_DEVICE_CONTEXT is a TLV that contains IHV-provided device context for NDIS_STATUS_WDI_INDICATION_IHV_TASK_REQUEST.

TLV Type

0xE0

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The IHV-provided device context information that is forwarded to the IHV task.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_IHV_TASK_REQUEST_PARAMETERS

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_IHV_TASK_REQUEST_PARAMETERS is a TLV that contains the requested priority for NDIS_STATUS_WDI_INDICATION_IHV_TASK_REQUEST.

TLV Type

0xDF

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The IHV-requested priority for this task. See **WDI_IHV_TASK_PRIORITY** for valid priority values.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_INCOMING_ASSOCIATION_REQUEST_INFO

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_INCOMING_ASSOCIATION_REQUEST_INFO is a TLV that contains information about the incoming association request.

TLV Type

0x8F

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_INCOMING_ASSOCIATION_REQUEST_PARAMETERS			The parameters for the incoming association request.
WDI_TLV_ASSOCIATION_REQUEST_FRAME			The association request frame.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ASSOCIATION_REQUEST_DEVICE_CONTEXT		X	The vendor-specific information that is passed down to the port.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_INCOMING_ASSOCIATION_REQUEST_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_INCOMING_ASSOCIATION_REQUEST_PARAMETERS is a TLV that contains association request parameters.

TLV Type

0x7D

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	The MAC address of the sender.
UINT8	A bit that indicates whether or not it is a reassociation request. A value of 1 indicates that it is a reassociation request.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_INDICATION_CAN_SUSTAIN_AP

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_INDICATION_CAN_SUSTAIN_AP is a TLV that contains the reason for a Can Sustain AP indication.

TLV Type

0xE7

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The Can Sustain AP reason. See WDI_CAN_SUSTAIN_AP_REASON for possible reason values.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

NDIS_STATUS_WDI_INDICATION_CAN_SUSTAIN_AP

WDI_TLV_INDICATION_STOP_AP

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_INDICATION_STOP_AP is a TLV that contains the reason for a Stop AP indication.

TLV Type

0xE6

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The Stop AP reason. See WDI_STOP_AP_REASON for possible reason values.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

NDIS_STATUS_WDI_INDICATION_STOP_AP

WDI_TLV_INDICATION_WAKE_PACKET

Article • 03/14/2023

WDI_TLV_INDICATION_WAKE_PACKET is a TLV that contains a wake packet for NDIS_STATUS_WDI_INDICATION_WAKE_REASON. When the wake reason is WDI_WAKE_REASON_CODE PACKET, the status must include the wake packet encapsulated in a WDI_TLV_INDICATION_WAKE_PACKET.

TLV Type

0x9D

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The wake packet. The size is the size of the flat memory version of the packet that will be indicated in the normal receive path.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_INDICATION_WAKE_PACKET_PATTERN_ID

Article • 03/14/2023

WDI_TLV_INDICATION_WAKE_PACKET_PATTERN_ID is a TLV that contains the ID of the pattern that matches a wake packet.

TLV Type

0xB0

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The ID of the pattern that matches the wake packet. The ID is defined when the pattern is added with OID_WDI_SET_ADD_WOL_PATTERN.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_INDICATION_WAKE_REASON

Article • 03/14/2023

WDI_TLV_INDICATION_WAKE_REASON is a TLV that contains a wake reason for NDIS_STATUS_WDI_INDICATION_WAKE_REASON.

TLV Type

0x9C

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the wake reason.

Valid wake reason values are:

Wake reason	Value	Description
WDI_WAKE_REASON_CODE_PACKET	0x0001	A received packet matches the wake pattern.
WDI_WAKE_REASON_CODE_MEDIA_DISCONNECT	0x0002	Media disconnection.
WDI_WAKE_REASON_CODE_MEDIA_CONNECT	0x0003	Media connection.
WDI_WAKE_REASON_CODE_NLO_DISCOVERY	0x1000	NLO discovery.
WDI_WAKE_REASON_CODE_AP_ASSOCIATION_LOST	0x1001	Access point association lost.
WDI_WAKE_REASON_CODE_GTK_HANDSHAKE_ERROR	0x1002	GTK handshake error.
WDI_WAKE_REASON_CODE_4WAY_HANDSHAKE_REQUEST	0x1003	4-Way Handshake request.
WDI_WAKE_REASON_CODE_EAPID_REQUEST	0x1004	Reserved for future use.
WDI_WAKE_ REASON _CODE_INCOMING_M1	0x1005	Use WDI_WAKE_REASON_CODE_4WAY_HANDSHAKE_REQUEST instead.
WDI_WAKE_REASON_CODE_FIRMWARE_STALLED	0x1010	Firmware hang is detected (for example, by the watchdog timer) and wake logic is still functioning to wake the host.
WDI_WAKE_REASON_CODE_GTK_HANDSHAKE_REQUEST	0x1020	Group Key rekey request.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_INTERFACE_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_INTERFACE_ATTRIBUTES is a TLV that contains the attributes of an interface.

TLV Type

0x21

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_INTERFACE_CAPABILITIES			The capabilities of the interface.
WDI_TLV_FIRMWARE_VERSION			An ASCII string that specifies the firmware version.
WDI_TLV_IHV_NON_WDI_OIDS_LIST		X	List of non-WDI OIDs that the adapter wants to advertise to the operating system. The adapter should not assume that the operating system has already filtered non-WDI OIDs to match this list.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_INTERFACE_CAPABILITIES

Article • 03/14/2023

WDI_TLV_INTERFACE_CAPABILITIES is a TLV that contains the capabilities of the Wi-Fi interface.

TLV Type

0xF

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	The maximum transfer unit (MTU) size.
UINT32	The multicast list size for the adapter.
UINT16	The backfill size in bytes. This value cannot be greater than 256 bytes.
WDI_MAC_ADDRESS	The permanent MAC address for the adapter. If the device supports multiple permanent MAC addresses, the first MAC address that will be used by the device should be returned.
UINT32	The maximum supported send rate for this adapter in kbps.
UINT32	The maximum supported receive rate for this adapter in kbps.
UINT8	Specifies whether the radio is enabled by hardware. Valid values are 0 (disabled) and 1 (enabled).
UINT8	Specifies whether they radio is enabled by software. Valid values are 0 (disabled) and 1 (enabled).

Type	Description
UINT8	Specifies whether the interface supports PLR. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether the interface supports FLR. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether sending and receiving action frames is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	The supported number of RX spatial streams.
UINT8	The supported number of TX spatial streams.
UINT8	The number of channels that the adapter can work in concurrently, regardless of operation mode.
UINT8	Specifies whether antenna diversity is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether eCSA is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether the adapter supports MAC address randomization. Valid values are 0 (not supported) and 1 (supported).
WDI_MAC_ADDRESS	A bit mask that specifies for each address bit whether it can be randomized (0) or should keep the same value as the permanent address (1). The default is all zeros.
WDI_BLUETOOTH_COEXISTENCE_SUPPORT (UINT32)	The supported level of Wi-Fi - Bluetooth coexistence.
UINT8	Specifies non-WDI OID support. Valid values are: • 0 : Not supported. OIDs that the Microsoft component does not understand are not forwarded to the adapter. • 1 : Supported. OIDs that the Microsoft component does not understand are forwarded to the adapter. These OIDs will not contain WDI headers. To identify the adapter's port that the request came in on, use **NdisPortNumber** in the NDIS_OID_REQUEST and match it to the one in WDI_TASK_CREATE_PORT.

Type	Description
UINT8	Specifies whether the Fast Transition is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether Mu-MIMO is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies if the interface cannot support Miracast Sink. Valid values are 0 (supported) and 1 (not supported).
UINT8	Specifies if 802.11v BSS transition is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies if the device supports IP docking capability. Valid values are 0 (not supported) and 1 (supported). Added in Windows 10, version 1607, WDI version 1.0.21.
UINT8	Specifies if the device supports SAE authentication. Valid values are 0 (not supported) and 1 (supported). Added in Windows 10, version 1903, WDI version 1.1.8.
UINT8	Specifies if the device supports Multiband Operation (MBO). Valid values are 0 (not supported) and 1 (supported). Added in Windows 10, version 1903, WDI version 1.1.8.
UINT8	Specifies if the adapter implements beacon report measurements. Valid values are 0 (the adapter does not implement beacon report measurements) and 1 (the adapter implements its own 11k beacon report). Added in Windows 10, version 1903, WDI version 1.1.8.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_IPV4_CHECKSUM_OFFLOAD

Article • 03/14/2023

WDI_TLV_IPV4_CHECKSUM_OFFLOAD is a TLV that contains checksum offload capabilities for IPv4.

TLV Type

0xCF

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CHECKSUM_OFFLOAD_V4_TX_PARAMETERS			Parameters for Tx checksum offload for IPv4.
WDI_TLV_CHECKSUM_OFFLOAD_V4_RX_PARAMETERS			Parameters for Rx checksum offload for IPv4.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_IPV4_LSO_V2 (0xD3)

Article • 03/14/2023

WDI_TLV_IPV4_LSO_V2 is a TLV that contains Large Send Offload V2 parameters for IPv4.

TLV Type

0xD3

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Encapsulation type. Valid values are: • WDI_ENCAPSULATION_IEEE_802_11
UINT32	The maximum offload size. Specified by the maximum number of bytes of TCP user data per packet.
UINT32	The minimum segment count. Specified by the minimum number of segments that should be present after segmentation.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_IPV6_CHECKSUM_OFFLOAD

Article • 03/14/2023

WDI_TLV_IPV6_CHECKSUM_OFFLOAD is a TLV that contains checksum offload capabilities for IPv6.

TLV Type

0xD0

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CHECKSUM_OFFLOAD_V6_TX_PARAMETERS			Parameters for Tx checksum offload for IPv6.
WDI_TLV_CHECKSUM_OFFLOAD_V6_RX_PARAMETERS			Parameters for Rx checksum offload for IPv6.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_IPV6_LSO_V2 (0xD4)

Article • 03/14/2023

WDI_TLV_IPV6_LSO_V2 is a TLV that contains Large Send Offload V2 parameters for IPv6.

Capability values are reported as documented in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**. Use NDIS_OFFLOAD_NOT_SUPPORTED and NDIS_OFFLOAD_SUPPORTED when indicating capabilities through OID_WDI_GET_ADAPTER_CAPABILITIES.

TLV Type

0xD4

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Encapsulation type. Valid values are: • WDI_ENCAPSULATION_IEEE_802_11
UINT32	The maximum offload size. Specified by the maximum number of bytes of TCP user data per packet.
UINT32	The minimum segment count. Specified by the minimum number of segments that should be present after segmentation.
UINT32	Specifies if offload of checksum of packets with IP extension headers is supported.
UINT32	Specifies if offload of checksum with TCP options is supported.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_KCK_CONTENT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_KCK_CONTENT is a TLV that contains an IEEE 802.11 key confirmation key (KCK).

TLV Type

0x168

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Specifies an IEEE 802.11 key confirmation key (KCK).

Requirements

Minimum supported client	Windows 10, version 2004
Minimum supported server	Windows Server 2016
Header	Wditypes.pp

WDI_TLV_KEK_CONTENT

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_KEK_CONTENT is a TLV that contains an IEEE 802.11 key encryption key (KEK).

TLV Type

0x169

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Specifies an IEEE 802.11 key encryption key (KEK).

Requirements

Minimum supported client	Windows 10, version 2004
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_LCI_REPORT_BODY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_LCI_REPORT_BODY is a TLV that contains the Location Configuration Report (LCI) for a Fine Timing Measurement (FTM) request.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x160

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The LCI report.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_LCI_REPORT_STATUS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_LCI_REPORT_STATUS is a TLV that contains the status result of a Location Configuration Information (LCI) report, if one was requested during a Fine Timing Measurement (FTM) request.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x15F

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_LCI_REPORT_STATUS	The status of the LCI report.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_LINK_QUALITY_BAR_MAP

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_LINK_QUALITY_BAR_MAP is a TLV that contains the mapping of signal quality to Wi-Fi signal strength bars.

TLV Type

0xD8

Length

The size (in bytes) of the array of WDI_LINK_QUALITY_BAR_MAP_PARAMETERS elements. The array must contain 1 or more elements.

Note WDI_LINK_QUALITY_BAR_MAP_PARAMETERS is not a WDI structure. It is defined in the WDI TLV parser generator, and is used for documentation purposes only.

Values

Type	Description
WDI_LINK_QUALITY_BAR_MAP_PARAMETERS[]	An array of signal strength bar mapping parameters.

WDI_LINK_QUALITY_BAR_MAP_PARAMETERS consists of the following elements.

Type	Description
UINT8	The lower limit link quality (0-100) for the current signal strength bar.
UINT8	The upper limit of link quality (0-100) for the current signal strength bar.

Type	Description
UINT8	The signal strength bar number.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

WDI_TLV_LINK_STATE_CHANGE_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_LINK_STATE_CHANGE_PARAMETERS is a TLV that contains link state change parameters for NDIS_STATUS_WDI_INDICATION_LINK_STATE_CHANGE.

TLV Type

0x56

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	Specifies the MAC address of the remote peer.
UINT32	Specifies the current TX link speed. This is a value, in kilobits per second, that is the current TX link speed for this virtualized port. The conversion is 1 kbps = 1000 bps.
UINT32	Specifies the current RX link speed. This is a value, in kilobits per second, that is the current RX link speed for this virtualized port. The conversion is 1 kbps = 1000 bps.
UINT8	Specifies the current link quality. This is a value between 0 and 100 that is the current link quality for this virtualized port.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_LOW_LATENCY_CONNECTION_QUALITY_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_LOW_LATENCY_CONNECTION_QUALITY_PARAMETERS is a TLV that contains low latency connection quality parameters.

TLV Type

0xF6

Length

The size (in bytes) of the array of all contained elements.

Values

Type	Description
UINT8	Specifies the maximum number of milliseconds that the port can be on a different channel during Active Scan or other multi-channel operations. The only instance in which this off-channel can be higher is if the adapter needs to do a passive scan.
UINT8	Specifies the link quality threshold for NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED. When the link quality is below this threshold, it is acceptable for the adapter to send NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

OID_WDI_SET_CONNECTION_QUALITY

NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED

WDI_TLV_LSO_V1_CAPABILITIES (0xCC)

Article • 03/14/2023

WDI_TLV_LSO_V1_CAPABILITIES is a TLV that contains Large Send Offload V1 capabilities.

Capability values are reported as documented in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**. Use NDIS_OFFLOAD_NOT_SUPPORTED and NDIS_OFFLOAD_SUPPORTED when indicating capabilities through OID_WDI_GET_ADAPTER_CAPABILITIES.

TLV Type

0xCC

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	The encapsulation type. Valid values are: • WDI_ENCAPSULATION_IEEE_802_11
UINT32	The maximum offload size. Specified by the maximum number of bytes of TCP user data per packet.
UINT32	The minimum number of segments that a large TCP packet must be divisible by before the transport can offload it to the hardware for segmentation.
UINT32	Specifies whether or not TCP options are supported for this offload.
UINT32	Specifies whether or not IP options are supported for this offload.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_LSO_V2_CAPABILITIES

Article • 03/14/2023

WDI_TLV_LSO_V2_CAPABILITIES is a TLV that contains Large Send Offload V2 capabilities.

TLV Type

0xCD

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_IPV4_LSO_V2			Large Send Offload V2 capabilities for IPv4.
WDI_TLV_IPV6_LSO_V2			Large Send Offload V2 capabilities for IPv6.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_MAC_STATISTICS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_MAC_STATISTICS is a TLV that contains per-peer MAC statistics for OID_WDI_GET_STATISTICS.

TLV Type

0xA6

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	The MAC address of the peer that these counts are set for. For multicast and broadcast packets, this value is set to FF-FF-FF-FF-FF-FF-FF.
UINT64	The number of MSDU packets and MMPDU frames that the IEEE MAC layer of the 802.11 station successfully transmitted.
UINT64	The number of MSDU packets and MMPDU frames that the IEEE MAC layer of the 802.11 station successfully received. This member should not be incremented for received packets that failed cipher decryption or MIC validation.

Type	Description
UINT64	The number of unencrypted received MPDU frames that the MAC layer discarded when the IEEE 802.11 dot11ExcludeUnencrypted management information base (MIB) object is enabled. For more information about this MIB object, see OID_DOT11_EXCLUDE_UNENCRYPTED. MPDU frames are considered unencrypted when the Protected Frame subfield of the Frame Control field in the IEEE 802.11 MAC header is set to zero.
UINT64	The number of received MSDU packets that the 802.11 station discarded because of MIC failures.
UINT64	The number of received MPDU frames that the 802.11 station discarded because of the TKIP replay protection procedure.
UINT64	The number of encrypted MPDU frames that the 802.11 station failed to decrypt because of a TKIP ICV error.
UINT64	The number of received MPDU frames that the 802.11 discarded because of an invalid AES-CCMP format.
UINT64	The number of received MPDU frames that the 802.11 station discarded because of the AES-CCMP replay protection procedure.
UINT64	The number of received MPDU frames that the 802.11 station discarded because of errors detected by the AES-CCMP decryption algorithm.
UINT64	The number of encrypted MPDU frames received for which a WEP decryption key was not available on the 802.11 station.
UINT64	The number of encrypted MPDU frames that the 802.11 station failed to decrypt because of a WEP ICV error.

Type	Description
UINT64	The number of received encrypted packets that the 802.11 station successfully decrypted. For the WEP and TKIP cipher algorithms, the port must increment this counter for each received encrypted MPDU that was successfully decrypted. For the AES-CCMP cipher algorithm, the port must increment this counter on each received encrypted MSDU packet that was successfully decrypted.
UINT64	The number of encrypted packets that the 802.11 station failed to decrypt. For the WEP and TKIP cipher algorithms, the port must increment this counter for each received encrypted MPDU that was not successfully decrypted. For the AES-CCMP cipher algorithm, the port must increment this counter on each received encrypted MSDU packet that was not successfully decrypted. The port must not increment this counter for packets that are decrypted successfully, but are discarded for other reasons. For example, the port must not increment this counter for packets that are discarded because of TKIP MIC failures or TKIP/CCMP replays.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_MULTICAST_CIPHER_ALGO_LIST

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_MULTICAST_CIPHER_ALGO_LIST is a TLV that contains a list of multicast cipher algorithms.

TLV Type

0x3D

Length

The size (in bytes) of the array of WDI_CIPHER_ALGORITHM structures. The array must contain 1 or more elements.

Values

Type	Description
WDI_CIPHER_ALGORITHM[]	An array of multicast cipher algorithms.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_MULTICAST_DATA_ALGORITHM_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_MULTICAST_DATA_ALGORITHM_LIST is a TLV that contains an array of multicast data algorithm pairs.

TLV Type

0x14

Length

The size (in bytes) of the array of WDI_ALGO_PAIRS elements. The array must contain 1 or more elements.

Note WDI_ALGO_PAIRS is not a WDI structure. It is defined in the WDI TLV parser generator, and is used for documentation purposes only.

Values

Type	Description
WDI_ALGO_PAIRS[]	An array of authentication and cipher algorithm pairs.

WDI_ALGO_PAIRS consists of the following elements.

Type	Description
UINT8	Authentication algorithm as defined in WDI_AUTH_ALGORITHM.
UINT8	Cipher algorithm as defined in WDI_CIPHER_ALGORITHM.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_MULTICAST_LIST

Article • 03/14/2023

WDI_TLV_MULTICAST_LIST is a TLV that contains an array of multicast MAC addresses.

TLV Type

0x6A

Length

The size (in bytes) of the array of WDI_MAC_ADDRESS structures. The array must contain 1 or more structures.

Values

Type	Description
WDI_MAC_ADDRESS[]	An array of multicast MAC addresses.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_MULTICAST_MGMT_ALGORITHM_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_MULTICAST_MGMT_ALGORITHM_LIST is a TLV that contains an array of multicast management algorithm pairs.

TLV Type

0x15

Length

The size (in bytes) of the array of WDI_ALGO_PAIRS elements. The array must contain 1 or more elements.

Note WDI_ALGO_PAIRS is not a WDI structure. It is defined in the WDI TLV parser generator, and is used for documentation purposes only.

The size (in bytes) of the array of algorithm pairs.

Values

Type	Description
WDI_ALGO_PAIRS[]	An array of authentication and cipher algorithm pairs.

WDI_ALGO_PAIRS consists of the following elements.

Type	Description
UINT8	Authentication algorithm as defined in WDI_AUTH_ALGORITHM.

Type	Description
UINT8	Cipher algorithm as defined in WDI_CIPHER_ALGORITHM.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_NEIGHBOR_REPORT_ENTRY

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_NEIGHBOR_REPORT_ENTRY is a TLV that contains a neighbor report.

TLV Type

0x123

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID			The BSSID of the AP in the neighbor report.
WDI_TLV_BSSID_INFO			The BSSID information of the AP.
WDI_TLV_OPERATING_CLASS			The operating class of the AP indicated by this BSSID.
WDI_TLV_CHANNEL_NUMBER			The last known operating channel of the AP indicated by this BSSID.
WDI_TLV_PHY_TYPE			The PHY type of the AP indicated by this BSSID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_NETWORK_LIST_OFFLOAD_CONFIG

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_NETWORK_LIST_OFFLOAD_CONFIG is a TLV that contains Network List Offload (NLO) configuration.

TLV Type

0xDA

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Reserved field.
UINT32	The delay (in seconds) before the scan schedule starts.
UINT32	The period (in seconds) to scan in the first phase.
UINT32	The number of iterations in the fast scan phase.
UINT32	The period (in seconds) to scan in the slow scan phase. This phase lasts indefinitely until a new NLO command is set.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_NETWORK_LIST_OFFLOAD_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_NETWORK_LIST_OFFLOAD_PARAMETERS is a TLV that contains Network List Offload (NLO) parameters for OID_WDI_SET_NETWORK_LIST_OFFLOAD.

TLV Type

0x59

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_NETWORK_LIST_OFFLOAD_CONFIG			Specifies NLO configuration.
WDI_TLV_SSID_OFFLOAD	X	X	Specifies offload SSIDs. When this element is absent, the firmware should stop NLO scanning.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_NETWORK_OFFLOAD_CHANNELS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_NETWORK_OFFLOAD_CHANNELS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_NEXT_DIALOG_TOKEN

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_NEXT_DIALOG_TOKEN is a TLV that contains the dialog token to be used in the next Action frame.

TLV Type

0xE1

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The dialog token to be used in the next Action frame.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

OID_WDI_GET_NEXT_ACTION_FRAME_DIALOG_TOKEN

WDI_TLV_OFFLOAD_SCOPE

Article • 03/14/2023

WDI_TLV_OFFLOAD_SCOPE is a TLV that contains the scope for network offloads.

TLV Type

0x143

Length

The size (in bytes) of the below values.

Values

Type	Description
UINT8	Specifies whether checksum offload parameters are applicable on all ports. Possible values: • 0: Not applicable • 1: Applicable
UINT8	Specifies whether LsoV1 offload parameters are applicable on all ports. Possible values: • 0: Not applicable • 1: Applicable
UINT8	Specifies whether LsoV2 offload parameters are applicable on all ports. Possible values: • 0: Not applicable • 1: Applicable
UINT8	Specifies whether RSC offload parameters are applicable on all ports. Possible values: • 0: Not applicable • 1: Applicable

Requirements

Minimum supported client: Windows 10, version 1709

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_TLV_OPERATING_CLASS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_OPERATING_CLASS is a TLV that contains the frequency band for a channel.

TLV Type

0xFA

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The frequency band for a channel.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_OPERATION_MODE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_OPERATION_MODE is a TLV that contains the desired operation mode.

TLV Type

0x95

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The desired operation mode, as defined in WDI_OPERATION_MODE.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_OS_POWER_MANAGEMENT_FEATURES

Article • 03/14/2023

WDI_TLV_OS_POWER_MANAGEMENT_FEATURES is a TLV that contains flags for OS power management features. This enables IHVs to indicate to the OS that they support an advanced power management feature called Nic Auto Power Saver (NAPS). NAPS permits the wireless adapter to enter *DX* in situations where network activity is idle.

TLV Type

0x144

Length

The size (in bytes) of the following values.

Values

Type	Description
WDI_OS_POWER_MANAGEMENT_FLAGS	A bitwise OR of **WDI_OS_POWER_MANAGEMENT_FLAGS** values that defines supported NAPS enablement scenarios.

Requirements

Minimum supported client: Windows 10, version 1803

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_TLV_OWE_DH_IE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_OWE_DH_IE is a Diffie-Hellman Extension IE blob that must be included in the association request sent by the station when auth type is OWE. This is applicable to any BSSID that the device would associate with and should be included in addition to the other associated req vendor IEs.

TLV Type

0x16A

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that contains the IEs that must be included in association requests sent by the port. These are applicable to any BSSID that the device associates with. They should be used in addition to the common and BSSID specific IEs.

Requirements

Minimum supported client	Windows 10, version 2004

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ACTION_FRAME_DEVICE_CONTEXT

Article • 03/14/2023

WDI_TLV_P2P_ACTION_FRAME_DEVICE_CONTEXT is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ACTION_FRAME_IES

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_ACTION_FRAME_IES is a TLV that contains action frame IEs.

TLV Type

0x90

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the set of IEs that are sent to the remote device.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ACTION_FRAME_RESPONSE_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_ACTION_FRAME_RESPONSE_PARAMETERS is a TLV that contains Wi-Fi Direct Action Frame response parameters.

TLV Type

0xAD

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_P2P_ACTION_FRAME_TYPE	The type of Response Frame to be sent.
WDI_MAC_ADDRESS	The device address of the target peer Wi-Fi Direct device.
UINT8	The Wi-Fi Direct Dialog Token for this transaction.
UINT32	The send timeout. Specifies the maximum time, in milliseconds, to send this action frame.
UINT32	The post-ACK dwell time. Specifies the time to remain on listen channel, in milliseconds, after the incoming packet is acknowledged.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ADVERTISED_PREFIX_ENTRY

Article • 03/14/2023

WDI_TLV_P2P_ADVERTISED_PREFIX_ENTRY is a TLV that contains a Wi-Fi Direct advertised prefix entry.

TLV Type

0x110

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_NAME			Name of the service, in UTF-8, with a maximum size of 255 bytes.
WDI_TLV_P2P_SERVICE_NAME_HASH			Hash of Service Name.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ADVERTISED_SERVICE_ENTRY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_ADVERTISED_SERVICE_ENTRY is a TLV that contains an advertised service entry.

TLV Type

0xFC

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_NAME			Name of the service, in UTF-8, up to 255 bytes.
WDI_TLV_P2P_SERVICE_NAME_HASH			Hash of Service Name.
WDI_TLV_P2P_SERVICE_INFORMATION		X	Service Information for this service.
WDI_TLV_P2P_SERVICE_STATUS			Service Status of this service.
WDI_TLV_P2P_ADVERTISEMENT_ID			An ID that uniquely identifies the service instance.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CONFIG_METHODS			Configuration methods as defined in WDI_WPS_CONFIGURATION_METHOD. Only PIN display, PIN keypad, and WFDS are applicable.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ADVERTISED_SERVICES

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_ADVERTISED_SERVICES is a TLV that contains a list of advertised services.

TLV Type

0xEF

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_ADVERTISED_SERVICE_ENTRY	X	X	A list of advertised services.
WDI_TLV_P2P_ADVERTISED_PREFIX_ENTRY	X	X	A list of advertised prefixes that are derived from the list of advertised services.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_ASP2_ADVERTISED_SERVICE_ENTRY	X	X	Added in Windows 10, version 1607, WDI version 1.0.21. A list of advertised ASP2 services.
WDI_TLV_P2P_SERVICE_UPDATE_INDICATOR			The service update indicator to include in ANQP responses if the driver supports responding to service information discovery ANQP requests.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ADVERTISEMENT_ID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_ADVERTISEMENT_ID is a TLV that contains an ID that uniquely identifies a service instance.

TLV Type

0xEA

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	An ID that uniquely identifies a service instance.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ASP2_ADVERTISED_SERVICE_ENTRY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_ASP2_ADVERTISED_SERVICE_ENTRY is a TLV that contains an ASP2 Advertised Service Entry.

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x12E

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_TYPE			Service Type of the service (UTF-8), up to 21 bytes.
WDI_TLV_P2P_SERVICE_TYPE_HASH			Hash of Service Type.
WDI_TLV_P2P_INSTANCE_NAME			Instance Type of the service (UTF-8), up to 63 bytes.
WDI_TLV_P2P_INSTANCE_NAME_HASH			Hash of "Instance Name, Service Type".
WDI_TLV_P2P_SERVICE_INFORMATION		X	Service Information for the service.
WDI_TLV_P2P_SERVICE_STATUS			Service Status of the service.
WDI_TLV_P2P_ADVERTISEMENT_ID			An ID that uniquely identifies the service instance.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CONFIG_METHODS			Configuration methods as defined in WDI_WPS_CONFIGURATION_METHOD. Only **WDI_WPS_CONFIGURATION_METHOD_DISPLAY**, **WDI_WPS_CONFIGURATION_METHOD_KEYPAD**, and **WDI_WPS_CONFIGURATION_METHOD_WFDS_DEFAULT** are applicable.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ASP2_SERVICE_INFORMATION_DISCOVERY_ENTRY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_ASP2_SERVICE_INFORMATION_DISCOVERY_ENTRY is a TLV that contains an ASP2 Service Information Discovery Entry.

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x12D

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_NAME			Name of the service (UTF-8), up to 21 bytes.
WDI_TLV_P2P_INSTANCE_NAME			Instance name of the service (UTF-8), up to 63 bytes.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_INFORMATION		X	Request service information to be used for the ANQP query request to download service information for this Service.
WDI_TLV_P2P_SERVICE_UPDATE_INDICATOR		X	Service Update indicator to be used for the ANQP query request.
WDI_TLV_P2P_SERVICE_TRANSACTION_ID		X	Service transaction ID to be used for the ANQP query request.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_P2P_ATTRIBUTES is a TLV that contains Wi-Fi Direct attributes.

TLV Type

0x25

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CAPABILITIES			The Wi-Fi Direct capabilities.
WDI_TLV_P2P_INTERFACE_ADDRESS_LIST			An array of Wi-Fi Direct interface MAC addresses.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_BACKGROUND_DISCOVER_MODE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_BACKGROUND_DISCOVER_MODE is a TLV that contains Wi-Fi Direct Background Discover Mode parameters.

TLV Type

0xCE

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_P2P_DISCOVER_TYPE	The type of discovery to be performed by the port.
WDI_P2P_SERVICE_DISCOVERY_TYPE	The type of Service Discovery to be performed by the port. The only valid values are WDI_P2P_SERVICE_DISCOVERY_TYPE_NO_SERVICE_DISCOVERY and WDI_P2P_SERVICE_DISCOVERY_TYPE_SERVICE_NAME_ONLY.
UINT32	The device visibility timeout. Specifies the maximum timeout (in milliseconds) for reporting a device entry. This is required for background scan only.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_CAPABILITIES

Article • 03/14/2023

WDI_TLV_P2P_CAPABILITIES is a TLV that contains Wi-Fi Direct capabilities.

TLV Type

0x17

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Specifies the concurrent Group Owner count.
UINT8	Specifies the concurrent Client count.
UINT32	Specifies the supported WPS version.
UINT8	Specifies whether Service discovery is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Wi-Fi Direct Service Names Discovery support. Specifies whether, when given a list of service name hashes, the adapter can probe for service hashes and indicate the responses as they arrive. Valid values are 0 (not supported) and 1 (supported).
UINT8	Wi-Fi Direct Service Information Discovery support. Specifies whether, when given a list of service name hashes, the adapter can perform probes and ANQP queries to get full service information. Valid values are 0 (not supported) and 1 (supported).

Type	Description
UINT32	Specifies the maximum supported number of Service Name Advertisements bytes (to be sent in the beacon and probe responses). This sets a hard limit on the number of services that can be advertised.
UINT32	Specifies the maximum supported number of Service Information Advertisement bytes the adapter can respond to using the GAS protocol. This is only valid if the device supports responding to Service Advertisement queries. This value is for firmware optimization so that the firmware does not wake up the host to respond to every query. The operating system does not limit the number of service advertisements if the firmware has a limitation because there is a fallback in the operating system. If the firmware cannot handle the ANQP query response, it should pass up the request and the operating system handles it.
UINT8	Background discovery of Wi-Fi Direct devices and services. Specifies whether the adapter can periodically query for Wi-Fi Direct devices and service names so any new devices show up within 5 minutes of becoming visible. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether Client Discoverability is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether infrastructure management is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	The maximum size of the secondary adapter type list.
UINT8[6]	The device address in network byte order.
UINT32	The discovery filter list size.
UINT8	The GO client table size.

Type	Description
UINT32	The maximum size, in bytes, of vendor specific extension IEs that can be added to WFD management frames.
UINT8	Specifies whether the adapter supports OID_WDI_P2P_LISTEN_STATE_PASSIVE_AVAILABILITY. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether the adapter supports indicating updates to the GO operating channel(s). Valid values are 0 (not supported) and 1 (supported).
UINT8	Added in Windows 10, version 1511, WDI version 1.0.10. Specifies whether the adapter supports operating a GO on the 5GHz band. Valid values are 0 (not supported) and 1 (supported).
UINT8	Added in Windows 10, version 1607, WDI version 1.0.21. Specifies if the adapter, when provided with a list of ASP2 Service name instances, can probe for service hashes and indicate the responses as they arrive. Valid values are 0 (not supported) and 1 (supported).
UINT8	Added in Windows 10, version 1607, WDI version 1.0.21. Specifies if the adapter, when given a set of service name instances, can perform probes and ANQP queries to get the full service information. Valid values are 0 (not supported) and 1 (supported).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_CHANNEL_ENTRY_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_CHANNEL_ENTRY_LIST is a TLV that contains a channel number list.

TLV Type

0xF9

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_OPERATING_CLASS			The frequency band for the channels.
WDI_TLV_CHANNEL_INFO_LIST			The channel number list.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_CHANNEL_INDICATE_REASON

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_CHANNEL_INDICATE_REASON is a TLV that contains a reason for sending an indication.

TLV Type

0x102

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The reason for sending an indication. See WDI_P2P_CHANNEL_INDICATE_REASON for possible reasons.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_CHANNEL_LIST_ATTRIBUTE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_CHANNEL_LIST_ATTRIBUTE is a TLV that contains channel list attributes.

TLV Type

0xD5

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_COUNTRY_REGION_LIST			The country/region list.
WDI_TLV_P2P_CHANNEL_ENTRY_LIST	X		The list of channels.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_P2P_CHANNEL_NUMBER

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_CHANNEL_NUMBER is a TLV that contains Wi-Fi Direct channel number information.

TLV Type

0x82

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8[3]	The country or region code where the operating class and channel number are valid.
UINT8	The operating class/frequency band for the channel number.
WDI_CHANNEL_NUMBER (UINT32)	The channel number for the Wi-Fi Direct Device or Group.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_CONFIG_METHODS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_CONFIG_METHODS is a TLV that contains Wi-Fi Direct configuration methods.

TLV Type

0xEB

Length

The size (in bytes) of a UINT16.

Values

Type	Description
UINT16	Configuration methods as defined in WDI_WPS_CONFIGURATION_METHOD. Only PIN display, PIN keypad, and WFDS are applicable.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DEVICE_ADDRESS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DEVICE_ADDRESS is a TLV that contains the device address of the Group Owner.

TLV Type

0x91

Length

The size (in bytes) of a WDI_MAC_ADDRESS structure.

Values

Type	Description
WDI_MAC_ADDRESS	The device address of the Group Owner.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DEVICE_CAPABILITY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DEVICE_CAPABILITY is a TLV that contains Wi-Fi Direct device capabilities.

TLV Type

0x84

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	A bitmap of the Wi-Fi Direct device capabilities as defined in Table 12 of the Wi-Fi Direct technical specification.
UINT8	A bitmap of the Wi-Fi Direct capabilities in the above device capability bitmap that are currently set by the operating system.
UINT32	A bitmask that indicates which WPS versions are enabled.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DEVICE_FILTER_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DEVICE_FILTER_LIST is a TLV that contains a list of Wi-Fi Direct devices and Group Owners to search for during Wi-Fi Direct device discovery.

TLV Type

0xCE

Length

The size (in bytes) of the array of WDI_MAC_ADDRESS structures. The array must contain 1 or more structures.

Values

Type	Description
WDI_MAC_ADDRESS[]	A list of Wi-Fi Direct devices and Group Owners to search for during Wi-Fi Direct device discovery.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DEVICE_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DEVICE_INFO is a TLV that contains Wi-Fi Direct device information.

TLV Type

0x96

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_DEVICE_INFO_PARAMETERS			The device information, including Wi-Fi Direct device address, supported configuration methods, and primary device type.
WDI_TLV_P2P_DEVICE_NAME			The device name for this device.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DEVICE_INFO_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DEVICE_INFO_PARAMETERS is a TLV that contains Wi-Fi Direct device information parameters.

TLV Type

0x86

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8[6]	The Wi-Fi Direct Device Address of the peer.
UINT16	The configuration methods supported by the device.
UINT16	Primary device type: Main type category identifier.
UINT8[4]	Primary device type: OUI assigned to this device type.
UINT16	Primary device type: Subcategory type identifier.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DEVICE_NAME

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DEVICE_NAME is a TLV that contains a device name.

TLV Type

0x92

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the device name for the device.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DISCOVER_MODE

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DISCOVER_MODE is a TLV that contains Wi-Fi Direct discovery mode information for OID_WDI_TASK_P2P_DISCOVER.

TLV Type

0xA9

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_P2P_DISCOVER_TYPE (UINT32)	The type of discovery to be performed by the port.
UINT8	A flag that indicates if a complete device discovery is required. Valid values are 0 (not required) and 1 (required). If this flag is set to 0, a partial discovery may be performed.
WDI_P2P_SCAN_TYPE (UINT32)	The type of scan to be performed by the port in scan phase.
WDI_P2P_SERVICE_DISCOVERY_TYPE (UINT32)	The type of Service Discovery to be performed.
UINT8	The scan repeat count. Specifies if the full scan procedure should be repeated. If set to 0, the scan should be repeated until the task is aborted by the host.

Type	Description
UINT32	The time between scans. If the scan repeat count is not set to 1, this time specifies how long (in milliseconds) device should wait before repeating the scan procedure after completing a full scan of the requested channels.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DISCOVERED_SERVICE_ENTRY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DISCOVERED_SERVICE_ENTRY is a TLV that contains a discovered service entry.

TLV Type

0x112

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_NAME			The name of the service, in UTF-8, up to 255 bytes.
WDI_TLV_P2P_SERVICE_INFORMATION		X	The Service Information for the service.
WDI_TLV_P2P_SERVICE_STATUS			The Service Status of the service.
WDI_TLV_P2P_ADVERTISEMENT_ID			An ID that uniquely identifies the service instance.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CONFIG_METHODS			The Configuration Methods as defined in WDI_WPS_CONFIGURATION_METHOD. Only PinDisplay, PinKeypad and WFDS are applicable.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_DISCOVERY_CHANNEL_SETTINGS

Article • 03/14/2023

> **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_DISCOVERY_CHANNEL_SETTINGS is a TLV that contains Wi-Fi Direct discovery channel settings.

TLV Type

0xE8

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_LISTEN_DURATION			The cycle duration and listen time.
WDI_TLV_BAND_CHANNEL	X		The list of channels to scan.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_P2P_GO_INTERNAL_RESET_POLICY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GO_INTERNAL_RESET_POLICY is a TLV that contains the policy used by the firmware for operating channel selection after a Wi-Fi Direct GO Reset is stopped/restarted.

TLV Type

0xB2

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_P2P_GO_INTERNAL_RESET_POLICY (UINT32)	If an Wi-Fi Direct GO Reset is stopped/restarted by the IHV component on its own (for example, for Bluetooth co-ex spatial stream downgrade), this configuration defines the policy to be adopted by the firmware for operating channel selection after the reset.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_P2P_GO_NEGOTIATION_CONFIRMATION_INFO

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GO_NEGOTIATION_CONFIRMATION_INFO is a TLV that contains Wi-Fi Direct GO Negotiation Confirmation information.

TLV Type

0x88

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_GO_NEGOTIATION_CONFIRMATION_PARAMETERS			The Wi-Fi Direct GO Negotiation Confirmation parameters.
WDI_TLV_P2P_GROUP_ID		X	The Wi-Fi Direct Group ID.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CHANNEL_NUMBER		X	The listen channel of the remote device. The GO negotiation confirmation frame must be sent on this channel whenever this is specified.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_GO_NEGOTIATION_CONFIRMATION_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GO_NEGOTIATION_CONFIRMATION_PARAMETERS is a TLV that contains incoming GO Negotiation Confirmation parameters.

TLV Type

0xAA

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The Wi-Fi Direct Status Code, as defined by the Wi-Fi Direct specification.
UINT8	Wi-Fi Direct Group capability bitmask. The bitmask matches those defined in Table 13-Group Capability Bitmap definition of the Wi-Fi Direct technical specification.
UINT8	The bits in the Group capability bitmap above that are set by the operating system.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_P2P_GO_NEGOTIATION_REQUEST_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GO_NEGOTIATION_REQUEST_INFO is a TLV that contains Wi-Fi Direct Group Owner negotiation request information.

TLV Type

0x6D

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_GO_NEGOTIATION_REQUEST_PARAMETERS			The Wi-Fi Direct Group Owner negotiation request parameters.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CHANNEL_NUMBER		X	The listen channel of the remote device. Whenever this is specified, the GO negotiation request frame must be sent on this channel.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_GO_NEGOTIATION_REQUEST_PARAMETERS

Article • 03/14/2023

> ⓘ Important
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GO_NEGOTIATION_REQUEST_PARAMETERS is a TLV that contains Wi-Fi Direct Group Owner negotiation request parameters.

TLV Type

0x6E

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Specifies the local Wi-Fi Direct GO Intent. Valid values are between 0 and 15.
UINT8	Specifies the tie-breaker field of GO Intent.
UINT16	Specifies the GO Configuration Timeout in milliseconds.
UINT16	Specifies the Client Configuration Timeout in milliseconds.
WDI_MAC_ADDRESS	Intended interface address. Specifies the local MAC address for future Wi-Fi Direct connections.
UINT8	Specifies the Wi-Fi Direct Group capability bitmask. The bitmask matches those defined in Table 13-Group Capability Bitmap definition of the Wi-Fi P2P technical specification.

Type	Description
UINT8	Specifies the bits set by the operating system in the Group capability bitmap above.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_GO_NEGOTIATION_RESPONSE_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GO_NEGOTIATION_RESPONSE_INFO is a TLV that contains Wi-Fi Direct Group Owner negotiation response information.

TLV Type

0x6F

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_GO_NEGOTIATION_RESPONSE_PARAMETERS			Specifies the Wi-Fi Direct Group Owner negotiation response parameters.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_GROUP_ID		X	Specifies the Group ID for local Wi-Fi Direct GO.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_GO_NEGOTIATION_RESPONSE_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GO_NEGOTIATION_RESPONSE_PARAMETERS is a TLV that contains incoming GO Negotiation Response parameters.

TLV Type

0x71

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Specifies the Wi-Fi Direct Status Code, as defined by the Wi-Fi Direct specification.
UINT8	Specifies the local Wi-Fi Direct GO Intent. This is a value between 0 and 15.
UINT8	Specifies the tie-breaker field of the GO Intent.
UINT16	Specifies the GO Configuration Timeout in milliseconds.
UINT16	Specifies the Client Configuration Timeout in milliseconds.
WDI_MAC_ADDRESS	Intended interface address. Specifies the local MAC Address for future Wi-Fi Direct connection.

Type	Description
UINT8	Specifies the Wi-Fi Direct Group capability bitmask. The bitmask matches those defined in Table 13-Group Capability Bitmap definition of the Wi-Fi P2P technical specification.
UINT8	Specifies the bits set by the operating system in the Group capability bitmap above.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_GROUP_BSSID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GROUP_BSSID is a TLV that contains the Group BSSID for local Wi-Fi Direct GO.

TLV Type

0x73

Length

The size (in bytes) of a WDI_MAC_ADDRESS structure.

Values

Type	Description
WDI_MAC_ADDRESS	Specifies the Group BSSID for local Wi-Fi Direct GO.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_GROUP_ID

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GROUP_ID is a TLV that contains the Group ID for Wi-Fi Direct GO.

TLV Type

0x75

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_DEVICE_ADDRESS			Specifies the device address of the Wi-Fi Direct GO.
WDI_TLV_SSID			Specifies the Group SSID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_GROUP_OWNER_CAPABILITY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_GROUP_OWNER_CAPABILITY is a TLV that contains Wi-Fi Direct Group Owner capability information.

TLV Type

0x77

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Specifies the Wi-Fi Direct Group capability bitmask. The bitmask matches those defined in Table 13-Group Capability Bitmap definition of the Wi-Fi P2P technical specification.
UINT8	Specifies the bits set by the operating system in the Group capability bitmap above.
UINT32	Maximum client count for this Group Owner.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_P2P_INCLUDE_LISTEN_CHANNEL

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INCLUDE_LISTEN_CHANNEL is a TLV that specifies whether the probe request should include the Listen Channel attribute during discovery.

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x128

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	This parameter specifies whether the probe request should include the Listen Channel attribute during discovery. Valid values are 0 (do not include) and 1 (include).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INCOMING_FRAME_INFORMATION

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INCOMING_FRAME_INFORMATION is a TLV that contains incoming Wi-Fi Direct action frame information.

TLV Type

0x79

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_INCOMING_FRAME_PARAMETERS			Specifies the incoming frame parameters.
WDI_TLV_P2P_ACTION_FRAME_IES			Specifies the IEs section of the received public action frame.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ACTION_FRAME_DEVICE_CONTEXT		X	Specifies the vendor-specific information that is passed back down if the host decides to send a response to this incoming message. To avoid lifetime management issues, the IHV component must not use pointers in this field.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INCOMING_FRAME_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INCOMING_FRAME_PARAMETERS is a TLV that contains incoming Wi-Fi Direct action frame parameters.

TLV Type

0x7A

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_P2P_ACTION_FRAME_TYPE	The type of the incoming action frame.
WDI_MAC_ADDRESS	The MAC address of the remote peer.
UINT8	The Wi-Fi Direct Dialog Token for the transaction.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INSTANCE_NAME

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INSTANCE_NAME is a TLV that contains the Instance Name of the service.

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x12B

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The Instance Name of the service in UTF-8, up to 63 bytes long.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INSTANCE_NAME_HASH

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INSTANCE_NAME_HASH is a TLV that contains the hash of "Instance Name, Service Type".

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x12C

Length

The size (in bytes) of a WDI_P2P_SERVICE_NAME_HASH structure.

Values

Type	Description
WDI_P2P_SERVICE_NAME_HASH	The hash of "Instance Name, Service Type".

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INTERFACE_ADDRESS_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INTERFACE_ADDRESS_LIST is a TLV that contains an address list for a Wi-Fi Direct interface.

TLV Type

0x18

Length

The size (in bytes) of the array of WDI_MAC_ADDRESS structures. The array must contain 1 or more structures.

Values

Type	Description
WDI_MAC_ADDRESS[]	An array of Wi-Fi MAC addresses.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INVITATION_REQUEST_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INVITATION_REQUEST_INFO is a TLV that contains Wi-Fi Direct Invitation Request information.

TLV Type

0x7B

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_INVITATION_REQUEST_PARAMETERS			The Wi-Fi Direct Invitation Request parameters.
WDI_TLV_P2P_GROUP_BSSID		X	The Group BSSID.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CHANNEL_NUMBER		X	The operating channel for Wi-Fi Direct GO.
WDI_TLV_P2P_GROUP_ID			The Group ID for target Wi-Fi Direct GO.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INVITATION_REQUEST_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INVITATION_REQUEST_PARAMETERS is a TLV that contains Wi-Fi Direct Invitation Request parameters.

TLV Type

0x7C

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT16	The Group Owner Configuration Timeout in milliseconds.
UINT16	The Client Configuration Timeout in milliseconds.
UINT8	The invitation flags as defined by the Wi-Fi Direct specification.
UINT8	A bit that indicates whether or not the outgoing Invitation Request is an invitation to a local Group Owner. Valid values are 0 and 1. This bit is set to 1 if it is an invitation to a local GO.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INVITATION_RESPONSE_INFO

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INVITATION_RESPONSE_INFO is a TLV that contains Wi-Fi Direct Invitation Response information.

TLV Type

0x7E

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_INVITATION_RESPONSE_PARAMETERS			The Wi-Fi Direct Invitation Response parameters.
WDI_TLV_P2P_GROUP_BSSID		X	The Group BSSID for local Wi-Fi Direct GO.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CHANNEL_NUMBER		X	The operating channel for Wi-Fi Direct GO.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_INVITATION_RESPONSE_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_INVITATION_RESPONSE_PARAMETERS is a TLV that contains Wi-Fi Direct Invitation Response parameters.

TLV Type

0x80

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The Wi-Fi Direct Status Code, as specified by the Wi-Fi Direct specification..
UINT16	The GO Configuration Timeout, in milliseconds.
UINT16	The Client Configuration Timeout, in milliseconds.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_LISTEN_CHANNEL

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_LISTEN_CHANNEL is a TLV that contains Wi-Fi Direct channel information.

TLV Type

0x114

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8[3]	The country or region code where the operating class and channel number are valid.
UINT8	The operating class/frequency band for the channel number.
WDI_CHANNEL_NUMBER (UINT32)	The channel number for the Wi-Fi Direct Device or Group.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_LISTEN_DURATION

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_LISTEN_DURATION is a TLV that contains Wi-Fi Direct listen duration information.

TLV Type

0xE9

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	The duration, in milliseconds, of the listen cycle. The adapter is in listen state during this time.
UINT32	The duration, in milliseconds, that the adapter is expected to actively listen during the listen cycle. This duration must be less than listen cycle duration.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_LISTEN_STATE

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_LISTEN_STATE is a TLV that contains a Wi-Fi Direct listen state.

TLV Type

0x81

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_P2P_LISTEN_STATE	The desired Wi-Fi Direct listen state.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_PERSISTENT_GROUP_ID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_PERSISTENT_GROUP_ID is a TLV that contains a Group ID of a Persistent Group to be used for a connection.

TLV Type

0xF1

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_DEVICE_ADDRESS			The device address of the Group Owner.
WDI_TLV_SSID			The Group SSID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_PROVISION_DISCOVERY_REQUEST_INFO

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_PROVISION_DISCOVERY_REQUEST_INFO is a TLV that contains Wi-Fi Direct Provision Discovery Request information.

TLV Type

0x83

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_PROVISION_DISCOVERY_REQUEST_PARAMETERS			The Wi-Fi Direct Provision Discovery Request parameters.
WDI_TLV_P2P_GROUP_ID		X	The Group ID for the target Wi-Fi Direct GO. The Group ID is optional. In the case of Wi-Fi Direct services, this is the Group ID for the local Wi-Fi Direct GO that the remote side should join.
WDI_TLV_P2P_PROVISION_SERVICE_ATTRIBUTES		X	The Wi-Fi Direct Provision Service attributes.
WDI_TLV_P2P_PERSISTENT_GROUP_ID		X	The Group IP for the Persistent Group to be used for the connection. This field is valid if the Persistent Group flag in WDI_TLV_P2P_PROVISION_SERVICE_ATTRIBUTES is set to 1.
WDI_TLV_P2P_SERVICE_SESSION_INFO		X	Service Session information. This field is valid if WDI_TLV_P2P_PROVISION_SERVICE_ATTRIBUTES is present.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_PROVISION_DISCOVERY_REQUEST_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_PROVISION_DISCOVERY_REQUEST_PARAMETERS is a TLV that contains Wi-Fi Provision Discovery Request parameters.

TLV Type

0x85

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Wi-Fi Direct Group capability bitmask. The bitmask matches those defined in Table 13-Group Capability Bitmap definition of the Wi-Fi Direct technical specification.
UINT8	The bits in the Group capability bitmap above that are set by the operating system.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_PROVISION_DISCOVERY_RESPONSE_INFO

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_PROVISION_DISCOVERY_RESPONSE_INFO is a TLV that contains Wi-Fi Direct Provision Discovery Response information.

TLV Type

0x87

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_PROVISION_DISCOVERY_RESPONSE_PARAMETERS			The provision discovery response parameters.
WDI_TLV_P2P_PROVISION_SERVICE_ATTRIBUTES		X	The Provision Service attributes.
WDI_TLV_P2P_GROUP_ID		X	The Group ID if Wi-Fi Direct Service is supported.
WDI_TLV_P2P_PERSISTENT_GROUP_ID		X	The Group IP for the Persistent Group to be used for the connection. This field is valid if the Persistent Group flag in WDI_TLV_P2P_PROVISION_SERVICE_ATTRIBUTES is set to 1.
WDI_TLV_P2P_SERVICE_SESSION_INFO		X	The Service Session information.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_PROVISION_DISCOVERY_RESPONSE_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_PROVISION_DISCOVERY_RESPONSE_PARAMETERS is a TLV that contains provision discovery response parameters.

TLV Type

0x113

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The Wi-Fi Direct Group capability bitmask. The bitmask matches those defined in Table 13-Group Capability Bitmap definition of the Wi-Fi P2P technical specification.
UINT8	The bits set by the operating system in the above Group capability bitmap.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_PROVISION_SERVICE_ATTRIBUTES

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_PROVISION_SERVICE_ATTRIBUTES is a TLV that contains Wi-Fi Direct Provision Service attributes.

TLV Type

0xC6

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Wi-Fi Direct Status Code, as defined by the Wi-Fi Direct specification.
WDI_MAC_ADDRESS	Local MAC Address for future Wi-Fi Direct connection.
UINT8	Connection Capability bitmask.
UINT32	Feature Capability bitmask.
UINT32	Advertisement ID for the Service Instance.
WDI_MAC_ADDRESS	Service address for the Service instance.
UINT32	Session ID that uniquely identifies the Session to the Service.
WDI_MAC_ADDRESS	Session address that uniquely identifies the Session to the Service.

Type	Description
UINT16	GO Configuration Timeout in milliseconds.
UINT16	Client Configuration Timeout in milliseconds.
UINT8	A flag indicating if a Persistent Group will be used for the connection. The flag is set to 1 if a Persistent Group will be used.
UINT8	A flag indicating if this frame is part of a follow-on provision discovery. The flag is set to 1 if it is part of a follow-on provision discovery.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_RESPONSE_FRAME_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_RESPONSE_FRAME_PARAMETERS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SECONDARY_DEVICE_TYPE_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SECONDARY_DEVICE_TYPE_LIST is a TLV that contains a list of secondary device types.

TLV Type

0x94

Length

The size (in bytes) of the array of WDI_P2P_DEVICE_TYPE elements. An array length of 0 is allowed.

Note WDI_P2P_DEVICE_TYPE is not a WDI structure. It is defined in the WDI TLV parser generator, and is used for documentation purposes only.

Values

Type	Description
WDI_P2P_DEVICE_TYPE[]	An array of Wi-Fi Direct device types.

WDI_P2P_DEVICE_TYPE consists of the following elements.

Type	Description
UINT16	The main type category ID.
UINT8[4]	The OUI that is assigned to this device type.

Type	Description
UINT16	The subcategory ID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SEND_ACTION_FRAME_RESULT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SEND_ACTION_FRAME_RESULT is a TLV that contains information about an Action Frame that was sent to a peer.

TLV Type

0xAF

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SEND_ACTION_FRAME_RESULT_PARAMETERS			The Wi-Fi Direct send Action Frame result parameters.
WDI_TLV_P2P_ACTION_FRAME_IES			The set of IEs sent to the remote device.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SEND_ACTION_FRAME_RESULT_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SEND_ACTION_FRAME_RESULT_PARAMETERS is a TLV that contains Wi-Fi Direct send Action Frame result parameters.

TLV Type

0xAE

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	The device address of the target Wi-Fi Direct device.
UINT8	The Wi-Fi Direct Dialog Token for this transaction.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SEND_ACTION_REQUEST_FRAME_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SEND_ACTION_REQUEST_FRAME_PARAMETERS is a TLV that contains parameters for sending a Wi-Fi Direct action request frame with OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME.

TLV Type

0x8B

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_P2P_ACTION_FRAME_TYPE	The type of request to send.
WDI_MAC_ADDRESS	The MAC address of the target peer device.
UINT8	The Direct Dialog Token for the transaction.
UINT32	The send timeout. The maximum time, in milliseconds, to send the action frame.
UINT32	The post-ACK dwell time. The time, in milliseconds, to remain on the listen channel after the incoming packet is acknowledged.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SEND_REQUEST_ACTION_FRAME_RESULT

Article • 03/14/2023

WDI_TLV_P2P_SEND_REQUEST_ACTION_FRAME_RESULT is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SEND_RESPONSE_ACTION_FRAME_RESULT

Article • 03/14/2023

WDI_TLV_P2P_SEND_RESPONSE_ACTION_FRAME_RESULT is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_INFORMATION

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_INFORMATION is a TLV that contains Wi-Fi Direct Service Information.

TLV Type

0xEE

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The Service Information for the service.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_INFORMATION_DISCOVERY_ENTRY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_INFORMATION_DISCOVERY_ENTRY is a TLV that contains a Service Information Discovery Entry.

TLV Type

0x117

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_NAME			Name of the service (UTF-8), up to 255 bytes.
WDI_TLV_P2P_SERVICE_NAME_HASH			Hash of Service Name.
WDI_TLV_P2P_SERVICE_INFORMATION		X	Request service information to be used for the ANQP query request to download service information for this Service.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SERVICE_UPDATE_INDICATOR		X	Service Update indicator to be used for the ANQP query request.
WDI_TLV_P2P_SERVICE_TRANSACTION_ID		X	Service transaction ID to be used for the ANQP query request.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_INFORMATION_ENTRY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_INFORMATION_ENTRY is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_NAME

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_NAME is a TLV that contains the name of a service.

TLV Type

0xEC

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The name of the service, in UTF-8, up to 255 bytes.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_NAME_HASH

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_NAME_HASH is a TLV that contains the hash of a service name.

TLV Type

0xED

Length

The size (in bytes) of a WDI_P2P_SERVICE_NAME_HASH structure.

Values

Type	Description
WDI_P2P_SERVICE_NAME_HASH	The hash of a WFDS Service Name.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_SESSION_INFO

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_SESSION_INFO is a TLV that contains Service Session information.

TLV Type

0xF0

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Service Session Information.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_STATUS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_STATUS is a TLV that contains the Service Status of a service.

TLV Type

0xFB

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The Service Status of a service.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_TRANSACTION_ID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_TRANSACTION_ID is a TLV that contains the Service transaction ID to be used for the ANQP query request.

TLV Type

0x116

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The Service transaction ID to be used for the ANQP query request.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_TYPE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_TYPE is a TLV that contains the Service Type of the service.

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x129

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The Service Type of the service in UTF-8, up to 21 bytes long.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_TYPE_HASH

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_TYPE_HASH is a TLV that contains the hash of Service Type.

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x12A

Length

The size (in bytes) of a WDI_P2P_SERVICE_NAME_HASH structure.

Values

Type	Description
WDI_P2P_SERVICE_NAME_HASH	The hash of Service Type.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_SERVICE_UPDATE_INDICATOR

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_SERVICE_UPDATE_INDICATOR is a TLV that contains a Wi-Fi Direct service update indicator.

TLV Type

0x115

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT16	The service update indicator to include in ANQP responses if the driver supports responding to service information discovery ANQP requests.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_P2P_WPS_ENABLED

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_P2P_WPS_ENABLED is a TLV that specifies if Wi-Fi Protected Setup is enabled.

TLV Type

0xF7

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	Specifies if Wi-Fi Protected Setup is enabled. Valid values are 0 (not enabled) and 1 (enabled).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

OID_WDI_SET_P2P_WPS_ENABLED

WDI_TLV_PACKET_FILTER_PARAMETERS

Article • 03/14/2023

WDI_TLV_PACKET_FILTER_PARAMETERS is a TLV that contains packet filter parameters for OID_WDI_SET_RECEIVE_PACKET_FILTER.

TLV Type

0x47

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_PACKET_FILTER_TYPE (UINT32)	Specifies the desired Wi-Fi packet filter.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PEER_MAC_ADDRESS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PEER_MAC_ADDRESS is a TLV that contains the MAC address of the peer.

TLV Type

0x4C

Length

The size (in bytes) of a WDI_MAC_ADDRESS structure.

Values

Type	Description
WDI_MAC_ADDRESS	Specifies the Wi-Fi MAC address of the peer.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_CAPABILITIES

Article • 03/14/2023

WDI_TLV_PHY_CAPABILITIES is a TLV that contains PHY capabilities.

TLV Type

0x1B

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_PHY_TYPE	Specifies the PHY types.
UINT8	Specifies whether or not the PHY supports CF Poll.
UINT32	Specifies the MPDU maximum length.
UINT32	Specifies the operating temperature class.
UINT32	Specifies the antenna diversity support.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_DATA_RATE_LIST

Article • 03/14/2023

WDI_TLV_PHY_DATA_RATE_LIST is a TLV that contains a list of data rates.

TLV Type

0x13

Length

The size (in bytes) of the array of WDI_DATA_RATE_LIST elements. The array must contain 1 or more elements.

Note WDI_DATA_RATE_LIST is not a WDI structure. It is defined in the WDI TLV parser generator, and is used for documentation purposes only.

Values

Type	Description
WDI_DATA_RATE_LIST[]	An array of data rates. Each data rate in the array must contain data rate flags and a data rate value.

WDI_DATA_RATE_LIST consists of the following elements.

Type	Description
UINT8	The data rate flags as defined in WDI_DATA_RATE_FLAGS.
UINT16	The data rate value.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_INFO

Article • 03/14/2023

WDI_TLV_PHY_INFO is a TLV that contains PHY information.

TLV Type

0x26

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PHY_CAPABILITIES			The phy capabilities.
WDI_TLV_PHY_TX_POWER_LEVEL_LIST			A list of TX power levels.
WDI_TLV_PHY_DATA_RATE_LIST			A list of data rates.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PHY_LIST is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_STATISTICS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PHY_STATISTICS is a TLV that contains per-PHY statistics for OID_WDI_GET_STATISTICS.

TLV Type

0xA7

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_PHY_TYPE	The type for this PHY.
UINT64	The number of MSDU packets and MMPDU frames that the IEEE PHY layer of the 802.11 station has successfully transmitted.
UINT64	The number of multicast or broadcast MSDU packets and MMPDU frames that the IEEE PHY layer of the 802.11 station has successfully transmitted.
UINT64	The number of MSDU packets and MMPDU frames that the 802.11 station failed to transmit after exceeding the retry limits defined by the 802.11 IEEE dot11ShortRetryLimit or dot11LongRetryLimit MIB counters.

Type	Description
UINT64	The number of MSDU packets and MMPDU frames that the 802.11 station successfully transmitted after one or more attempts.
UINT64	The number of MSDU packets and MMPDU frames that the 802.11 station successfully transmitted after more than one retransmission attempts. For MSDU packets, the port must increment this counter for each packet that was transmitted successfully after one or more of its MPDU fragments required retransmission.
UINT64	The number of MSDU packets and MMPDU frames that the 802.11 station failed to transmit because of a timeout as defined by the IEEE 802.11 dot11MaxTransmitMSDULifetime MIB object.
UINT64	The number of MPDU frames that the 802.11 station transmitted and acknowledged through a received 802.11 ACK frame.
UINT64	The number of times that the 802.11 station received a Clear To Send (CTS) frame in response to a Request To Send (RTS) frame. If this cannot be maintained per port, it can be maintained per channel.
UINT64	The number of times that the 802.11 station did not receive a CTS frame in response to an RTS frame. If this cannot be maintained per port, it can be maintained per channel.
UINT64	The number of times that the 802.11 station expected and did not receive an Acknowledgment (ACK) frame. If this cannot be maintained per port, it can be maintained per channel.

Type	Description
UINT64	The number of MSDU packets and MMPDU frames that the 802.11 station has successfully received. For MSDU packets, the port must increment this counter for each packet whose MPDU fragments were received and passed frame check sequence (FCS) verification and replay detection. The port must increment this member regardless of whether the received MSDU packet or MPDU fragment fail MAC-layer cipher decryption.
UINT64	The number of multicast or broadcast MSDU packets and MMPDU frames that the 802.11 station has successfully received. For MSDU packets, the port must increment this counter for each packet whose MPDU fragments were received and passed FCS verification and replay detection. The port must increment this member regardless of whether the received MSDU packet or MPDU fragment fail MAC-layer cipher decryption.
UINT64	The number of MSDU packets or MMPDU frames received by the 802.11 station when a promiscuous packet filter is enabled. If this cannot be maintained per port, it can be maintained per channel.
UINT64	The number if MSDU packets and MMPDU frames that the 802.11 station discarded because of a timeout as defined by the IEEE 802.11 dot11MaxReceiveLifetime MIB object. If this cannot be maintained per port, it can be maintained per channel.
UINT64	The number of duplicate MPDU frames that the 802.11 station received. The 802.11 station determines duplicate frames through the Sequence Control field of the 802.11 MAC header. If this cannot be maintained per port, it can be maintained per channel.
UINT64	The number of MPDU frames received by the 802.11 station for MSDU packets or MMPDU frames.

Type	Description
UINT64	The number of MPDU frames received by the 802.11 station for MSDU packets or MMPDU frames when a promiscuous packet filter was enabled.
UINT64	The number of MPDU frames that the 802.11 station received with FCS errors. If this cannot be maintained per port, it can be maintained per channel.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_SUPPORTED_RX_DATA_RATES_LIST

Article • 03/14/2023

WDI_TLV_PHY_SUPPORTED_RX_DATA_RATES_LIST is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_SUPPORTED_TX_DATA_RATES_LIST

Article • 03/14/2023

WDI_TLV_PHY_SUPPORTED_TX_DATA_RATES_LIST is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_TX_POWER_LEVEL_LIST

Article • 03/14/2023

WDI_TLV_PHY_TX_POWER_LEVEL_LIST is a TLV that contains a list of power levels.

TLV Type

0x1C

Length

The size (in bytes) of the array of UINT32 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT32[]	An array of UINT32 elements that specifies power levels.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_TYPE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PHY_TYPE is a TLV that contains a PHY type.

TLV Type

0x122

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_PHY_TYPE (UINT32)	The PHY type.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_TYPE_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PHY_TYPE_LIST is a TLV that contains an array of PHY types.

TLV Type

0x19

Length

The size (in bytes) of the array of WDI_PHY_TYPE values. The array must contain 1 or more values.

Values

Type	Description
WDI_PHY_TYPE[]	An array of PHY type values.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PHY_TYPE_LIST (unused)

Article • 03/14/2023

WDI_TLV_PHY_TYPE_LIST (0x69) is an unused TLV.

TLV Type

0x69

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

WDI_TLV_PHY_TYPE_LIST

WDI_TLV_PLDR_SUPPORT

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PLDR_SUPPORT is a TLV that specifies if PLDR (Platform Level Reset) is supported.

Note This TLV was added in Windows 10, version 1511, WDI version 1.0.10.

TLV Type

0x11A

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	Specifies if PLDR is supported. This value is set to 0 if the device or bus does not support reset functionality (usually by querying the ACPI or PCI methods). A non-zero value specifies that reset functionality is supported.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

PLDR

WDI_TLV_PM_CAPABILITIES (0x42)

Article • 03/14/2023

WDI_TLV_PM_CAPABILITIES is a TLV that contains power management capabilities.

TLV Type

0x42

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the power management supported flags. Valid flags are: • NDIS_PM_WAKE_PACKET_INDICATION_SUPPORTED • NDIS_PM_SELECTIVE_SUSPEND_SUPPORTED (0x00000002)
UINT32	Specifies the supported Wake-on-LAN patterns. Valid patterns are: • NDIS_PM_WOL_BITMAP_PATTERN_SUPPORTED (0x00000001) • NDIS_PM_WOL_MAGIC_PACKET_SUPPORTED (0x00000002) • NDIS_PM_WOL_IPV4_TCP_SYN_SUPPORTED (0x00000004) • NDIS_PM_WOL_IPV6_TCP_SYN_SUPPORTED (0x00000008) • NDIS_PM_WOL_IPV4_DEST_ADDR_WILDCARD_SUPPORTED (0x00000200) • NDIS_PM_WOL_IPV6_DEST_ADDR_WILDCARD_SUPPORTED (0x00000800) • NDIS_PM_WOL_EAPOL_REQUEST_ID_MESSAGE_SUPPORTED (0x00010000)
UINT32	Specifies the total number of Wake-on-LAN patterns.
UINT32	Specifies the maximum Wake-on-LAN pattern size.
UINT32	Specifies the maximum Wake-on-LAN pattern offset.

Type	Description
UINT32	Specifies the maximum Wake-on-LAN packet save buffer.
UINT32	Specifies the supported protocol offloads. Valid offloads are: • NDIS_PM_PROTOCOL_OFFLOAD_ARP_SUPPORTED (0x00000001) • NDIS_PM_PROTOCOL_OFFLOAD_NS_SUPPORTED (0x00000002) • NDIS_PM_PROTOCOL_OFFLOAD_80211_RSN_REKEY_SUPPORTED (0x00000080)
UINT32	Specifies the number of ARP offload IPv4 addresses.
UINT32	Specifies the number of NS offload IPv6 addresses.
NDIS_DEVICE_POWER_STATE	Specifies the minimum magic packet wake-up.
NDIS_DEVICE_POWER_STATE	Specifies the minimum pattern wake-up.
NDIS_DEVICE_POWER_STATE	Specifies the minimum link change wake-up.
UINT32	Specifies the supported wake-up events. Valid events are: • NDIS_PM_WAKE_ON_MEDIA_CONNECT_SUPPORTED (0x00000001) • NDIS_PM_WAKE_ON_MEDIA_DISCONNECT_SUPPORTED (0x00000002)
UINT32	Specifies the media-specific wake-up events. Valid events are: • NDIS_WLAN_WAKE_ON_NLO_DISCOVERY_SUPPORTED (0x00000001) • NDIS_WLAN_WAKE_ON_AP_ASSOCIATION_LOST_SUPPORTED (0x00000002) • NDIS_WLAN_WAKE_ON_GTK_HANDSHAKE_ERROR_SUPPORTED (0x00000004) • NDIS_WLAN_WAKE_ON_4WAY_HANDSHAKE_REQUEST_SUPPORTED (0x00000008)

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY

Article • 03/14/2023

WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY is a TLV that contains RSN Rekey protocol offload parameters. If TCK/iGTK key info is configured, drivers must return it when queried in OID_WDI_GET_PM_PROTOCOL_OFFLOAD via this TLV.

TLV Type

0x63

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_TLV_RSN_KEY_INFO	Rsn Eapol key parameters.
LIST<WDI_TLV_CONFIGURED_CIPHER_KEY>	A list of configured ciphers to be set in OID_WDI_GET_PM_PROTOCOL_OFFLOAD. Drivers must return any GTK or iGTK keys that are currently configured.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PM_PROTOCOL_OFFLOAD_GET

Article • 03/14/2023

WDI_TLV_PM_PROTOCOL_OFFLOAD_GET is a TLV that contains a protocol offload ID for OID_WDI_GET_PM_PROTOCOL_OFFLOAD.

TLV Type

0xA8

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the protocol offload ID. This is an OS-provided value that identifies the offloaded protocol. Before the OS sends an Add request down or completes the request to the overlying driver, the OS sets ProtocolOffloadId to a value that is unique among the protocol offloads on a network adapter.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv4ARP

Article • 03/14/2023

WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv4ARP is a TLV that contains IPv4 ARP protocol offload parameters.

TLV Type

0x61

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the protocol offload ID. This is an OS-provided value that identifies the offloaded protocol. Before the OS sends an Add request down or completes the request to the overlying driver, the OS sets ProtocolOffloadId to a value that is unique among the protocol offloads on a network adapter.
UINT8[4]	Specifies an optional IPv4 address to match with the Source Protocol Address (SPA) field of the ARP request. If the incoming ARP request has an SPA value that matches this IPv4 address, the network adapter sends an ARP response when it is in a low power state. If this is set to zero, the network adapter should respond to ARP requests from any remote IPv4 address.
UINT8[4]	Specifies the host IPv4 address the network adapter uses for the Source Protocol Address (SPA) field when sending an ARP response.
WDI_MAC_ADDRESS	Specifies the MAC address that the network adapter must use for the Source Hardware Address (SHA) field of the ARP response packet that it generates. However, it should use the current MAC address of the network adapter as the source address in the MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv6NS

Article • 03/14/2023

WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv6NS is a TLV that contains IPv6 NS protocol offload parameters.

TLV Type

0x62

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the protocol offload ID. This is an OS-provided value that identifies the offloaded protocol. Before the OS sends an Add request down or completes the request to the overlying driver, the OS sets ProtocolOffloadId to a value that is unique among the protocol offloads on a network adapter.
UINT8[16]	Specifies an optional IPv6 address to match with the Source Address field in the IPv6 header of the NS message. If the incoming NS message has a Source Address value that matches this IPv6 address, the network adapter sends a neighbor advertisement (NA) message when it is in a low power state. If this is set to zero, the network adapter should respond to NS messages from any remote IPv6 address.
UINT8[16]	Specifies the solicited node IPv6 address.
UINT8[16]	Specifies one or two IPv6 addresses to match the Target Address field of an incoming NS message. If there is only one address, that address is stored in Target address 1, and Target address 2 is filled with zeros. If one of these addresses matches the Target Address field of an incoming NS message, the network adapter sends an NA message in response.
UINT8[16]	See description of Target address 1.

Type	Description
WDI_MAC_ADDRESS	Specifies the MAC address that the network adapter must use for the target link-layer address (TLLA) field of the NA message that it generates. However, it should use the current MAC address of the network adapter as the source address in the MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PM_PROTOCOL_RSN_OFFLOAD_KEYS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PM_PROTOCOL_RSN_OFFLOAD_KEYS is a TLV that contains currently configured Rsn Eapol key information. This TLV is used in the NDIS_STATUS_WDI_INDICATION_CIPHER_KEY_UPDATED status indication.

TLV Type

0x149

Values

Type	Description
WDI_RSN_OFFLOAD_KEYS_CONTAINER	The currently configured Rsn Eapol key information.

Requirements

Minimum supported client: Windows 10, version 1803

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_TLV_PM_PROTOCOL_OFFLOAD_REMOVE

Article • 03/14/2023

WDI_TLV_PM_PROTOCOL_OFFLOAD_REMOVE is a TLV that contains the protocol offload ID to be removed with OID_WDI_SET_REMOVE_PM_PROTOCOL_OFFLOAD.

TLV Type

0x6C

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the protocol offload ID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PMKID

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PMKID is a TLV that contains a PMKID value.

TLV Type

0x9F

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	A 16-byte PMKID value.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PORT_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_PORT_ATTRIBUTES is a TLV that contains port attributes.

TLV Type

0x29

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_MAC_ADDRESS	Specifies the MAC address associated with the port.
UINT16	Specifies the port number.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_POWER_MANAGMENT_CAPABILITIES

Article • 03/14/2023

WDI_TLV_POWER_MANAGMENT_CAPABILITIES is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_POWER_STATE

Article • 03/14/2023

WDI_TLV_POWER_STATE is a TLV that contains a power state.

TLV Type

0x44

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies a power state. Valid values are: • 0x0001: Exit low power (D0) • 0x0003: Enter low power (D2) • 0x0004: Enter power off (D3, may not actually be powered off on some platforms)

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PRIVACY_EXEMPTION_ENTRY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PRIVACY_EXEMPTION_ENTRY is a TLV that contains a privacy exemption entry.

TLV Type

0x48

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT16	Specifies the IEEE EtherType in big-endian byte order.
WDI_EXEMPTION_ACTION_TYPE	Specifies the action type of the exemption.
WDI_EXEMPTION_PACKET_TYPE	Specifies the type of packet that the exemption applies to.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PROBE_RESPONSE_FRAME

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PROBE_RESPONSE_FRAME is a TLV that contains a probe response frame.

TLV Type

0x9

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the probe response frame.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_PROTOCOL_OFFLOAD_ID

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_PROTOCOL_OFFLOAD_ID is a TLV that contains the protocol offload identifier.

TLV Type

0x166

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	Specifies the the protocol offload identifier.

Requirements

Minimum supported client	Windows 10, version 2004
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_RADIO_STATE

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_RADIO_STATE is a TLV that contains the state of the radio in hardware and software.

TLV Type

0xA1

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The current state of the radio in hardware. Valid values are 0 and 1.
UINT8	The current state of the radio in software. Valid values are 0 and 1.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_RADIO_STATE_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_RADIO_STATE_PARAMETERS is a TLV that contains radio state parameters for OID_WDI_TASK_SET_RADIO_STATE.

TLV Type

0xA0

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The desired radio state. Valid values are 0 (the radio is turned off) and 1 (the radio is enabled).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_RECEIVE_COALESCE_OFFLOAD_CAPABILITIES

Article • 03/14/2023

WDI_TLV_RECEIVE_COALESCE_OFFLOAD_CAPABILITIES is a TLV that contains Rx coalesce offload capabilities.

TLV Type

0xCE

Length

The size (in bytes) of the below values.

Values

Type	Description
UINT8	Specifies whether or not Rx coalesce is enabled for IPv4. Valid values are 0 (not enabled) and 1 (enabled).
UINT8	Specifies whether or not Rx coalesce is enabled for IPv6. Valid values are 0 (not enabled) and 1 (enabled).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_RECEIVE_COALESCING_CAPABILITIES

Article • 03/14/2023

WDI_TLV_RECEIVE_COALESCING_CAPABILITIES is a TLV that contains hardware assisted receive filter capabilities.

TLV Type

0x9A

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Enabled filter types. A bitwise OR of flags that specify the types of receive filters that are enabled. The following flags are valid. NDIS_RECEIVE_FILTER_VMQ_FILTERS_ENABLED Specifies that VMQ filters are enabled. NDIS_RECEIVE_FILTER_PACKET_COALESCING_FILTERS_ENABLED Specifies that NDIS packet coalescing receive filters are enabled.
UINT32	Enabled queue types. A bitwise OR of flags that specify the types of receive queues that are enabled. The following flag is valid. NDIS_RECEIVE_FILTER_VM_QUEUES_ENABLED Specifies that virtual machine (VM) queues are enabled. VM queues are used when the miniport driver is enabled to use the VMQ interface.
UINT32	The number of VM queues that the network adapter supports.
UINT32	Supported VM queue properties. A bitwise OR of flags that specify the VM queue properties that the network adapter supports. The following flags are valid. NDIS_RECEIVE_FILTER_MSI_X_SUPPORTED The network adapter assigned an MSI-X table entry for each receive queue. Network adapters must not use one MSI-X table entry for multiple receive queues. This flag is mandatory for miniport drivers that support the VMQ or SR-IOV interface. NDIS_RECEIVE_FILTER_VM_QUEUE_SUPPORTED

Type	Description
	The network adapter provides the minimum requirements to support VM queue packet filtering. The miniport driver must set this flag if it is enabled to use the VMQ or SR-IOV interface. For more information about VMQ requirements for VM queue packet filtering, see Setting and Clearing VMQ Filters. For more information about SR-IOV requirements for VM queue packet filtering, see Setting a Receive Filter on a Virtual Port. NDIS_RECEIVE_FILTER_LOOKAHEAD_SPLIT_SUPPORTED The network adapter supports VM queues that split an incoming received packet at the lookahead offset. This offset is equal to or greater than the requested lookahead size. The network adapter uses DMA to transfer the lookahead and post-lookahead data to separate shared memory segments. **Note** Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported. Miniport drivers that support this version of NDIS must not set this flag. NDIS_RECEIVE_FILTER_DYNAMIC_PROCESSOR_AFFINITY_CHANGE_SUPPORTED The network adapter supports the ability to dynamically change one of the following processor affinity attributes: • The processor affinity of a VM queue in the VMQ interface. The processor affinity is changed through an OID set request of OID_RECEIVE_FILTER_QUEUE_PARAMETERS. • The processor affinity of a nondefault virtual port (VPort), which was created in the SR-IOV interface and is attached to the PCI Express (PCIe) physical function (PF) of the network adapter. The processor affinity is changed through an OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS. NDIS_RECEIVE_FILTER_INTERRUPT_VECTOR_COALESCING_SUPPORTED The network adapter supports interrupt coalescing for received packets on any of the following: • Multiple VM queues in the VMQ interface. • Multiple VPorts that are attached to the PF in the SR-IOV interface. If this flag is set, the network adapter must coalesce receive interrupts for VM queues or VPorts that have the same processor affinity. NDIS_RECEIVE_FILTER_IMPLAT_MIN_OF_QUEUES_MODE Indicates that the number of VM queues available is the minimum number of queues available from any member of a Load Balancing Failover (LBFO) team. This flag applies to LBFO filters only. This flag is not set for miniports. NDIS_RECEIVE_FILTER_IMPLAT_SUM_OF_QUEUES_MODE Indicates that the number of VM queues available is the sum of all the queues available from every member of an LBFO team. This flag applies to LBFO filters

Type	Description
	only. This flag is not set for miniports. NDIS_RECEIVE_FILTER_PACKET_COALESCING_SUPPORTED_ON_DEFAULT_QUEUE The network adapter supports NDIS packet coalescing. Packet coalescing is only supported on the default receive queue of the network adapter. This receive queue has an identifier of NDIS_DEFAULT_RECEIVE_QUEUE_ID.
UINT32	Supported filter tests. A bitwise OR of flags that specify the test operations that a miniport driver supports. The following flags are valid. NDIS_RECEIVE_FILTER_TEST_HEADER_FIELD_EQUAL_SUPPORTED The network adapter supports testing the selected header field to determine whether it is equal to a given value. **Note** If the miniport driver supports the VMQ or SR-IOV interfaces, it must set this flag. NDIS_RECEIVE_FILTER_TEST_HEADER_FIELD_MASK_EQUAL_SUPPORTED The network adapter supports masking (that is, a bitwise AND) of the selected header field to determine whether the result is equal to a specified value. NDIS_RECEIVE_FILTER_TEST_HEADER_FIELD_NOT_EQUAL_SUPPORTED The network adapter supports testing the selected header field to determine whether it is not equal to a specified value.
UINT32	Supported headers. A bitwise OR of flags that specify the types of network packet headers that a miniport driver can inspect. The following flags are valid. NDIS_RECEIVE_FILTER_MAC_HEADER_SUPPORTED The network adapter can inspect the media access control (MAC) header of a network packet. The **SupportedMacHeaderFields** member defines the various fields from the MAC header that can be inspected. NDIS_RECEIVE_FILTER_ARP_HEADER_SUPPORTED The network adapter can inspect the Address Resolution Protocol (ARP) header of a network packet. The **SupportedArpHeaderFields** member defines the various fields from the ARP header that can be inspected. NDIS_RECEIVE_FILTER_IPV4_HEADER_SUPPORTED The network adapter can inspect the IP version 4 (IPv4) header of a network packet. The **SupportedIPv4HeaderFields** member defines the various fields from the IPv4 header that can be inspected. NDIS_RECEIVE_FILTER_IPV6_HEADER_SUPPORTED The network adapter can inspect the IP version 6 (IPv6) header of a network packet. The **SupportedIPv6HeaderFields** member defines the various fields from the IPv6 header that can be inspected. NDIS_RECEIVE_FILTER_UDP_HEADER_SUPPORTED The network adapter can inspect the User Datagram Protocol (UDP) header of a network packet. The **SupportedIPv6HeaderFields** member defines the various fields from the UDP header that can be inspected.

Type	Description
UINT32	Supported MAC header fields. A bitwise OR of flags that specify the types of MAC header fields that a miniport driver can inspect. The following flags are valid. NDIS_RECEIVE_FILTER_MAC_HEADER_DEST_ADDR_SUPPORTED The network adapter supports inspecting and filtering that are based on the destination MAC address in the MAC header. **Note** Starting with NDIS 6.30, miniport drivers that support the VMQ or SR-IOV interface must set this flag. NDIS_RECEIVE_FILTER_MAC_HEADER_SOURCE_ADDR_SUPPORTED The network adapter supports inspecting and filtering that are based on the source MAC address in the MAC header. NDIS_RECEIVE_FILTER_MAC_HEADER_PROTOCOL_SUPPORTED The network adapter supports inspecting and filtering that are based on the EtherType identifier in the MAC header. For example, the EtherType identifier for IPv4 packets is 0x0800. NDIS_RECEIVE_FILTER_MAC_HEADER_VLAN_ID_SUPPORTED The network adapter supports inspecting and filtering that are based on the VLAN identifier in the MAC header. NDIS_RECEIVE_FILTER_MAC_HEADER_PRIORITY_SUPPORTED The network adapter supports inspecting and filtering that are based on the priority tag in the MAC header. NDIS_RECEIVE_FILTER_MAC_HEADER_PACKET_TYPE_SUPPORTED The network adapter supports inspecting and filtering that are based on the packet type field of the IEEE 802.2 subnetwork access protocol (SNAP) header in an 802.3 MAC header.
UINT32	The maximum number of MAC header filters that the miniport driver supports.
UINT32	Maximum queue groups. This value is reserved.
UINT32	Maximum queues per queue group. This value is reserved.
UINT32	The minimum size, in bytes, that the network adapter supports for lookahead packet buffers. **Note** Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported. Miniport drivers that support this version of NDIS must set this member to zero.

Type	Description
UINT32	The maximum size, in bytes, that the network adapter supports for lookahead packet buffers. **Note** Starting with NDIS 6.30, splitting packet data into separate lookahead buffers is no longer supported. Miniport drivers that support this version of NDIS must set this member to zero.
UINT32	Supported ARP header fields. A bitwise OR of flags that specify the types of ARP header fields that a miniport driver can inspect. The following flags are valid. NDIS_RECEIVE_FILTER_ARP_HEADER_OPERATION_SUPPORTED The network adapter supports receive filtering on the ARP operation field. NDIS_RECEIVE_FILTER_ARP_HEADER_SPA_SUPPORTED The network adapter supports receive filtering on the ARP source protocol address (SPA) field. NDIS_RECEIVE_FILTER_ARP_HEADER_TPA_SUPPORTED The network adapter supports receive filtering on the ARP target protocol address (TPA) field.
UINT32	Supported IPv4 header fields. A bitwise OR of flags that specify the types of IPv4 header fields that a miniport driver can inspect. The following flag is valid. NDIS_RECEIVE_FILTER_IPV4_HEADER_PROTOCOL_SUPPORTED The network adapter supports receive filtering on the IPv4 protocol field.
UINT32	Supported IPv6 header fields. A bitwise OR of flags that specify the types of IPv6 header fields that a miniport driver can inspect. The following flag is valid. NDIS_RECEIVE_FILTER_IPV6_HEADER_PROTOCOL_SUPPORTED The network adapter supports receive filtering on the IPv6 protocol field.
UINT32	Supported UDP header fields. A bitwise OR of flags that specify the types of IPv6 header fields that a miniport driver can inspect. The following flag is valid. NDIS_RECEIVE_FILTER_UDP_HEADER_DEST_PORT_SUPPORTED The network adapter supports receive filtering on the UDP destination port field. **Note** If the received UDP packet contains IPv4 options or IPv6 extension headers, the network adapter can automatically drop the received packet and treat it as if it failed the UDP filter test.

Type	Description
UINT32	The maximum number of tests on packet header fields that can be specified for a single packet coalescing filter. For more information about packet coalescing, see NDIS Packet Coalescing. **Note** Network adapters that support packet coalescing must support five or more packet header fields that can be specified for a single packet coalescing filter. If the adapter does not support packet coalescing, the miniport driver must set this value to zero.
UINT32	The maximum number of packet coalescing receive filters that are supported by the network adapter. **Note** Network adapters that support packet coalescing must support ten or more packet coalescing filters. If the adapter does not support packet coalescing, the miniport driver must set this value to zero.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

NDIS_RECEIVE_FILTER_CAPABILITIES

WDI_TLV_RECEIVE_COALESCING_CONFIG

Article • 03/14/2023

WDI_TLV_RECEIVE_COALESCING_CONFIG is a TLV that contains receive coalescing configuration.

TLV Type

0xDB

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	A unique queue ID to queue packets matching this filter.
UINT32	A filter ID with a value from 1 to the number of filters supported.
UINT32	The maximum coalescing delay in milliseconds.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

OID_WDI_SET_RECEIVE_COALESCING

WDI_TLV_RECEIVE_FILTER_FIELD (0x65)

Article • 03/14/2023

WDI_TLV_RECEIVE_FILTER_FIELD is a TLV that contains a receive filter test criterion for one field in a network header.

TLV Type

0x65

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies a bitwise OR of flags. The possible flag value is WDI_RECEIVE_FILTER_FIELD_MAC_HEADER_VLAN_UNTAGGED_OR_ZERO. If this flag is set, the network adapter must only indicate received packets that pass the following criteria: • The packet's MAC address matches the specified MAC header field test. • The packet either does not contain a VLAN tag or has a VLAN tag with an ID of zero.
NDIS_FRAME_HEADER (UINT32)	Frame header. Specifies the type of the frame header.
NDIS_RECEIVE_FILTER_TEST (UINT32)	Receive filter test. Specifies the type of test that the receive filter performs.
UINT32	Header field. Specifies the protocol-specific header field type with the union as documented in the NDIS_RECEIVE_FILTER_FIELD_PARAMETERS.HeaderField.
UINT8[16]	Field value. Specifies the value that the miniport adapter compares to the corresponding header field value in incoming packets. The location of the header field value is determined by the field type that is specified in the *header field* element. This value is in network byte order and is specified with the union as documented in the NDIS_RECEIVE_FILTER_FIELD_PARAMETERS.FieldValue.

Type	Description
UINT8[16]	Test result value. If the *receive filter test* element is set to ReceiveFilterTestMaskEqual, the network adapter first calculates a result from the value in the *field value* member and the header field value as specified by the *header field* member. The adapter then compares the calculated result with *result value*. This value is specified with the union as documented in the NDIS_RECEIVE_FILTER_FIELD_PARAMETERS.ResultValue.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_REPLAY_COUNTER

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_REPLAY_COUNTER is a TLV that contains a UINT64 value that represents a replay counter.

TLV Type

0x164

Length

The size (in bytes) of a UINT64.

Values

Type	Description
UINT64	A replay counter

Requirements

Minimum supported client	Windows 10, version 2004
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_REQUEST_LCI_REPORT

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_REQUEST_LCI_REPORT is a TLV that contains information for whether a Location Configuration Information (LCI) report should be requested from a target BSS during a Fine Timing Measurement (FTM) request.

This TLV is used in WDI_TLV_FTM_TARGET_BSS_ENTRY.

TLV Type

0x158

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	Possible values: • 0: LCI report not needed. • 1: LCI report should be requested.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_RETRY_AFTER

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_RETRY_AFTER is a TLV that contains the duration, in seconds, that should pass before trying to request a new Fine Timing Measurement (FTM) from a target BSS.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x15A

Length

The size (in bytes) of a UINT16.

Values

Type	Description
UINT16	The duration, in seconds, that should pass before trying to request a new Fine Timing Measurement (FTM) from the target BSS.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_ROAMING_NEEDED_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_ROAMING_NEEDED_PARAMETERS is a TLV that contains the reason for a roam trigger. This is used in the NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED payload.

TLV Type

0x55

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_ASSOC_STATUS	Specifies the reason for a roam trigger. When a OID_WDI_TASK_ROAM is triggered, this reason is forwarded to it.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_RSN_KEY_INFO

Article • 03/14/2023

WDI_TLV_RSN_KEY_INFO is a TLV that contains Rsn Eapol key parameters. This TLV is a value of the WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY TLV.

TLV Type

0x148

Length

The size (in bytes) of the following values.

Values

Type	Description
UINT32	A UINT32 value that specifies the protocol offload ID. This is an OS-provided value that identifies the offloaded protocol. Before the OS sends an Add request down or completes the request to the overlying driver, the OS sets ProtocolOffloadId to a value that is unique among the protocol offloads on a network adapter.
UINT64	A UINT64 value that specifies the replay counter.
UINT8[16]	A UINT8 array that specifies the IEEE 802.11 key confirmation key (KCK).
UINT8[16]	A UINT8 array that specifies the IEEE 802.11 key encryption key (KEK).

Requirements

Minimum supported client: Windows 10, version 1803

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_TLV_RTT

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_RTT is a TLV that contains the measured roundtrip time (RTT), in picoseconds, for a Fine Timing Measurement (FTM) request.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x15C

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The RTT.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_RTT_ACCURACY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_RTT_ACCURACY is a TLV that contains the accuracy, or expected degree of closeness, of a roundtrip time (RTT) measurement to the true value for a Fine Timing Measurement (FTM) request. The unit is in picoseconds.

For example, if the current RTT is 66712.82 picoseconds (10 meters away from the target AP), but it is known through hardware profiling that the measurement could be off by +/-1 meter, then the RTT accuracy is 6671.28 picoseconds. It is the responsibility of the IHV to provide as specific an accuracy as possible based on the profiling of its hardware and the matching conditions when the actual FTM is taken. There are multiple variables affecting FTM accuracy and multiple possibilities for which of these variables can be measured and considered. The reason a more specific accuracy is desirable is because this is useful information that upper layers can consume, such as preferring measurements with higher accuracy when computing a position or to vary the computed position error based off the FTM accuracies. When profiling, a minimum 90% CDF should be used.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x15D

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The RTT.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_RTT_VARIANCE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_RTT_VARIANCE is a TLV that contains the statistical variance of the measurements used to calculate roundtrip time (RTT) during a Fine Timing Measurement (FTM) request, if more than one measurement was used.

This TLV is used in WDI_TLV_FTM_RESPONSE.

TLV Type

0x15E

Length

The size (in bytes) of a UINT64.

Values

Type	Description
UINT64	The statistical variance of the measurements used to calculate the RTT.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_ANTI_CLOGGING_TOKEN

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_ANTI_CLOGGING_TOKEN is a TLV that contains the anti-clogging token for a Simultaneous Authentication of Equals (SAE) Commit request.

This TLV is used in WDI_TLV_SAE_COMMIT_REQUEST.

TLV Type

0x155

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The anti-clogging token.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_COMMIT_REQUEST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_COMMIT_REQUEST is a TLV that contains parameters for a Simultaneous Authentication of Equals (SAE) Commit request.

This TLV is used in the command parameters for OID_WDI_SET_SAE_AUTH_PARAMS.

TLV type

0x150

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SAE_FINITE_CYCLIC_GROUP	UINT16			The Finite Cyclic Group used for SAE authentication.
WDI_TLV_SAE_SCALAR	TLV<LIST<UINT8>>			The Finite Field Element (FFE).

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SAE_ELEMENT	TLV<LIST<UINT8>>			The Encoded Field Element (EFE).
WDI_TLV_SAE_ANTI_CLOGGING_TOKEN	TLV<LIST<UINT8>>			The anti-clogging token as requested by the BSSID.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_COMMIT_RESPONSE

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_COMMIT_RESPONSE is a TLV that contains the Simultaneous Authentication of Equals (SAE) Commit response frame.

This TLV is used in the payload data of NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED.

TLV Type

0x14D

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The SAE Commit response frame.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_CONFIRM

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_CONFIRM is a TLV that contains the Confirm field for a Simultaneous Authentication of Equals (SAE) Confirm request.

This TLV is used in WDI_TLV_SAE_CONFIRM_REQUEST.

TLV Type

0x157

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The Confirm field.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_CONFIRM_REQUEST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_CONFIRM_REQUEST is a TLV that contains parameters for a Simultaneous Authentication of Equals (SAE) Confirm request.

This TLV is used in the command parameters for OID_WDI_SET_SAE_AUTH_PARAMS.

TLV type

0x151

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SAE_SEND_CONFIRM	UINT16			The Send Confirm field, used as an anti-replay counter.
WDI_TLV_SAE_CONFIRM	TLV<LIST<UINT8>>			The Confirm field.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_CONFIRM_RESPONSE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_CONFIRM_RESPONSE is a TLV that contains the Simultaneous Authentication of Equals (SAE) Confirm response frame.

This TLV is used in the payload data of NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED.

TLV Type

0x14E

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The SAE Confirm response frame.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_ELEMENT

Article • 03/14/2023

> **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_ELEMENT is a TLV that contains the Encoded Field Element (EFE) for a Simultaneous Authentication of Equals (SAE) Commit request.

This TLV is used in WDI_TLV_SAE_COMMIT_REQUEST.

TLV Type

0x154

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	A part of the EFE.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_FINITE_CYCLIC_GROUP

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_FINITE_CYCLIC_GROUP is a TLV that contains the Finite Cyclic Group used in a Commit request for Simultaneous Authentication of Equals (SAE) authentication.

This TLV is used in WDI_TLV_SAE_COMMIT_REQUEST.

TLV Type

0x152

Length

The size (in bytes) of a UINT16.

Values

Type	Description
UINT16	The Finite Cyclic Group used for SAE authentication.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_INDICATION_TYPE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_INDICATION_TYPE is a TLV that contains the type of information needed to continue SAE authentication with a target BSSID, or notification that authentication cannot continue.

This TLV is used in the payload data of NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED.

TLV Type

0x14B

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_SAE_INDICATION_TYPE	The type of information needed to continue SAE authentication with a target BSSID, or notification that authentication cannot continue.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_REQUEST_TYPE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_REQUEST_TYPE is a TLV that contains the type of Simultaneous Authentication of Equals (SAE) request frame to send to a target BSSID.

This TLV is used in the command parameters of OID_WDI_SET_SAE_AUTH_PARAMS.

TLV Type

0x14F

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_SAE_REQUEST_TYPE	The type of SAE request frame to send to the BSSID.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_SCALAR

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_SCALAR is a TLV that contains the Finite Field Element (FFE) for a Simultaneous Authentication of Equals (SAE) Commit request.

This TLV is used in WDI_TLV_SAE_COMMIT_REQUEST.

TLV Type

0x153

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	The FFE.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_SEND_CONFIRM

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_SEND_CONFIRM is a TLV that contains the Send Confirm field for a Simultaneous Authentication of Equals (SAE) Confirm request. The Send Confirm field is used as an anti-replay counter.

This TLV is used in WDI_TLV_SAE_CONFIRM_REQUEST.

TLV Type

0x156

Length

The size (in bytes) of a UINT16.

Values

Type	Description
UINT16	The Send Confirm field.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAE_STATUS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SAE_STATUS is a TLV that contains Simultaneous Authentication of Equals (SAE) authentication failure error status.

This TLV is used in the command parameters of OID_WDI_SET_SAE_AUTH_PARAMS and in the payload data of NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED.

TLV Type

0x14C

Length

The size (in bytes) of a UINT32.

Values

Type	Description
WDI_SAE_STATUS	The SAE authentication failure error status.

Requirements

Minimum supported client: Windows 10, version 1903 **Minimum supported server**: Windows Server 2016 **Header**: Wditypes.hpp

WDI_TLV_SAFE_MODE_PARAMETERS

Article • 03/14/2023

WDI_TLV_SAFE_MODE_PARAMETERS is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SCAN_DWELL_TIME

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SCAN_DWELL_TIME is a TLV that contains scanning dwell time settings.

TLV Type

0x7

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the time in milliseconds to dwell on active channels. This is a hint and if the adapter decides to use its own dwell time, it must meet the Maximum Scan Time requirement.
UINT32	Specifies the time in milliseconds to dwell on passive channels. This is a hint and if the adapter decides to use its own dwell time, it must meet the Maximum Scan Time requirement.
UINT32	Specifies the time in milliseconds for total scan. If the adapter limits its dwell times to below the values specified above, it can ignore the Maximum Scan Time parameter.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_SCAN_MODE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SCAN_MODE is a TLV that contains scan mode parameters.

TLV Type

0x6

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The number of times the full scan procedure should be repeated. If this value is set to 0, the scan should be repeated until the task is aborted by the host.
WDI_SCAN_TYPE	Specifies the type of scan that should be performed. If WDI_SCAN_TYPE_ACTIVE is set, the device must only scan active channels.
UINT8	Specifies if live updates are needed and discovered entries must be reported when they are found, with the recommended throttling logic above. This value is always true when the Microsoft component manages the BSS list cache.
WDI_SCAN_TRIGGER	Specifies the trigger for the scan.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SEND_ACTION_FRAME_REQUEST_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SEND_ACTION_FRAME_REQUEST_PARAMETERS is a TLV that contains parameters for OID_WDI_TASK_SEND_REQUEST_ACTION_FRAME.

TLV Type

0xBF

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_CHANNEL_NUMBER (UINT32)	The channel on which to send the action frame and also to linger on as specified in the post-ACK dwell time.
WDI_BAND_ID (UINT32)	The ID of the band on which to send the action frame.
WDI_MAC_ADDRESS	The MAC address of the target access point or peer adapter.
UINT32	The send timeout. Specifies the maximum time (in milliseconds) to send this Action Frame.
UINT32	The post-acknowledgment dwell time. Specifies the time (in milliseconds) to remain on listen channel after the incoming packet is acknowledged.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SEND_ACTION_FRAME_RESPONSE_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SEND_ACTION_FRAME_RESPONSE_PARAMETERS is a TLV that contains parameters for OID_WDI_TASK_SEND_RESPONSE_ACTION_FRAME.

TLV Type

0xE2

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
WDI_CHANNEL_NUMBER (UINT32)	The channel on which to send the action frame and also to linger on as specified in the post-ACK dwell time.
WDI_BAND_ID (UINT32)	The ID of the band on which to send the action frame.
WDI_MAC_ADDRESS	The MAC address of the target access point or peer adapter.
UINT32	The send timeout. Specifies the maximum time (in milliseconds) to send this Action Frame.
UINT32	The post-acknowledgment dwell time. Specifies the time (in milliseconds) to remain on listen channel after the incoming packet is acknowledged.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SET_AUTO_POWER_SAVE

Article • 03/14/2023

WDI_TLV_SET_AUTO_POWER_SAVE is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SET_CIPHER_KEY_INFO

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SET_CIPHER_KEY_INFO is a TLV that contains cipher key mapping key information for OID_WDI_SET_ADD_CIPHER_KEYS.

TLV Type

0x52

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PEER_MAC_ADDRESS		X	Specifies the MAC address of the peer that this key is associated with. If not present, assume this is a default key. At least one of peer MAC Address or cipher key ID must be present. This field must be present when key type is set to WDI_CIPHER_KEY_TYPE_PAIRWISE_KEY, and may be present when key type is set to WDI_CIPHER_KEY_TYPE_GROUP_KEY.
WDI_TLV_CIPHER_KEY_ID		X	Specifies the ID for this cipher key. At least one of peer MAC address or cipher key ID must be present. This field is not required for pairwise keys.
WDI_TLV_CIPHER_KEY_TYPE_INFO			Specifies the cipher key type information.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CIPHER_KEY_RECEIVE_SEQUENCE_COUNT		X	Specifies the initial 48-bit value of Packet Number (PN), which is used for replay protection. This is optional if the cipher algorithm is WDI_CIPHER_ALGO_WEP40, WDI_CIPHER_ALGO_WEP104, or WDI_CIPHER_ALGO_WEP.
WDI_TLV_CIPHER_KEY_CCMP_KEY		X	Specifies the CCMP cipher algorithm key data. This is only present if the cipher algorithm is WDI_CIPHER_ALGO_CCMP.
WDI_TLV_CIPHER_KEY_GCMP_256_KEY		X	Contains GCMP_256 cipher algorithm key data. This is only present if the cipher algorithm is WDI_CIPHER_ALGO_GCMP_256.
WDI_TLV_CIPHER_KEY_TKIP_INFO		X	Specifies the TKIP information. This is only present if the cipher algorithm is WDI_CIPHER_ALGO_TKIP.
WDI_TLV_CIPHER_KEY_BIP_KEY		X	Specifies the BIP key. This is only present if the cipher algorithm is WDI_CIPHER_ALGO_BIP.
WDI_TLV_CIPHER_KEY_BIP_GMAC_256_KEY		X	Contains GMAC_256 cipher algorithm key data. This is only present if cipher algorithm is WDI_CIPHER_ALGO_BIP_GMAC_256.
WDI_TLV_CIPHER_KEY_WEP_KEY		X	Specifies the WEP key. This is only present if the cipher algorithm is WDI_CIPHER_ALGO_WEP40, WDI_CIPHER_ALGO_WEP104, or WDI_CIPHER_ALGO_WEP
WDI_TLV_CIPHER_KEY_IHV_KEY		X	Specifies the IHV cipher key. This is only present if WDI_TLV_CIPHER_KEY_TYPE_INFO is in the range WDI_CIPHER_ALGO_IHV_START to WDI_CIPHER_ALGO_IHV_END.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SET_CLEAR_RECEIVE_COALESCING

Article • 03/14/2023

WDI_TLV_SET_CLEAR_RECEIVE_COALESCING is a TLV that contains a filter ID for OID_WDI_SET_CLEAR_RECEIVE_COALESCING.

TLV Type

0x9B

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The ID of the filter.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SET_ENCAPSULATION_OFFLOAD_V4_PARAMETERS

Article • 03/14/2023

WDI_TLV_SET_ENCAPSULATION_OFFLOAD_V4_PARAMETERS is a TLV that is used by OID_WDI_SET_ENCAPSULATION_OFFLOAD to indicate if IPv4 offloading should be started.

TLV Type

0xFD

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	Specifies if IPv4 offloading should be started. This value is set to NDIS_OFFLOAD_SET_ON if enabled, and set to NDIS_OFFLOAD_SET_OFF if disabled.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

NDIS_OFFLOAD_PARAMETERS

WDI_TLV_SET_ENCAPSULATION_OFFLOAD_V6_PARAMETERS

Article • 03/14/2023

WDI_TLV_SET_ENCAPSULATION_OFFLOAD_V6_PARAMETERS is a TLV that is used by OID_WDI_SET_ENCAPSULATION_OFFLOAD to indicate if IPv6 offloading should be started.

TLV Type

0xFE

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	Specifies if IPv6 offloading should be started. This value is set to NDIS_OFFLOAD_SET_ON if enabled, and set to NDIS_OFFLOAD_SET_OFF if disabled.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

NDIS_OFFLOAD_PARAMETERS

WDI_TLV_SET_POWER_DX_REASON

Article • 03/14/2023

WDI_TLV_SET_POWER_DX_REASON is a TLV that contains the reason for a set power Dx.

TLV Type

0x103

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The reason for a set power Dx. Valid values are: - WDI_SET_POWER_DX_REASON_SELETIVE_SUSPEND (1) When this value is set, it implies waking on any interesting external events without explicit **WDI_TLV_ENABLE_WAKE_EVENTS**. This is an idle low power where the device functions transparently to end users as if it were in D0. See WDI USB remote wake sequence for more information.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SET_RECEIVE_COALESCING

Article • 03/14/2023

WDI_TLV_SET_RECEIVE_COALESCING is a TLV that contains received packet coalescing parameters for OID_WDI_SET_RECEIVE_COALESCING.

TLV Type

0x64

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_RECEIVE_COALESCING_CONFIG			Specifies coalescing filter configuration.
WDI_TLV_RECEIVE_FILTER_FIELD	X	X	Specifies a receive filter field.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SSID

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SSID is a TLV that contains an SSID.

TLV Type

0x3B

Length

The size (in bytes) of the array of UINT8 elements. An array length of 0 is allowed.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies an SSID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SSID_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SSID_LIST is an unused TLV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SSID_OFFLOAD

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SSID_OFFLOAD is a TLV that contains an SSID and hints about the SSID.

TLV Type

0x9E

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SSID			The SSID.
WDI_TLV_UNICAST_ALGORITHM_LIST			The unicast algorithm list.
WDI_TLV_CHANNEL_LIST			The channel list.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_START_AP_PARAMETERS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_START_AP_PARAMETERS is a TLV that contains the parameters for OID_WDI_TASK_START_AP.

TLV Type

0xAB

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	The beacon period. If non-zero, this parameter specifies the beacon interval.
UINT32	The DTIM period. If non-zero, this parameter specifies the number of beacon intervals between transmissions of beacon frames that contain a TIM element with a DTIM Count field that equals zero. This value is transmitted in the DTIM Period field of beacon frames.
UINT8	This parameter sets the dot11ExcludeUnencrypted MIB. Valid values are 0 and 1.

Type	Description
UINT8	This parameter specifies if the device supports 802.11b speeds. Valid values are 0 (not supported) and 1 (supported). When this value is set to 1, the access point should allow clients using 11b rates to connect to it.
UINT8	Added in Windows 10, version 1511, WDI version 1.0.10. This parameter specifies whether to allow legacy SoftAP clients to connect. Valid values are 0 (not allowed) and 1 (allowed).
UINT8	Added in Windows 10, version 1511, WDI version 1.0.10. MustUseSpecifiedChannels. This parameter specifies whether the AP can only be started on the channels specified in OID_WDI_TASK_START_AP task parameters with **WDI_TLV_AP_BAND_CHANNEL**. Valid values are 0 and 1. If it is set to 1, the AP can only be started from the specified list. If it is not set, the list is meant to be a recommendation of channels that the firmware can pick from, and it may pick another channel if it is not possible to start the AP on any of the specified channels.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

OID_WDI_TASK_START_AP

WDI_TLV_STATION_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_STATION_ATTRIBUTES is a TLV that contains the attributes of a station.

TLV Type

0x22

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_STATION_CAPABILITIES			The station capabilities.
WDI_TLV_UNICAST_ALGORITHM_LIST		X	The supported unicast algorithms.
WDI_TLV_MULTICAST_DATA_ALGORITHM_LIST		X	The supported multicast data algorithms.
WDI_TLV_MULTICAST_MGMT_ALGORITHM_LIST		X	The supported multicast management algorithms.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_STATION_CAPABILITIES

Article • 03/14/2023

WDI_TLV_STATION_CAPABILITIES is a TLV that contains the capabilities of a station.

TLV Type

0x11

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	The scan SSID list size.
UINT32	The desired BSSID list size.
UINT32	The desired SSID list size.
UINT32	The privacy exemption list size.
UINT32	The key mapping table size.
UINT32	The default key table size.
UINT32	The maximum length of the WEP key value.
UINT32	The maximum number per STA default key tables.
UINT8	Supported QoS flags. Specifies whether the device supports WMM. Valid values are 0 (not supported) and 1 (supported).

Type	Description
UINT8	Specifies whether host-FIPS mode is implemented. If the field is set to DOT11_EXTSTA_ATTRIBUTES_SAFEMODE_OID_SUPPORTED with no other bits set, the driver implements the 802.11 safe mode of operation. If the field is set to DOT11_EXTSTA_ATTRIBUTES_SAFEMODE_CERTIFIED, the NIC has received a validation certificate from the National Institute of Standards and Technology (NIST) under Federal Information Processing Standards (FIPS) Publication 140-2, Security Requirements for Cryptographic Modules. In this mode, the hardware is responsible for ensuring compliance to FIPS standard. If the field is set to zero (0), FIPS mode is not implemented by the NIC.
UINT8	Specifies whether 802.11w MFP capability is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether auto power save is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether the adapter maintains the Station BSS List cache. Valid values are 0 (no) and 1 (yes).
UINT8	Specifies whether the adapter may attempt association to a BSSID that is not specified in the Preferred BSSID list during a Station connect. Valid values are 0 (no) and 1 (yes).
UINT32	The maximum supported Network Offload List size.
UINT8	Specifies whether or not the adapter can track HESSIDs associated with SSIDs and connect/roam only to those APs that match the specified SSID+HESSID. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether the adapter can offload connectivity to networks belonging to specific HESSIDs.
UINT8	Specifies whether disconnected standby is supported. Valid values are 0 (not supported) and 1 (supported).
UINT8	Specifies whether the driver supports the Fine Time Measurement (FTM) protocol as an initiator. Valid values are 0 (not supported) and 1 (supported).
UINT8	The maximum number of target STAs that can be queried per FTM request task.

Type	Description
UINT8	Specifics whether the driver supports the FIPS mode. If the field is set to DOT11_EXTSTA_ATTRIBUTES_SAFEMODE_OID_SUPPORTED with no other bits set, the driver implements the 802.11 safe mode of operation. If the field is set to DOT11_EXTSTA_ATTRIBUTES_SAFEMODE_CERTIFIED the NIC has received a validation certificate from the National Institute of Standards and Technology (NIST) under Federal Information Processing Standards (FIPS) Publication 140-2, Security Requirements for Cryptographic Modules. In this mode the hardware is responsible for ensuring compliance to FIPS standard. If the field is set to zero (0), FIPS mode is not implemented by the NIC. **Operating system support for FIPS is anticipated in a future release of Windows.** **NOTE** that FIPS mode requires the driver to indicate support for WDI_AUTH_ALGO_WPA3 auth and WDI_CIPHER_ALGO_GCMP_256 cipher pairs in the unicast and multicast algo pairs. It must also indicate support for WDI_AUTH_ALGO_WPA3 auth and WDI_CIPHER_ALGO_BIP_GMAC_256 cipher in the Multicast Management algo pairs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_STATUS

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_STATUS is a TLV that contains a status value.

TLV Type

0x1

Length

The size (in bytes) of an NDIS_STATUS.

Values

Type	Description
NDIS_STATUS	The NDIS_STATUS value.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_SUPPORTED_GUIDS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_SUPPORTED_GUIDS is a TLV that contains a supported NDIS GUID.

Note This TLV was added in Windows 10, version 1607, WDI version 1.0.21.

TLV Type

0x130

Length

The size (in bytes) of a NDIS_GUID structure.

Values

Type	Description
NDIS_GUID	A supported NDIS GUID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

OID_WDI_GET_ADAPTER_CAPABILITIES

WDI_TLV_TCP_OFFLOAD_CAPABILITIES

Article • 03/14/2023

WDI_TLV_TCP_OFFLOAD_CAPABILITIES is a TLV that contains TCP/IP offload capabilities.

Capability values are reported as documented in **NDIS_TCP_IP_CHECKSUM_OFFLOAD**. Use NDIS_OFFLOAD_NOT_SUPPORTED and NDIS_OFFLOAD_SUPPORTED when indicating capabilities through OID_WDI_GET_ADAPTER_CAPABILITIES. For a connection with FIPS mode, offloads are turned OFF by the UE.

TLV Type

0xCA

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CHECKSUM_OFFLOAD_CAPABILITIES			Checksum offload capabilities.
WDI_TLV_LSO_V1_CAPABILITIES			Large Send Offload V1 capabilities.
WDI_TLV_LSO_V2_CAPABILITIES			Large Send Offload V2 capabilities.
WDI_TLV_RECEIVE_COALESCE_OFFLOAD_CAPABILITIES			Receive Offload capabilities.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_OFFLOAD_SCOPE			Indicates whether offloads apply to the STA port only or on all ports. Currently applicable to 802.11ad adapters only.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_TCP_RSC_STATISTICS_PARAMETERS

Article • 03/14/2023

WDI_TLV_TCP_RSC_STATISTICS_PARAMETERS is a TLV that contains TCP RSC statistics for OID_WDI_TCP_RSC_STATISTICS.

TLV Type

0xF3

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT64	The total number of packets that were coalesced.
UINT64	The total number of bytes that were coalesced.
UINT64	The total number of coalescing events, which is the total number of packets that were formed from coalescing packets.
UINT64	The total number of RSC abort events, which is the number of exceptions other than the IP datagram length being exceeded. This count should include the cases where a packet is not coalesced because of insufficient hardware resources.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_TCP_SET_OFFLOAD_PARAMETERS

Article • 03/14/2023

WDI_TLV_TCP_SET_OFFLOAD_PARAMETERS is a TLV that contains TCP offload capabilities of a miniport adapter for OID_WDI_SET_TCP_OFFLOAD_PARAMETERS.

TLV Type

0xF2

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The IPv4 checksum setting of the miniport adapter. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. - **NDIS_OFFLOAD_PARAMETERS_TX_RX_DISABLED** - Disabled. - **NDIS_OFFLOAD_PARAMETERS_TX_ENABLED_RX_DISABLED** - Enabled for transmit and disabled for receive. - **NDIS_OFFLOAD_PARAMETERS_RX_ENABLED_TX_DISABLED** - Enabled for receive and disabled for transmit. - **NDIS_OFFLOAD_PARAMETERS_TX_RX_ENABLED** - Enabled for transmit and receive.

Type	Description
UINT8	The IPv4 checksum setting for TCP packets. Valid values are: • **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_DISABLED** - Disabled. • **NDIS_OFFLOAD_PARAMETERS_TX_ENABLED_RX_DISABLED** - Enabled for transmit and disabled for receive. • **NDIS_OFFLOAD_PARAMETERS_RX_ENABLED_TX_DISABLED** - Enabled for receive and disabled for transmit. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_ENABLED** - Enabled for transmit and receive.
UINT8	The IPv4 checksum setting for UDP packets. Valid values are: • **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_DISABLED** - Disabled. • **NDIS_OFFLOAD_PARAMETERS_TX_ENABLED_RX_DISABLED** - Enabled for transmit and disabled for receive. • **NDIS_OFFLOAD_PARAMETERS_RX_ENABLED_TX_DISABLED** - Enabled for receive and disabled for transmit. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_ENABLED** - Enabled for transmit and receive.
UINT8	The IPv6 checksum setting for TCP packets. Valid values are: • **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_DISABLED** - Disabled. • **NDIS_OFFLOAD_PARAMETERS_TX_ENABLED_RX_DISABLED** - Enabled for transmit and disabled for receive. • **NDIS_OFFLOAD_PARAMETERS_RX_ENABLED_TX_DISABLED** - Enabled for receive and disabled for transmit. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_ENABLED** - Enabled for transmit and receive.

Type	Description
UINT8	The IPv6 checksum setting for UDP packets. Valid values are: • **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_DISABLED** - Disabled. • **NDIS_OFFLOAD_PARAMETERS_TX_ENABLED_RX_DISABLED** - Enabled for transmit and disabled for receive. • **NDIS_OFFLOAD_PARAMETERS_RX_ENABLED_TX_DISABLED** - Enabled for receive and disabled for transmit. • **NDIS_OFFLOAD_PARAMETERS_TX_RX_ENABLED** - Enabled for transmit and receive.
UINT8	The Large Send Offload version 1 (LSOV1) setting. Valid values are: • **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. • **NDIS_OFFLOAD_PARAMETERS_LSOV1_ENABLED** - LSOV1 is enabled. • **NDIS_OFFLOAD_PARAMETERS_LSOV1_DISABLED** - LSOV1 is disabled.
UINT8	The Internet Protocol Security (IPsec) offload setting. Valid values are: • **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. • **NDIS_OFFLOAD_PARAMETERS_IPSECV1_DISABLED** - IPsec offload is disabled. • **NDIS_OFFLOAD_PARAMETERS_IPSECV1_AH_ENABLED** - The IPsec offload Authentication Header (AH) feature should be enabled for transmit and receive. • **NDIS_OFFLOAD_PARAMETERS_IPSECV1_ESP_ENABLED** - The IPsec offload Encapsulating Security Payload (ESP) feature should be enabled for transmit and receive. • **NDIS_OFFLOAD_PARAMETERS_IPSECV1_AH_AND_ESP_ENABLED** - The IPsec offload AH and ESP features are enabled for transmit and receive.

Type	Description
UINT8	The IPv4 Large Send Offload version 2 (LSOV2) setting. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. - **NDIS_OFFLOAD_PARAMETERS_LSOV2_ENABLED** - LSOV2 for IPv4 is enabled. - **NDIS_OFFLOAD_PARAMETERS_LSOV2_DISABLED** - LSOV2 for IPv4 is disabled.
UINT8	The IPv6 Large Send Offload version 2 (LSOV2) setting. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. - **NDIS_OFFLOAD_PARAMETERS_LSOV2_ENABLED** - LSOV2 for IPv6 is enabled. - **NDIS_OFFLOAD_PARAMETERS_LSOV2_DISABLED** - LSOV2 for IPv6 is disabled.
UINT8	The IPv4 connection offload setting. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting.
UINT8	The IPv6 connection offload setting. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting.
UINT8	Indicates Receive Segment Coalescing state for IPv4. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The RSC state is unchanged. - **NDIS_OFFLOAD_PARAMETERS_RSC_ENABLED** - The RSC state is enabled. - **NDIS_OFFLOAD_PARAMETERS_RSC_DISABLED** - The RSC state is disabled.

Type	Description
UINT8	Indicates Receive Segment Coalescing state for IPv6. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The RSC state is unchanged. - **NDIS_OFFLOAD_PARAMETERS_RSC_ENABLED** - The RSC state is enabled. - **NDIS_OFFLOAD_PARAMETERS_RSC_DISABLED** - The RSC state is disabled.
UINT32	The value is a bitwise OR of flags. This must be set to 0. There are no flags currently defined.
UINT8	The Internet protocol security (IPsec) offload version 2 setting of a miniport adapter that supports both IPv6 and IPv4. This specifies the setting for both IPv6 and IPv4 support. Valid values are: - **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. - **NDIS_OFFLOAD_PARAMETERS_IPSECV2_DISABLED** - IPsec offload version 2 is disabled. - **NDIS_OFFLOAD_PARAMETERS_IPSECV2_AH_ENABLED** - The IPsec offload version 2 Authentication Header (AH) feature should be enabled for transmit and receive. - **NDIS_OFFLOAD_PARAMETERS_IPSECV2_ESP_ENABLED** - The IPsec offload version 2 Encapsulating Security Payload (ESP) feature should be enabled for transmit and receive. - **NDIS_OFFLOAD_PARAMETERS_IPSECV2_AH_AND_ESP_ENABLED** - The IPsec offload version 2 AH and ESP features are enabled for transmit and receive.

Type	Description
UINT8	The Internet protocol security (IPsec) offload version 2 setting of a miniport adapter that supports IPv4 and does not support IPv6. If the miniport driver supports IPv6, the IPsecV2 member specifies the IPv4 setting and this member is not used. Valid values are: • **NDIS_OFFLOAD_PARAMETERS_NO_CHANGE** - The miniport driver should not change the current setting. • **NDIS_OFFLOAD_PARAMETERS_IPSECV2_DISABLED** - IPsec offload version 2 is disabled. • **NDIS_OFFLOAD_PARAMETERS_IPSECV2_AH_ENABLED** - The IPsec offload version 2 Authentication Header (AH) feature should be enabled for transmit and receive. • **NDIS_OFFLOAD_PARAMETERS_IPSECV2_ESP_ENABLED** - The IPsec offload version 2 Encapsulating Security Payload (ESP) feature should be enabled for transmit and receive. • **NDIS_OFFLOAD_PARAMETERS_IPSECV2_AH_AND_ESP_ENABLED** - The IPsec offload version 2 AH and ESP features are enabled for transmit and receive.
UINT8	Encapsulated Packet Task Offload. A protocol driver sets this field to one of the following values. • **NDIS_OFFLOAD_SET_NO_CHANGE** (0) - The NVGRE task offload state is unchanged. • **NDIS_OFFLOAD_SET_ON** (1) - Enables NVGRE task offloads. • **NDIS_OFFLOAD_SET_OFF** (2) - Disables NVGRE task offloads.
UINT8	Encapsulation types. This field is effective only when the Encapsulated Packet Task Offload is set to **NDIS_OFFLOAD_SET_ON**. If the Encapsulated Packet Task Offload member is not set to **NDIS_OFFLOAD_SET_ON**, this member is zero. A protocol driver must set Encapsulation Types to the bitwise OR of the flags corresponding to encapsulation types that it requires. It can select from the following flags. • **NDIS_ENCAPSULATION_TYPE_GRE_MAC** (0x00000001) - Specifies GRE MAC encapsulation (NVGRE).

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

See also

NDIS_OFFLOAD_PARAMETERS

OID_WDI_SET_TCP_OFFLOAD_PARAMETERS

WDI_TLV_TKIP_MIC_FAILURE_INFO

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_TKIP_MIC_FAILURE_INFO is a TLV that contains TKIP-MIC failure information.

TLV Type

0x57

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	Specifies which cipher key type detected that the TKIP-MIC failure occurred. If this value is 1, the TKIP-MIC failure was detected through a default cipher key. If this value is 0, the TKIP-MIC failure was detected through a key mapping cipher key.
UINT32	Specifies the index of the cipher key in the default key array. Valid value range is from 0 through 3.
WDI_MAC_ADDRESS	Specifies the MAC address of the peer that transmitted the packet that failed MIC verification.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Wditypes.hpp

WDI_TLV_UNICAST_ALGORITHM_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_UNICAST_ALGORITHM_LIST is a TLV that contains an array of unicast data algorithm pairs.

TLV Type

0x13

Length

The size (in bytes) of the array of WDI_ALGO_PAIRS elements. The array must contain 1 or more elements.

Note WDI_ALGO_PAIRS is not a WDI structure. It is defined in the WDI TLV parser generator, and is used for documentation purposes only.

Values

Type	Description
WDI_ALGO_PAIRS[]	An array of authentication and cipher algorithm pairs.

WDI_ALGO_PAIRS consists of the following elements.

Type	Description
UINT8	Authentication algorithm as defined in WDI_AUTH_ALGORITHM.
UINT8	Cipher algorithm as defined in WDI_CIPHER_ALGORITHM.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_UNICAST_CIPHER_ALGO_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_UNICAST_CIPHER_ALGO_LIST is a TLV that contains a list of unicast cipher algorithms.

TLV Type

0x3E

Length

The size (in bytes) of the array of WDI_CIPHER_ALGORITHM structures. The array must contain 1 or more elements.

Values

Type	Description
WDI_CIPHER_ALGORITHM[]	An array of unicast cipher algorithms.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_UNREACHABLE_DETECTION_THRESHOLD

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_UNREACHABLE_DETECTION_THRESHOLD is a TLV that contains the unreachable detection threshold.

TLV Type

0xB1

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The unreachable detection threshold. Specifies the maximum amount of time, in milliseconds, before the 802.11 station determines that it has lost connectivity to a peer device. The station must include missed beacons in making this connectivity loss determination but can also use any other heuristics it desires.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_VENDOR_SPECIFIC_IE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_VENDOR_SPECIFIC_IE is a TLV that contains a list of vendor-specific IEs.

TLV Type

0x5

Length

The size (in bytes) of the array of UINT8 elements. The array must contain 1 or more elements.

Values

Type	Description
UINT8[]	An array of UINT8 elements that specifies the vendor-specific IEs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_VIRTUALIZATION_ATTRIBUTES

Article • 03/14/2023

WDI_TLV_VIRTUALIZATION_ATTRIBUTES is a TLV that contains virtualization attributes.

TLV Type

0x24

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_VIRTUALIZATION_CAPABILITIES			The virtualization capabilities.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_VIRTUALIZATION_CAPABILITIES

Article • 03/14/2023

WDI_TLV_VIRTUALIZATION_CAPABILITIES is a TLV that contains virtualization capabilities.

TLV Type

0x10

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT8	The number of supported ExtSTA ports.
UINT8	The number of supported Wi-Fi Direct Group ports. This count should only include Role ports. If this value is non-zero, it is assumed that a Wi-Fi Direct Device port is available.
UINT8	The number of supported legacy ExtAP ports.
UINT8	The maximum number of supported simultaneous AP/WFD Group Owners.
UINT8	The maximum number of separate channels that the device can operate in and maintain data connections on simultaneously. This limit should not include temporary multichannel operations like scans and Wi-Fi Direct negotiations.
UINT8	The maximum number of supported simultaneous STA/WFD clients.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_WAKE_PACKET_BITMAP_PATTERN

Article • 03/14/2023

WDI_TLV_WAKE_PACKET_BITMAP_PATTERN is a TLV that contains a wake-on-LAN pattern.

TLV Type

0x5B

Length

The sum (in bytes) of the sizes of all contained TLVs.

Values

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_WAKE_PACKET_BITMAP_PATTERN_ID			Specifies the wake-on-LAN pattern ID.
WDI_TLV_BITMAP_PATTERN			Specifies the wake-on-LAN pattern.
WDI_TLV_BITMAP_PATTERN_MASK			Specifies the wake-on-LAN pattern mask. The length is (PatternLength + 7)/8.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_WAKE_PACKET_BITMAP_PATTERN_ID

Article • 03/14/2023

WDI_TLV_WAKE_PACKET_BITMAP_PATTERN_ID is a TLV that contains a wake-on-LAN pattern ID.

The pattern ID is an OS-provided value that identifies the wake-on-LAN pattern and is set to a value that is unique among the wake-on-LAN patterns on a network adapter. The pattern ID is set before the OS sends an add to the underlying drivers or completes the request to the overlying driver.

TLV Type

0xE3

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	The wake-on-LAN pattern ID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

See also

OID_WDI_SET_ADD_WOL_PATTERN

WDI_TLV_WAKE_PACKET_EAPOL_REQUEST_ID_MESSAGE

Article • 03/14/2023

WDI_TLV_WAKE_PACKET_EAPOL_REQUEST_ID_MESSAGE is a TLV that contains the wake-on-LAN pattern ID of a EAPOL request ID message.

TLV Type

0x5F

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the wake-on-LAN pattern ID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_WAKE_PACKET_IPv4_TCP_SYNC

Article • 03/14/2023

WDI_TLV_WAKE_PACKET_IPv4_TCP_SYNC is a TLV that contains wake-on-LAN IPv4 TCP sync packet information.

TLV Type

0x5D

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the wake-on-LAN pattern ID.
UINT8[4]	Specifies the IPv4 source address in the TCP SYN packet.
UINT8[4]	Specifies the IPv4 destination address in the TCP SYN packet.
UINT16	Specifies the TCP source port number in the TCP SYN packet.
UINT16	Specifies the TCP destination port number in the TCP SYN packet.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_WAKE_PACKET_IPv6_TCP_SYNC

Article • 03/14/2023

WDI_TLV_WAKE_PACKET_IPv6_TCP_SYNC is a TLV that contains wake-on-LAN IPv6 TCP sync packet information.

TLV Type

0x5E

Length

The sum (in bytes) of the sizes of all contained elements.

Values

Type	Description
UINT32	Specifies the WoL pattern ID.
UINT8[16]	Specifies the IPv6 source address in the TCP SYN packet.
UINT8[16]	Specifies the IPv6 destination address in the TCP SYN packet.
UINT16	Specifies the TCP source port number in the TCP SYN packet.
UINT16	Specifies the TCP destination port number in the TCP SYN packet.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_WAKE_PACKET_MAGIC_PACKET

Article • 03/14/2023

WDI_TLV_WAKE_PACKET_MAGIC_PACKET is a TLV that contains a pattern ID of a magic packet for OID_WDI_SET_ADD_WOL_PATTERN.

TLV Type

0x5C

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the wake-on-LAN magic packet pattern ID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_WAKE_PACKET_PATTERN_REMOVE

Article • 03/14/2023

WDI_TLV_WAKE_PACKET_PATTERN_REMOVE is a TLV that contains the wake packet pattern ID to be removed with OID_WDI_SET_REMOVE_WOL_PATTERN.

TLV Type

0x6B

Length

The size (in bytes) of a UINT32.

Values

Type	Description
UINT32	Specifies the wake packet pattern ID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI_TLV_WFD_ASSOCIATION_STATUS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

WDI_TLV_WFD_ASSOCIATION_STATUS is a TLV that contains the status code to be set when an association request is denied.

TLV Type

0x126

Length

The size (in bytes) of a UINT8.

Values

Type	Description
UINT8	The DOT11_WFD_STATUS_CODE to be set when an association request is denied.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Wditypes.hpp

WDI USB Selective Suspend Topics

Article • 03/14/2023

To save power, a USB device (such as a Wi-Fi NIC on USB) may go to low power state when idle. WDI supports the LE by utilizing NDIS native support for USB Selective Suspend.

In this section:

WDI NDIS idle detection

WDI and WLAN Selective Suspend capability

WDI USB suspend sequence

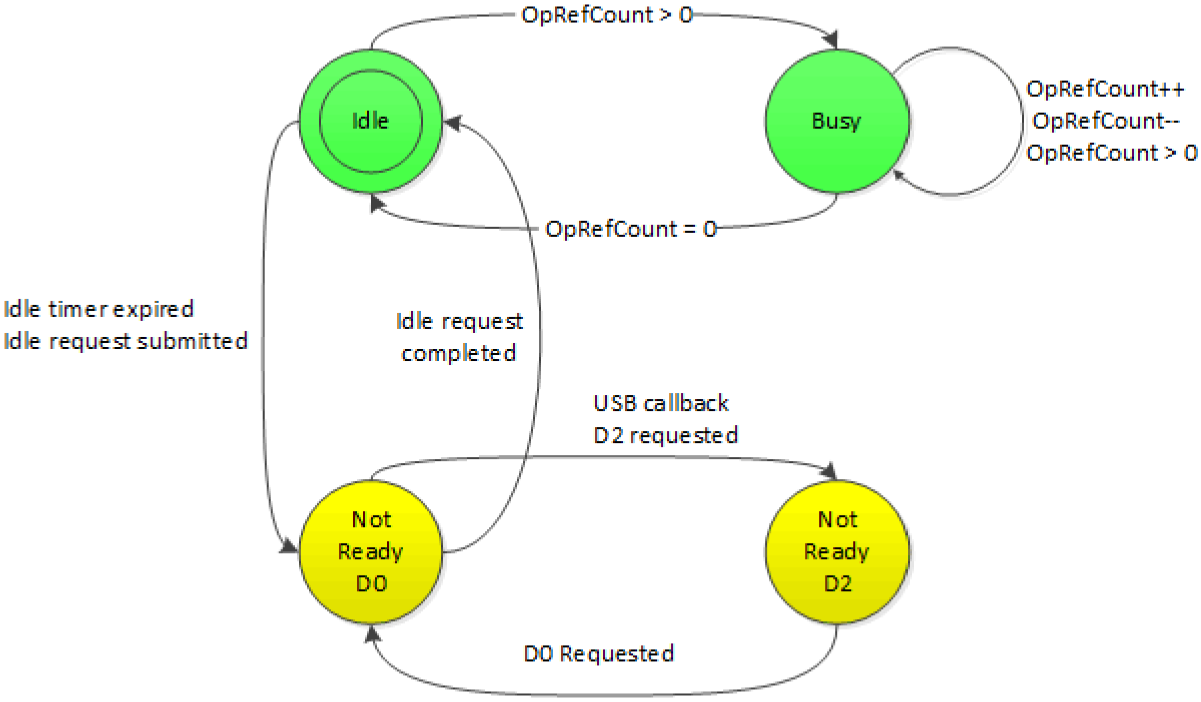
WDI USB resume sequence

WDI USB remote wake sequence

WDI NDIS idle detection

Article • 03/14/2023

This following diagram shows a simple state diagram of NDIS idle detection, which is used to drive USB selective suspend.



If the WDI device/driver supports USB selective suspend, NDIS detects its idle state to send the device into low power state (D2).

WDI and WLAN Selective Suspend capability

Article • 03/14/2023

To enable USB Selective Suspend support, the LE must report the capability. NDIS defines keywords for this feature. For more information, see Standardized INF Keywords for NDIS Selective Suspend.

- *SelectiveSuspend : {Enable, Disable}

- *SSIdleTimeout : idle timeout in seconds

WDI enables support based on the following sources.

- Device INF: This is written to the next item at device setup from the above keyword.
- Registry settings: This is set from the INF or the Advanced property sheet for the device in Device Manager.
- Power management capabilities in the return from OID_WDI_GET_ADAPTER_CAPABILITIES.
- Idle handlers in **NDIS_MINIPORT_DRIVER_WDI_CHARACTERISTICS**.
  - *MiniportWdiIdleNotification*
  - *MiniportWdiCancelIdleNotification*

The WDI driver exposes two callback functions for the LE.

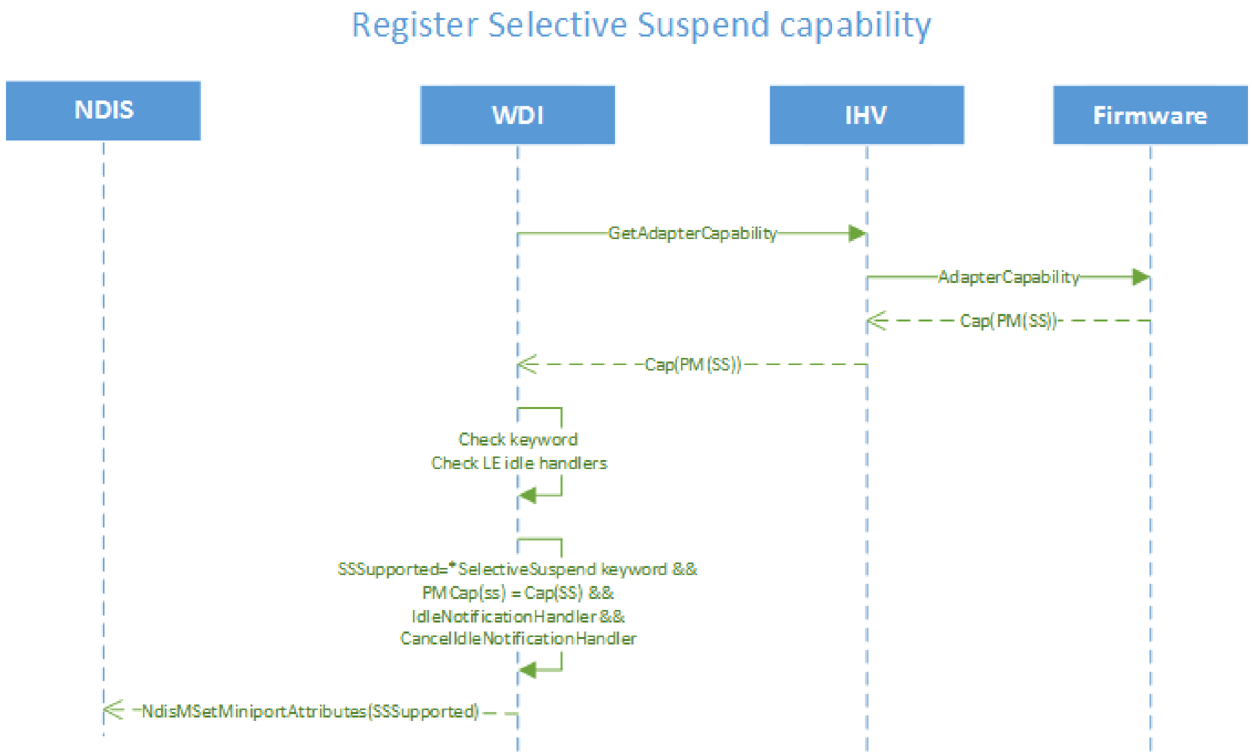
- **IdleNotificationComplete**

- **IdleNotificationConfirm**

WDI Selective Suspend capability registration

Article • 03/14/2023

The following is a flow diagram for registering the USB Selective Suspend capability.


Register Selective Suspend capability
NDIS
WDI
IHV
Firmware
GetAdapterCapability
AdapterCapability
Cap(PM(SS))
Cap(PM (SS))
Check keyword
Check LE idle handlers
SSSupported=*SelectiveSuspend keyword &&
PM Cap(ss) = Cap(SS) &&
IdleNotificationHandler &&
CancelIdleNotificationHandler
NdisMSetMiniportAttributes(SSSupported)

AdapterCap(PM(ss)), *SelectiveSuspend, **LeIdleNotificationHandler**, and **LeCancelIdleNotificationHandler** must be true or valid for WDI to register that WLAN supports Selective Suspend.

When WDI decides that Selective Suspend can be supported, WDI also registers an optional handler to NDIS.

Related topics

MiniportWdiCancelIdleNotification

MiniportWdiIdleNotification

WDI USB suspend sequence

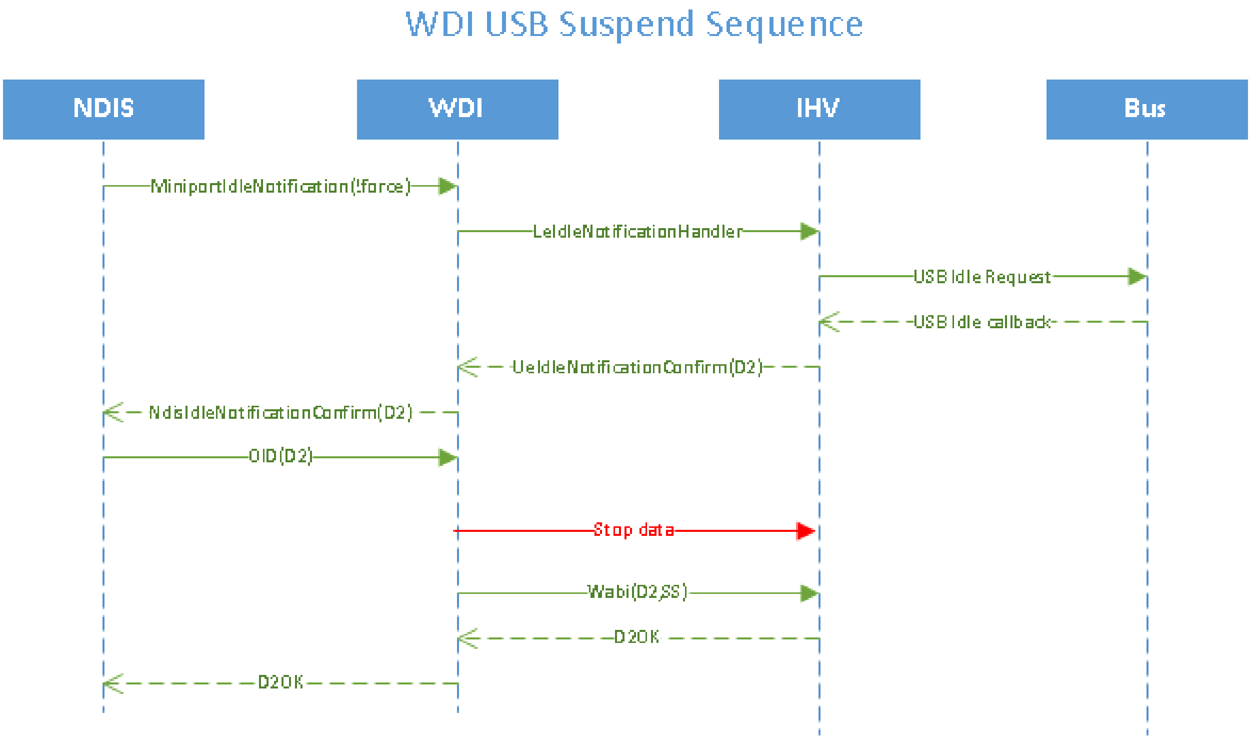
Article • 03/14/2023

When NDIS detects idle for longer than the Selective Suspend idle timeout (*SSIdleTimeout*), NDIS calls the UE. The UE may veto the idle notification by returning NDIS_STATUS_BUSY. If the UE does not veto the idle notification, the UE calls the LE with **LeIdleNotificationHander**, and the LE may veto or accept.

The UE accepts idle notifications when there is no pending data, commands, or timers that may expire to send data or commands down to the LE.

When WDI receives a D2 OID, it processes the OID as if it is a regular D2, except that it sends the WDI OID with the reason code set to **WDI_SET_POWER_DX_REASON_SELETIVE_SUSPEND**.

The following flow diagram shows the suspend sequence.


WDI USB Suspend Sequence
NDIS
WDI
IHV
Bus
MiniportIdleNotification(!force)
LeIdleNotificationHandler
USB Idle Request
USB Idle callback
UeIdleNotificationConfirm(D2)
NdisIdleNotificationConfirm(D2)
OID(D2)
Stop data
Wabi(D2,SS)
D2OK
D2OK

Related topics

MiniportWdiIdleNotification

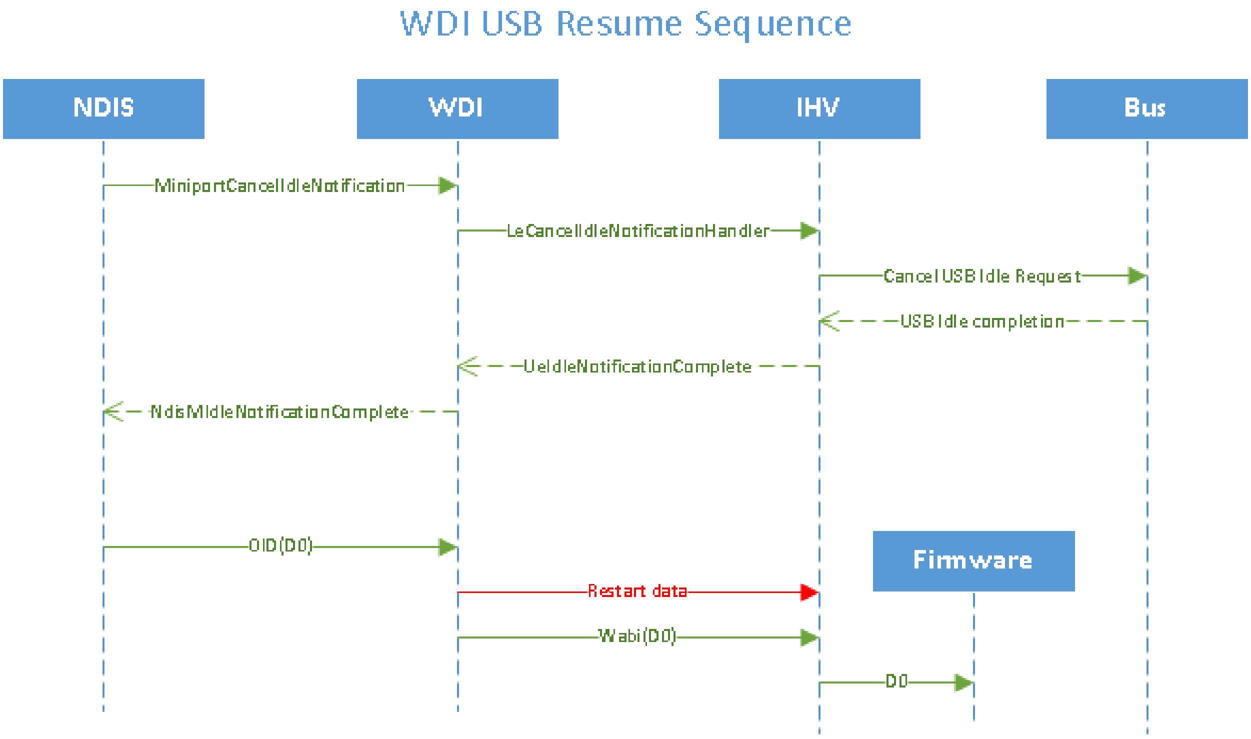
WDI_TLV_SET_POWER_DX_REASON

WDI USB resume sequence

Article • 03/14/2023

When the operating system needs to use the NIC, it cancels the idle state and resumes the NIC to the D0 working state. NDIS initiates the resume.

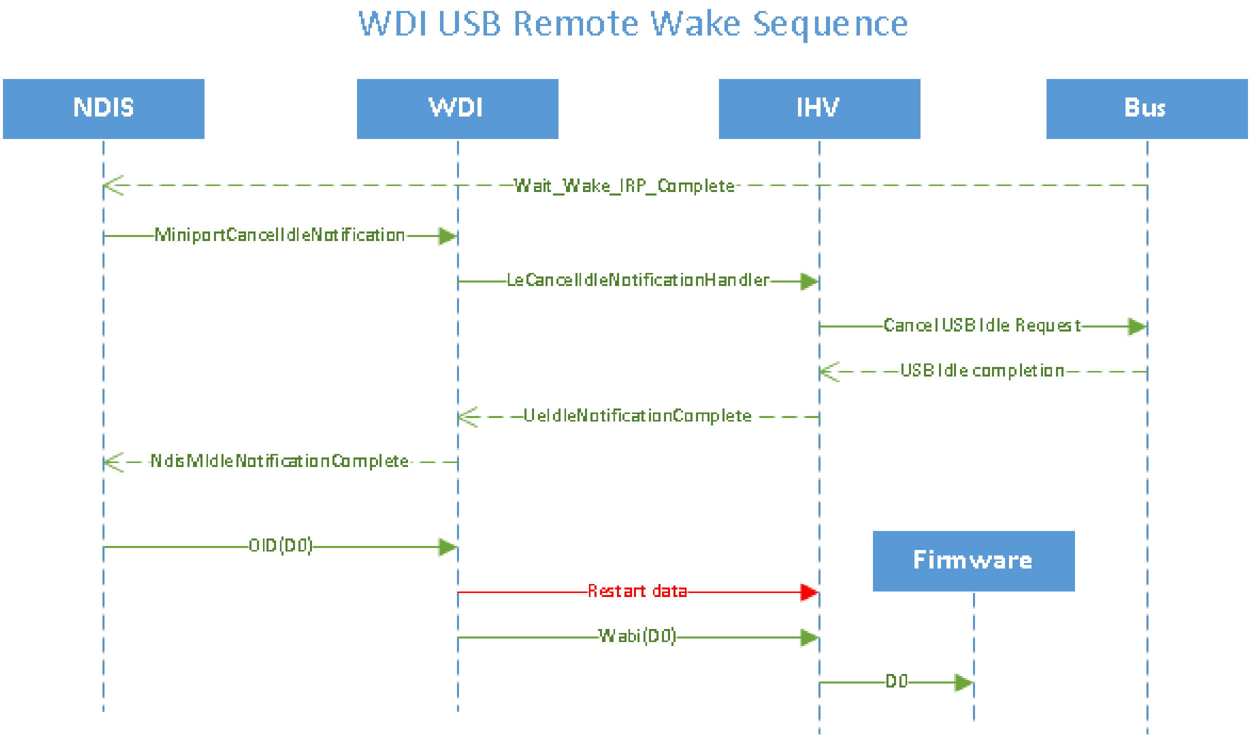
The following flow diagram shows the resume sequence.


WDI USB Resume Sequence
NDIS
WDI
IHV
Bus
MiniportCancelIdleNotification
LeCancelIdleNotificationHandler
Cancel USB Idle Request
USB Idle completion
UeIdleNotificationComplete
NdisMIdleNotificationComplete
OID(D0)
Firmware
Restart data
Wabi(D0)
D0

WDI USB remote wake sequence

Article • 03/14/2023

When the device receives external events, it wakes the stack. The following flow diagram shows the remote wake sequence.


WDI USB Remote Wake Sequence
NDIS
WDI
IHV
Bus
Wait_Wake_IRP_Complete
MiniportCancelIdleNotification
LeCancelIdleNotificationHandler
Cancel USB Idle Request
USB Idle completion
UeIdleNotificationComplete
NdisMIdleNotificationComplete
OID(D0)
Firmware
Restart data
Wabi(D0)
D0

WDI design guide topics

Article • 03/14/2023

This section describes requirements and recommendations for supporting features in the WLAN driver.

WDI low latency connection quality

WDI Extended channel switch announcement

WDI IHV extensible types

Develop and validate WDI drivers for Reset Recovery

WDI low latency connection quality

Article • 03/14/2023

A port can be configured for low latency mode operation if there is an application running on the system that needs low latency data traffic (for example, VoIP applications). When in this mode of operation, the driver should modify any behavior (such as scanning or better AP roaming) that would cause it to move off of the channel of the port that is configured for low latency mode. It should also follow the specified guidance for the NDIS_STATUS_WDI_INDICATION_LINK_STATE_CHANGE indication. The host provides **WDI_TLV_LOW_LATENCY_CONNECTION_QUALITY_PARAMETERS** that the port should use when it is in this mode. This specifies the maximum time that the port should be off channel and the minimum link quality value that the connection must fall down to before initiating a low latency roam (including sending NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED).

For scans, the host provides the maximum channel dwell time (there are different values for active and passive channels) and the adapter should not go above the maximum time. The host also throttles unnecessary scans. However, the adapter can throttle the scan further if the **WDI_SCAN_TRIGGER** is **WDI_SCAN_TRIGGER_BACKGROUND** or **WDI_SCAN_TRIGGER_ROAM**. If the adapter performs its own scans in this mode, it is recommended that it includes the SSID it is looking for (unless it is after a resume from sleep) to reduce the dwell time on a channel. In addition, it should avoid scanning multiple channels in single off-channel scan so that it is under the overall off-channel time limit.

The host considers NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED a strong request from the adapter to roam, so when in this mode, the adapter should be careful about how often this indication is sent up. If the adapter performs its own roaming/AP selection decisions, it must employ appropriate mechanisms (such as neighbor reports or PMKIDs) to find and select/rank APs.

To optimize the association process, the adapter should use the cached BSS entry for TSF timer synchronization during join if possible. The cached entry should be good enough for TSF timer synchronization, which is fresh enough most of time because it was obtained from a recent probe request. TSF synchronization can be done later, even when the driver decides to pick an AP that does not have an up-to-date cached probe response. The driver can disable Wi-Fi power save until it receives the next beacon, which usually occurs within 100ms.

When operating in multi-channel concurrency mode, it is recommended that the adapter employ ECSA or other mechanisms for enabling seamless/no jitter experience when performing channel multiplexing.

WDI Extended channel switch announcement (ECSA)

Article • 03/14/2023

To minimize the cases where the Wi-Fi Direct port causes the system to operate in Multi-Channel mode, multi-channel uses cases are not as performant as single channel use cases. We recommend that the device (driver/firmware) implements ECSA. This feature should exist completely on the IHV side.

Here are the suggested driver/firmware changes.

- Support bi-directional ECSA on the Wi-Fi Direct port.
- When the device is the Group Owner and is in Multi-Channel mode:
  - The driver must detect if the remote peer supports ECSA.
  - If the remote peer supports ECSA, engage ECSA to move the peer into the channel configuration that yields a single channel.
- When the device is the Client and is in Multi-Channel mode:
  - If an ECSA request comes from the remote peer, then support it.
- Send channel change notifications to the operating system with NDIS_STATUS_WDI_INDICATION_P2P_GROUP_OPERATING_CHANNEL.

WDI IHV extensible types

Article • 03/14/2023

The WDI model allows the IHV to support custom PHY type, authentication algorithms, and cipher algorithms. The IHV extension PHY type is used mainly for reporting purposes and does not change the operating system behavior. The IHV extension authentication and cipher algorithms are used with the IHV extensibility module and IHV profiles. When these are used for a connection, the host does not perform any matching for security settings before forwarding the candidate BSS list to the adapter.

Develop and validate WDI drivers for Reset Recovery

Article • 03/14/2023

The UE has a built-in hook for stressing reset and recovery by simulating firmware hangs. It exercises the UE and LE but not the actual firmware, which likely remains functional in the simulation. The code is in M1 UE. It is ideal if the IHV has mechanisms to inject firmware hangs. This exercises Reset Recovery on modules below the LE, specifically Bus, ACPI, and UEFI. There have been hard-to-debug issues in these lower layers regarding to Reset Recovery where the lower layer failed to actually reset the firmware.

OID_WDI_TASK_CHANGE_OPERATION_MODE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_CHANGE_OPERATION_MODE configures the operation mode for the port.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	No	4	1

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_OPERATION_MODE			The desired operation mode.

Task completion indication

NDIS_STATUS_WDI_INDICATION_CHANGE_OPERATION_MODE_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_CLOSE

Article • 03/14/2023

OID_WDI_TASK_CLOSE requests that the IHV component closes the adapter. This includes disabling interrupts and shutting down hardware. During a halt, this task is passed to the IHV through the CloseAdapterHandler handler registered by the IHV.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Adapter	No	1	5

Task parameters

None

Task completion indication

NDIS_STATUS_WDI_INDICATION_CLOSE_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_CONNECT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_CONNECT requests that the IHV component connects to an Access Point or to a Wi-Fi Direct GO.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The abort must be followed by a dot11 reset.	4	10

As part of the connect, the IHV component must synchronize with, authenticate to, and associate to the BSS. The host provides the BSS entries that the IHV component can attempt to connect to. Once the IHV component has successfully connected to one of those entries, it should complete the connect process. If it is unable to connect to any of the BSS entries, it should complete the connect process with a failure.

The IHV component does not need to perform a scan to find candidate BSS entries. It can use the list provided by the host for the connect. It can attempt to connect to each one, one after another. The host sorts the networks by RSSI, but the IHV component can use its own order for connection. If the adapter does not specify "Connect BSS Selection Override", it must only use the entries provided by the host for the connect. The host may issue an abort on an outstanding connect. On receiving the abort, the port must end the connection attempts and report a completion to the host.

If the adapter specifies "Connect BSS Selection Override", it can perform scans on its own to find candidate BSS entries. It can connect to any BSS entry it finds as long as it meets the parameters configured by the host. It should optimize this selection to ensure that it meets any configured connection quality requirements. This could include optimizing roam scan, optimize AP selection, optimize association process, and minimize the security handshake needed. During a scan, if the device needs additional association parameters for a found BSS entry (for example, PMKID for roaming), it can send a NDIS_STATUS_WDI_INDICATION_ASSOCIATION_PARAMETERS_REQUEST indication to get the parameters. When available, the host configures these parameters with OID_WDI_SET_ASSOCIATION_PARAMETERS.

If the connect fails or is aborted, the port should not reset any settings that may have been configured outside of the connect command. It must support the host issuing a second connect call on the same port.

The status of the connect attempt for each BSS entry must be reported by the port at the end of the association attempt. This includes the successful attempt and also any failed attempts. At any time, the port must be associated with no more than one Access Point or Wi-Fi Direct GO.

While a connect is ongoing, the port must maintain any connections established on other ports (for example, Infrastructure or Wi-Fi Direct). However, the port may reduce the amount of medium access provided to the other ports to finish the connection. During the connect, the host can submit packet send requests on other ports.

If the authentication algorithm that is used for the connection requires 802.1x port authorization for network access, the host authorizes the port after the association operation has completed successfully.

The 802.11 station uses the PMKID cache for pre-authentication to access points that have enabled the Robust Security Network Association (RSNA) authentication algorithm. If the 802.11 station is associating or reassociating to a BSSID that has a provided PMKID, the 802.11 station must use the PMKID data in the RSN information element (RSN IE) of its Association or Reassociation frame.

If the port declares support for Host FIPS mode in **WDI_TLV_STATION_ATTRIBUTES**, HostFIPSModeEnabled may be set to 1 in the connection parameters.

If HostFIPSModeEnabled is set to 1, the following rules apply.

- The port must follow the guidelines for sending/receiving data frames in Send operations in FIPS mode and Receive operations in FIPS mode.
- The port must not declare support for any QoS protocol in the association request sent to a non-HT access point. QoS support is required for HT connections.
- The port must not negotiate TSpec and must not perform transmit MSDU aggregation.
- The port must ensure that the SPP A-MSDU capable bit (bit 10) of the RSN capabilities IE it transmits is set to zero. Only PP A-MSDU are supported in this mode.

The connection parameters must not have MFPEnabled and HostFIPSModeEnabled both set to 1. Management Frame Protection (802.11w) requires the port to encrypt/decrypt certain management and action frames, so it cannot be enabled for a connection using Host FIPS mode. In addition, Wake on Wireless LAN features are not applicable in Host-FIPS mode.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CONNECT_PARAMETERS			The connection parameters.
WDI_TLV_CONNECT_BSS_ENTRY	X		The preferred list of candidate connect BSS entries. The port should attempt to connect to any of these BSS entries until the list is exhausted or the connection completed successfully. The port can reprioritize the entries if needed. If the adapter has set the Connect BSS Selection Override bit, then it can pick a BSS that is not in this list as long as it follows the Allowed/Disallowed list.

Task completion indication

NDIS_STATUS_WDI_INDICATION_CONNECT_COMPLETE

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_ASSOCIATION_RESULT

NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_CREATE_PORT

Article • 03/14/2023

OID_WDI_TASK_CREATE_PORT requests that a new 802.11 entity is created by the IHV component.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Adapter	No	6	1

The operation mode of the created port is set to **WDI_OPERATION_MODE_STA** unless it has been specified in the task parameters.

If the MAC is to function as a Wi-Fi Direct device port, **uOpmodeMask** contains **WDI_OPERATION_MODE_P2P_DEVICE**. In this case, the IHV component driver must assign the MAC address reserved for the Wi-Fi Direct Device to this port and return it in the request completion indication.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CREATE_PORT_PARAMETERS			Parameters for port creation.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CREATE_PORT_MAC_ADDRESS		X	This TLV is used when the UE recreates the non-primary port during resume from hibernation. When this TLV is present, the firmware must use this MAC address to create the port. This MAC address is guaranteed to be the MAC address that the firmware created for the port type prior to hibernation. The goal is to use the same NDIS port number and MAC address in order to match the states of the upper layers. Note that the WFC_PORT_ID can be different at recreation, but the port ID should not collide with any port ID of an existing port. This information is only used between the UE and LE/firmware.

Task completion indication

NDIS_STATUS_WDI_INDICATION_CREATE_PORT_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_DELETE_PORT

Article • 03/14/2023

OID_WDI_TASK_DELETE_PORT requests that the IHV component releases all resources (including MAC and PHY) allocated to the specified port.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Adapter	No	6	1

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DELETE_PORT_PARAMETERS			The delete port parameters.

Task completion indication

NDIS_STATUS_WDI_INDICATION_DELETE_PORT_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_DISCONNECT

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_DISCONNECT is used to terminate a connection with a peer.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	No	2	1

This command is used to disconnect from an Access Point or a Wi-Fi Direct GO, and also to disconnect clients of the port. When the disconnect is received, the port must disassociate and deauthenticate from the peer and clear the state associated with that peer. However, it must not reset any of the connection parameters that are not specific to this peer. The task must be completed only after the disconnect activity has been completed.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DISCONNECT_PARAMETERS			The disconnect parameters.

Task completion indication

NDIS_STATUS_WDI_INDICATION_DISCONNECT_COMPLETE

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_DISASSOCIATION When the port gets disconnected from the network, it sends the disassociation indication to the OS. The disconnect may be triggered by a command from the OS, or may be triggered from the network. Network triggered disconnects may be explicit from received disassociation or deauthentication packets, or may be implicit when the port cannot detect the presence of the peer it is connected to.

Before the disassociation indication is sent, the port must clear the state associated with the peer. This includes any keys and 802.1x port authorization information associated with the peer. The port must not trigger a roam on its own.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_DOT11_RESET

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_DOT11_RESET requests that the IHV component resets the MAC and PHY state on a specified port.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	No	1	1

Prior to issuing a dot11 reset command, the WDI driver stops issuing new commands to IHV component and aborts any task in progress on the port. It also flushes its Rx and TX queues.

The dot11 reset combines the semantics of the 802.11 MLME and PLME reset primitive. When the IHV component receives a dot11 reset request, it should perform the following tasks.

- Reset the port's MAC entity to its initial state.
- Reset the port's MIB attributes so they are set to their default values, if bSetDefaultMIB is true.
- Reset the TX/Rx state machines for the PHY entity and set it to Rx state only to ensure no more frames are transmitted.
- Flush the adapter's Rx queue and complete the send for each packet in the TX queues.
- If the MAC address parameter is present, reset the port's MAC address to the specified value.
- Set the port state to INIT before completing the dot11 reset operation.

If the port being reset was operating as a STA, AP, or a Wi-Fi Direct Client or GO, the host would have triggered the disconnect task to request the IHV component to send disassociation to the peers before the reset. As such, the IHV component does not need to do it again.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DOT11_RESET_PARAMETERS			Parameters for the dot11 reset.
WDI_TLV_CONFIGURED_MAC_ADDRESS		X	The MAC address that should be used for the port.

Task completion indication

NDIS_STATUS_WDI_INDICATION_DOT11_RESET_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_IHV

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_IHV is used to start an IHV-initiated task.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	Priority depends on IHV-requested settings.	10

The task is initiated by the sending NDIS_STATUS_WDI_INDICATION_IHV_TASK_REQUEST, and is prioritized based on the value requested by the IHV.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_IHV_TASK_DEVICE_CONTEXT		X	The context data provided by the IHV component. This is forwarded from NDIS_STATUS_WDI_INDICATION_IHV_TASK_REQUEST.

Task completion indication

NDIS_STATUS_WDI_INDICATION_IHV_TASK_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_OPEN

Article • 03/14/2023

OID_WDI_TASK_OPEN requests that the IHV component initializes the adapter.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Adapter	No	1	2

Adapter initialization includes downloading firmware to the adapter, and setting up interrupts and other hardware resources. During initialization, this task is passed to the IHV using the OpenAdapterHandler handler registered by the IHV. On resume from low power state, this is passed to the IHV using OID_WDI_TASK_OPEN.

Task parameters

None

Task completion indication

NDIS_STATUS_WDI_INDICATION_OPEN_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_P2P_DISCOVER

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_P2P_DISCOVER is issued to the device to perform Wi-Fi Direct discovery.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	6	15

The command contains properties which define either a specific set of Wi-Fi Direct devices to search for, or wildcard discovery.

Wi-Fi Direct discovery is mutually exclusive from standard Wi-Fi scanning. While this task is running, broadcast probe requests shall not be sent without a "DIRECT-" SSID, or a specific GO SSID. These probe requests must also include all necessary Wi-Fi Direct IEs.

The host may have search criteria which is not provided as part of the task parameters down to the device. The host may use the task abort mechanism if it has matched the required criteria, therefore it is important that the device can abort Wi-Fi Direct Discovery tasks quickly so as not to degrade scenario performance.

When the task has been completed (either normally or due to an abort), the port should be in a good state such that another discover request can be issued on that port.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_DISCOVER_MODE			Discovery mode information, such as scan type, count, and time between scans.
WDI_TLV_SCAN_DWELL_TIME			Scanning dwell time settings.
WDI_TLV_P2P_DISCOVERY _CHANNEL_SETTINGS	X	X	Scan duration and list of channels to scan. When specified, the listen settings override those specified in WDI_TLV_SCAN_DWELL_TIME. If this list is empty, the port must scan on all supported channels and use the listen settings from WDI_TLV_SCAN_DWELL_TIME.
WDI_TLV_SSID	X	X	A list of SSIDs that the port should scan for. There can be multiple SSIDs in this list and one of them can be a wildcard. When doing an active scan on a channel, the port must send a probe request for each SSID in the list. If this list is empty, the port must scan for all SSIDs.
WDI_TLV_P2P_SERVICE_NAME_HASH	X	X	A list of Service Hash names to be queried. Required if WDI_P2P_SERVICE_DISCOVERY_TYPE_SERVICE_NAME_ONLY or WDI_P2P_SERVICE_DISCOVERY_TYPE_ASP2_SERVICE_NAME_ONLY is specified.
WDI_TLV_VENDOR_SPECIFIC_IE		X	One or more IEs that must be included in the probe requests sent by the port. These IEs are not used for passive scan.
WDI_TLV_P2P_SERVICE_INFORMATION_DISCOVERY_ENTRY	X	X	An optional list of Service Information Discovery Entries to be queried. This is required if WDI_P2P_SERVICE_DISCOVERY_TYPE_SERVICE_INFORMATION is specified. The driver is expected to perform a P2P service discovery over probe request/response using the service name hash. For each service entry that contains service information, the driver is expected to perform an ANQP query request/response to query the service information.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_ASP2_SERVICE_INFORMATION_DISCOVERY_ENTRY	X	X	Added in Windows 10, version 1607, WDI version 1.0.21. An optional list of ASP2 Service Information Discovery Entries to be queried. This is required if WDI_P2P_SERVICE_DISCOVERY_TYPE_ASP2_SERVICE_INFORMATION is specified. The driver is expected to perform a P2P service discovery over probe request/response using the service name hash. For each service entry that contains service information, the driver is expected to perform an ANQP query request/response to query the service information.
WDI_TLV_P2P_INCLUDE_LISTEN_CHANNEL		X	Added in Windows 10, version 1607, WDI version 1.0.21. Specifies whether the probe request should include the Listen Channel attribute during discovery.

Task completion indication

NDIS_STATUS_WDI_INDICATION_P2P_DISCOVERY_COMPLETE

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME is issued to the device to send a Wi-Fi Direct Public Action Frame Request.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	3	5

This command is different than OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME, which is a significantly more time-sensitive operation.

When the device receives an acknowledgment for a request frame, it shall dwell on the same channel for 100ms and indicate any Wi-Fi Direct Public Action Frames it receives to the host.

While the maximum timeout has not expired, the device shall retry sending the Wi-Fi Direct Public Action frame to the remote device on the remote device’s listen channel.

The task is complete either when the local device receives an acknowledgment from the remote device for the action frame that was sent, the timeout expires, or the host aborts the operation. The device may indicate task completion after the same-channel dwell time has expired.

The host may decide to abort this operation and continue/retry the Wi-Fi Direct action frame exchange, so it is important that the device is able to abort this operation quickly.

Validation

For miniport drivers that support WDI version 1.1.8 and later, additional validation of the P2P IEs on outgoing P2P Action Frames has been added. This validation addresses a common problem in which the **Configuration Timeout** attribute of the P2P IE has not been converted form units of milliseconds, as provided to the LE in OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME and OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME, to units of tens of milliseconds, which is the IE format.

The Wi-Fi Direct and Wi-Fi Direct Services HLK tests will fail for drivers supporting WDI version 1.1.8 and later if the **Configuration Timeout** attribute of the P2P IE is not encoded correctly on an outgoing action frame. For WDI versions 1.1.7 and earlier, the tests will print a warning to the test output.

The WDI interface itself is unchanged and continues to use units of milliseconds just as it did in versions 1.1.7 and earlier.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SEND_ACTION_REQUEST_FRAME_PARAMETERS			Parameters such as action frame type, device address of target peer adapter, and dialog token.
WDI_TLV_P2P_GO_NEGOTIATION_REQUEST_INFO		X	GO Negotiation Request Parameters. THe port shall only examine this structure if wfdRequestFrameType is a GO Negotiation request.
WDI_TLV_P2P_INVITATION_REQUEST_INFO		X	Invitation Request Parameters. The port shall only examine this structure if wfdRequestFrameType is an Invitation request.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_PROVISION_DISCOVERY_REQUEST_INFO		X	Provision Discovery Request Parameters. The port shall only examine this structure if wfdRequestFrameType is an Provision Discovery request.
WDI_TLV_BSS_ENTRY			The device discovery entry as returned by the Wi-Fi Direct Discovery task from the port. This is provided so the port does not need to remember its discovery database in order to send Wi-Fi Direct Action Frame Requests to remote Wi-Fi Direct devices without requiring a discovery.
WDI_TLV_VENDOR_SPECIFIC_IE		X	One or more IEs that must be included in the frame sent by the port.

Task completion indication

NDIS_STATUS_WDI_INDICATION_P2P_SEND_REQUEST_ACTION_FRAME_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME is issued to the IHV component to send a Wi-Fi Direct Public Action Frame Request to a peer.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	3	5

When port receives an acknowledgment for a request frame, it shall dwell on the same channel for 100ms and indicate any Wi-Fi Direct Public Action Frames it receives to the host.

This task is time sensitive. The Wi-Fi Direct specification requires that sending Wi-Fi Direct action responses are only serviced within 100 milliseconds of receiving this packet.

While the maximum timeout has not expired, the port shall retry sending the Wi-Fi Direct to the remote device on the appropriate channel as defined by the following table. The table defines the explicit channel requirements for where to send the packets when the command is issued. The general rule is that the response packet shall be sent out on the same channel as the prior request.

Response Action Frame Type	Target Transmit Channel
GO Negotiation Response	Local Listen Channel
GO Negotiation Confirmation	Remote Listen Channel
Invitation Response	Local Listen or Local GO Operational Channel
Provision Discovery Response	Local Listen Channel or Remote GO Operational Channel

The task is complete either when local device receives an acknowledgment from the remote device for the action frame that was sent, the timeout expires, or the host aborts the operation. The device may indicate task completion after the same-channel dwell time has expired.

The host may decide to abort this operation and continue/retry the Wi-Fi Direct action frame exchange, so it is important that the device is able to abort this operation quickly.

Validation

For miniport drivers that support WDI version 1.1.8 and later, additional validation of the P2P IEs on outgoing P2P Action Frames has been added. This validation addresses a common problem in which the **Configuration Timeout** attribute of the P2P IE has not been converted form units of milliseconds, as provided to the LE in OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME and OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME, to units of tens of milliseconds, which is the IE format.

The Wi-Fi Direct and Wi-Fi Direct Services HLK tests will fail for drivers supporting WDI version 1.1.8 and later if the **Configuration Timeout** attribute of the P2P IE is not encoded correctly on an outgoing action frame. For WDI versions 1.1.7 and earlier, the tests will print a warning to the test output.

The WDI interface itself is unchanged and continues to use units of milliseconds just as it did in versions 1.1.7 and earlier.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_ACTION_FRAME_RESPONSE_PARAMETERS			Parameters such as action frame type, device address of target peer adapter, and dialog token.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_GO_NEGOTIATION_RESPONSE_INFO		X	GO Negotiation Response Parameters. The port shall only examine this structure if wfdRequestFrameType is a GO Negotiation Response.
WDI_TLV_P2P_GO_NEGOTIATION_CONFIRMATION_INFO		X	GO Negotiation Confirmation Parameters. The port shall only examine this structure if wfdRequestFrameType is a GO Negotiation Confirmation.
WDI_TLV_P2P_INVITATION_RESPONSE_INFO		X	Invitation Response Parameters. The port shall only examine this structure if wfdRequestFrameType is an Invitation Response.
WDI_TLV_P2P_PROVISION_DISCOVERY_RESPONSE_INFO		X	Provision Discovery Response Parameters. The port shall only examine this structure if wfdRequestFrameType is an Provision Discovery Response.
WDI_TLV_P2P_INCOMING_FRAME_INFORMATION			Information that was indicated from the previously received P2P Action Frame. The received indication is provided back to the port.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_VENDOR_SPECIFIC_IE		X	One or more IEs that must be included in the frame sent by the port.

Task completion indication

NDIS_STATUS_WDI_INDICATION_P2P_SEND_RESPONSE_ACTION_FRAME_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_REQUEST_FTM

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_REQUEST_FTM is issued to the LE to initiate Fine Timing Measurement (FTM) procedures with the listed BSS targets. The number of targets is less than or equal to the value of **FTMNumberOfSupportedTargets**, obtained from the station attributes.

This task should be completed as soon as all the FTM sessions with the targets are completed, the timeout has expired, or the host has aborted the operation.

When this task is completed, the driver should send an NDIS_STATUS_WDI_INDICATION_REQUEST_FTM_COMPLETE status indication that contains a list of FTM responses for each of the targets requested.

After this task is completed, the port should be in a good state and should be ready to process a new FTM request, because the host might immediately re-attempt the task with a new set of targets.

If the LE maintains a station BSS list cache, it can use this cache to obtain the parameters needed to issue FTM request to the targets and ignore the provided data. However, if the target BSSID is not found in the cache the LE needs to use the provided data to attempt the FTMs.

For each target, it is indicated if a Location Configuration Information (LCI) report should be requested. If indicated, the LE should request one from the target.

Task parameters

TLV	Type	Multiple TLV instances allowed	Optional	Description

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_FTM_REQUEST_TIMEOUT	UINT32			The maximum time, in milliseconds, to complete the FTM. The timeout is set to 150 ms multiplied by the number of targets.
WDI_TLV_FTM_TARGET_BSS_ENTRY	WDI_FTM_TARGET_BSS_ENTRY	X		A list of the BSS targets with which FTM procedures should be completed.

Task completion indication

NDIS_STATUS_WDI_INDICATION_REQUEST_FTM_COMPLETE

Requirements

Minimum supported client	Windows 10, version 1903
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_ROAM

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_ROAM requests that the adapter tries to roam from the currently connected AP to a new one.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. If aborted after disassociation, it must be followed by a dot11 reset.	4	10

The Microsoft component provides the list of preferred BSS entries that the adapter should consider for roaming.

When this command issued, if its currently associated, the adapter would need to disassociate from the currently connected AP and then roam to the new AP. In this case it would first indicate disassociation for the old AP, then indicate association result for the new AP and then complete the task.

It can also determine not to perform the roam and stay connected to the current AP. In this case it would only complete the task without any association or disassociation indications.

The scan and AP selection requirements for this task are same as for OID_WDI_TASK_CONNECT.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CONNECT_PARAMETERS			Connection parameters.
WDI_TLV_CONNECT_BSS_ENTRY	X		The preferred list of candidate connect BSS entries. The port should attempt to connect to these BSS entries until the list is exhausted, or the connection completed successfully. The port can reprioritize the entries if needed. If the adapter has set the Connect BSS Selection Override bit, then it can pick a BSS that is not in this list as long as it follows the Allowed/Disallowed list requirements.

Task completion indication

NDIS_STATUS_WDI_INDICATION_ROAM_COMPLETE

Unsolicited indications

NDIS_STATUS_WDI_INDICATION_ASSOCIATION_RESULT

NDIS_STATUS_WDI_INDICATION_DISASSOCIATION

NDIS_STATUS_WDI_INDICATION_FT_ASSOC_PARAMS_NEEDED

NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_SCAN

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_SCAN requests a survey of BSS networks. The port performs a scan according to the requirements of the IEEE 802.11 specification.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	6 (background scan) 5 (user-initiated scan)	4

A task started message containing a WDI_TLV_STATUS is indicated once the port has started the scan and is ready to receive other commands.

Once a scan is started when enabled by LiveUpdatesNeeded, the port must provide incremental updates (using unsolicited indications of NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST) about discovered BSS entries. BSS entries that had previously been discovered but were not found by the port in the current scan should not be reported by the port. For power and performance reasons, the port should throttle indications and send updates to the host only when it has discovered 3 or more, or when it has discovered less than 3 entries but has not reported them to the host for more than 500 milliseconds. After the scan is completed, if the adapter does not manage BSS entries, it does not need to remember the BSS entries that it has discovered. Once the scan operation has finished, the port must send the task complete notification to the operating system and stop reporting BSS entries to the host. The scan command is used for finding legacy (non-Wi-Fi Direct networks) and the port should not include the Wi-Fi Direct IEs in the probe requests.

If the adapter does not manage BSS entries, the host remembers the BSS entries reported by the port from a scan for a finite period of time. It times out its cached entries and flushes them. If the adapter manages the BSS entries, it caches and times them out. The host may send the OID_WDI_SET_FLUSH_BSS_ENTRY command to explicitly flush the entries.

The host tracks BSS entries using their BSSID. If the port reports two BSS entries for the same BSSID, the host overwrites one with another.

While the scan is ongoing, the port must maintain the existing connections (for example, Infrastructure or Wi-Fi Direct). If connections already exist, the port should scan a subset of channels at a time and in between subsets, provide the other connections access to the medium. During the scan, the host can submit packet send requests to any port on the adapter.

In the indicated BSS entries, the port can include device specific context information. This context information is passed back to the device if the port is asked to connect to that BSS entry. However, this context may be cleared by the host automatically if the BSS entry is flushed.

The scan command can be aborted. On receiving the abort command, the port should stop trying to find new BSS networks and complete the scan task as soon as possible. When the task has been completed (either normally or due to an abort), the port should be in a good state such that another scan can be issued on that port.

The adapter must not violate regulatory restrictions when performing a scan.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID			BSSID of the network to scan for. If this is the broadcast MAC address, the station scans for all BSSIDs.
WDI_TLV_SSID	X		A list of SSID list that the port should scan for. There can be multiple SSIDs in this list and one of them can be a wildcard. When doing an active scan on a channel, the port must send a probe request for each SSID in the list. If this list is empty, the port must scan for all SSIDs.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_VENDOR_SPECIFIC_IE		X	One or more IEs that must be included in the probe requests sent by the port. These IEs are not used for passive scan.
WDI_TLV_SCAN_MODE			Scan mode parameters.
WDI_TLV_SCAN_DWELL_TIME			Dwell time parameters.
WDI_TLV_BAND_CHANNEL	X	X	A list of recommended channels to scan. The adapter can perform a scan on a subset or superset of the channel list as long as it meets the Maximum Scan Time requirements. If this list is empty, the port must scan on all supported channels.

Task completion indication

NDIS_STATUS_WDI_INDICATION_SCAN_COMPLETE

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST

This notification is used by the device to tell the host about updates to the BSS entries. It can be sent at any time.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_SEND_AP_ASSOCIATION_RESPONSE

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_SEND_AP_ASSOCIATION_RESPONSE requests that the IHV component sends an Association Response to a peer device that has recently sent an association request.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	3	1

If the send fails for any reason, an empty NDIS_STATUS_WDI_INDICATION_SEND_AP_ASSOCIATION_RESPONSE_COMPLETE is expected, with the correct status included in the populated header.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ASSOCIATION_RESPONSE_PARAMETERS			Association response parameters.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_VENDOR_SPECIFIC_IE		X	Additional IEs that the port must append to Association Response IE set before sending response to peer adapter.
WDI_TLV_INCOMING_ASSOCIATION_REQUEST_INFO			Information about the incoming association request.
WDI_TLV_WFD_ASSOCIATION_STATUS		X	The Status value to set when the association request is denied.

Task completion indication

NDIS_STATUS_WDI_INDICATION_SEND_AP_ASSOCIATION_RESPONSE_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_SEND_REQUEST_ACTION_FRAME

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_SEND_REQUEST_ACTION_FRAME requests that the device sends an Action Frame Request to another device.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	3	5

This command is different from OID_WDI_TASK_SEND_RESPONSE_ACTION_FRAME, which is a significantly more time-sensitive operation.

When the device receives an acknowledgment for a request frame, it shall dwell on the same channel for the Post-ACK Dwell time as specified in the Task Parameters, and shall indicate to the host any Action Frames it receives and doesn't handle itself.

As long as the maximum timeout has not expired, the device shall retry sending the Public Action frame to the remote device on the remote device's listen channel.

The task is complete either when local device receives an acknowledgment from the remote device for the action frame that was sent, the timeout expires, or the host aborts the operation. The device may indicate task completion after the same-channel dwell time has expired.

The host may decide to abort this operation and continue/retry the public action frame exchange, so it is important that the device is able to abort this operation quickly.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SEND_ACTION_FRAME_REQUEST_PARAMETERS			Parameters for sending an Action Frame Request.
WDI_TLV_ACTION_FRAME_BODY			The Action Frame body.

Task completion indication

NDIS_STATUS_WDI_INDICATION_SEND_REQUEST_ACTION_FRAME_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_SEND_RESPONSE_ACTION_FRAME

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_SEND_RESPONSE_ACTION_FRAME requests that the IHV component sends Response Action Frames.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The port must be in a clean state after the abort.	3	5

This task is time sensitive and must be serviced within 100 milliseconds of receiving this packet.

While the maximum timeout has not expired, the port shall retry sending the frame to the remote device on the specified channel.

The task is complete either when local device receives an acknowledgment from the remote device for the action frame that was sent, the timeout expires, or the host aborts the operation. The device may indicate task completion after the same-channel dwell time has expired.

The host may decide to abort this operation and continue/retry the action frame exchange, so it is important that the device is able to abort this operation quickly.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SEND_ACTION_FRAME_RESPONSE_PARAMETERS			Parameters for sending an Action Frame Response.
WDI_TLV_ACTION_FRAME_BODY			The Action Frame body.

Task completion indication

NDIS_STATUS_WDI_INDICATION_SEND_RESPONSE_ACTION_FRAME_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_SET_RADIO_STATE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_SET_RADIO_STATE is used to set the Wi-Fi radio state for the adapter.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Adapter	No	1	1

The task must be completed only after the disconnect activity has been completed.

The IHV component may also send unsolicited indications about radio state changes to the host.

Before the host turns off the radio, it disconnects all peers and stops any Group Owner that is running. The adapter is not expected to remember the station/GO profile information across a radio OFF/ON transition.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_RADIO_STATE_PARAMETERS			The desired state of the radio. If this set to 1, the radio is enabled. If this is set to 0, the radio is turned off.

Task completion indication

NDIS_STATUS_WDI_INDICATION_SET_RADIO_STATE_COMPLETE

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_RADIO_STATUS This indication is used to report changes in the radio state for the adapter. This is sent both when a software radio change is triggered by the host and when a hardware radio state change is detected by the adapter.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_START_AP

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_START_AP requests that the IHV component configures a port to start a Wi-Fi Direct Group Owner on the specified port.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	Yes. The abort must be followed by a dot11 reset.	4	1

During initialization, the driver sets the GO on 5GHz band capability in WDI_TLV_P2P_CAPABILITIES to indicate whether it can start the access point on the 5 GHz band.

If GO on 5 GHz band support is set, the adapter should start the AP on the Advertised Operating channel, and if that fails, it should try the list specified in the AP band channel list parameter. The operating system will provide a hint to the driver about whether this AP would provide internet connectivity by setting the **DOT11_WFD_GROUP_CAPABILITY_CROSS_CONNECTION_SUPPORTED** flag in WDI_TLV_P2P_GROUP_OWNER_CAPABILITY.

If **MustUseSpecifiedChannel** in WDI_TLV_START_AP_PARAMETERS is specified, the AP may return one of the following errors if it is unable to start the AP on the specified band/channel(s).

NDIS_STATUS_DOT11_AP_CHANNEL_CURRENTLY_NOT_AVAILABLE: Unable to start the AP on the specified channel(s) right now . Retry on the specified channel(s) later.

NDIS_STATUS_DOT11_AP_BAND_CURRENTLY_NOT_AVAILABLE: Unable to start the AP on the specified band(s) right now. Retry on the specified band(s) later.

NDIS_STATUS_DOT11_AP_CHANNEL_NOT_ALLOWED: Unable to start the AP on the specified channel(s) due to regulatory reasons.

NDIS_STATUS_DOT11_AP_BAND_NOT_ALLOWED: Unable to start the AP on the specified band(s) due to regulatory reasons.

Task parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SSID			The SSID to be used by the AP.
WDI_TLV_START_AP_PARAMETERS			Additional parameters for this task.
WDI_TLV_AUTH_ALGO_LIST			The list of authentication algorithms that the connection can use.
WDI_TLV_MULTICAST_CIPHER_ALGO_LIST			The list of multicast cipher algorithms that the connection can use.
WDI_TLV_UNICAST_CIPHER_ALGO_LIST			The list of multicast cipher algorithms that the connection can use.
WDI_TLV_P2P_CHANNEL_NUMBER		X	If specified, this defines the operating channel determined in group formation. This may only be specified when the operating mode is Wi-Fi Direct GO.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_AP_BAND_CHANNEL	X	X	Added in Windows 10, version 1511, WDI version 1.0.10. Optional list of bands and channels to start the access point on. If MustUseSpecifiedChannels is set to 1, the AP can only be started from this list. If it is not set, this list is meant to be a recommendation of channels that the firmware can pick from, and it may pick another channel if it is not possible to start the AP on any of the specified channels.

Task completion indication

NDIS_STATUS_WDI_INDICATION_START_AP_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_TASK_STOP_AP

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_TASK_STOP_AP requests that the IHV component disconnects all connected clients on the specified port and stops beaconing and responding to probe requests. AP configuration and MIB attributes are preserved.

Object	Abort capable	Default priority (host driver policy)	Normal execution time (seconds)
Port	No	2	1

Task parameters

None

Task completion indication

NDIS_STATUS_WDI_INDICATION_STOP_AP_COMPLETE

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_ABORT_TASK

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_ABORT_TASK is a property that is sent down to cancel a specific pending task.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

This command follows property semantics. It should be treated as a signal, should be handled as quickly as possible, and should be completed independently of task completion. The IHV component must then attempt to complete the pending task as soon as possible.

Command parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CANCEL_PARAMETERS			Information for the command that is being cancelled.

Command result

Contains a status of NDIS_STATUS_SUCCESS. There is no additional payload.

Examples

Original input task command:

Field	Subfield	Type	Value
NDIS_OID_REQUEST	Oid	NDIS_OID	OID(WDI_TASK_SCAN)
---	InputBufferLength	UINT32	0x210 (example)
---	InformationBuffer	PVOID	Pointer to memory block containing WDI_MESSAGE_HEADER + TLV payload

Field	Subfield	Type	Value
WDI_MESSAGE_HEADER	PortId	UINT16	0x0001 (example)
---	Reserved	UINT16	N/A
---	WiFiStatus	NDIS_STATUS	N/A
---	TransactionId	UINT32	0x1111 (example)
---	IhvSpecificId	UINT32	N/A
TLV Payload	TLV Payload	Various	Payload data

Abort task input command (with message header):

Field	Subfield	Type	Value
NDIS_OID_REQUEST	Oid	NDIS_OID	OID(WDI_ABORT_TASK)
---	InputBufferLength	UINT32	sizeof(WDI_MESSAGE_HEADER) + sizeof(WDI_TLV_CANCEL_PARAMETERS)
---	InformationBuffer	PVOID	Pointer to memory block containing WDI_MESSAGE_HEADER + TLV payload
WDI_MESSAGE_HEADER	PortId	UINT16	0x0001 (example)
---	Reserved	UINT16	N/A
---	WiFiStatus	NDIS_STATUS	N/A
---	TransactionId	UINT32	0x2222 (example)
---	IhvSpecificId	UINT32	0
WDI_TLV_CANCEL_PARAMETERS	OriginalTaskOid	NDIS_OID	OID(WDI_TASK_SCAN)
---	OriginalPortId	UINT16	0x0001 (example)
---	OriginalTransactionId	UINT32	0x1111 (example)

Abort task command result:

Field	Subfield	Type	Value
NDIS_OID_REQUEST	Oid	NDIS_OID	OID(WDI_TASK_SCAN)
---	OutputBufferLength	UINT32	sizeof(WDI_MESSAGE_HEADER)
---	InformationBuffer	PVOID	Pointer to memory block containing WDI_MESSAGE_HEADER
WDI_MESSAGE_HEADER	PortId	UINT16	0x0001 (example)

Field	Subfield	Type	Value
---	Reserved	UINT16	N/A
---	WiFiStatus	NDIS_STATUS	NDIS_STATUS_SUCCESS
---	TransactionId	UINT32	0x2222 (example)
---	IhvSpecificId	UINT32	N/A

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_GET_ADAPTER_CAPABILITIES

Article • 03/14/2023

OID_WDI_GET_ADAPTER_CAPABILITIES is a read-only property that is issued from the host to the adapter during initialization and requests the adapter's capabilities.

Scope	Set serialized with task	Normal execution time (seconds)
Adapter	Set not supported	1

Get property parameters

No additional parameters. The data in the header is sufficient.

Get property results

If the adapter supports Wi-Fi Direct, both WDI_TLV_AP_ATTRIBUTES and WDI_TLV_P2P_ATTRIBUTES must be specified.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_COMMUNICATION_CONFIGURATION_ATTRIBUTES		X	Host-adapter communication protocol configuration attributes.
WDI_TLV_INTERFACE_ATTRIBUTES			Interface attributes.
WDI_TLV_STATION_ATTRIBUTES			Station attributes.
WDI_TLV_AP_ATTRIBUTES		X	Access point attributes.
WDI_TLV_VIRTUALIZATION_ATTRIBUTES		X	Virtualization attributes.
WDI_TLV_P2P_ATTRIBUTES		X	The Wi-Fi Direct attributes.
WDI_TLV_DATAPATH_ATTRIBUTES		X	Datapath attributes.
WDI_TLV_BAND_INFO	X	X	Band information.
WDI_TLV_PHY_INFO	X	X	PHY information.
WDI_TLV_PM_CAPABILITIES		X	Power management capabilities.
WDI_TLV_COUNTRY_REGION_LIST		X	Country or region codes.
WDI_TLV_RECEIVE_COALESCING_CAPABILITIES		X	Hardware assisted receive filter capabilities.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_TCP_OFFLOAD_CAPABILITIES		X	TCP/IP offload capabilities.
WDI_TLV_SUPPORTED_GUIDS	X	X	Added in Windows 10, version 1607, WDI version 1.0.21. A list of GUIDs that are passed on to NDIS when WDI is queried for OID_GEN_SUPPORTED_GUIDS.
WDI_TLV_OS_POWER_MANAGEMENT_FEATURES			Added in Windows 10, version 1803, WDI version 1.1.6. Used to enable advanced OS power management features.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_GET_AUTO_POWER_SAVE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_GET_AUTO_POWER_SAVE gets the power save state of the port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Not applicable	1

There is a trade-off between power saving and latency. When auto power save mode is set to be enabled with the OID_WDI_SET_CONNECTION_QUALITY command, the firmware tries to interact with the connected access point to go to power save mode as much as appropriate to save power. The firmware is also responsible for detecting if the connected access point confirms to the 802.11 specification and follows the power save mode protocol. If the access point does not conform (does not support power save mode correctly), the firmware should not go into power save mode, even when Auto Power Save is set to enabled. When Auto Power Save is set to disabled, the firmware focuses on low latency of sending and receiving packets. Examples of this are when streaming mode is on, and when the system is using AC power so low latency is preferred to saving power.

Get property parameters

No additional parameters. The data in the header is sufficient.

Get property results

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_GET_AUTO_POWER_SAVE			Auto power save information.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_GET_BSS_ENTRY_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_GET_BSS_ENTRY_LIST is used to ask the adapter to indicate the list of BSS networks that have been cached by the port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Set not supported	1

This is only used for an adapter that perform BSS list caching. When acting as a client, the port must report the BSS entry for the access point. In addition, if the port is performing background scans, it should report BSS entries that it has discovered in its scan.

When this request is received by the adapter, it must send NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST indications to the Microsoft component. These indications must contain the BSS entries that match the Get parameters. The adapter can send one or more NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST indications, but they must be completed before the property completes.

The Microsoft component uses the list of indicated entries to report the BSS list to the operation system. It is also used to populate parameters for roam and connect tasks.

Get property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SSID			The SSID that the host needs the BSS list update for.

Get property results

No additional data. The data in the header is sufficient.

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_GET_NEXT_ACTION_FRAME_DIALOG_TOKEN

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_GET_NEXT_ACTION_FRAME_DIALOG_TOKEN requests the DialogToken to be used in the next Action frame.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

Get property parameters

No additional parameters. The data in the header is sufficient.

Get property results

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_NEXT_DIALOG_TOKEN			A dialog token.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_GET_PM_PROTOCOL_OFFLOAD

Article • 03/14/2023

OID_WDI_GET_PM_PROTOCOL_OFFLOAD requests a list of protocol offloads for power management.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Not applicable	1

Get property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PM_PROTOCOL_OFFLOAD_GET			Protocol offload ID.

Get property results

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv4ARP		X	IPv4 ARP protocol offload parameters.
WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv6NS		X	IPv6 NS protocol offload parameters.
WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY		X	RSN Rekey protocol offload parameters.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_ADD_PM_PROTOCOL_OFFLOAD

OID_WDI_SET_REMOVE_PM_PROTOCOL_OFFLOAD

OID_WDI_GET_RECEIVE_COALESCING_MATCH_COUNT

Article • 03/14/2023

OID_WDI_GET_RECEIVE_COALESCING_MATCH_COUNT requests the number of packets that have matched receive filters on the network port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Get property parameters

No additional parameters. The data in the header is sufficient.

Get property results

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_COALESCING_FILTER_MATCH_COUNT			The number of packets that have matched receive filters on the network port.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_GET_STATISTICS

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_GET_STATISTICS requests that the IHV component returns MAC and PHY layer statistics.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Set not supported	1

The MAC statistics must all be maintained per port. PHY statistics must also be maintained per port unless exempted. If PHY statistics cannot be maintained per port (as allowed by exemption), the statistics can be maintained per "channel" (PHY statistics for two ports operating on the same channel can be combined).

Get property parameters

No additional parameters. The data in the header is sufficient.

Get property results

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_MAC_STATISTICS	X		Per-peer MAC statistics.
WDI_TLV_PHY_STATISTICS	X		Per-port PHY statistics.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_GET_SUPPORTED_DEVICE_SERVICES

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_GET_SUPPORTED_DEVICE_SERVICES queries the IHV driver for all of its supported device services (with each device service identified by a GUID). If no device services are supported, LE should fail the command with **STATUS_NOT_SUPPORTED**.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

WLAN will provide a pipeline for any user mode component (UMDF driver or UM service) to communicate with the IHV WLAN driver. Among other things, this pipeline can be used for engaging SAR scenarios by being the platform to send information between the OEM sensors and the IHV firmware. The definition/format of the data to be sent to the IHV/LE driver and what the commands must do is not specified by UE. All contracts with the IHV must be defined by the OEM, and the device service WDI commands (and the corresponding user mode WLAN APIs) will just serve as a generic pipeline.

Get property parameters

No additional parameters. The data in the header is sufficient.

Get property results

TLV	Multiple TLV instances allowed	Optional	Description

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DEVICE_SERVICE_GUID_LIST			The list of device services that the underlying IHV driver exposes to UM components.

Requirements

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
Header	Dot11wdi.h

OID_WDI_IHV_REQUEST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_IHV_REQUEST is used to forward information that an IHV extensibility module has sent to the miniport.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

This command is not serialized with any tasks. It is serialized with other properties and with the M1-M3 of a task.

Command parameter

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_IHV_DATA		X	The information from the IHV extensibility module.

Response result

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_IHV_DATA		X	The response to be sent to the IHV extensibility module. The data value is forwarded as-is to the IHV extensibility module.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_ADAPTER_CONFIGURATION

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_ADAPTER_CONFIGURATION configures the adapter. It is an optional property and can only be sent before any ports are created.

Scope	Set serialized with task	Normal execution time (seconds)
Adapter	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CONFIGURED_MAC_ADDRESS		X	MAC address.
WDI_TLV_UNREACHABLE_DETECTION_THRESHOLD		X	Unreachable detection threshold.
WDI_TLV_P2P_GO_INTERNAL_RESET_POLICY		X	Policy used by the firmware for operating channel selection after a Wi-Fi Direct GO Reset is stopped/restarted.
WDI_TLV_BAND_ID_LIST		X	List of band IDs.
WDI_TLV_LINK_QUALITY_BAR_MAP			Mapping of signal quality to Wi-Fi signal strength bars. This field should be ignored by the adapter and it should use the behavior specified in NDIS_STATUS_WDI_INDICATION_LINK_STATE_CHANGE for doing Link Quality notifications.
WDI_TLV_ADAPTER_NLO_SCAN_MODE		X	Indicates whether the NLO scans should be performed in active or passive mode.
WDI_TLV_PLDR_SUPPORT			Added in Windows 10, version 1511, WDI version 1.0.10. Specifies if PLDR is supported.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_ADD_CIPHER_KEYS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_ADD_CIPHER_KEYS adds or overwrites cipher keys in the key table of a port. This is a set-only property.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Cipher keys that are marked as Static should not be cleared on a roam. They can only be cleared on a OID_WDI_TASK_DOT11_RESET or if they are overwritten with a new OID_WDI_SET_ADD_CIPHER_KEYS.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SET_CIPHER_KEY_INFO	X		The cipher keys to be added or overwritten in the key table of the port.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Dot11wdi.h

See also

OID_WDI_SET_DELETE_CIPHER_KEYS

OID_WDI_SET_ADD_PM_PROTOCOL_OFFLOAD

Article • 03/14/2023

OID_WDI_SET_ADD_PM_PROTOCOL_OFFLOAD adds a list of one or more protocol offloads for power management.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

This property provides information to enable the device/firmware to implement these protocols while the main CPU is asleep. In this state, the firmware and device handles the offloaded tasks without waking up the host.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv4ARP		X	IPv4 ARP protocol offload parameters.
WDI_TLV_PM_PROTOCOL_OFFLOAD_IPv6NS		X	IPv6 NS protocol offload parameters.
WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY		X	RSN Rekey protocol offload parameters.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_GET_PM_PROTOCOL_OFFLOAD

OID_WDI_SET_REMOVE_PM_PROTOCOL_OFFLOAD

OID_WDI_SET_ADD_WOL_PATTERN

Article • 03/14/2023

OID_WDI_SET_ADD_WOL_PATTERN adds a wake-on-LAN (WOL) pattern to the firmware.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

The host defines the packet pattern types to add to the firmware. The firmware detects matching packets that arrive in Dx. If detected, the firmware asserts the wake interrupt.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_WAKE_PACKET_BITMAP_PATTERN	X	X	WOL pattern information.
WDI_TLV_WAKE_PACKET_MAGIC_PACKET		X	Pattern ID of the magic packet.
WDI_TLV_WAKE_PACKET_IPv4_TCP_SYNC	X	X	WOL IPv4 TCP sync packet information.
WDI_TLV_WAKE_PACKET_IPv6_TCP_SYNC	X	X	WOL IPv4 TCP sync packet information.
WDI_TLV_WAKE_PACKET_EAPOL_REQUEST_ID_MESSAGE		X	WOL pattern ID of a EAPOL request ID message.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_REMOVE_WOL_PATTERN

OID_WDI_SET_ADVERTISEMENT_INFORMATION

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_ADVERTISEMENT_INFORMATION configures the Information Elements (IEs) and other advertisement settings to be included in the probe request, probe response, and beacons sent by the specified port. This request is only applicable to Wi-Fi Direct ports.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

When this command is received by the device, it shall update any relevant Wi-Fi Direct IEs, and append any necessary additional IEs in future outgoing messages sent by this port.

Set property parameters

WDI can provide a pre-configured set of prefix hashes for the advertised services. If a peer sends a hash, the driver first tries to match with a service name hash as defined in WDI_TLV_P2P_ADVERTISED_PREFIX_ENTRY. If a match is found from the prefix hashes, the driver searches for the service(s) in WDI_TLV_P2P_ADVERTISED_SERVICE_ENTRY that have the prefix and responds with those. If a match is not found, the driver tries to match the requested service name hash in WDI_TLV_P2P_ADVERTISED_SERVICE_ENTRY.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ADDITIONAL_IES		X	Additional IEs to be included.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_DEVICE_INFO		X	Wi-Fi Direct device information.
WDI_TLV_P2P_DEVICE_CAPABILITY		X	Wi-Fi Direct device capabilities.
WDI_TLV_P2P_GROUP_OWNER_CAPABILITY		X	Wi-Fi Direct Group Owner capability information
WDI_TLV_P2P_SECONDARY_DEVICE_TYPE_LIST		X	List of Wi-Fi Direct secondary device types.
WDI_TLV_P2P_ADVERTISED_SERVICES		X	Wi-Fi Direct advertised services.

Set property results

No additional data. The data in the header is sufficient.

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_ACTION_FRAME_RECEIVED The adapter must indicate ANQP Action Frame requests for the Service Information if it receives an ANQP request (or any other unknown action frame) from a peer.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_ASSOCIATION_PARAMETERS

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_ASSOCIATION_PARAMETERS specifies parameters that the adapter can use during association to a set of BSSIDs.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

This command replaces the previously configured list of BSSID-specific association parameters.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CONNECT_BSS_ENTRY	X		The BSS entries and parameters.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Dot11wdi.h

OID_WDI_SET_CLEAR_RECEIVE_COALESCING

Article • 03/14/2023

OID_WDI_SET_CLEAR_RECEIVE_COALESCING is used by the host to remove a packet filter for packet coalescing.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SET_CLEAR_RECEIVE_COALESCING			The packet filter ID to be removed.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_RECEIVE_COALESCING

OID_WDI_SET_CONNECTION_QUALITY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_CONNECTION_QUALITY provides a hint to the IHV component to enforce connection quality for a given virtualized port. This hint allows the port to optimize channel usage in different scenarios.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Note This property specifies data path quality of service hints, which may cause conflicts with other properties or tasks that are issued to the adapter.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_CONNECTION_QUALITY_PARAMETERS			The desired Wi-Fi connection quality hint.
WDI_TLV_LOW_LATENCY_CONNECTION_QUALITY_PARAMETERS		X	The behavior for low latency connection quality. This is only required if the connection quality is set to WDI_CONNECTION_QUALITY_LOW_LATENCY.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_DEFAULT_KEY_ID

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_DEFAULT_KEY_ID sets the default key ID for packet transmission on a port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DEFAULT_TX_KEY_ID_PARAMETERS			The default key ID for packet transmission on the port.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_DELETE_CIPHER_KEYS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_DELETE_CIPHER_KEYS deletes cipher keys from the device's cipher key table.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DELETE_CIPHER_KEY_INFO	X		The cipher keys to be deleted from the device's key table.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_ADD_CIPHER_KEYS

OID_WDI_SET_ENCAPSULATION_OFFLOAD

Article • 03/14/2023

OID_WDI_SET_ENCAPSULATION_OFFLOAD is sent by the OS to indicate that the lower edge driver (LE) should start doing the TCP Checksum/LSO offloads.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

When this message is received, the LE should indicate its current encapsulation offload configuration with NDIS_STATUS_WDI_INDICATION_TASK_OFFLOAD_CURRENT_CONFIG. For receive operations, the LE should not start the checksum offload until after it receives the OID_WDI_SET_ENCAPSULATION_OFFLOAD message.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SET_ENCAPSULATION_OFFLOAD_V4_PARAMETERS			Specifies if IPv4 offloading should be started.
WDI_TLV_SET_ENCAPSULATION_OFFLOAD_V6_PARAMETERS			Specifies if IPv6 offloading should be started.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_END_DWELL_TIME

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_END_DWELL_TIME is typically sent during an Action Frame exchange, either when WDI has to wait some time before sending a followup Action Frame, or when the protocol sequence is complete. This command can be sent on the device port or station port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

On receipt of this command, the firmware can choose to stop dwelling on the channel that had been specified when WDI sent the command to send the Action Frame. If the Dwell Time had already expired, the firmware should ignore this command.

Set property parameters

No additional parameters. The data in the header is sufficient.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_FAST_BSS_TRANSITION_PARAMETERS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_FAST_BSS_TRANSITION_PARAMETERS is sent in response to NDIS_STATUS_WDI_INDICATION_FT_ASSOC_PARAMS_NEEDED. It has the parameters required to send the (Re)Association request. The command is sent to the driver as a direct OID.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_STATUS			If this status is success, the rest of the fields (RSNIE, MDE, FTE) are present. This indicates that there are no problems or errors with the Authentication response (for example, MIC check failure) and the IHV can proceed with the reassociation request.
WDI_TLV_FT_RSNIE		X	The RSN IE byte blob.
WDI_TLV_FT_MDE		X	The MDE byte blob.
WDI_TLV_FT_FTE		X	The FTE byte blob.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_FLUSH_BSS_ENTRY

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_FLUSH_BSS_ENTRY is sent to the device to flush the list of BSS entries maintained by the adapter. This command can only be sent on the station port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

Set property parameters

No additional parameters. The data in the header is sufficient.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_MULTICAST_LIST

Article • 03/14/2023

OID_WDI_SET_MULTICAST_LIST specifies the multicast address list for a given port. This command can be set at any time.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

The IHV component should only fail the command if the list size exceeds the limit specified in **WDI_TLV_INTERFACE_ATTRIBUTES**.

After the host enables multicast packet filtering on the port using OID_WDI_SET_RECEIVE_PACKET_FILTER, the device must indicate received multicast frames with a destination address matching an address in the port's multicast list to the host. The device must clear the multicast list as part of processing of OID_WDI_TASK_DOT11_RESET. When the command is sent with no multicast list specified, the driver must clear its multicast list. In this case, no packets should be indicated up unless OID_WDI_SET_RECEIVE_PACKET_FILTER has the WDI_PACKET_FILTER_ALL_MULTICAST bit set.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_MULTICAST_LIST		X	List of multicast MAC addresses. The list must not be empty.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_NEIGHBOR_REPORT_ENTRIES

Article • 03/14/2023

ⓘ Important

This topic is part of the WDI driver model released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. WiFiCx is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_NEIGHBOR_REPORT_ENTRIES sends the list of neighbor reports received from the AP to the LE. This is sent as soon as the UE receives the neighbor report from the currently connected AP.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_NEIGHBOR_REPORT_ENTRY	X		The list of neighbor reports.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_NETWORK_LIST_OFFLOAD

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_NETWORK_LIST_OFFLOAD sets a list of preferred SSIDs for the firmware to scan for APs.

Scope	Set serialized with task	Normal execution time (seconds)
Primary port	Yes	1

There are two types of Network List Offload (NLO). One type is offload to NICs on Always On Always Connected (AOAC) systems. The other is Instant Connect NLO which, in Windows 8 and Windows 8.1, was only used for non-AOAC systems to quickly reconnect Wi-Fi at resume from hibernation. For Instant Connect, the list is sent down before the system goes into hibernation. Going forward, Instant Connect is used for resume from hibernation on AOAC systems that support it.

Instant Connect

WDI handles Instant Connect NLO and uses a combination of targeted scans to fulfill the request from the OS. IHV drivers do not need to handle this Instant Connect OS request.

When the OS resumes from hibernation, the OS sends an Instant Connect NLO. WDI makes a union of all channel hints for a targeted scan OID. IHV drivers should support such a targeted scan as defined in OID_WDI_TASK_SCAN. The following section applies to Network List Offload to capable NICs on AOAC systems.

Network List Offload

The OS does not request periodic background scans when in CS. A NLO scan is the preferred method in CS because the screen is off when users do not need to see all visible APs. SSIDs that users have auto-connect profiles set to auto-connect are the only useful APs. The list of SSIDs to offload from the OS has a preferred authentication and cipher pair, and up to four channel hints. When the list has a least one SSID, the firmware should start to do a NLO scan autonomously, following the schedule of fast scan and slow scan phases. The WDI compliant driver translates the operating system request to a firmware request. The firmware is expected to do a NLO scan according the schedule for the APs. The APs should support the preferred authentication and cipher pair associated with the SSIDs.

The request to the firmware has a list of channel hints for all offload SSIDs. The WDI compliant driver combines them for the firmware. For example, if SSID1[auth1, cipher1] has channel hints of 1 and 6, and SSID2[auth2, cipher2] has channel hints of 6 and 11, the request to the firmware is a list of SSIDs { SSID1[auth1, cipher1], SSID2[auth2, cipher2] } and list of channels to scan { 1, 6, 11 }.

In each scan period, the firmware scans for SSIDs that match the criteria on the list of channels, but not necessary constrained on the list of channels. The discovered AP information should be cached for the host to retrieve. The firmware indicates NLO discovery when at least one BSSID matches the SSID, algorithm, and cipher, but the channel match is not required.

Each OID_WDI_SET_NETWORK_LIST_OFFLOAD that the UE sends to the LE represents a fresh NLO scan request. Any previous such requests or states are renewed. LE scans for NLO and only indicates once for a found AP per request. The UE replumbs (12 times; this is subject to change) NLO at Dx transitions if a found AP is not connected successfully (due to reasons such as: an AP is found but devices move around, the AP signal fades, and the connection fails; or prolong EAP authentication fails partway through). The LE and firmware should delay the NLO scan schedule based on the delay configuration in **WDI_TLV_NETWORK_LIST_OFFLOAD_CONFIG**. This is a number that the UE uses to conform to the schedule of the operating system's original NLO command.

The default scan type for NLO is WDI_SCAN_TYPE_AUTO. When actively scanning a channel, the firmware should use the wildcard SSID. Visible APs should be compared with SSIDs on the offload list to decide a match. This is to reduce privacy exposure.

Indicating NLO discovery has two cases.

1. When the NIC is in D2, it must do the following steps.

    - Trigger the wake interrupt and wait for set power to D0 before continuing to the following steps.
    - Indicate that the firmware woke the stack with the reason of NLO discovery.
    - Return D0 command.

- Indicate NLO discovery with all of the found AP information.

2. When the NIC is in D0, it must do the following step.

   - Indicate NLO discovery with all of the found AP information.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_NETWORK_LIST_OFFLOAD_PARAMETERS			The NLO parameters.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_P2P_LISTEN_STATE

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_P2P_LISTEN_STATE sets the Wi-Fi Direct listen state on the port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

There are different levels of listen state, and the port is expected to adhere to concurrency requirements across ports.

This property is only applicable to virtualized Wi-Fi Direct Adapter Port interfaces.

When the listen state is active, the port is expected to park the radio on a social channel for a certain period of time.

If the adapter has a virtualized port operating on a non-social channel, the port may become discoverable on that channel. If this behavior is used, the port must be very highly available to allow other adapters to quickly discover it when in the scan phase of Wi-Fi Direct discovery. This is provided as a trade-off to avoid channel hopping in low latency scenarios.

Note This property specifies a radio time slice requirement to the port, which may cause conflicts with other properties or tasks issued to the port.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_LISTEN_STATE			Desired listen state.

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CHANNEL_NUMBER		X	The host’s desired listen channel when enabling the Wi-Fi Direct listen state. If this option is not specified, the port may select a listen channel on its own.
WDI_TLV_P2P_LISTEN_DURATION			Cycle duration and listen time.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_P2P_START_BACKGROUND_DISCOVERY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_P2P_START_BACKGROUND_DISCOVERY instructs the adapter to periodically perform Wi-Fi Direct discovery in the background

Scope	Set serialized with task	Normal execution time (seconds)	Affects data throughput/latency
Port	No	1	Yes

The adapter is required to scan the specified channels at regular intervals and be able to find a device that becomes discoverable within the device visibility timeout (typically 5 minutes). The behavior is similar to the regular Wi-Fi Direct Discovery scan (as defined in OID_WDI_TASK_P2P_DISCOVER), but it is not time-bound, and the adapter may schedule the scans at some later point in time. The adapter must perform at least one scan within each device visibility timeout. If the device visibility timeout is 0, the adapter should continue to scan regularly using its own cycle time. If a DISCOVER or SCAN task request is made during this time, the adapter should suspend the background discovery for the duration of the task and continue when the task is finished. Upon completing a background scan, the device should send the NDIS_STATUS_WDI_INDICATION_P2P_DISCOVERY_COMPLETE indication (with transaction ID equal to 0) to let the operating system know that it has completed a scan. The adapter must send this indication every time it completes a background scan.

If the channel list is provided, the adapter should only scan on the specified channels. Otherwise, it should scan all channels. If the firmware happened to discover a device outside of the specified channels, it should still send the information to the operating system.

When Listen Duration and channels (**WDI_TLV_P2P_DISCOVERY_CHANNEL_SETTINGS**) are specified, they refer to the listen times for the remote devices. Based on all the values of Listen Duration and channels, the adapter needs to come up with a schedule to scan the requested channels in the most efficient manner. The operating system may also specify multiple instances of Listen Duration and channels. In this case, the adapter should first come up with the scan schedule for those entries which have non-zero values of Listen Duration and Channel list. Then, the adapter should use default values in the following cases:

1. If the Listen duration is 0, the adapter should use the default scan times for the specified channels.
2. If the channel list is empty, the adapter should scan all of the channels in that band using the listen and cycle times specified for that band. The scan times would not apply to any channels that have separate listen durations specified by the operating system.

When the NIC is in D0, the adapter indicates the responses to the probe requests for the specific service name(s) as NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST notifications to the operating system. WDI caches the response information for the OS for the higher layer services, and notifies them as necessary.

When the NIC is in D2, it suspends background discovery until it goes back to D0.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_BACKGROUND_DISCOVER_MODE			Wi-Fi Direct Background Discover Mode parameters.
WDI_TLV_P2P_DISCOVERY_CHANNEL_SETTINGS	X	X	List of recommended channels to scan.
WDI_TLV_P2P_DEVICE_FILTER_LIST		X	List of Wi-Fi Direct devices and Group Owners to search for during Wi-Fi Direct device discover.
WDI_TLV_P2P_SERVICE_NAME_HASH	X	X	List of Service Hash names to be queried. This is required if WDI_P2P_SERVICE_DISCOVERY_TYPE_SERVICE_NAME_ONLY is specified.
WDI_TLV_VENDOR_SPECIFIC_IE		X	One or more IEs that must be included in the probe requests sent by the port.

Set property results

No additional data. The data in the header is sufficient.

Unsolicited indication

NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_P2P_STOP_BACKGROUND_DISCOVERY

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_P2P_STOP_BACKGROUND_DISCOVERY instructs the adapter to cancel the background discovery and stop any active scans in progress.

Scope	Set serialized with task	Normal execution time (seconds)	Affects data throughput/latency
Port	No	1	No

Set property parameters

No additional parameters. The data in the header is sufficient.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_P2P_WPS_ENABLED

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_P2P_WPS_ENABLED requests that the adapter enables or disables Wi-Fi Protected Setup (WPS) on the NIC.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_WPS_ENABLED			Specifies whether to enable or disable WPS.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_POWER_STATE

Article • 03/14/2023

OID_WDI_SET_POWER_STATE sets the power state of the device.

Scope	Set serialized with task	Normal execution time (seconds)
Adapter	Yes	10

A NIC comes up in D0 (device fully powered) when the system boots or when the NIC is plugged in to the system. When the condition is right (on AOAC platforms, this is when NIC Active reference is 0 on the NIC), the operating system prepares and puts the NIC in D0. When users are not present, the host goes to low power state to save power. The host may set the NIC into a lower power state where the NIC can keep connections autonomously for the host. The NIC wakes up the host for external events that the host expresses interest in.

OID_WDI_SET_POWER_STATE sets the device into D0, D1, D2, and D3. The D states are device class and platform specific. A Wi-Fi NIC usually supports only a subset of the states. For example, for Wi-Fi devices on SD bus, the supported set consists of D0, D2, and D3. The meaning of D2 and D3 are device-specific as well. For a Wi-Fi NIC on SDIO bus, it is defined to be able to wake from D2, but in D3, the NIC is halted.

A PCIe bus NIC supports D0 and D3, where D3 can be D3Hot or D3Cold. On the host software stack, there is only D3. D3hot or D3Cold depends on the host scenarios and underlying platform support. For example, in connected standby scenarios, the host offloads wake events to the NIC and sets the NIC in D3, which is D3hot with platform support to keep the NIC powered so that the NIC can watch for external events for the host. In the hibernation scenario, the host sets the NIC in D3 and the platform turns off the power to the NIC so the NIC does not use any power.

For an AOAC system that supports hibernation, the following is a summary of important system power states. On an AOAC system, a system sleep state is a connected standby state. This is the state where NICs are set to low power (D2 for SDBus NICs, D3 for PCIe NICs) and armed to wake. If the driver is suspended to the hard drive, it is the driver’s responsibility to resume firmware states as the driver does not go through reinitialization again (for example, DriverEntry is not called).

	Sleep	Hibernation	Hybrid shutdown	Full shutdown

	Sleep	**Hibernation**	**Hybrid shutdown**	**Full shutdown**
Request by	Power button (default)	shutdown /h	shutdown /s /hybrid	shutdown /s
UI	**Start** > **Power** > **Sleep**	--	**Start** > **Power** > **Shutdown**	--
System state	Connected standby	Hibernation	Hybrid shutdown	Power off
Driver state	Alive - armed to wake	Suspend to hard drive	Suspend to hard drive	Power off

For an AOAC system where hibernation is not required or supported, here is the summary of driver power states.

	Sleep	**Full shutdown**
Request by	Power button (default)	shutdown /s
UI	**Start** > **Power** > **Sleep**	**Start** > **Power** > **Shutdown**
System state	Connected standby	Power off
Driver state	Alive - armed to wake	Power off

Set power commands cannot fail. The firmware should never fail such commands. The Microsoft component ensures that there are no outstanding tasks or commands when it sends any set power command. While the set power command is outstanding, the Microsoft component also guarantees that no other commands or tasks are sent to the IHV component.

Power state	**Description**
D0 (fully powered)	The NIC is fully powered and ready to receive commands. The host never requests changes between low power states. For example, if the host wants to set the NIC power state from D2 to D3, it first sets the power state to D0, and then to D3.
D2 and armed for wake (SDBus NICs)	In D2, the host never sends requests to the firmware except the Set D0 command. See later sections in this topic for relevant flow charts.

Power state	Description
D3: power off (SDBus NICs), armed for wake (PCIe NICs)	For SDBus NICs, this state is powered off. For PCIe bus NICs, the operating system may arm NICs for wakes (D3Hot) or may turn off the power (D3Cold). Note that from the driver stack perspective, there is only D3 state. Multiple components are involved to enable the D3Hot state, including the ACPI table and the processing of NDIS system power IRPs that come from the operating system depending on end-user actions or inactions, such as hibernation, Connection Standby, and hybrid shutdown.
Dx for non-default ports	Dx is either D2 or D3. When the NIC is put into Dx all non-default ports are reset, which means all non-default ports are disconnected in Dx.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_POWER_STATE			The power state. This applies to the primary port.
WDI_TLV_ENABLE_WAKE_EVENTS		X	This field may only appear when the NIC is being put into low power and is armed to wake on any of the specified events (such as D2 on SD IO).
WDI_TLV_SET_POWER_DX_REASON		X	The set power reason.

Set property results

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ADAPTER_RESUME_REQUIRED		X	If the value is true, it signals to the OS that the firmware needs assistance in resuming its context. This should only occur when the driver is suspended to storage. The IHV component must reset the software state because the operating system issues a series of Wi-Fi commands to bring the firmware context and IHV component context up to date.

Enable wake events

A NIC specifies the set of events that it can detect to wake the stack. The operating system plumbs down a subset or full set of the events to the NIC with the low power command. Some wake event parameters are set much earlier than the Dx command. Others are set right before the Dx command to the firmware. All events only become enabled with the Dx command.

In this interface, the event that is set to enabled is plumbed down in the optional WDI_TLV_ENABLE_WAKE_EVENTS TLV as part of the OID_WDI_SET_POWER command for device power state Dx. The TLV is absent if the operating system does not want to arm the NIC to wake.

When the firmware receives a Dx command with WDI_TLV_ENABLE_WAKE_EVENTS, it may detect a wake event before it completes the Dx command. It should buffer the event, finish processing the command, and then assert the wake interrupt.

Each and every wake by the Wi-Fi NIC should be followed by a wake reason for why the NIC wakes the stack. A NIC wakes the stack by asserting the wake interrupt line, which is typically serviced by the bus or ACPI methods. The methods wake the CPU and required components to handle the wake event, and complete the Wi-Fi Wait Wake IRP for the stack. Subsequently, the operating system issues a D0 request to the driver and firmware. This request is a power OID to the driver that sends a D0 command to the firmware. The firmware holds the indication of the wake reason until it receives and completes the D0 command.

Note If the NIC receives the D0 command for some other reason (for example, the NIC does not wake the host), the NIC should not indicate a wake reason.

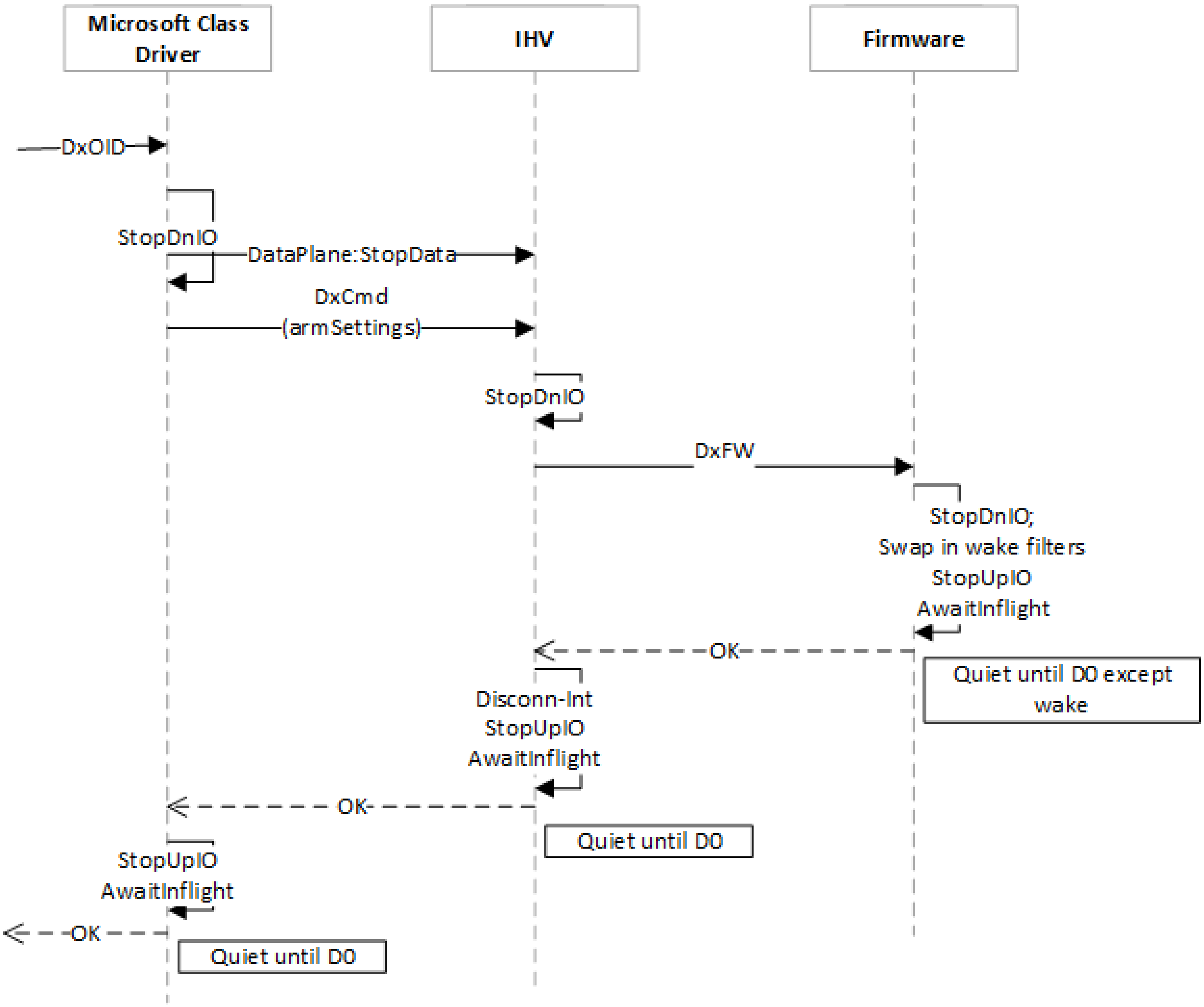
No enabled wake events

If there is no WDI_TLV_ENABLE_WAKE_EVENTS present, the operating system does not need the NICs to run at low power. The NICs may be completely powered off. If suspended to a hard drive, the NICs drivers are expected to resume firmware context at resume.

Power state interaction and transition examples

The following diagrams show interactions and sequences of transitions between D0 and Dx (D2 or D3) for the NIC. In this context, the "Miniport" represents the host or WDI compliant driver.

D0 to Dx (armed to wake)

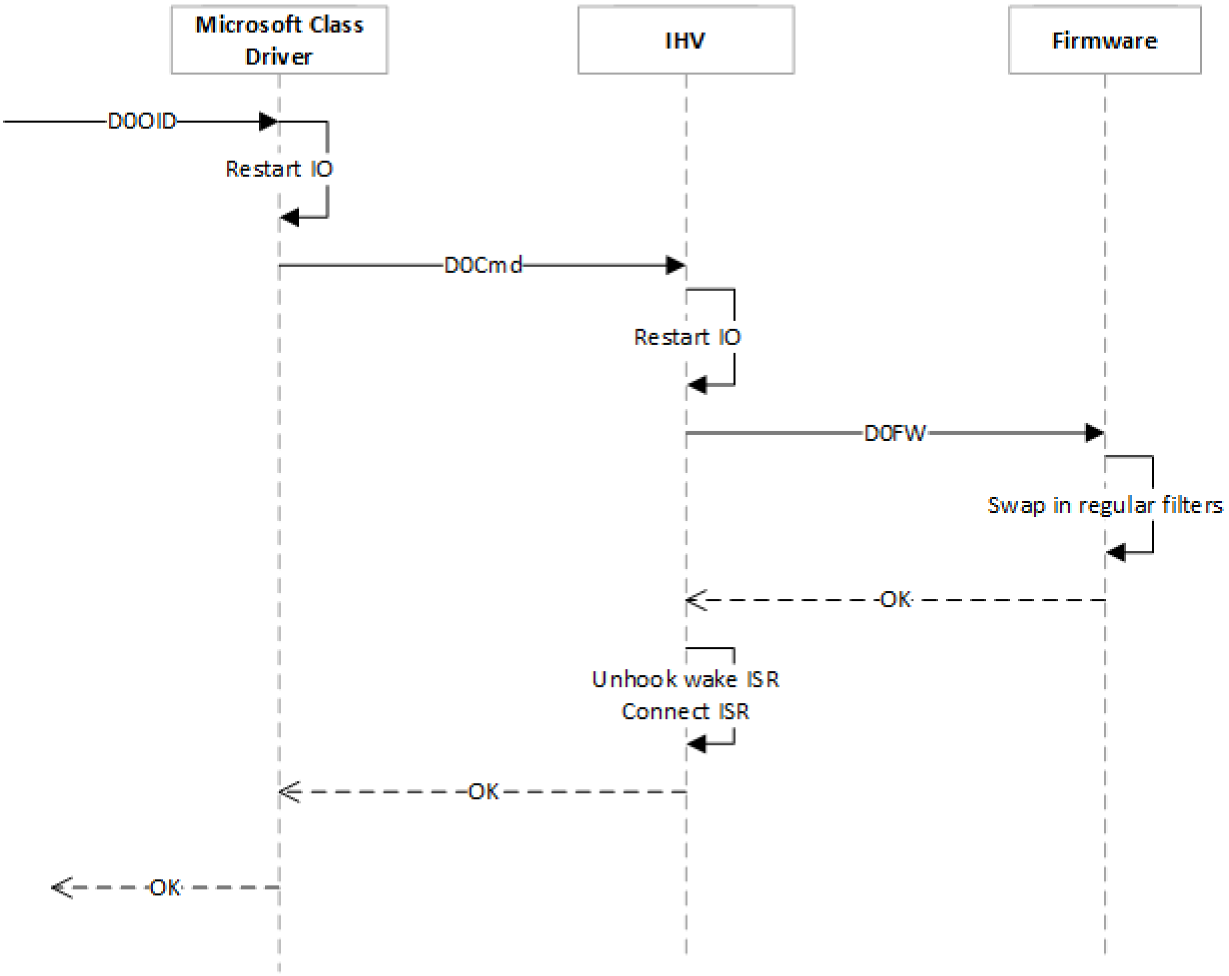


- Stop [DnIO|UpIO]: DnIO are messages (controls and data) to lower layer. UpIO are messages to upper layer.
  - Reject new requests from above layer (fail fast).
  - Stop initiating IO from this layer (except this Dx command).
  - Allow lower layer to inject TXs needed to go into Dx.
  - Flush queues.

- AwaitInflight: Waiting for IO calls to return, including DMA in progress. Flush queues.

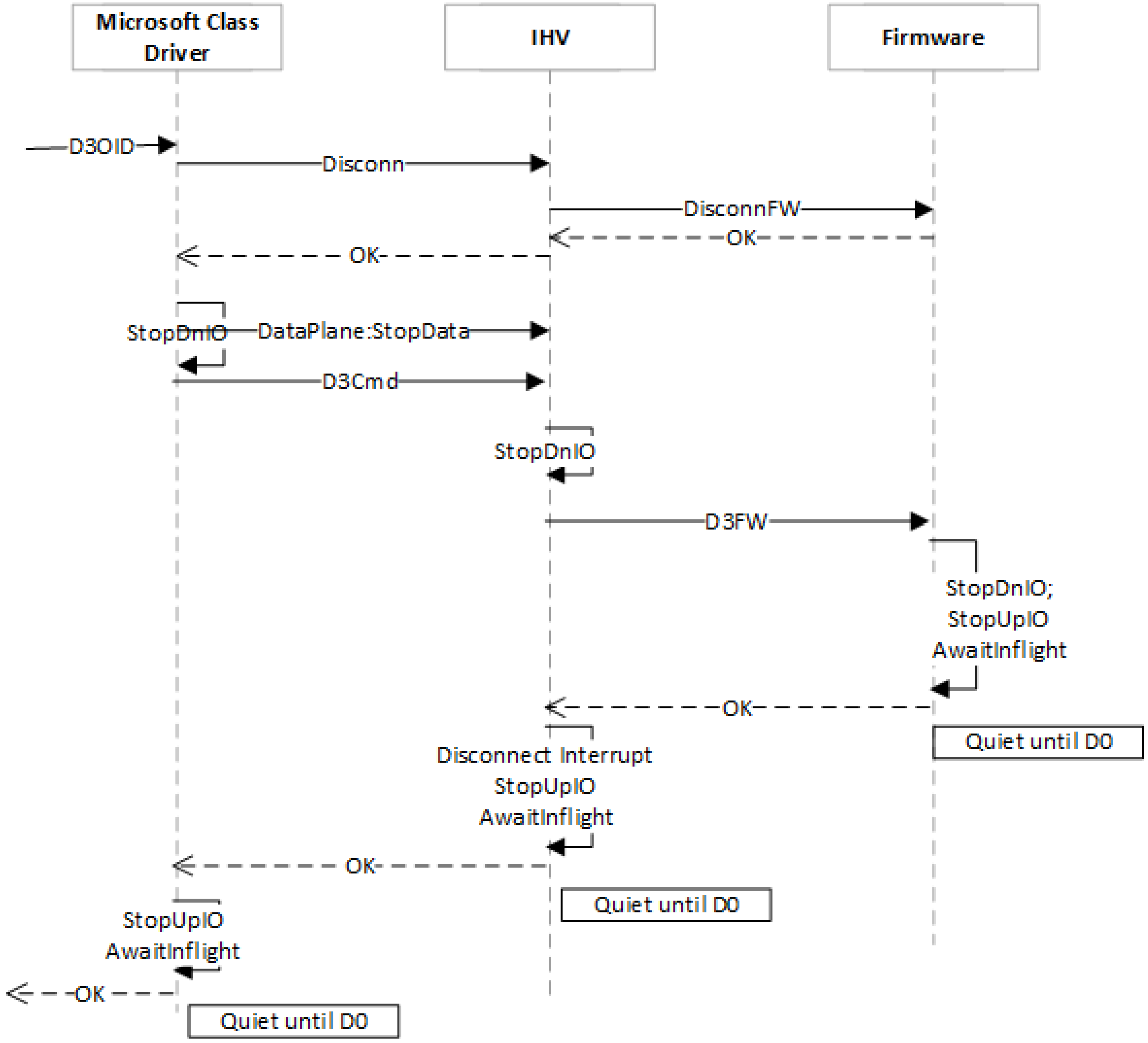
- Dx is any non-D0 state. For SDBus Wi-Fi, this is D2. For PCIe bus, this is D3Hot. Firmware shall not lose power.

Dx (armed to wake) to D0 transition



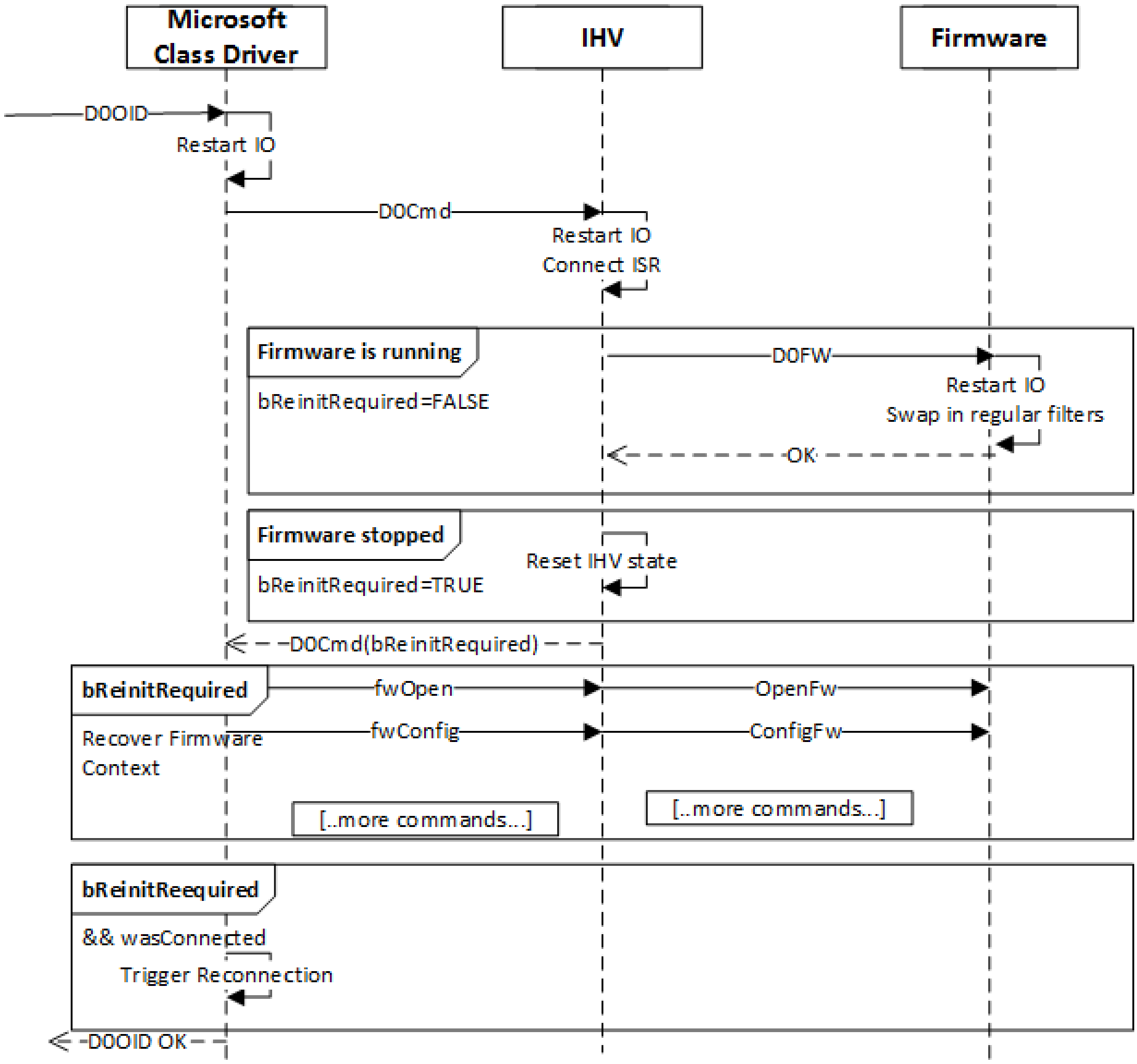
- If the NIC is armed to wake, it can't be D3Cold. Firmware must continue running in Dx.

D0 to D3 (not armed to wake) transition



- Stop [DnIO|UpIO]: DnIO are messages (controls and data) to lower layer. UpIO are messages to upper layer.
  - Reject new requests from above layer (fail fast).
  - Stop initiating IO from this layer (except this Dx command).
  - Allow lower layer to inject TXs needed to go into Dx.
  - Flush queues.
- AwaitInflight: Waiting for IO calls to return, including DMA in progress. Flush queues.
- D3 without PmParameters. The NIC may (D3Cold) or may not be powered off (for example, a shared power rail with a D0 device).

Dx (not armed to wake) to D0 transition



- D2 notArmToWake: Kept power, no reinitialization required.
- D3 notArmtoWake: Might be Hot or Cold. Cold requires that context be restored.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_PRIVACY_EXEMPTION_LIST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_PRIVACY_EXEMPTION_LIST is used by the host to provide the list of exemptions for packet description. The adapter applies these exemptions to packets it receives that match the IEEE EtherType value specified for the exemption.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PRIVACY_EXEMPTION_ENTRY	X	X	List of privacy exemption entries.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_RECEIVE_COALESCING

Article • 03/14/2023

OID_WDI_SET_RECEIVE_COALESCING is used by the host to add a packet filter for packet coalescing.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

When the host receives a request from the OS to set packet coalescing filters, it uses this command to add a packet filter for packet coalescing. To clear a packet filter for packet coalescing, see OID_WDI_SET_CLEAR_RECEIVE_COALESCING.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SET_RECEIVE_COALESCING			The packet coalescing parameters to be set.

Set property results

No additional parameters. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_CLEAR_RECEIVE_COALESCING

OID_WDI_SET_RECEIVE_PACKET_FILTER

Article • 03/14/2023

OID_WDI_SET_RECEIVE_PACKET_FILTER defines a bitmask filter for data packets to be indicated for a given virtualized port.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

If set, the port shall only notify the host of packets which match the provided filter. These filters are similar to the required 802.11 filters provided to OID_GEN_CURRENT_PACKET_FILTER.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PACKET_FILTER_PARAMETERS			The bitmask filter for data packets.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

OID_WDI_SET_REMOVE_PM_PROTOCOL_OFFLOAD

Article • 03/14/2023

OID_WDI_SET_REMOVE_PM_PROTOCOL_OFFLOAD removes the protocol offload specified by the protocol offload ID.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PM_PROTOCOL_OFFLOAD_REMOVE			Protocol offload ID.

Set property results

No additional parameters. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_GET_PM_PROTOCOL_OFFLOAD

OID_WDI_SET_ADD_PM_PROTOCOL_OFFLOAD

OID_WDI_SET_REMOVE_WOL_PATTERN

Article • 03/14/2023

OID_WDI_SET_REMOVE_WOL_PATTERN removes a wake-on-LAN (WOL) pattern from the firmware.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_WAKE_PACKET_PATTERN_REMOVE			WOL pattern ID.

Set property results

No additional parameters. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_ADD_WOL_PATTERN

OID_WDI_SET_SAE_AUTH_PARAMS

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

OID_WDI_SET_SAE_AUTH_PARAMS is sent by WDI in response to an NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED indication from the driver. It contains the parameters required to send the Simultaneous Authentication of Equals (SAE) Commit or Confirm request, or an error message indicating a failure to perform SAE with the BSSID.

This command is sent as a Direct OID request to the driver.

For more information about SAE authentication, see WPA3 SAE authentication.

Command parameters

TLV	**Type**	**Multiple TLV instances allowed**	**Optional**	**Description**
WDI_TLV_BSSID	**WDI_MAC_ADDRESS**			The BSSID of the AP.
WDI_TLV_SAE_REQUEST_TYPE	**WDI_SAE_REQUEST_TYPE**			The type of SAE request frame to send to the BSSID.
WDI_TLV_SAE_COMMIT_REQUEST	WDI_SAE_COMMIT_REQUEST		X	The SAE Commit request parameters.
WDI_TLV_SAE_CONFIRM_REQUEST	WDI_SAE_CONFIRM_REQUEST		X	The SAE Confirm request parameters.

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SAE_STATUS	**WDI_SAE_STATUS**		X	SAE authentication failure error status.

Requirements

Minimum supported client: Windows 10, version 1903

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

OID_WDI_SET_TCP_OFFLOAD_PARAMETERS

Article • 03/14/2023

OID_WDI_SET_TCP_OFFLOAD_PARAMETERS is sent down to the device from the OS to set the TCP offload parameters.

Scope	Set serialized with task	Normal execution time (seconds)
Port	Yes	1

This command is sent in some cases such as when there is a need to turn off the offloads due to a performance issue.

The lower edge driver (LE) must use the contents of **WDI_TLV_TCP_SET_OFFLOAD_PARAMETERS** to update the currently reported TCP offload capabilities. After the update, the LE must report the current task offload capabilities with NDIS_STATUS_WDI_INDICATION_TASK_OFFLOAD_CURRENT_CONFIG. This status indication ensures that all of the overlying protocol drivers are updated with the new capabilities information.

Set property parameters

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_TCP_SET_OFFLOAD_PARAMETERS			The TCP offload parameters to be set.

Set property results

No additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Dot11wdi.h

OID_WDI_TCP_RSC_STATISTICS

Article • 03/14/2023

OID_WDI_TCP_RSC_STATISTICS is a get command that queries the RSC statistics of the hardware.

Scope	Set serialized with task	Normal execution time (seconds)
Port	No	1

Get property parameters

No additional parameters. The data in the header is sufficient.

Get property results

TLV	Multiple TLV instances allowed	Optional	Description
WDI_TLV_TCP_RSC_STATISTICS_PARAMETERS			TCP RSC statistics of the hardware.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_ACTION_FRAME_RECEIVED

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_ACTION_FRAME_RECEIVED to indicate that an Action Frame has been received.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID			The BSSID of the source.
WDI_TLV_BSS_ENTRY_CHANNEL_INFO			The logical channel number and band ID for the BSS entry.
WDI_TLV_ACTION_FRAME_BODY			The incoming Action Frame body.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_ADVERTISEMENT_INFORMATION

NDIS_STATUS_WDI_INDICATION_AP_ASSOCIATION_REQUEST_RECEIVED

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_AP_ASSOCIATION_REQUEST_RECEIVED to indicate that a Wi-Fi Association Request Frame has been received for an operational Wi-Fi Direct Group Owner. The host may issue an OID_WDI_TASK_SEND_AP_ASSOCIATION_RESPONSE for this request.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_INCOMING_ASSOCIATION_REQUEST_INFO			The incoming Association Request information.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_ASSOCIATION_PARAMETERS_REQUEST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_ASSOCIATION_PARAMETERS_REQUEST to request association parameters for a set of BSSIDs from the host.

Object
Port

This indication can be sent by the adapter when it finds a BSS entry that is a candidate for association based on the current settings. Upon receiving this indication, the host checks if the association parameters are available, and if so, sends them with OID_WDI_SET_ASSOCIATION_PARAMETERS.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ASSOCIATION_PARAMETERS_REQUESTED_TYPE			The list of requested association parameters.
WDI_TLV_BSS_ENTRY	X	X	The list of BSSIDs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_CONNECT

NDIS_STATUS_WDI_INDICATION_ASSOCIATION_RESULT

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_ASSOCIATION_RESULT to indicate association results.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ASSOCIATION_RESULT	X		A list of association results.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_CONNECT

OID_WDI_TASK_ROAM

NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_BSS_ENTRY_LIST to inform the host about updates to the BSS entries. This is an unsolicited indication and can be sent at any time.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSS_ENTRY	X	X	The list of updated BSSIDs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_SCAN

OID_WDI_TASK_P2P_DISCOVER

OID_WDI_SET_P2P_START_BACKGROUND_DISCOVERY

NDIS_STATUS_WDI_INDICATION_CAN_SUSTAIN_AP

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_CAN_SUSTAIN_AP to indicate that the port is ready to receive a OID_WDI_TASK_START_AP request, after previously indicating NDIS_STATUS_WDI_INDICATION_STOP_AP.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_INDICATION_CAN_SUSTAIN_AP			The reason the adapter can now sustain 802.11 AP functionality.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_CHANGE_OPERATION_MODE_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_CHANGE_OPERATION_MODE_COMPLETE to indicate the completion of OID_WDI_TASK_CHANGE_OPERATION_MODE.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_CIPHER_KEY_UPDATED

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers send this indication to indicate that the cipher key(s) have been updated.

This indication is sent only while the driver has not offload the RSN GTK rekey (via the WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY filed in the OID_WDI_SET_ADD_PM_PROTOCOL_OFFLOAD command). If the driver is currently in the offload state for the Rsn GTK Rekey, then it should not indicate via this method and should allow the updated key information to be queried via the OID_WDI_GET_PM_PROTOCOL_OFFLOAD command when it comes out of the offload state.

For example, the driver would send this notification if it or the firmware receives a new GTK/iGTK in the WNM-Sleep mode response.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PM_PROTOCOL_RSN_OFFLOAD_KEYS			The currently configured Rsn Eapol key information.

Requirements

Minimum supported client: Windows 10, version 1803

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_CLOSE_COMPLETE

Article • 03/14/2023

Miniport drivers use NDIS_STATUS_WDI_INDICATION_CLOSE_COMPLETE to indicate the completion of OID_WDI_TASK_CLOSE.

Object
Adapter

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_CONNECT_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_CONNECT_COMPLETE to indicate the completion of OID_WDI_TASK_CONNECT.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_CREATE_PORT_COMPLETE

Article • 03/14/2023

Miniport drivers use NDIS_STATUS_WDI_INDICATION_CREATE_PORT_COMPLETE to indicate the completion of OID_WDI_TASK_CREATE_PORT.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_PORT_ATTRIBUTES			The attributes of the created port.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_DELETE_PORT_COMPLETE

Article • 03/14/2023

Miniport drivers use NDIS_STATUS_WDI_INDICATION_DELETE_PORT_COMPLETE to indicate the completion of OID_WDI_TASK_DELETE_PORT.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_DEVICE_SERVICE_EVENT

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

The NDIS_STATUS_WDI_INDICATION_DEVICE_SERVICE_EVENT status indication is used by an IHV miniport driver to pass on unsolicited information about a device to a user mode client.

Device service indications must be sent by the miniport driver only when in the *D0* power state, and it must not cause the device to wake from *Dx*. WDI will drop this indication without forwarding it up the stack if it receives it when in *Dx*.

This indication is currently handled only on the default port (station).

The miniport driver should send a separate notification for every device service GUID and opcode pair whenever necessary.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DEVICE_SERVICE_PARAMS_DATA_BLOB		X	The information received from the IHV miniport driver.
WDI_TLV_DEVICE_SERVICE_PARAMS_GUID			The GUID that identifies the device service to which this indication belongs (as defined by the IHV/OEM).

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DEVICE_SERVICE_PARAMS_OPCODE			The opcode specific to the device service.

Requirements

Minimum supported client: Windows 10, version 1809

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_DISASSOCIATION

Article • 03/14/2023

> ⓘ **Important**
>
> This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_DISASSOCIATION to indicate that a port disconnected from the network.

Object
Port

The disconnect may be triggered by a command from the operating system or triggered from the network. Network triggered disconnect may be explicit from received disassociation or deauthentication packets, or may be implicit when the port cannot detect the presence of the peer it is connected to.

Before the disassociation indication is sent, the port must clear the state associated with this peer. This includes any keys and 802.1x port authorization information associated with this peer. The port must not trigger a roam on its own.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DISASSOCIATION_INDICATION_PARAMETERS			The disassociation indication parameters.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_DISCONNECT_DEAUTH_FRAME		X	The deauthentication frame that was received. This does not include the 802.11 MAC header.
WDI_TLV_DISCONNECT_DISASSOCIATION_FRAME		X	The disassociation frame that was received. This does not include the 802.11 MAC header.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_DISCONNECT

OID_WDI_TASK_ROAM

NDIS_STATUS_WDI_INDICATION_DISCONNECT_COMPLETE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_DISCONNECT_COMPLETE to indicate the completion of OID_WDI_TASK_DISCONNECT.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_DOT11_RESET_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_DOT11_RESET_COMPLETE to indicate the completion of OID_WDI_TASK_DOT11_RESET.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_FIRMWARE_STALLED

Article • 03/14/2023

NDIS_STATUS_WDI_INDICATION_FIRMWARE_STALLED is used to indicate that the firmware stalled.

Object
Adapter

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_FT_ASSOC_PARAMS_NEEDED

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_FT_ASSOC_PARAMS_NEEDED to request parameters for 802.11r roaming.

Object
Port

During OID_WDI_TASK_ROAM, WDI provides the parameters to send the 802.11 Authentication Request (PmkR0Name, R0KH-ID, SNonce, MDIE). Upon receiving the Authentication response, the LE requests additional needed parameters for the reassociation request, such as PMKR1Name and R1KH-ID. The LE also sends the parameters received in the Authentication Response (ANonce, SNonce, and R1KHID).

For a connection where Initial Mobility Domain is successfully done, the LE should only perform 11r roams (Fast roams). The LE can use the candidate list provided by the operating system, or use their own for the roams. If the LE uses its own candidate list, it must use the parameters (MDE, FTE, and PMKR0Name) provided in any one of the candidates suggested by the operating system to do a 11r roam. 11r is disabled whenever the connection is in FIPS mode. 11r fast roaming is currently only supported for FT over 1x authentication type.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID			The BSSID of the AP.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_FT_AUTH_REQUEST			The authentication request byte blob.
WDI_TLV_FT_AUTH_RESPONSE			The authentication response byte blob.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_ROAM

NDIS_STATUS_WDI_INDICATION_IHV_EVENT

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_IHV_EVENT to pass IHV specific information to the IHV extensibility module.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_IHV_DATA		X	The event to be sent to the IHV extensibility module.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_IHV_TASK_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

NDIS_STATUS_WDI_INDICATION_IHV_TASK_COMPLETE indicates the completion of OID_WDI_TASK_IHV.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient. The completion status from the message is not forwarded to anyone.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_IHV_TASK_REQUEST

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_IHV_TASK_REQUEST to request that the Microsoft component queue an IHV task.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_IHV_TASK_REQUEST_PARAMETERS			The IHV-requested priority for this task. Refer to the WDI_IHV_TASK_PRIORITY enum for valid values.
WDI_TLV_IHV_TASK_DEVICE_CONTEXT		X	The IHV-provided context information that is forwarded to OID_WDI_TASK_IHV.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Dot11wdi.h

See also

OID_WDI_TASK_IHV

NDIS_STATUS_WDI_INDICATION_LINK_STATE_CHANGE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_LINK_STATE_CHANGE to indicate any of the following situations:

- The link speed changed.
- The link quality changed by more than a threshold value. The threshold is 1 if the connection quality hint is set to WDI_CONNECTION_QUALITY_LOW_LATENCY (defined in WDI_CONNECTION_QUALITY_HINT). Otherwise, the threshold is 5.

Object
Port

This information from this indication is used by the host to keep track of metadata about the current link, and it may be propagated to the user.

In Station and P2P Client cases, the Peer MAC Address is set to the BSSID of the connected network. In AP/P2P GO cases, the Peer MAC Address is set to the MAC address of a given connected device.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_LINK_STATE_CHANGE_PARAMETERS			The link state change parameters.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_NLO_DISCOVERY

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_NLO_DISCOVERY to indicate Network List Offload (NLO) discovery.

Object
Port

The firmware detects APs for SSIDs pushed down in NLO. NLO is used in non-AOAC systems for fast connection when resuming from system sleep. It is also used in AOAC systems to scan APs for SSIDs that are pushed to the firmware.

The OS does not request periodic background scans when in CS. NLO scan is the preferred method in CS because the screen is off when users don't need to see all visible APs but those for SSIDs that users have auto-connect profiles to auto connect to. The list of SSIDs to offload from the OS has a preferred authentication and cipher pair and up to 4 channel hints. When the list has at least one SSID, the firmware should start an NLO scan autonomously following the schedule of fast scan and slow scan phases. The class driver translates the OS request to a firmware request. The firmware is expected to do NLO scan according the schedule for the APs that support the preferred authentication and cipher pair associated with the SSIDs.

In each scan period, the firmware scans for SSIDs that match the criteria on the list of channels but not necessary constrained on the list of channels. The discovered AP information should be cached for indication.

When any matches are found, the firmware indicates NLO discovery and caches the list of discovered AP information for the host to retrieve.

The indication of NLO discovery happens in the following two cases.

- When the NIC is in Dx:

    1. Trigger the wake interrupt and wait for set power to D0 to continue the following steps.
    2. Indicate NLO discovery.
    3. Indicate that the firmware woke the stack with the reason of NLO discovery.

- When the NIC is in D0:
    - Indicate NLO discovery.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSS_ENTRY	X		A list of BSSIDs. The list must at least contain the entry that triggered this discovery status.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_OPEN_COMPLETE

Article • 03/14/2023

Miniport drivers use NDIS_STATUS_WDI_INDICATION_OPEN_COMPLETE to indicate the completion of OID_WDI_TASK_OPEN.

Object
Adapter

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_P2P_ACTION_FRAME_RECEIVED

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_P2P_ACTION_FRAME_RECEIVED to indicate that a Wi-Fi Direct Action Frame has been received.

Object
Port

The host may issue an OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME for this request.

The port must indicate these packets in any of the following situations:

- The port is in listen state.
- The port has a GO in operational state.
- The port is dwelling on a remote listen channel when an OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME or OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME has been recently issued.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_INCOMING_FRAME_INFORMATION			The incoming Wi-Fi Direct Action Frame information. This information is forwarded back to the port when the host issues OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME

OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME

NDIS_STATUS_WDI_INDICATION_P2P_DISCOVERY_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_P2P_DISCOVERY_COMPLETE to indicate the completion of OID_WDI_TASK_P2P_DISCOVER.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_P2P_GROUP_OPERATING_CHANNEL

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_P2P_GROUP_OPERATING_CHANNEL to indicate which operating channel a given Wi-Fi Direct port is operating on.

For a Wi-Fi Direct Client port, this must be indicated during the connect (before the connect completion).

For a Wi-Fi Direct GO port, this must be indicated during OID_WDI_TASK_START_AP (before the OID completion).

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CHANNEL_NUMBER			The operating channel the given Wi-Fi Direct port is operating on.
WDI_TLV_P2P_CHANNEL_INDICATE_REASON			The reason for sending the indication.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_P2P_OPERATING_CHANNEL_ATTRIBUTES

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_P2P_OPERATING_CHANNEL_ATTRIBUTES to indicate the preferred operating channel to start a GO, the preferred listen channel if asked to enter listen state, and the full set of supported channels at any point of time. The indication is sent once when adapter initializes, and then sent each time one of these parameters changes due to events such as roaming or connecting or disconnecting from an access point.

Object
Port

The operating channel and channel list values are local settings and do not account for the actual channel negotiation during GO negotiation/invitation. The driver is still expected to negotiate the channel when GO negotiation/invitation is performed.

It is expected that the listen channel reported by the driver is honored if listen state is turned on. It is expected that this indication is fired if the host configured a listen channel that is different from the preferred listen channel reported earlier via this indication.

Payload data

Type	Multiple TLV instances allowed	Optional	Description

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_CHANNEL_NUMBER			The Wi-Fi Direct Operating channel attribute.
WDI_TLV_P2P_CHANNEL_LIST_ATTRIBUTE			The full set of channels supported by the local adapter.
WDI_TLV_P2P_LISTEN_CHANNEL			The Wi-Fi Direct Listen channel attribute.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_P2P_SEND_REQUEST_ACTION_FRAME_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_P2P_SEND_REQUEST_ACTION_FRAME_COMPLETE to indicate information about the Request Action frame sent by OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SEND_ACTION_FRAME_RESULT		This TLV is only required if the status is success.	Information about the Request Action frame that was sent to the peer.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_P2P_SEND_RESPONSE_ACTION_FRAME_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_P2P_SEND_RESPONSE_ACTION_FRAME_COMPLETE to indicate information about the Response Action frame sent by OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_P2P_SEND_ACTION_FRAME_RESULT		This TLV is only required if the status is success.	Information about the Response Action frame that was sent to the peer.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016

Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_RADIO_STATUS

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_RADIO_STATUS to indicate changes in the adapter's radio state. This unsolicited indication is sent when a software radio change is triggered by the host, and when a hardware radio state change is detected by the adapter.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_RADIO_STATE			The current state of the radio in hardware and software.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

WDI_TASK_SET_RADIO_STATE

NDIS_STATUS_WDI_INDICATION_REQUEST_FTM_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers send the **NDIS_STATUS_WDI_INDICATION_REQUEST_FTM_COMPLETE** status indication to the host as a task completion indication for OID_WDI_TASK_REQUEST_FTM. This notification contains a list of Fine Timing Measurement (FTM) responses received from each requested target.

Payload data

Type	TLV	Multiple TLV instances allowed	Optional	Description
WDI_STATUS	A field in the header.		The general completion status of the event.	
WDI_TLV_FTM_RESPONSE	Multiple TLV<WDI_TLV_FTM_RESPONSE>	X		A list of FTM responses for each target.

Requirements

Minimum supported client: Windows 10, version 1903

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_ROAM_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_ROAM_COMPLETE to indicate the completion of OID_WDI_TASK_ROAM.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_ROAM

NDIS_STATUS_WDI_INDICATION_ROAMING_NEEDED

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_DISASSOCIATION to indicate that the host should try to find a better peer to connect to. This notification is used when the link quality with the currently connected peer falls below a certain threshold. On sending this notification, the host may trigger a roam scan and/or a roam operation. The Microsoft component does not perform a disconnect before it starts the roam operation.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ROAMING_NEEDED_PARAMETERS			The reason for the roam trigger. When a OID_WDI_TASK_ROAM is triggered, this reason is forwarded to it.

Requirements

Minimum supported client	Windows 10

Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_ROAM

NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

The Wi-Fi adapter sends this indication to request parameters for Simultaneous Authentication of Equals (SAE) authentication.

When the miniport driver is requested to perform SAE authentication with a target BSSID, it needs to request information at various stages of authentication. Initially, it requests parameters for the Commit request frame, then the Confirm request frame if successful. If the driver encounters an irrecoverable timeout or error, it also indicates that to the OS.

This indication is sent during the SAE authentication process. For more information, see WPA3-SAE authentication.

Payload data

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_BSSID	**WDI_MAC_ADDRESS**			The BSS ID of the AP.

TLV	Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_SAE_INDICATION_TYPE	**WDI_SAE_INDICATION_TYPE**			The type of information needed to continue SAE authentication with the BSSID, or notification that authentication cannot continue.
WDI_TLV_SAE_COMMIT_RESPONSE	TLV<LIST<UINT8>>		X	The SAE Commit Response frame.
WDI_TLV_SAE_CONFIRM_RESPONSE	TLV<LIST<UINT8>>		X	The SAE Confirm Response frame.
WDI_TLV_SAE_STATUS	**WDI_SAE_STATUS**		X	The SAE authentication failure error status.

Requirements

Minimum supported client: Windows 10, version 1903

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_SCAN_COMPLETE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_SCAN_COMPLETE to indicate the completion of OID_WDI_TASK_SCAN.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_SCAN

NDIS_STATUS_WDI_INDICATION_SEND_AP_ASSOCIATION_RESPONSE_COMPLETE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_SEND_AP_ASSOCIATION_RESPONSE_COMPLETE to indicate information about the AP association response sent by OID_WDI_TASK_SEND_AP_ASSOCIATION_RESPONSE.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ASSOCIATION_RESPONSE_RESULT_PARAMETERS			The association response parameters.

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_ASSOCIATION_RESPONSE_FRAME			The received association response. This does not include the 802.11 MAC header.
WDI_TLV_BEACON_IES			The beacon IEs from the association.
WDI_TLV_PHY_TYPE_LIST			The list of PHY types.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_SEND_REQUEST_ACTION_FRAME_COMPLETE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_SEND_REQUEST_ACTION_FRAME_COMPLETE to indicate the completion of OID_WDI_TASK_SEND_REQUEST_ACTION_FRAME.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_SEND_REQUEST_ACTION_FRAME

NDIS_STATUS_WDI_INDICATION_SEND_RESPONSE_ACTION_FRAME_COMPLETE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_SEND_RESPONSE_ACTION_FRAME_COMPLETE to indicate the completion of OID_WDI_TASK_SEND_RESPONSE_ACTION_FRAME.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_SEND_RESPONSE_ACTION_FRAME

NDIS_STATUS_WDI_INDICATION_SET_RADIO_STATE_COMPLETE

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_SET_RADIO_STATE_COMPLETE to indicate the completion of OID_WDI_TASK_SET_RADIO_STATE.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_SET_RADIO_STATE

NDIS_STATUS_WDI_INDICATION_START_AP_COMPLETE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_START_AP_COMPLETE to indicate the completion of OID_WDI_TASK_START_AP.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_START_AP

NDIS_STATUS_WDI_INDICATION_STOP_AP

Article • 03/14/2023

ⓘ **Important**

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_STOP_AP to indicate that the adapter cannot sustain 802.11 Access Point (AP) functionality on any of the PHYs. The adapter should send this indication only after the NIC has stopped any APs that are operating on the available PHYs. The host blocks all OID_WDI_TASK_START_AP requests until the adapter sends NDIS_STATUS_WDI_INDICATION_CAN_SUSTAIN_AP.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_INDICATION_STOP_AP			The reason the adapter cannot sustain 802.11 AP functionality on any of the PHYs.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_START_AP

NDIS_STATUS_WDI_INDICATION_CAN_SUSTAIN_AP

NDIS_STATUS_WDI_INDICATION_STOP_AP_COMPLETE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_STOP_AP_COMPLETE to indicate the completion of OID_WDI_TASK_STOP_AP.

Object
Port

Payload data

This indication contains no additional data. The data in the header is sufficient.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_TASK_STOP_AP

NDIS_STATUS_WDI_INDICATION_TASK_OFFLOAD_CURRENT_CONFIG

Article • 03/14/2023

Miniport drivers use NDIS_STATUS_WDI_INDICATION_TASK_OFFLOAD_CURRENT_CONFIG to indicate when there is a change in the TCP offload capabilities of the hardware.

Object
Port

When there is a change in the TCP offload capabilities of the hardware, the LE sends this unsolicited indication to the UE, with the new TCP checksum/LSO capabilities. Use the values **NDIS_OFFLOAD_SET_OFF** and **NDIS_OFFLOAD_SET_ON** for members in WDI_TLV_TCP_OFFLOAD_CAPABILITIES for indicating changes in offload capabilities. When the UE sends down a OID_WDI_SET_TCP_OFFLOAD_PARAMETERS, the LE should update the offload capabilities and then send this indication so that the OS is updated with the latest offload capabilities information.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_TCP_OFFLOAD_CAPABILITIES		X	The TCP/IP checksum and Large Send Offload capabilities.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

See also

OID_WDI_SET_TCP_OFFLOAD_PARAMETERS

NDIS_STATUS_WDI_INDICATION_TKIP_MIC_FAILURE

Article • 03/14/2023

ⓘ Important

This topic is part of the **WDI driver model** released in Windows 10. The WDI driver model is in maintenance mode and will only receive high priority fixes. **WiFiCx** is the Wi-Fi driver model released in Windows 11. We recommend that you use WiFiCx to take advantage of the latest features.

Miniport drivers use NDIS_STATUS_WDI_INDICATION_TKIP_MIC_FAILURE to indicate when a received packet that was successfully decrypted by the TKIP cipher algorithm fails the message integrity code (MIC) verification.

Object
Port

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_TKIP_MIC_FAILURE_INFO			The TKIP MIC failure information.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

NDIS_STATUS_WDI_INDICATION_WAKE_REASON

Article • 03/14/2023

Miniport drivers use NDIS_STATUS_WDI_INDICATION_WAKE_REASON to indicate the reason for a wake when the NIC wakes the host. The wake reason is used for debugging purposes and has no functional effect.

Object
Port

When the host goes to low power state, it offloads a few functions to the NIC and arms the NIC for wake. When a wake event occurs, the NIC asserts the wake interrupt line to wake the host. The host then brings the NIC into D0 (running power state). The NIC must indicate the wake reason once it enters D0.

If the wake reason is a wake packet, the NIC should also include the wake packet and the wake pattern ID that matches the packet. The packet is encapsulated as WDI_TLV_INDICATION_WAKE_PACKET. The wake reason should also include WDI_TLV_INDICATION_WAKE_PACKET_PATTERN_ID to specify the pattern ID which matches the packet.

Payload data

Type	Multiple TLV instances allowed	Optional	Description
WDI_TLV_INDICATION_WAKE_REASON			The wake reason.
WDI_TLV_INDICATION_WAKE_PACKET		X	The wake packet.
WDI_TLV_INDICATION_WAKE_PACKET_PATTERN_ID		X	The ID of the pattern that matches the wake packet. The ID is obtained from the Add command of the pattern.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Dot11wdi.h

WDI_BAND_ID

Article • 03/03/2023

The WDI_BAND_ID data type is a UINT32 value that defines a band ID.

```C++
typedef UINT32 WDI_BAND_ID;
```

Remarks

Possible band ID values are as follows:

Name	Value	Description
WDI_BAND_ID_ANY	0xFFFFFFFF	All bands
WDI_BAND_ID_2400	1	2.4 GHz
WDI_BAND_ID_5000	2	5 GHz
WDI_BAND_ID_60000	3	60 GHz
WDI_BAND_ID_900	4	900 MHz
WDI_BAND_ID_CUSTOM_START	0x80000000	Specifies the start of the range that is used to define a band ID reported by an IHV.
WDI_BAND_ID_CUSTOM_END	0x81000000	Specifies the end of the range that is used to define a band ID reported by an IHV.

Requirements

Minimum supported client: Windows 10

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_CHANNEL_NUMBER

Article • 03/03/2023

The WDI_CHANNEL_NUMBER data type is a UINT32 value that defines a channel number.

C++

```
typedef UINT32 WDI_CHANNEL_NUMBER;
```

Requirements

Minimum supported client: Windows 10

Minimum supported server: Windows Server 2016

Header: Wditypes.hpp

WDI_EXTENDED_TID

Article • 03/03/2023

The WDI_EXTENDED_TID data type is a UINT8 value that defines a traffic identifier (TID).

```c++
typedef UINT8 WDI_EXTENDED_TID;
```

Remarks

Possible values are as follows:

Value	Description
0-15	802.11 TIDs
16 (WDI_EXT_TID_NON_QOS)	Non-QoS data
17-24	Reserved for use with IHV-injected frames. Frames with extended TID in the interval 17-24 are considered higher priority than those with a smaller extended TID in the same interval 17-24.
25-30	Unused values
31 (WDI_EXT_TID_UNKNOWN)	Unknown/unspecified

Requirements

Minimum supported client: Windows 10

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

WDI_FRAME_ID

Article • 03/03/2023

The WDI_FRAME_ID data type is a UINT16 value that defines a frame ID. This is only an identifier. It does not convey information about the ordering of frames.

```c++
typedef UINT16 WDI_FRAME_ID;
```

Requirements

Minimum supported client: Windows 10

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

WDI_PEER_ID

Article • 03/03/2023

The WDI_PEER_ID data type is a UINT16 value that defines a peer ID.

```c++
typedef UINT16 WDI_PEER_ID;
```

Remarks

If you want to specify any peer (wildcard), you can use the WDI_PEER_ANY (0xFFFF) value.

Requirements

Minimum supported client: Windows 10

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

WDI_PORT_ID

Article • 03/03/2023

The WDI_PORT_ID data type is a UINT16 value that defines a port ID.

C++

```cpp
typedef UINT16 WDI_PORT_ID;
```

Remarks

If you want to specify any port (wildcard), you can use the WDI_PORT_ANY (0xFFFF) value.

Requirements

Minimum supported client: Windows 10

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h

TAL_TXRX_HANDLE

Article • 12/15/2021

The TAL_TXRX_HANDLE data type is an NDIS_HANDLE value.

```c++
typedef NDIS_HANDLE TAL_TXRX_HANDLE, *PTAL_TXRX_HANDLE;
```

Requirements

Minimum supported client: Windows 10

Minimum supported server: Windows Server 2016

Header: Dot11wdi.h (include Ndis.h)

Features not carried over in WDI

Article • 03/14/2023

The following features are available in the previous Native WLAN driver model, but are not available in WDI.

- Soft AP
- IBSS
- Netmon

The following Soft AP required features are ported to Wi-Fi Direct.

- Added 802.11b.
- Moved all tethering requirements for Soft AP to Wi-Fi Direct.

WPA3-SAE authentication

Article • 12/15/2021

WPA3-SAE, also known as WPA3-Personal, is supported in Windows with WDI version 1.1.8 and later. Frame content generation and parsing for SAE (Secure Authentication of Equals) authentication is done within Windows, but the OS requires driver support for sending and receiving WPA3-SAE authentication frames.

WPA3-SAE capabilities

Miniport drivers indicate SAE support by doing the following:

1. Set SAE supported capability.
   The driver sets the **SAEAuthenticationSupported** capability in WDI_TLV_INTERFACE_ATTRIBUTES during the call to OID_WDI_GET_ADAPTER_CAPABILITIES.
2. Set MFP capability.
   The driver sets the **MFPCapable** capability in WDI_TLV_STATION_ATTRIBUTES during the call to OID_WDI_GET_ADAPTER_CAPABILITIES.
3. Add the **WDI_AUTH_ALGO_WPA3_SAE** auth method.
   The driver includes **WDI_AUTH_ALGO_WPA3_SAE** in the list of auth-cipher combinations returned in the call to OID_WDI_GET_ADAPTER_CAPABILITIES. This should be added in the following sections:

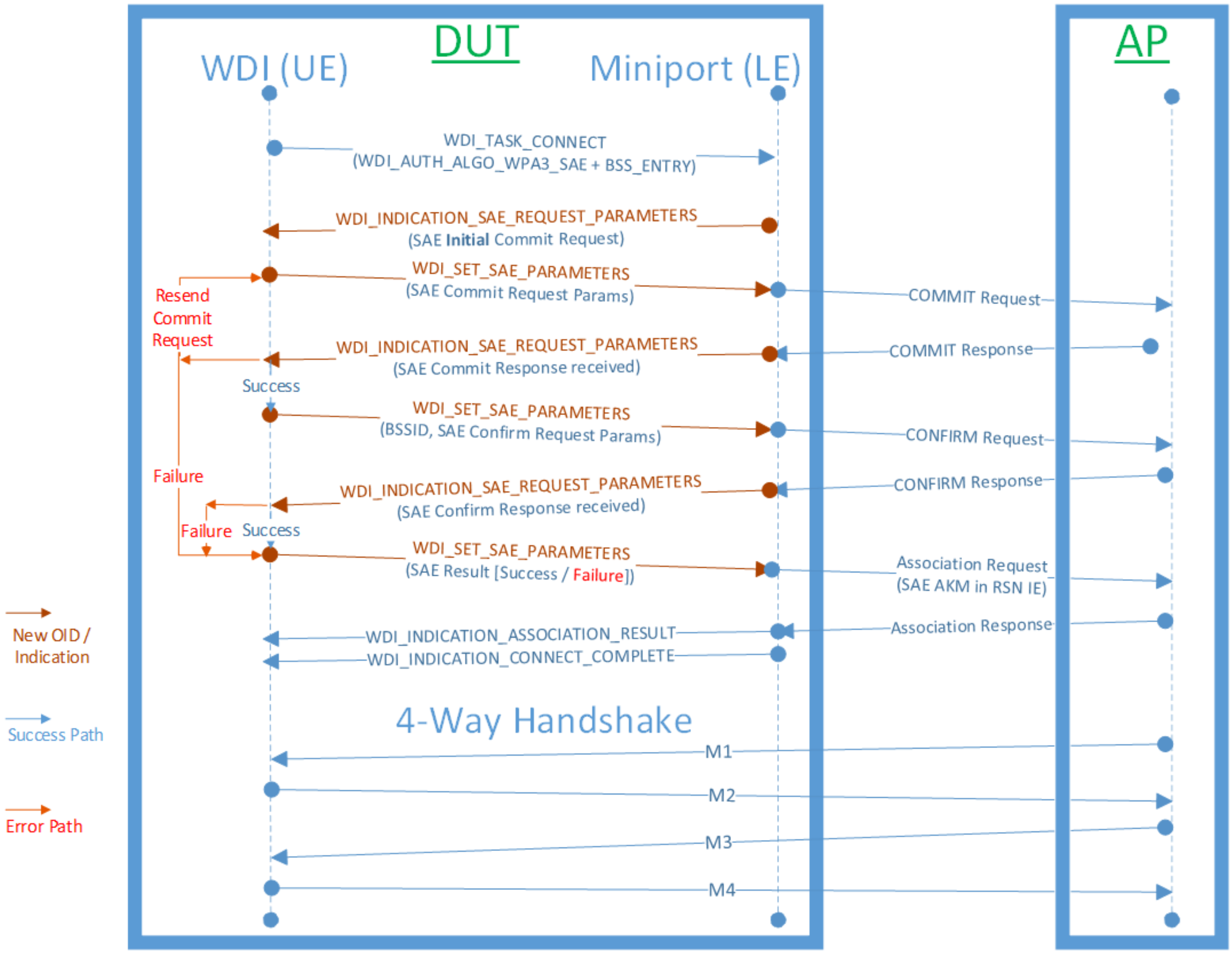
   - WDI_TLV_STATION_ATTRIBUTES : : WDI_TLV_UNICAST_ALGORITHM_LIST
   - WDI_TLV_STATION_ATTRIBUTES : : WDI_TLV_MULTICAST_DATA_ALGORITHM_LIST

WPA3-SAE authentication flow

Connection initiation

SAE connections are initiated with OID_WDI_TASK_CONNECT or OID_WDI_TASK_ROAM. WDI specifies **WDI_AUTH_ALGO_WPA3_SAE** as the auth method when the driver is required to do SAE authentication. If WDI provides the PMKID in the BSS list in the Connect/Roam task, then the driver skips SAE authentication and performs Open Authentication instead, followed by a reassociation request with the PMKID.

Authentication flow


DUT
WDI (UE)
Miniport (LE)
AP
WDI_TASK_CONNECT
(WDI_AUTH_ALGO_WPA3_SAE + BSS_ENTRY)
WDI_INDICATION_SAE_REQUEST_PARAMETERS
(SAE Initial Commit Request)
Resend
Commit
Request
WDI_SET_SAE_PARAMETERS
(SAE Commit Request Params)
COMMIT Request
WDI_INDICATION_SAE_REQUEST_PARAMETERS
(SAE Commit Response received)
COMMIT Response
Success
WDI_SET_SAE_PARAMETERS
(BSSID, SAE Confirm Request Params)
CONFIRM Request
Failure
WDI_INDICATION_SAE_REQUEST_PARAMETERS
(SAE Confirm Response received)
CONFIRM Response
Failure
Success
WDI_SET_SAE_PARAMETERS
(SAE Result [Success / Failure])
Association Request
(SAE AKM in RSN IE)
New OID /
Indication
WDI_INDICATION_ASSOCIATION_RESULT
WDI_INDICATION_CONNECT_COMPLETE
Association Response
Success Path
4-Way Handshake
M1
M2
M3
M4
Error Path

Initial request for SAE parameters

The driver first selects a BSS to which to connect or roam and, if WDI did not provide the PMKID for that BSS, the driver requests Commit parameters from WDI with NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED. In this initial indication, the driver sets the indication type to **WDI_SAE_INDICATION_TYPE_COMMIT_REQUEST_PARAMS_NEEDED**. In response, WDI sends OID_WDI_SET_SAE_AUTH_PARAMS to the driver with one of the following options.

- Send Commit request (**WDI_SAE_REQUEST_TYPE_COMMIT_REQUEST**)
- Fail SAE authentication (**WDI_SAE_REQUEST_TYPE_FAILURE**)

Upon receiving a Commit response

On receiving a Commit response, the driver sends NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED with the type set to **WDI_SAE_INDICATION_TYPE_COMMIT_RESPONSE**. In response, WDI sends OID_WDI_SET_SAE_AUTH_PARAMS with one of the following requests:

- Send Commit request (**WDI_SAE_REQUEST_TYPE_COMMIT_REQUEST**)
- Send Confirm request (**WDI_SAE_REQUEST_TYPE_CONFIRM_REQUEST**)

- Fail SAE authentication (**WDI_SAE_REQUEST_TYPE_FAILURE**)

Upon receiving a Confirm response

On receiving a Confirm response, the driver sends NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED with the type set to **WDI_SAE_INDICATION_TYPE_CONFIRM_RESPONSE**. WDI then sends OID_WDI_SET_SAE_AUTH_PARAMS with the SAE status field set to success or failure. If SAE authentication fails in the driver due to timeouts or other reasons, the driver sends an NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED indication with the type se to **WDI_SAE_INDICATION_TYPE_ERROR** and the failure reason specified in WDI_TLV_SAE_STATUS.

Timeouts and retransmissions

These are handled by the driver.

WPA3-SAE association

The device connects to an SAE network using one of the following options.

(Re)Association following SAE exchange

This is normally the first association attempt to an SAE network. The driver sets the SAE AKM in the RSN IE in the Association Request frame.

(Re)Association using PMKID

If WDI provided a PMKID for the BSS entry in the connect/roam task, then the driver does the following:

1. The driver performs an Open authentication followed by inclusion of the PMKID in the (Re)association request.
2. If the device does not receive a response from the AP within a short time, or if the AP returns an association error in the response, the driver skips SE authentication with this AP and either moves to another AP, or falls back to doing full SAE authentication with this AP.

SAE connection completes once the SAE authentication/association is complete. As before, the driver sends the following indications on conclusion of the connect or roam task:

- NDIS_STATUS_WDI_INDICATION_ASSOCIATION_RESULT
- NDIS_STATUS_WDI_INDICATION_CONNECT_COMPLETE

Error handling

Resending the SAE Commit request frame

If the driver needs to resend a Commit frame due to a timeout, it can either resend the original Scalar/Element values that were provided by WDI, or request a new set of Scalar/Element values from WDI with an NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED indication.

Resending the SAE Confirm response frame

If the driver needs to resend a Confirm frame due to a timeout, it should request a new set of **SendConfirm** and **Confirm** values from WDI with an NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED indication, setting the type to **WDI_SAE_INDICATION_TYPE_CONFIRM_REQUEST_RESEND_REQUEST**.

WDI doc change history

Article • 03/14/2023

Windows 10, version 2004

Documentation updated to WDI version 1.1.9.

Topic	Description
WDI message structure	Modified TLV structure and aggregate container to allow for variable-size KCK/KEK.
OID_WDI_TASK_REQUEST_FTM	**ScanTrigger** enum value added. Description updated for LE with BSS list cache.
WDI_AUTH_ALGORITHM	Added new WDI_AUTH_ALGORITHM **WDI_AUTH_ALGO_OWE**.
WDI_CIPHER_ALGORITHM	**WDI_CIPHER_ALGO_GCMP_256** new cipher added. **WDI_CIPHER_ALGO_BIP_GMAC_256** new cipher added.
WDI_TLV_CONFIGURED_CIPHER_KEY	Added entries for WDI_TLV_CIPHER_KEY_GCMP_256_KEY and WDI_TLV_CIPHER_KEY_BIP_GMAC_256_KEY.
WDI_TLV_CIPHER_KEY_BIP_GMAC_256_KEY	Newly added TLV type.
WDI_TLV_CIPHER_KEY_GCMP_256_KEY	Newly added TLV type.
WDI_TLV_CONNECT_PARAMETERS	Added reference for new TLV type WDI_TLV_OWE_DH_IE.
WDI_TLV_FTM_RESPONSE	**BandwidthUsed** field added. **PropegationProperty** field added. **RTT** field changed to signed integer.
WDI_TLV_KCK_CONTENT	Newly added TLV type.
WDI_TLV_KEK_CONTENT	Newly added TLV type.
WDI_TLV_OWE_DH_IE	Newly added TLV type.
WDI_TLV_PROTOCOL_OFFLOAD	Newly added TLV type.
WDI_TLV_REPLAY_COUNTER	Newly added TLV type.
WDI_TLV_STATION_CAPABILITIES	**Host-WPA3-FIPS** mode added.

Windows 10, version 1903

Documentation updated to WDI version 1.1.8.

Topic	Description
WDI_TLV_STATION_CAPABILITIES	Added support for the driver to indicate support for Fine Timing Measurement (FTM).
OID_WDI_TASK_REQUEST_FTM	Newly added task OID that enables WDI to request that the adapter initiate FTM procedures to obtain roundtrip time (RTT) and the Location Configuration Information (LCI) report from BSS targets.
WDI_TLV_FTM_REQUEST_TIMEOUT	Newly added TLV for FTM request.
WDI_TLV_FTM_TARGET_BSS_ENTRY	Newly added TLV for FTM request.

Topic	Description
WDI_TLV_REQUEST_LCI_REPORT	Newly added TLV for FTM request.
NDIS_STATUS_WDI_INDICATION_REQUEST_FTM_COMPLETE	Newly added status indication sent by the host as a task completion indication for OID_WDI_TASK_REQUEST_FTM. Contains a list of FTM responses from BSS targets.
WDI_TLV_FTM_RESPONSE	Newly added TLV for FTM response.
WDI_TLV_FTM_RESPONSE_STATUS	Newly added TLV for FTM response.
WDI_TLV_RETRY_AFTER	Newly added TLV for FTM response.
WDI_TLV_FTM_NUMBER_OF_MEASUREMENTS	Newly added TLV for FTM response.
WDI_TLV_RTT	Newly added TLV for FTM response.
WDI_TLV_RTT_ACCURACY	Newly added TLV for FTM response.
WDI_TLV_RTT_VARIANCE	Newly added TLV for FTM response.
WDI_TLV_LCI_REPORT_STATUS	Newly added TLV for FTM response.
WDI_TLV_LCI_REPORT_BODY	Newly added TLV for FTM response.
WDI_TLV_INTERFACE_CAPABILITIES	Added new capabilities for the driver to indicate support for Multiband Operation (MBO) and beacon report offloading.
WDI_ASSOC_STATUS	Added **WDI_ASSOC_STATUS_ASSOCIATION_DISALLOWED** status.
WPA3-SAE authentication	New overview of WPA3-SAE (Secure Authentication of Equals) authentication.
WDI_TLV_INTERFACE_CAPABILITIES	Added new capability for the driver to indicate support for SAE authentication.
WDI_AUTH_ALGORITHM	Added definition for **WDI_AUTH_ALGO_WPA3_SAE**.
NDIS_STATUS_WDI_INDICATION_SAE_AUTH_PARAMS_NEEDED	Newly added status indication sent by the driver to request SAE authentication parameters from WDI.
WDI_TLV_SAE_INDICATION_TYPE	Newly added TLV for SAE authentication parameters requests.
WDI_TLV_SAE_COMMIT_RESPONSE	Newly added TLV for SAE authentication parameters requests.
WDI_TLV_SAE_CONFIRM_RESPONSE	Newly added TLV for SAE authentication parameters requests.
WDI_TLV_SAE_STATUS	Newly added TLV for SAE authentication parameters requests and for setting SAE authentication parameters.
OID_WDI_SET_SAE_AUTH_PARAMS	Newly added property OID that contains the parameters required to send the SAE Commit or Confirm request, or an error message indicating a failure to perform SAE with the BSSID.
WDI_TLV_SAE_REQUEST_TYPE	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_COMMIT_REQUEST	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_FINITE_CYCLIC_GROUP	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_SCALAR	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_ELEMENT	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_ANTI_CLOGGING_TOKEN	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_CONFIRM_REQUEST	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_SEND_CONFIRM	Newly added TLV for setting SAE authentication parameters.
WDI_TLV_SAE_CONFIRM	Newly added TLV for setting SAE authentication parameters.

Topic	Description
OID_WDI_TASK_P2P_SEND_REQUEST_ACTION_FRAME	Added additional validation of P2P IEs on outgoing action frames.
OID_WDI_TASK_P2P_SEND_RESPONSE_ACTION_FRAME	Added additional validation of P2P IEs on outgoing action frames.

Windows 10, version 1809

Documentation updated to WDI version 1.1.7.

Topic	Description
WDI_PHY_TYPE	Added support for 802.11ax PHY.
WDI_CONNECTION_QUALITY_HINT	Changed the name of the **WDI_CONNECTION_QUALITY_HIGH_CHANNEL_AVAILABILITY** value to **WDI_CONNECTION_QUALITY_HIGH_THROUGHPUT**. No change to the description of this value.
NDIS_STATUS_WDI_INDICATION_DEVICE_SERVICE_EVENT	Added support for unsolicited device service notifications.

Windows 10, version 1803

Documentation updated to WDI version 1.1.6.

Topic	Description
WDI_TLV_OS_POWER_MANAGEMENT_FEATURES	Added this TLV to OID_WDI_GET_ADAPTER_CAPABILITIES to indicate which OS power management (PM) features that the driver supports.
WDI_TLV_PM_PROTOCOL_OFFLOAD_80211RSN_REKEY	Updated this TLV to specify that drivers must now return GTK/iGTK key info, if configured, when queried in OID_WDI_GET_PM_PROTOCOL_OFFLOAD.
NDIS_STATUS_WDI_INDICATION_CIPHER_KEY_UPDATED	Added this indication for drivers to provide notifications of GTK/iGTK key updates when the keys are updated, while the driver is not in the Offload state.
MINIPORT_WDI_TX_SUSPECT_FRAME_LIST_ABORT	Updated *TxSuspectFrameListAbortHandle* to *TxSuspectFrameListAbort*.

Windows 10, version 1709

Documentation updated to WDI version 1.1.5.

Topic	Description
WDI_TLV_TCP_OFFLOAD_CAPABILITIES	Added new **WDI_TLV_OFFLOAD_SCOPE** parameter to indicate whether offloads specified apply to the STA port only or to all ports.
NDIS_STATUS_WDI_INDICATION_SEND_AP_ASSOCIATION_RESPONSE_COMPLETE	Changed the **WDI_TLV_PHY_TYPE_LIST** parameter to make it required.
User-initiated feedback with IHV trace logging	Added a new section describing how to add IHV logging to user-initiated feedback scenarios.

Windows 10, version 1607

Documentation updated to WDI version 1.0.21.

Topic	Description

Topic	Description
OID_WDI_TASK_P2P_DISCOVER	Added new task parameters: - WDI_TLV_P2P_ASP2_SERVICE_INFORMATION_DISCOVERY_ENTRY - WDI_TLV_P2P_INCLUDE_LISTEN_CHANNEL
OID_WDI_GET_ADAPTER_CAPABILITIES	Added new get property result: WDI_TLV_SUPPORTED_GUIDS
WDI_CIPHER_ALGORITHM	Added new value: WDI_CIPHER_ALGO_GCMP
WDI_PHY_TYPE	Added new value: WDI_PHY_TYPE_DMG
WDI_P2P_SERVICE_DISCOVERY_TYPE	Added new values: - WDI_P2P_SERVICE_DISCOVERY_TYPE_ASP2_SERVICE_NAME_ONLY - WDI_P2P_SERVICE_DISCOVERY_TYPE_ASP2_SERVICE_INFORMATION
WDI_TLV_P2P_ASP2_ADVERTISED_SERVICE_ENTRY WDI_TLV_P2P_ASP2_SERVICE_INFORMATION_DISCOVERY_ENTRY WDI_TLV_P2P_INCLUDE_LISTEN_CHANNEL WDI_TLV_P2P_INSTANCE_NAME WDI_TLV_P2P_INSTANCE_NAME_HASH WDI_TLV_P2P_SERVICE_TYPE WDI_TLV_P2P_SERVICE_TYPE_HASH WDI_TLV_SUPPORTED_GUIDS	Newly added TLVs.
WDI_TLV_P2P_ADVERTISED_SERVICES	Added contained TLV: WDI_TLV_P2P_ASP2_ADVERTISED_SERVICE_ENTRY
WDI_TLV_INTERFACE_CAPABILITIES	Added a new value that specifies if the device supports IP docking capability.
WDI_TLV_P2P_CAPABILITIES	Added a new value that specifies if ASP2 Service Names Discovery is supported. Added a new value that specifies if ASP2 Service Information Discovery is supported.

March 2016

Topic	Description
MINIPORT_WDI_TX_TARGET_DESC_DEINIT	Added note that the IHV miniport is not permitted to make any indication in the context of this call.
MINIPORT_WDI_TX_TARGET_DESC_INIT	Added note that the IHV miniport is not permitted to make any indication in the context of this call.

Windows 10, version 1511

Documentation updated to WDI version 1.0.10.

Topic	Description
OID_WDI_TASK_START_AP	Added a new task parameter: WDI_TLV_AP_BAND_CHANNEL.
OID_WDI_SET_ADAPTER_CONFIGURATION	Added a new task parameter: WDI_TLV_PLDR_SUPPORT.

Topic	Description
WDI_TLV_AP_BAND_CHANNEL	Newly added TLV type.
WDI_TLV_P2P_CAPABILITIES	Added a new value that specifies whether the adapter supports operating a GO on the 5GHz band.
WDI_TLV_PLDR_SUPPORT	Newly added TLV type.
WDI_TLV_START_AP_PARAMETERS	Added a new value that specifies whether to allow legacy SoftAP clients to connect. Added a new value that specifies whether the AP can only be started on the channels specified in OID_WDI_TASK_START_AP task parameters with **WDI_TLV_AP_BAND_CHANNEL**.

Windows 10

Initial version.

Native 802.11 IHV Extensions Topics

Article • 12/15/2021

This section describes how an independent hardware vendor (IHV) can add functionality to the Native 802.11 framework. For more information about the Native 802.11 framework, see Native 802.11 Software Architecture.

This section includes the following topics:

Overview of IHV Extensibility

Article • 12/15/2021

The Native 802.11 framework provides support for an independent hardware vendor (IHV) to add functionality to the Native 802.11 framework.

For example, the IHV can provide support for any of the following:

- Proprietary or non-standard authentication algorithms for port-based network access. For more information, see Extending Support for 802.11 Authentication Algorithms.

- Proprietary or non-standard cipher algorithms for data encryption. For more information, see Extending Support for 802.11 Cipher Algorithms.

- Proprietary PHY configurations. For more information, see Extending Support for 802.11 PHY Configurations.

In order to extend the Native 802.11 functionality, the IHV must provide the following components:

- A Native 802.11 miniport driver that supports the Extensible Station (ExtSTA) operation mode. For more information about this mode, see Extensible Station Operation Mode. For more information about ways the ExtSTA operation mode can extend Native 802.11 functionality, see Extending Native 802.11 Functionality.

- An IHV Extensions DLL, which processes the security packets exchanged through the proprietary authentication algorithms that the IHV supports. The IHV Extensions DLL is also responsible for cipher key derivation through these authentication algorithms, as well as the validation of user data that pertains to the security extensions supported by the IHV.

  For more information about the IHV Extensions DLL, see Native 802.11 IHV Extensions DLL.

- An IHV User Interface (UI) Extensions DLL, which extends the Native 802.11 user interface to configure connectivity and security settings that are validated and processed by the IHV Extensions DLL.

  For more information about the IHV UI Extensions DLL, see Native 802.11 IHV UI Extensions DLL.

For more information about the modules provided by the IHV, see Native 802.11 Software Architecture.

To provide a secure execution environment, the IHV should do the following:

1. Do not log any sensitive information, such as encryption keys, in event or debug logs.

2. Use CryptProtectMemory to protect sensitive encryption keys stored in memory, and SecureZeroMemory to clear memory when done with the keys.

3. Treat the IHV extension portions of the network profile as untrusted data that may have been manipulated by an attacker. IHV extension portions of profiles are opaque to the 802.11 Auto Configuration Module (ACM) and Media Specific Module (MSM) and will not be validated. (See Native 802.11 Software Architecture for descriptions of these modules and configuration control paths.) This IHV extension data should be appropriately parsed to prevent any buffer overflows or attacks that could lead to a local escalation of privileges.

Installing Native 802.11 IHV Extensions

Article • 06/17/2022

To install the Native 802.11 IHV Extensions DLL and Native 802.11 IHV UI Extensions DLL, the independent hardware vendor (IHV) must make the following changes to the DDInstall section within the INF file that is used for the installation of the IHV's wireless LAN (WLAN) adapter.

- Add a CopyFiles directive, with an associated *file-list-section*, to the INF file. The name of each DLL developed by the IHV must be within the *file-list-section*.

  For example, if the IHV is installing IhvExt.dll and IhvUIExt.dll, the INF file would have the following CopyFiles directive and *file-list-section*:

  ```
  CopyFiles = Sample-File-List-Section

  [Sample-File-List-Section]
  IhvExt.dll,,,2
  IhvUIExt.dll,,,2
  ```

  For more information about the CopyFiles directive, see **INF CopyFiles Directive**.

- Make sure that a DestinationDirs section declares the destination of the *file-list-section* used in the CopyFiles directive.

  In the previous example, the DestinationDirs section would have the following values:

  ```
  [DestinationDirs]
  DefaultDestDir = 13
  Sample-File-List-Section = 13
  ```

  For more information about the DestinationDirs section, see **INF DestinationDirs Section**.

- Make sure that an AddReg directive, with an associated *add-registry-section*, is added to the INF file for each WLAN adapter. For more information about the AddReg directive, see **INF AddReg Directive**.

Within the *add-registry-section*, the following keys must be declared.

HKR, Ndi\IHVExtensions, ExtensibilityDLL, 0, *destination-file-name*
This key identifies the name of the IHV Extensions DLL. For example, to associate the IhvExt.dll with the management of the WLAN adapter, the following key would be declared.

```
HKR,Ndi\IHVExtensions, ExtensibilityDLL, 0, %13%\IhvExt.dll
```

HKR, Ndi\IHVExtensions, UIExtensibilityCLSID, 0, *CLSID*
This key identifies the COM class identifier (CLSID), which was registered on the target system for the IHV UI Extensions DLL. The *CLSID* value must be enclosed in quotation marks. This key associates the IHV UI Extensions DLL with the IHV Extensions DLL installed through the **ExtensibilityDLL** key.

Note The **UIExtensibilityCLSID** key is required only if the IHV installs an IHV UI Extensions DLL.

HKR, Ndi\IHVExtensions, AdapterOUI, 0x00010001, *OUI*
This key identifies the IEEE-assigned organizationally unique identifier (OUI), which identifies the IHV. The OUI value must be declared as a 24-bit hexadecimal value.

The AdapterOUI key is used to verify that the OUI of the WLAN adapter matches the value of the **OUI** attribute of the **IHV** XML element. For more information about the **IHV** element and the Native 802.11 XML schema, refer to the Microsoft Windows SDK documentation.

For more information about the INF file and its sections, see Creating an INF File.

Native 802.11 IHV Extensions DLL Topics

Article • 12/15/2021

This section discusses the IHV Extensions DLL and has the following topics:

Native 802.11 IHV Extensions DLL Overview

Native 802.11 IHV Extensibility functions

Native 802.11 IHV Handler functions

Native 802.11 IHV Extensions DLL Implementation Guidelines

DLL Start/Stop Operations

802.11 WLAN Adapter Management

Network Profile Management

Interaction with the User

Pre-Association Operations

Post-Association Operations

Send and Receive Operations

Notification Operations

Virtual Station

Native 802.11 IHV Extensions DLL Overview

Article • 12/15/2021

Through an IHV Extensions DLL, the independent hardware vendor (IHV) can support the following:

- Proprietary or non-standard authentication algorithms. Through this support, the IHV Extensions DLL sends and receives all security packets related to the authentication algorithm.

  The IHV Extensions DLL can also support standard authentication algorithms for network configurations that are not supported by the operating system. For example, the DLL can support the Wi-Fi Protected Access with preshared keys (WPA-PSK) authentication algorithm over independent basic service set (IBSS) networks, which is a configuration not supported by Windows Vista.

- Proprietary or non-standard cipher algorithms. Through this support, the IHV Extensions DLL is responsible for deriving the cipher key and downloading the keys to the Native 802.11 miniport driver.

  The IHV Extensions DLL can also support standard cipher algorithms for network configurations that are not supported by the operating system. For example, the DLL can support the Temporal Key Integrity Protocol (TKIP) over IBSS networks, which is a configuration not supported by Windows Vista.

- Verification of proprietary extensions to a network profile. For example, the IHV Extensions DLL is responsible for the validation of user settings for IHV-defined security options.

- Configuration of the Native 802.11 miniport driver. For example, prior to starting a connection operation with the miniport driver, the operating system will call the *Dot11ExtIhvPerformPreAssociate* function so that the IHV Extensions DLL can configure the driver with proprietary extensions related to the connection to a BSS network.

- Interface to the IHV UI Extensions DLL. Through this interface, the IHV Extensions DLL can request user input or notification. For more information about the IHV UI Extensions DLL, see Native 802.11 IHV UI Extensions DLL.

The Native 802.11 IHV Extensibility Host process loads the IHV Extensions DLL into its process space upon the first arrival and detection of a wireless LAN (WLAN) adapter for which the DLL was installed. For more information about the Native 802.11 IHV Extensibility Host process and Native 802.11 framework, see Native 802.11 Software Architecture.

The Native 802.11 IHV Extensibility Host process provides an API through its IHV Extensibility functions. Through this API, the IHV Extensions DLL can interface the Native 802.11 miniport driver or IHV UI Extensions DLL. For more information about the IHV Extensibility functions, see Native 802.11 IHV Extensibility Functions.

Similarly, the IHV Extensions DLL provides an API through its IHV Handler functions. The Native 802.11 IHV Extensibility Host process uses this API for various operations, such as initiating pre- or post-association operations. For more information about the IHV Handler functions, see Native 802.11 IHV Handler Functions.

Native 802.11 IHV Extensibility functions

Article • 12/15/2021

ⓘ **Important**

The **Native 802.11 Wireless LAN** interface is deprecated in Windows 10 and later. Please use the WLAN Device Driver Interface (WDI) instead. For more information about WDI, see **WLAN Universal Windows driver model**.

The Native 802.11 IHV Extensibility functions are provided by the operating system and are called by the IHV Extensions DLL to do the following:

- Allocate and free buffers that are used within the Native 802.11 framework.
- Send packets, such as a packet defined by an authentication algorithm, through the IHV's wireless LAN (WLAN) adapter.
- Configure the IHV's WLAN adapter with various security settings for any authentication and cipher algorithms supported by the IHV Extensions DLL.
- Interface with the IHV UI Extensions DLL (if installed) to process event notifications. For example, the IHV Extensions DLL could notify the IHV UI Extensions DLL about the various stages involved in a basic service set (BSS) network connection.

For more information about the IHV UI Extensions DLL, see Native 802.11 IHV UI Extensions DLL.

ⓘ **Note**

The IHV Extensions DLL calls each Native 802.11 IHV Extensibility function through a function pointer associated with a member of the **DOT11EXT_APIS** structure. When the operating system calls the **Dot11ExtIhvInitService** IHV Handler function, it passes the list of pointers to the IHV Extensibility functions through the *pDot11ExtAPI* parameter.

The following table lists the Native 802.11 IHV Extensibility Functions that can be called by the IHV Extensions DLL. Each IHV Extensibility function can only be called under these conditions.

- **Called After Service Initialization**
  The IHV Extensibility function can only be called after the Dot11ExtIhvInitService IHV Handler function has been called to initialize the IHV Extensions DLL. Also, the

Extensions DLL cannot call the IHV Extensibility function after the Dot11ExtIhvDeinitService IHV Handler function has been called.

- **Called after Adapter Initialization**
  The IHV Extensibility function can only be called after the Dot11ExtIhvInitAdapter IHV Handler function has been called to initialize the interface to the IHV's WLAN adapter.
  The IHV Extensibility function requires a handle, which identifies the WLAN adapter. When *Dot11ExtIhvInitAdapter* is called, the IHV Extensions DLL is passed this handle through the *hDot11SvcHandle* parameter.
  The Extensions DLL cannot call the IHV Extensibility function after the Dot11ExtIhvDeinitAdapter IHV Handler function has been called.
- **Called after Pre-Association**
  The IHV Extensibility function can only be called after the Dot11ExtIhvPerformPreAssociate IHV Handler function has been called to initiate a pre-association operation with a basic service set (BSS) network.
  The IHV Extensibility function requires a handle, which identifies the BSS network connection. When *Dot11ExtIhvPerformPreAssociate* is called, the IHV Extensions DLL is passed this handle through the *hConnection* parameter.
  The Extensions DLL cannot call the IHV Extensibility function after the Dot11ExtIhvDeinitAdapter or Dot11ExtIhvAdapterReset IHV Handler functions have been called.
- **Called after Post-Association**
  The IHV Extensibility function can only be called after the Dot11ExtIhvPerformPostAssociate IHV Handler function has been called to initiate a post-association operation with a basic service set (BSS) network.
  The IHV Extensibility function requires a handle, which identifies the security session with the BSS network connection. When *Dot11ExtIhvPerformPostAssociate* is called, the IHV Extensions DLL is passed this handle through the *hSecuritySessionID* parameter.
  The Extensions DLL cannot call the IHV Extensibility function after the Dot11ExtIhvDeinitAdapter or Dot11ExtIhvAdapterReset IHV Handler functions have been called.

Function	Called after service initialization	Called after adapter initialization	Called after pre-association	Called after post-association
Dot11ExtAllocateBuffer	X			
Dot11ExtFreeBuffer	X			
Dot11ExtGetProfileCustomUserData			X	

Function	Called after service initialization	Called after adapter initialization	Called after pre-association	Called after post-association
Dot11ExtNicSpecificExtension		X		
Dot11ExtStartOneX				X
Dot11ExtStopOneX				X
Dot11ExtPostAssociateCompletion				X
Dot11ExtPreAssociateCompletion			X	
Dot11ExtProcessOneXPacket				X
Dot11ExtQueryVirtualStationProperties		X		
Dot11ExtReleaseVirtualStation		X		
Dot11ExtRequestVirtualStation		X		
Dot11ExtSendNotification		X		
Dot11ExtSendUIRequest		X		
Dot11ExtSetAuthAlgorithm		X		
Dot11ExtSetCurrentProfile			X	
Dot11ExtSetDefaultKey		X		
Dot11ExtSetDefaultKeyId		X		
Dot11ExtSetEtherTypeHandling		X		
Dot11ExtSetExcludeUnencrypted		X		
Dot11ExtSetKeyMappingKey		X		
Dot11ExtSetMulticastCipherAlgorithm		X		
Dot11ExtSetProfileCustomUserData		X		
Dot11ExtSetUnicastCipherAlgorithm		X		
Dot11ExtSetVirtualStationAPProperties			X	

For more information about IHV Handler functions, see Native 802.11 IHV Handler Functions.

Native 802.11 IHV Handler functions

Article • 12/15/2021

ⓘ **Important**

The **Native 802.11 Wireless LAN** interface is deprecated in Windows 10 and later. Please use the WLAN Device Driver Interface (WDI) instead. For more information about WDI, see **WLAN Universal Windows driver model**.

The Native 802.11 IHV Handler functions are provided by the IHV Extensions DLL and are called by the operating system to do the following:

- Allocate and free buffers that are used within the Native 802.11 framework.
- Send packets, such as a packet defined by an authentication algorithm, through the IHV's wireless LAN (WLAN) adapter.
- Receive packets based on a specified list of IEEE EtherType values and privacy exemption rules.
- Configure the IHV's WLAN adapter with various security settings for any proprietary authentication and cipher algorithms.
- Interface with the IHV UI Extensions DLL (if installed) to process event notifications. For example, the IHV Extensions DLL could notify the UI Extensions DLL about the various stages involved in a basic service set (BSS) network connection.

For more information about the IHV UI Extensions DLL, see Native 802.11 IHV UI Extensions DLL.

ⓘ **Note**

With the exception of **Dot11ExtIhvGetVersionInfo** and **Dot11ExtIhvInitService**, the operating system calls the IHV Handler functions through a function pointer associated with a member of the **DOT11EXT_IHV_HANDLERS** structure. When the operating system calls the *Dot11ExtIhvInitService* IHV Handler function, the IHV Extensions DLL returns the list of pointers to the IHV Handler functions through the *pDot11IHVHandlers* parameter.

This section describes the following Native 802.11 IHV Handler functions.

- Dot11ExtIhvAdapterReset
- Dot11ExtIhvControl
- Dot11ExtIhvCreateDiscoveryProfiles

- Dot11ExtIhvDeinitAdapter
- Dot11ExtIhvDeinitService
- Dot11ExtIhvGetVersionInfo
- Dot11ExtIhvInitAdapter
- Dot11ExtIhvInitService
- Dot11ExtIhvInitVirtualStation
- Dot11ExtIhvIsUIRequestPending
- Dot11ExtIhvOneXIndicateResult
- Dot11ExtIhvPerformCapabilityMatch
- Dot11ExtIhvPerformPostAssociate
- Dot11ExtIhvPerformPreAssociate
- Dot11ExtIhvProcessSessionChange
- Dot11ExtIhvProcessUIResponse
- Dot11ExtIhvQueryUIRequest
- Dot11ExtIhvReceiveIndication
- Dot11ExtIhvReceivePacket
- Dot11ExtIhvSendPacketCompletion
- Dot11ExtIhvStopPostAssociate
- Dot11ExtIhvValidateProfile

Native 802.11 IHV Extensions DLL Implementation Guidelines

Article • 12/06/2022

The IHV Extensions DLL is implemented as a run-time dynamic-link library (DLL). For more information about DLLs, see About Dynamic-Link Libraries.

Refer to the following guidelines when implementing an IHV Extensions DLL.

- The structures and function prototypes referenced by the IHV Extensions DLL are declared in Wlanihv.h.
- The IHV Extensions DLL must implement the *Dot11ExtIhvGetVersionInfo* and *Dot11ExtIhvInitService* functions. Also, these functions must be exported through the module-definition (.def) file used to build the DLL. The operating system resolves the address for these functions through the **GetProcAddress** function.
- The IHV Extensions DLL must implement all of the IHV Handler functions. The DLL returns a list of function pointers to these functions when the operating system calls the *Dot11ExtIhvInitService* function.

  For more information about the IHV Handler functions, see Native 802.11 IHV Handler Functions.
- For Windows Vista, the IHV Extensions DLL must support the interface version of zero. When *Dot11ExtIhvGetVersionInfo* is called, the DLL must define the minimum and maximum supported interface versions to be zero.

DLL Start/Stop Operations Topics

Article • 12/15/2021

This section discusses how the operating system starts and stops the IHV Extensions DLL, and includes the following topics:

DLL Start Operations

DLL Stop Operations

DLL Start Operations

Article • 12/15/2021

Immediately after loading the IHV Extensions DLL, the operating system calls the following IHV Handler functions in this sequence.

1. The operating system calls the *Dot11ExtIhvGetVersionInfo* IHV Handler function to determine the interface versions supported by the IHV Extensions DLL. This function is passed a pointer to a **DOT11_IHV_VERSION_INFO** structure, which the DLL formats with the minimum and maximum interface versions that it supports. **Note** For Windows Vista, the IHV Extensions DLL must set the **dwVerMin** and **dwVerMax** members of the DOT11_IHV_VERSION_INFO structure to zero.

2. If the IHV Extensions DLL supports an interface version that is supported by the operating system, the operating system calls the *Dot11ExtIhvInitService* IHV Handler function to initialize the DLL.

The IHV Extensions DLL must follow these guidelines when *Dot11ExtIhvInitService* is called.

- The *pDot11ExtAPI* parameter contains a pointer to a **DOT11EXT_APIS** structure, which is formatted with the addresses of the IHV Extensibility functions supported by the operating system. The IHV Extensions DLL must copy the DOT11EXT_APIS structure, which is referenced by the *pDot11ExtAPI* parameter, to a globally-declared DOT11EXT_APIS structure.

- The *pDot11IHVHandlers* parameter contains a pointer to a **DOT11EXT_IHV_HANDLERS** structure, which the IHV Extensions DLL formats with the addresses of the IHV Handler functions that it supports. **Note** The DLL must not set any of the members of the DOT11EXT_IHV_HANDLERS structure to **NULL**.

- The IHV Extensions DLL should perform any internal initialization and resource allocation in preparation for calls to its IHV Handler functions after the DLL returns from *Dot11ExtIhvInitService*.

For more information about the IHV Extensibility functions, see Native 802.11 IHV Extensibility Functions.

For more information about the IHV Handler functions, see Native 802.11 IHV Handler Functions.

DLL Stop Operations

Article • 12/06/2022

The operating system stops and unloads the IHV Extensions DLL whenever.

- The last wireless LAN (WLAN) adapter managed by the DLL is either removed or disabled.
- The host computer is reset or shut down.

The operating system follows this sequence when stopping and unloading the IHV Extensions DLL.

1. The operating system first calls the *Dot11ExtIhvDeinitAdapter* IHV Handler function for every WLAN adapter managed by the IHV Extensions DLL. For more information about this operation, see 802.11 WLAN Adapter Removal.

   After the call to *Dot11ExtIhvDeinitAdapter*, the IHV Extensions DLL must not call any IHV Extensions function related to adapter-specific operations, such as **Dot11ExtNicSpecificExtension**.

2. The operating system then calls the *Dot11ExtIhvDeinitService* IHV Handler function. When this function is called, the IHV Extensions DLL must free all allocated resources and prepare itself for unloading.

   After the call to *Dot11ExtIhvDeinitService*, the IHV Extensions DLL must not call any IHV Extensions function.

3. Finally, the operating system calls the *DllMain* function in the IHV Extensions DLL with the *fdwReason* parameter set to DLL_PROCESS_DETACH. For more information about *DllMain* and DLLs, see About Dynamic-Link Libraries.

For more information about the IHV Extensibility functions, see Native 802.11 IHV Extensibility Functions.

802.11 WLAN Adapter Management Topics

Article • 12/15/2021

This section discusses wireless LAN (WLAN) adapter management by the IHV Extensions DLL, and includes the following topics:

802.11 WLAN Adapter Arrival

802.11 WLAN Adapter Removal

802.11 WLAN Adapter Reset

802.11 WLAN Adapter Communication Channel

802.11 WLAN Adapter Arrival

Article • 12/15/2021

When the operating system detects a wireless LAN (WLAN) adapter for which an IHV Extensions DLL has been installed, the operating system calls the *Dot11ExtIhvInitAdapter* IHV Handler function. The operating system calls this function whenever a WLAN adapter becomes available and enabled for use, such as when a PCMCIA adapter is inserted.

When the *Dot11ExtIhvInitAdapter* function is called, the IHV Extensions DLL does the following:

- Allocates an array for the WLAN adapter context data, as well as any resources the DLL needs for the WLAN adapter.

- Registers a list of IEEE EtherTypes for the security packets received and consumed by the IHV Extensions DLL.

- Configures the adapter with any proprietary settings defined by the IHV.

The IHV Extensions DLL must follow these guidelines when *Dot11ExtIhvInitAdapter* is called.

- The *hDot11SvcHandle* parameter contains a unique handle value assigned by the operating system for the WLAN adapter. The IHV Extensions DLL must save this handle value and pass it to the *hDot11SvcHandle* parameter of the IHV Extensibility functions related to the adapter-specific processing, such as **Dot11ExtSetKeyMappingKey**.

  Typically, the DLL saves this handle value within a member of its WLAN adapter context array.

- The IHV Extensions DLL must return a unique handle value for the WLAN adapter through the *phIhvExtAdapter* parameter. The operating system passes the handle value to the *hIhvExtAdapter* parameter of the IHV Handler functions related to the adapter-specific processing, such as *Dot11ExtIhvReceiveIndication*.

  Typically, the DLL returns the address of the WLAN adapter context array as the handle value.

- The IHV Extensions DLL calls **Dot11ExtSetEtherTypeHandling** to register a list of the IEEE EtherTypes for the security packets that the DLL will receive. The IHV

Extensions DLL can also specify a list of EtherTypes that will be excluded from payload decryption. For more information about registering EtherTypes, see IEEE EtherType Handling.

After EtherTypes are registered, the operating system calls the *Dot11ExtIhvReceivePacket* IHV Handler function for every packet whose EtherType matches an entry in the list.

- The operating system configures the adapter with standard 802.11 parameters through set requests of the Native 802.11 object identifiers (OIDs). For more information about these OIDs, see Native 802.11 Wireless LAN OIDs.

  However, the DLL can configure the adapter with proprietary parameters through calls to the **Dot11ExtNicSpecificExtension** function. Through this function call, the DLL can communicate directly with the Native 802.11 miniport driver that manages the WLAN adapter and issue query or set requests to the driver based on a proprietary format defined by the IHV.

  For more information about the interface through which the DLL and WLAN adapter communicate, see 802.11 WLAN Adapter Communication Channel.

802.11 WLAN Adapter Removal

Article • 12/15/2021

When a wireless LAN (WLAN) adapter is removed or disabled, the operating system calls *Dot11ExtIhvDeinitAdapter* to notify the IHV Extensions DLL of the adapter's removal. The operating system also calls the *Dot11ExtIhvDeinitAdapter* function for every adapter managed by the IHV Extensions DLL before the operating system unloads the DLL.

When *Dot11ExtIhvDeinitAdapter* is called, the IHV Extensions DLL must follow these guidelines.

- The IHV Extensions DLL must free any allocated resources for the WLAN adapter. In particular, all memory allocated through calls to **Dot11ExtAllocateBuffer** must be freed through calls to **Dot11ExtFreeBuffer**.

- The handle used by the operating system to reference the WLAN adapter is no longer valid when *Dot11ExtIhvDeinitAdapter* is called. The operating system passes its handle to the IHV Extensions DLL through the *hDot11SvcHandle* parameter when *Dot11ExtIhvInitAdapter* is called.

  Within the call to the *Dot11ExtIhvDeinitAdapter* function and after returning from the call, the DLL must not use the handle value when calling any IHV Extensibility function that declares an *hDot11SvcHandle* parameter, such as **Dot11ExtSendPacket**.

- If the IHV Extensions DLL had a pending pre-association operation, which was initiated through a call to the *Dot11ExtIhvPerformPreAssociate* IHV Handler function, the operating system regards the operation as canceled through the call to the *Dot11ExtIhvDeinitAdapter* function. Within the call, the DLL must cancel the pre-association operation internally but must not call **Dot11ExtPreAssociateCompletion** to complete the pre-association operation.

  For more information about the pre-association operation, see Pre-Association Operations.

- If the IHV Extensions DLL had a pending post-association operation, which was initiated through a call to the *Dot11ExtIhvPerformPostAssociate* IHV Handler function, the operating system cancels the operation by calling the *Dot11ExtIhvStopPostAssociate* function before it calls *Dot11ExtIhvDeinitAdapter*.

For more information about the post-association operation, see Post-Association Operations.

- The operating system calls the *Dot11ExtIhvDeinitAdapter* function for every adapter managed by the IHV Extensions DLL before the operating system unloads the DLL. In this situation, the operating system calls the *Dot11ExtIhvDeinitService* IHV Handler function after the last WLAN adapter has been halted through a call to *Dot11ExtIhvDeinitAdapter*.

  For more information about this operation, see DLL Stop Operations.

802.11 WLAN Adapter Reset

Article • 12/15/2021

The operating system calls *Dot11ExtIhvAdapterReset* whenever it becomes necessary to restore the wireless LAN (WLAN) adapter to its initialized state. The operating system calls this function whenever one of the following events occurs.

- The WLAN adapter performs a disconnection operation. For more information about this operation, see Disconnection Operations.
- The operating system resets the Native 802.11 miniport driver, which manages the adapter, through a set request of OID_DOT11_RESET_REQUEST.

When *Dot11ExtIhvAdapterReset* is called, the IHV Extensions DLL must follow these guidelines.

- The IHV Extensions DLL must restore its state to the same state it was in after the *Dot11ExtIhvInitAdapter* function was called. If the DLL configured proprietary settings on the WLAN adapter, it must restore these settings to the same state they were in after *Dot11ExtIhvInitAdapter* was called.
- If the IHV Extensions DLL had a pending pre-association operation, which was initiated through a call to the *Dot11ExtIhvPerformPreAssociate* IHV Handler function, the DLL must call **Dot11ExtPreAssociateCompletion** to cancel the operation. In this situation, the DLL sets the *dwWin32Error* parameter of **Dot11ExtPreAssociateCompletion** to ERROR_CANCELLED.

  For more information about the pre-association operation, see Pre-Association Operations.
- If the DLL had a pending post-association operation, which was initiated through a call to the *Dot11ExtIhvPerformPostAssociate* IHV Handler function, the DLL must call **Dot11ExtPostAssociateCompletion** to cancel the operation. In this situation, the DLL sets the *dwWin32Error* parameter of **Dot11ExtPostAssociateCompletion** to ERROR_CANCELLED.

  For more information about the post-association operation, see Post-Association Operations.

802.11 WLAN Adapter Communication Channel

Article • 12/15/2021

The operating system provides a pass-through communication channel between the IHV Extensions DLL and the Native 802.11 miniport driver. The IHV Extensions DLL accesses the communication channel for the following operations.

Sending/Receiving Proprietary Configuration Data
The IHV Extensions DLL sends NDIS 6.0 or later object identifier (OID) method requests to the Native 802.11 miniport driver through calls to the **Dot11ExtNicSpecificExtension** function. Internally, this function issues a method request of OID_DOT11_NIC_SPECIFIC_EXTENSION to the miniport driver. For more information about NDIS OID method requests, see **NDIS_OID_REQUEST**.

Typically, the IHV Extensions DLL calls **Dot11ExtNicSpecificExtension** to do the following:

- Set proprietary configuration parameters for the miniport driver or WLAN adapter.
- Query proprietary configuration parameters or status data from the miniport driver or WLAN adapter.

Receiving Notifications/Indications
The IHV Extensions DLL asynchronously receives notifications from the Native 802.11 miniport driver through calls to the *Dot11ExtIhvReceiveIndication* IHV Handler function. The operating system calls this function whenever the miniport driver makes a media-specific indication through a call to **NdisMIndicateStatusEx**. For more information about this type of indication, see **NDIS_STATUS_MEDIA_SPECIFIC_INDICATION**.

Sending 802.11 Packets
The IHV Extensions DLL sends 802.11 packets to the Native 802.11 miniport driver through calls to the **Dot11ExtSendPacket** function. The miniport driver queues the packet on the WLAN adapter for transmission. When the packet has been transmitted, the operating system calls the *Dot11ExtIhvSendPacketCompletion* IHV Handler function. For more information about sending packets by the IHV Extensions DLL, see Send Operations.

Typically, the IHV Extensions DLL calls **Dot11ExtSendPacket** to send security packets during the post-association operation. The security packets are based on the authentication algorithm supported by the DLL and enabled on the WLAN adapter.

Receiving 802.11 Packets

The IHV Extensions DLL receives 802.11 packets from the Native 802.11 miniport driver through calls to the *Dot11ExtIhvReceivePacket* function. The operating system calls this function for every received packet that has an IEEE EtherType that matches an entry in the list of EtherTypes registered by the DLL through a call to **Dot11ExtSetEtherTypeHandling**. For more information about receiving packets by the IHV Extensions DLL, see Receive Operations.

The following points apply to the communication channel between the IHV Extensions DLL and the Native 802.11 miniport driver.

- Configuration, notification, or indication data transferred over this channel has a format defined by the independent hardware vendor (IHV), which is opaque to the operating system.
- All data received through this channel is serialized and delivered in the order the data was sent by the IHV Extensions DLL or Native 802.11 miniport driver.

Network Profile Management

Article • 12/06/2022

This section discusses the management and processing of network profiles by the IHV Extensions DLL. Network profiles define the attributes for the connection operation to a basic service (BSS) network.

The IHV Extensions DLL is responsible for verifying or creating proprietary extensions to a network profile. These extensions are XML data fragments, with each fragment declared within an **IHV** element of the Native 802.11 XML schema. The data within the <IHV> and </IHV> tags of the **IHV** element is in a format defined by the IHV.

This section includes the following topics:

Network Profile Overview

Creating Network Profile Extensions

Validating Network Profile Extensions

Network Profile Overview

Article • 12/06/2022

A network profile defines the attributes for a connection to a basic service set (BSS) network. Network profiles consist of XML data fragments. For Windows Vista, a network profile contains the following XML fragments.

Profile Name (required)
The name of the network profile, which is the service set identifier (SSID) of the BSS network.

Standard 802.11 Connectivity Settings (required)
This XML fragment consists of standard 802.11 settings for network connectivity, such as the BSS network type (infrastructure or independent) or type of wireless LAN (WLAN) security. The operating system processes the standard connectivity settings and configures the wireless WLAN adapter with them.

IHV Connectivity Extensions (optional)
This XML fragment consists of the extensions to network connectivity as defined by the IHV. The operating system passes the connectivity extensions to the IHV Extensions DLL for processing. The DLL is responsible for configuring the WLAN adapter with the proprietary extensions.

Standard 802.11 Security Settings (optional)
This XML fragment consists of the standard 802.11 authentication and cipher settings, such as the type of authentication and cipher algorithm to use on the BSS network connection. The operating system processes the standard security settings and configures the WLAN adapter with them.

IHV Security Extensions (optional)
This XML fragment consists of the extensions to network security as defined by the IHV. The IHV extensions can specify either of the following:

- Standard security settings.

  For the WLAN adapter that is managed by the IHV Extensions DLL, the DLL is responsible for the security algorithms, such as the Robust Security Network Association (RSNA) authentication algorithm or the AES-CCMP cipher algorithm. The operating system is no longer responsible. In this situation, the IHV Extensions DLL can either process the algorithms or provide proprietary methods for offloading the processing to the WLAN adapter.

- Proprietary security settings.

  The IHV Extensions DLL can provide support for security algorithms not supported by the operating system, such as non-standard or proprietary algorithms. The DLL is responsible for processing the algorithms or provide proprietary methods for offloading the processing to the WLAN adapter.

Creating Network Profile Extensions

Article • 12/15/2021

After the underlying wireless LAN (WLAN) adapter completes a scan operation, it returns a list of the detected basic service set (BSS) network to the operating system. The operating system calls the *Dot11ExtIhvCreateDiscoveryProfiles* function for every BSS network for which the user has not created a network profile. When this function is called, the IHV Extensions DLL can return temporary connectivity and security profile fragments that could be used to connect to the BSS network.

For more information about the scan operation, see Native 802.11 Scan Operations.

When *Dot11ExtIhvCreateDiscoveryProfiles* is called, the IHV Extensions DLL must follow these guidelines.

- The operating system passes to the *pConnectableBssid* parameter a list of IEEE 802.11 Beacon and Probe Response frames received during the last scan operation. This list is formatted as a DOT11_BSS_ENTRY structure. Each Beacon or Probe response within the list was sent by an access point (AP) with the same service set identifier (SSID).

  Note For Windows Vista, the IHV Extensions DLL supports only infrastructure basic service set (BSS) networks.

  The IHV Extensions DLL must parse each of the fixed-length fields and variable-length information elements (IEs) in order to create the appropriate profile fragments.

- The connectivity and security profile fragment must contain valid settings that can be used to connect to each of the APs, whose BSS identifiers (BSSIDs) are referenced through the *pConnectableBssid* parameter.

- Each connectivity and security profile fragment contains the XML data for the profile extensions defined by the IHV. The XML data within the profile fragment must be delimited by <IHV> and </IHV> tags.

Validating Network Profile Extensions

Article • 12/15/2021

The operating system calls IHV Handler functions to validate IHV-defined connectivity and security settings under the following conditions.

- The user creates a new network profile that contains settings for the IHV-defined connectivity and/or security profile extensions. In this situation, the operating system calls the *Dot11ExtIhvValidateProfile* IHV Handler function to validate the user settings.

- The WLAN adapter completes a scan operation and returns its results to the operating system. The operating system calls the *Dot11ExtIhvPerformCapabilityMatch* IHV Handler function to determine whether a detected basic service set (BSS) network matches the IHV-defined connectivity and security settings from a network profile.

  The operating system passes a list of the 802.11 Beacon and Probe Response frames from the BSS network to the *pConnectableBssid* parameter of the **Dot1ExtIhvPerformCapabilityMatch** function. The operating system also passes the connectivity and security profile extensions to the *pIhvConnProfile* and *pIhvSecProfile* parameters, respectively.

  If all of the entries in the list of 802.11 Beacon and Probe Response frames advertise the connectivity and security attributes defined in the profile fragments, the *Dot11ExtIhvPerformCapabilityMatch* function returns ERROR_SUCCESS.

- The operating system initiates a pre-association operation by calling the *Dot11ExtIhvPerformPreAssociate* function. In this situation, the IHV Extensions DLL must verify that the connectivity and security settings are valid. If the settings are valid, the function returns ERROR_SUCCESS and the DLL proceeds with the pre-association operation. Otherwise, the function returns an appropriate error code as defined in Winerror.h.

  For more information about the pre-association operation, see Pre-Association Operations.

For more information about the IHV Handler functions, see Native 802.11 IHV Handler Functions.

Interaction with the User

Article • 12/15/2021

The IHV Extensions DLL does not provide a user interface (UI) component. Instead, it interfaces with the IHV UI Extensions DLL to display the appropriate UI pages to the user. For more information about the IHV UI Extensions DLL, see Native 802.11 IHV UI Extensions DLL.

This section discusses the ways in which the IHV Extensions DLL requests user interaction, and includes the following topics:

Requesting User Interaction

Processing Session Changes

Requesting User Interaction

Article • 12/15/2021

At any time after the call to *Dot11ExtIhvInitAdapter*, the IHV Extensions DLL can request interaction with the user by calling the **Dot11ExtSendUIRequest** function. The operating system forwards all user interaction requests to the IHV UI Extensions DLL, which will process the request and display the appropriate user interface (UI) pages to the user.

When the request has been completed, the operating system calls the *Dot11ExtIhvProcessUIResponse* function to forward the results from the IHV UI Extensions DLL for the user interaction. For more information about the IHV UI Extensions DLL, see Native 802.11 IHV UI Extensions DLL.

For example, the IHV Extensions DLL can request user interaction for any of the following.

- Notify the user regarding the stages of a pre or post-association operation.
- Prompt the user to enter his/her credentials for authentication during the post-association operation.

When it calls the **Dot11ExtSendUIRequest** function, the IHV Extensions DLL passes a pointer to a **DOT11EXT_IHV_UI_REQUEST** structure to the *pIhvUIRequest* parameter. The DOT11EXT_IHV_UI_REQUEST structure specifies the request, such as the globally unique ID (GUID), which identifies the UI request as well as the COM class ID (CLSID) of the target UI page that will handle this request.

When the IHV UI Extensions DLL has completed the user notification, the operating system calls the *Dot11ExtIhvProcessUIResponse* function. If the user had entered any data through the notification, the operating system passes a pointer to the buffer, which contains the data, to the *pvResponseBuffer* parameter.

The operating system might periodically query the status of pending notification requests. In this situation, the operating system calls the *Dot11ExtIhvIsUIRequestPending* and passes the GUID of the UI request to the *guidUIRequest* parameter.

When calling **Dot11ExtSendUIRequest**, the IHV Extensions DLL must follow these guidelines.

- The IHV Extensions DLL does not need to serialize the calls to **Dot11ExtSendUIRequest**. The DLL can have more than one pending UI request at any time.

- The UI request for a particular GUID is completed only when *Dot11ExtIhvProcessUIResponse* is called for that GUID. In this situation, the IHV Extensions DLL must not free any allocated resources for the UI request until *Dot11ExtIhvProcessUIResponse* is called.

- All pending UI requests are canceled whenever *Dot11ExtIhvAdapterReset* or *Dot11ExtIhvDeinitAdapter* is called. All pending UI requests are also canceled whenever *Dot11ExtIhvProcessSessionChange* is called with the *uEventType* parameter set to WTS_SESSION_LOGOFF.

  In these situations, the IHV Extensions DLL must free all allocated resources for each pending UI request.

The operating system can initiate user interaction itself whenever the connection state changes on the basic service set (BSS) network. In this situation, the operating system calls the *Dot11ExtIhvQueryUIRequest* function. The IHV Extensions DLL allocates a buffer and formats it as a **DOT11EXT_IHV_UI_REQUEST** structure. The DLL sets the members of the DOT11EXT_IHV_UI_REQUEST structure to reference a UI page that is appropriate for the change in connection status. The operating system is responsible for displaying the UI page.

Note The IHV Extensions must allocate the buffer that contains the **DOT11EXT_IHV_UI_REQUEST** structure through **Dot11ExtAllocateBuffer**. The DLL must not free the buffer after returning from *Dot11ExtIhvQueryUIRequest*.

Processing Session Changes

Article • 12/15/2021

If the user's session changes state, such as when the user logs in or out, the operating system notifies the IHV Extensions DLL about the session change by calling the *Dot11ExtIhvProcessSessionChange* function. The operating system passes the reason for the session change to the *uEventType* parameter.

If the *uEventType* parameter is set to WTS_SESSION_LOGOFF, the user has logged off of the current session. In this situation, all pending user interface (UI) requests must be canceled internally by the IHV Extensions DLL, and the DLL must free any allocated resources for each pending UI request.

Pre-Association Operations Topics

Article • 12/15/2021

This section discusses the pre-association operation and how it is performed by the IHV Extensions DLL. This section has the following topics:

Pre-Association Operation Overview

Pre-Association Operation Guidelines

Pre-Association Operation Overview

Article • 12/15/2021

After the user has selected a profile for a basic service set (BSS) network connection, the operating system calls the *Dot11ExtIhvPerformPreAssociate* function to initiate a pre-association operation. When this function is called, the IHV Extensions DLL does the following:

- Verifies the IHV-defined extensions to the connectivity and security profile.

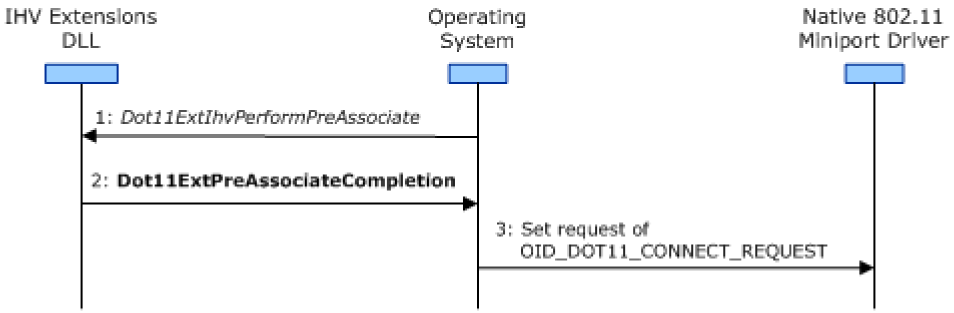
  If the IHV Extensions DLL determines that the profile is incorrect, it returns the appropriate error code as defined in Winerror.h. In this situation, the operating system notifies the user that the network profile cannot be used.

- Initiates the pre-association operation based on the IHV-defined extensions to the connectivity and security profiles.

  After the pre-association operation is initiated, it must be completed asynchronously from the call to *Dot11ExtIhvPerformPreAssociate*.

The IHV Extension DLL completes the pre-association operation through a call to **Dot11ExtPreAssociateCompletion**. Following this call, the operating system initiates the connection operation by issuing a set request of OID_DOT11_CONNECT_REQUEST to the Native 802.11 miniport driver, which manages the WLAN adapter.

The following figure shows the steps involved during the pre-association operation.


IHV Extensions DLL
Operating System
Native 802.11 Miniport Driver
1: Dot11ExtIhvPerformPreAssociate
2: Dot11ExtPreAssociateCompletion
3: Set request of OID_DOT11_CONNECT_REQUEST

When *Dot11ExtIhvPerformPreAssociate* is called, the operating system passes the IHV-defined extensions to the connectivity and security profile through the following parameters.

pIhvProfileParams
This parameter is passed a pointer to a **DOT11EXT_IHV_PROFILE_PARAMS** structure, which specifies the attributes of the basic service set (BSS) network to which the network

profile will be applied. For example, the **DOT11EXT_IHV_PROFILE_PARAMS** structure specifies the service set identifier (SSID) and type of the BSS network.

pIhvConnProfile
This parameter is passed a pointer to a DOT11EXT_IHV_CONNECTIVITY_PROFILE structure that contains the settings for the connectivity profile. The operating system only passes the extensions to the connectivity profile defined by the IHV and selected by the user.

pIhvSecProfile
This parameter is passed a pointer to a DOT11EXT_IHV_SECURITY_PROFILE structure that contains the settings for the security profile. The operating system only passes the extensions to the security profile defined by the IHV and selected by the user.

pConnectableBssid
This parameter is passed a pointer to a DOT11_BSS_LIST structure, which contains one or more 802.11 Beacon or Probe Response frames for the service set identifier (SSID) of the BSS network with which the DLL will perform the pre-association operation.

When performing the pre-association operation, the IHV Extensions DLL can do the following:

- Call the Dot11ExtNicSpecificExtension function to issue proprietary configuration requests for network connectivity to the Native 802.11 miniport driver.

  Through the *pIhvConnProfile* and *pIhvProfileParams* parameters, the IHV Extensions DLL can determine which proprietary connectivity settings were selected by the user.

  Through the *pConnectableBssid* parameter, the IHV Extensions DLL can determine the attributes of the BSS network and can configure proprietary network settings accordingly.

- Configure the WLAN adapter with the proprietary authentication and cipher algorithms to be used over the BSS network connection.

  Through the *pszXmlFragmentIhvSecurity* parameter, the IHV Extensions DLL can determine which proprietary security algorithms were selected by the user.

  The following IHV Extensibility functions can be called to set the security algorithms.
  - Dot11ExtSetAuthAlgorithm
  - Dot11ExtSetUnicastCipherAlgorithm
  - Dot11ExtSetMulticastCipherAlgorithm

- Call the **Dot11ExtSendUIRequest** function to request that the IHV UI Extensions DLL prompt the user for security parameters, such as the user's credentials.
- Call the **Dot11ExtSetEtherTypeHandling** function to register a list of the IEEE EtherTypes for the security packets that the DLL will receive. After the list is registered, the operating system calls the *Dot11ExtIhvReceivePacket* IHV Handler function for every packet whose EtherType matches an entry in the list.

  The IHV Extensions DLL can also specify a list of EtherTypes that will be excluded from payload decryption. For more information about registering EtherTypes, see IEEE EtherType Handling.
- Call the **Dot11ExtSetProfileCustomUserData** function to save data in the registry that is specific to the user and current BSS network profile.
- Call the **Dot11ExtGetProfileCustomUserData** function to retrieve data from the registry that is specific to the user and current BSS network profile.

For more information about the IHV Extensibility functions, see Native 802.11 IHV Extensibility Functions.

For more information about connection operations with BSS networks, see Connection Operations.

Pre-Association Operation Guidelines

Article • 12/15/2021

The IHV Extensions DLL must follow these guidelines when performing the pre-association operation.

- When the *Dot11ExtIhvPerformPreAssociate* function is called, the IHV Extensions DLL must do the following:
    - Verify the IHV extensions for the connectivity and security profile. If the profile parameters are invalid, the *Dot11ExtIhvPerformPreAssociate* function returns an appropriate error code as defined in Winerror.h.
    - Create and begin a new thread for the completion of the pre-association operation. Because the pre-association operation must be completed asynchronously from the call to *Dot11ExtIhvPerformPreAssociate*, the IHV Extensions DLL must call **Dot11ExtPreAssociateCompletion** from this thread after the operation completes.
    - Return ERROR_SUCCESS from the function call. At this point, the operating system is notified that the network profile is valid and the pre-association operation is in progress.

- The IHV Extensions DLL can call the **Dot11ExtNicSpecificExtension** function to configure the wireless LAN (WLAN) adapter. This function can be called either from within the call to *Dot11ExtIhvPerformPreAssociate* or from the thread that handles the pre-association operation after *Dot11ExtIhvPerformPreAssociate* returns.

- Calls to the **Dot11ExtSetProfileCustomUserData**, **Dot11ExtGetProfileCustomUserData**, and **Dot11ExtSetCurrentProfile** must not be made from within the call to *Dot11ExtIhvPerformPreAssociate*. These functions can only be called after *Dot11ExtIhvPerformPreAssociate* returns ERROR_SUCCESS.

- After the IHV Extensions DLL calls **Dot11ExtPreAssociateCompletion** to complete the pre-association operation, the handle for the connection session is no longer valid. The operating system passes this handle through the *hConnectSession* parameter of *Dot11ExtIhvPerformPreAssociate*. The DLL must not use this handle value when calling any IHV Extensibility functions that declare an *hConnectSession* parameter.

    For more information about the IHV Extensibility functions, see Native 802.11 IHV Extensibility Functions.

- If the *Dot11ExtIhvAdapterReset* function is called, the IHV Extensions DLL must cancel the pre-association operation by calling **Dot11ExtPreAssociateCompletion**. For more information about the reset operation, see 802.11 WLAN Adapter Reset.

- If the *Dot11ExtIhvDeinitAdapter* function is called, the IHV Extensions DLL must cancel the pre-association operation internally. However, it must not call any of the IHV Extensibility functions that can be called only after adapter initialization, including **Dot11ExtPreAssociateCompletion**.

Post-Association Operations Overview

Article • 12/15/2021

When the wireless LAN (WLAN) adapter successfully completes an association operation with an access point (AP), the operating system creates a data port for the association. The operating system then initiates a post-association operation on the data port by calling the *Dot11ExtIhvPerformPostAssociate* function.

Note For Windows Vista, the IHV Extensions DLL supports only infrastructure basic service set (BSS) networks.

When performing the post-association operation, the IHV Extensions DLL can do the following:

- Allocate any resources needed for the new data port.
- Perform proprietary security processing for the data port, including sending and receiving packets for the authentication algorithm configured during the pre-association operation. For more information about this operation, see Pre-Association Operations.
- Derive cipher keys and download them to the WLAN adapter.

When the WLAN adapter completes a disassociation operation with the AP, the operating system terminates the post-association operation on the data port by calling the *Dot11ExtIhvStopPostAssociate* function. Following this call, the operating system deletes the data port for the association.

The following topics describe what the IHV Extensions DLL must do to perform or stop a post-association operation.

Performing a Post-Association Operation

Stopping a Post-Association Operation

Interface to the Native 802.11 802.1X Module

For more information about the association operation, see Association Operations.

For more information about the disassociation operation, see Disassociation Operations.

For more information about the procedures involved in port management, see Port-Based Network Access.

Performing a Post-Association Operation

Article • 12/15/2021

When the wireless LAN (WLAN) adapter successfully completes an 802.11 association operation with an access point (AP), the Native 802.11 miniport driver notifies the operating system by making an NDIS_STATUS_DOT11_ASSOCIATION_COMPLETION indication. For more information about the association operation, see Association Operations.

Note For Windows Vista, the IHV Extensions DLL supports only infrastructure basic service set (BSS) networks.

After the operating system receives the NDIS_STATUS_DOT11_ASSOCIATION_COMPLETION indication, it calls the *Dot11ExtIhvPerformPostAssociate* function to notify the IHV Extensions DLL of the following:

- The creation of a new data port for the association with the AP. The IHV Extensions DLL is passed the current state of the data port through the *pPortState* parameter of the *Dot11ExtIhvPerformPostAssociate* function. For more information about the port state parameter, see **DOT11_PORT_STATE**.
- The parameters of the association between the wireless LAN (WLAN) adapter and the AP. The IHV Extensions DLL is passed the association parameters through the *pDot11AssocParams* parameter of the *Dot11ExtIhvPerformPostAssociate* function. For more information about the association parameters, see **DOT11_ASSOCIATION_COMPLETION_PARAMETERS**.

When *Dot11ExtIhvPerformPostAssociate* is called, the IHV Extensions DLL initiates a post-association operation with the AP to authenticate the data port. Through this operation, the IHV Extensions DLL can do the following:

- Allocate any resources needed for the new data port.
- Perform proprietary security processing on the data port for the association. The IHV Extensions DLL can determine the current state of the data port from *pPortState* parameter of the *Dot11ExtIhvPerformPostAssociate* function.

- Call the **Dot11ExtSendUIRequest** function to request the IHV UI Extensions DLL to prompt the user for security parameters, such as the user's credentials.

- Authenticate with the AP using the authentication algorithm enabled through **Dot11ExtSetAuthAlgorithm**. The IHV Extensions DLL calls **Dot11ExtSetAuthAlgorithm** during the pre-association operation. For more information about this operation, see Pre-Association Operations.

- Send security packets to the AP through calls to the **Dot11ExtSendPacket** function.

  When the security packet has been sent, the operating notifies the IHV Extensions DLL through a call to the *Dot11ExtIhvSendPacketCompletion* function.

  For more information about sending security packets, see Send Operations.

- Receive security packets from the AP. The operating system calls the *Dot11ExtIhvReceivePacket* function for each security packet received by the WLAN adapter.

  Each received security packet is serialized and indicated in the order they were received from the WLAN adapter. The operating system only calls the *Dot11ExtIhvReceivePacket* function to indicate received security packets that match an entry in the list of IEEE EtherTypes, which were specified by the IHV Extensions DLL through a call to the **Dot11ExtSetEtherTypeHandling** function.

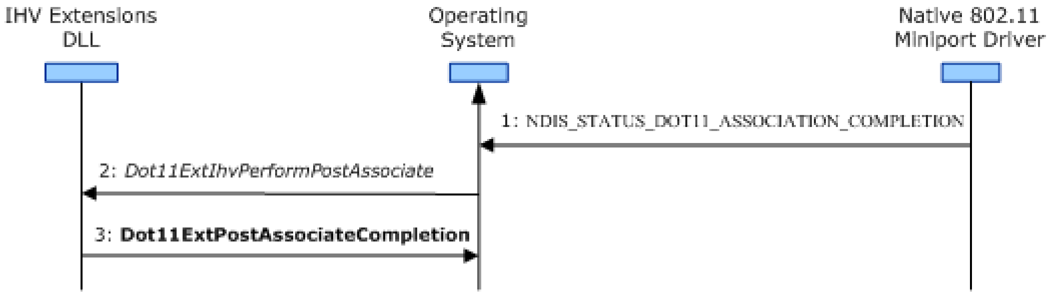
  For more information about receiving security packets, see Receive Operations.

- Configure the WLAN adapter with the cipher keys that are derived through the authentication algorithm. The following IHV Extensibility functions can be called to download the cipher keys to the WLAN adapter.
  - **Dot11ExtSetDefaultKey**
  - **Dot11ExtSetDefaultKeyId**
  - **Dot11ExtSetKeyMappingKey**

- Configure the WLAN adapter to exclude unencrypted packets through a call to the **Dot11ExtSetExcludeUnencrypted** IHV Extensibility function.

After the data port has been authenticated, the IHV Extensions DLL must call **Dot11ExtPostAssociateCompletion** to complete the post-association operation.

The following figure shows the steps involved during the post-association operation.


IHV Extensions DLL
Operating System
Native 802.11 Miniport Driver
1: NDIS_STATUS_DOT11_ASSOCIATION_COMPLETION
2: Dot11ExtIhvPerformPostAssociate
3: Dot11ExtPostAssociateCompletion

The IHV Extensions DLL must follow these guidelines when performing the post-association operation.

- The IHV Extensions DLL must call **Dot11ExtPostAssociateCompletion** asynchronously from the call to *Dot11ExtIhvPerformPostAssociate*.

- After completing the post-association operation, the IHV Extensions DLL can call **Dot11ExtPostAssociateCompletion** whenever the authentication status of the data port changes.

- If the *Dot11ExtIhvAdapterReset* function is called, the IHV Extensions DLL must cancel all pending post-association operations by calling **Dot11ExtPostAssociateCompletion**. For more information about the reset operation, see 802.11 WLAN Adapter Reset.

- If the *Dot11ExtIhvDeinitAdapter* function is called, the IHV Extensions DLL must cancel all pending post-association operations internally. However, it must not call any of the IHV Extensibility functions that can be called only after adapter initialization, including **Dot11ExtPostAssociateCompletion**. For more information about the IHV Extensibility functions, see Native 802.11 IHV Extensibility Functions.

Stopping a Post-Association Operation

Article • 12/15/2021

The operating system terminates the post-association operation by calling the *Dot11ExtIhvStopPostAssociate* IHV Handler function whenever one of the following occurs.

- The wireless LAN (WLAN) adapter completes a disassociation operation with the AP. In this situation, the Native 802.11 station, which manages the adapter, makes a media-specific NDIS_STATUS_DOT11_DISASSOCIATION indication. For more information about the disassociation operation, see Disassociation Operations.

- The WLAN adapter is disabled or removed. In this situation, the operating system calls the *Dot11ExtIhvStopPostAssociate* function before it calls the *Dot11ExtIhvDeinitAdapter* function.

The operating system calls the *Dot11ExtIhvStopPostAssociate* function to notify the IHV Extensions DLL that the data port created for the association with an AP is down. The operating system calls this function regardless of whether the DLL has completed the post-association operation through a call to **Dot11ExtPostAssociateCompletion**.

When *Dot11ExtIhvStopPostAssociate* is called, the IHV Extensions must release all of the resources allocated for the data port. If the post-association operation was not completed with a call to **Dot11ExtPostAssociateCompletion**, the IHV Extensions DLL must cancel the operation internally but must not call **Dot11ExtPostAssociateCompletion**.

Interface to the Native 802.11 802.1X Module

Article • 12/15/2021

After the operating system receives an NDIS_STATUS_DOT11_ASSOCIATION_COMPLETION indication from the Native 802.11 miniport driver, it calls the *Dot11ExtIhvPerformPostAssociate* function to initiate a post-association operation by the IHV Extensions DLL.

While it performs the post-association operation or after the operation has completed, the IHV Extensions DLL can use the extensible authentication protocol (EAP) algorithms that are supported by the operating system to authenticate the user with the access point (AP). In this situation, the IHV Extensions DLL interfaces with the 802.1X module of the Native 802.11 framework for the processing of EAP packets that are sent by the AP in the EAP over LAN (EAPOL) format.

For more information about the EAPOL format, refer to Clause 7 of the IEEE 802.1X-2001 standard.

For more information about the 802.1X module and the Native 802.11 framework, see Native 802.11 Software Architecture.

When interfacing the 802.1X module for user authentication, the IHV Extensions DLL must follow these guidelines:

- For Windows Vista, the IHV Extensions DLL can initiate 802.1X authentication operations through the 802.1X module only for infrastructure basic service set (BSS) network connections.

- The IHV Extensions DLL must register with the operating system to receive EAPOL packets. In this situation, the DLL must call the **Dot11ExtSetEtherTypeHandling** function and add the IEEE EAPOL EtherType (0x888E) to the list of registered EtherTypes that are passed in through the *pusRegistration* parameter. After the EtherType is registered, the operating system forwards received EAPOL packets to the IHV Extensions DLL through calls to the *Dot11ExtIhvReceivePacket* IHV Handler function.

  For more information about registering EtherTypes, see IEEE EtherType Handling.

- While it is performing the post-association operation, the IHV Extensions DLL initiates the 802.1X authentication operation by calling the **Dot11ExtStartOneX** function. When this function is called, the operating system does the following:
  - Display the properties page for the configuration of the 802.1X authentication. This information includes the EAP algorithm used for the authentication.
  - Prompt the user for credentials.
  - Send an EAPOL-Start packet to the AP to initiate the 802.1X authentication.

  The IHV Extensions DLL can call **Dot11ExtStartOneX** either within the call to *Dot11ExtIhvPerformPostAssociate* or after the function call returns.

- The IHV Extensions DLL can call the **Dot11ExtStartOneX** function only after the Native 802.11 miniport driver has completed an association operation with the AP. In this situation, the IHV Extensions DLL must not call the **Dot11ExtStartOneX** function under any of the following conditions:
  - Before the operating system calls *Dot11ExtIhvPerformPostAssociate*. The operating system calls this function after the miniport driver has successfully completed an association operation. For more information about this operation, see Association Operations.
  - After the operating system calls *Dot11ExtIhvStopPostAssociate*. The operating system calls this function after the miniport driver has completed a disassociation operation with the AP. For more information about this operation, see Disassociation Operations.
  - After the operating system calls *Dot11ExtIhvAdapterReset*. The operating system calls this function after the miniport driver has completed a disconnection operation with the basic service set (BSS) network. For more information about this operation, see Disconnection Operations.

- While the 802.1X authentication operation is in progress, the IHV Extensions DLL can cancel the operation by calling **Dot11ExtStopOneX**.

- While the 802.1X authentication operation is in progress, the IHV Extensions DLL must call **Dot11ExtProcessOneXPacket** to forward EAPOL packets to the operating system for processing. **Note** The IHV Extensions DLL is responsible for processing EAPOL-Key packets received from the AP. The DLL must not pass these packets to the operating system through calls to **Dot11ExtProcessOneXPacket**.

- When the 802.1X authentication operation completes, the operating system calls the *Dot11ExtIhvOneXIndicateResult* IHV Handler function. After this function is called, the IHV Extensions DLL is responsible for processing all EAPOL packets received from the AP, such as the EAPOL-Key packets used for derivation of the cipher keys.

- If the 802.1X authentication operation completed successfully, the operating system passes the MPPE-Send-Key value to the DOT11_MSONEX_RESULT_PARAMS structure pointed to by the *pDot11MsOneXResultParams* parameter of *Dot11ExtIhvOneXIndicateResult*. The MPPE-Send-Key value pointed to by the **pbMPPESendKey** member of DOT11_MSONEX_RESULT_PARAMS is derived through the authentication process and is used by the IHV Extensions DLL when sending EAPOL-Key packets to the AP. This key is encrypted and should be decrypted by calling the **CryptUnprotectData** function that is documented in the Windows SDK.

- The algorithm that is used to derive the cipher keys is dependent upon the implementation of the independent hardware vendor (IHV). The IHV Extensions DLL can support standard key derivation algorithms, such as the algorithm defined in Clause 8.5 of the IEEE 802.11i-2004 standard, as well as it can support a proprietary key derivation algorithm.

- After it derives the keys, the IHV Extensions DLL can call the following functions to download the cipher keys to the Native 802.11 miniport driver, which manages the wireless LAN (WLAN) adapter.

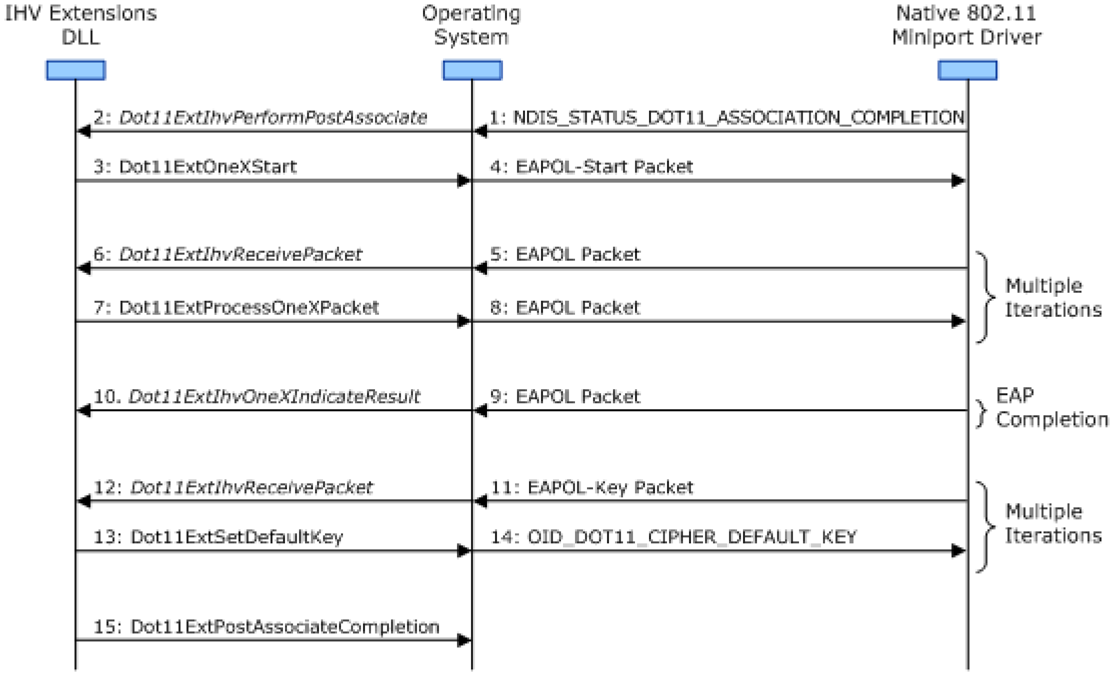
  - Dot11ExtSetDefaultKey

  - Dot11ExtSetDefaultKeyId

  - Dot11ExtSetKeyMappingKey

- The IHV Extensions DLL completes the post-association operation by calling the Dot11ExtPostAssociateCompletion function. After the post-association operation completes, the IHV Extensions DLL can initiate another 802.1X authentication operation if the DLL determines that the user must be reauthenticated.

The following figure shows the sequence of events when the IHV Extensions DLL initiates an 802.1X authentication operation during a post-association operation.


IHV Extensions DLL
Operating System
Native 802.11 Miniport Driver
1: NDIS_STATUS_DOT11_ASSOCIATION_COMPLETION
2: *Dot11ExtIhvPerformPostAssociate*
3: Dot11ExtOneXStart
4: EAPOL-Start Packet
5: EAPOL Packet
6: *Dot11ExtIhvReceivePacket*
7: Dot11ExtProcessOneXPacket
8: EAPOL Packet
Multiple Iterations
9: EAPOL Packet
10. *Dot11ExtIhvOneXIndicateResult*
EAP Completion
11: EAPOL-Key Packet
12: *Dot11ExtIhvReceivePacket*
13: Dot11ExtSetDefaultKey
14: OID_DOT11_CIPHER_DEFAULT_KEY
Multiple Iterations
15: Dot11ExtPostAssociateCompletion

Native 802.11 IHV extension send and receive operations

Article • 12/15/2021

This section describes the guidelines the IHV Extensions DLL follows when sending or receiving packets and contains the following topics:

IEEE EtherType Handling

Send Operations

Receive Operations

IEEE EtherType Handling

Article • 12/15/2021

The IHV Extensions DLL can specify a list of IEEE EtherTypes for special handling of packets received by the wireless LAN (WLAN) adapter. The following types of EtherType handling can be specified.

Privacy Exemptions
The IHV Extensions DLL can specify packet decryption exemptions for received packets. For example, the DLL can specify that a packet with a specified EtherType is allowed to be received unencrypted even if a matching cipher key is configured on the WLAN adapter.

EtherType Registration
The IHV Extensions DLL can register the EtherTypes that it will process and consume. The operating system forwards packets that match a registered EtherType to the DLL through calls to the *Dot11ExtIhvReceivePacket* function.

The IHV Extensions DLL specifies EtherType handling through a call to the **Dot11ExtSetEtherTypeHandling** function. When calling this function, the IHV Extensions DLL must follow these guidelines.

- The IHV Extensions DLL can only call **Dot11ExtSetEtherTypeHandling** any time prior to completing a pre-association operation. For more information about this operation, see Pre-Association Operations.

- The IHV Extensions DLL specifies its list of privacy exemptions through an array of zero or more **DOT11_PRIVACY_EXEMPTION** structures. If the IHV Extensions DLL does not call **Dot11ExtSetEtherTypeHandling**, the operating system defaults to an empty list of privacy exemptions for any 802.11 association with an access point (AP). **Note** For Windows Vista, the IHV Extensions DLL supports only infrastructure basic service set (BSS) networks.

- The IHV Extensions DLL registers a list of zero or more EtherTypes that it will receive. Typically, the DLL registers the EtherTypes for the security packets it processes during the post-association operation. For more information about this operation, see Post-Association Operations.

If the IHV Extensions DLL does not call **Dot11ExtSetEtherTypeHandling**, the operating system defaults to an empty list of registered EtherTypes for any 802.11 association with an AP.

- After the IHV Extensions DLL completes the pre-association operation by calling **Dot11ExtPreAssociateCompletion**, the list of privacy exemptions and EtherType registrations specified through a call to **Dot11ExtSetEtherTypeHandling** is applied to every 802.11 association made by the WLAN adapter while connected to the basic service set (BSS) network.

- The operating system clears the list of privacy exemptions and EtherType registrations before it calls *Dot11ExtIhvAdapterReset*.

Send Operations

Article • 12/15/2021

When performing a post-association operation, initiated through a call to *Dot11ExtIhvPerformPostAssociate*, the IHV Extensions DLL can send packets through the wireless LAN (WLAN) adapter. For more information about the post-association operation, see Post-Association Operations.

Typically, the DLL sends security packets to an access point (AP) for data port authentication by using the algorithm enabled through **Dot11ExtSetAuthAlgorithm**. The IHV Extensions DLL calls **Dot11ExtSetAuthAlgorithm** during the pre-association operation. For more information about this operation, see Pre-Association Operations.

Note For Windows Vista, the IHV Extensions DLL supports only infrastructure basic service set (BSS) networks.

When sending packets, the IHV Extensions DLL must follow these guidelines.

- The IHV Extensions DLL must allocate the memory for a complete 802.11 data packet, including 802.11 media access control (MAC) header, LLC encapsulation (if necessary), and payload data.

  The following table describes which fields and subfields within the 802.11 MAC header are set by the IHV Extensions DLL or WLAN adapter.

Field name	Subfield name	Set by IHV Extension DLL	Set by WLAN adapter
Frame Control	Protocol Version	X	
Frame Control	Type	X	
Frame Control	Subtype	X	
Frame Control	To DS	X	
Frame Control	From DS	X	
Frame Control	More Fragments		X
Frame Control	Retry		X

Field name	Subfield name	Set by IHV Extension DLL	Set by WLAN adapter
Frame Control	Pwr Mgt		X
Frame Control	More Data		X
Frame Control	Protected Frame		X
Frame Control	Order	X	
Duration/ID			X
Address 1		X	
Address 2		X	
Address 3		X	
Sequence Control	Fragment Number		X
Sequence Control	Sequence Number		X

- The IHV Extensions DLL calls the **Dot11ExtSendPacket** function to send the packet through the wireless LAN (WLAN) adapter. The DLL passes a unique handle value, which identifies the packet, to the function's *hSendCompletion* parameter. Typically, the DLL passes the address of the allocated buffer that contains the packet to the *hSendCompletion* parameter. **Note** Only unicast packets can be sent through calls to the **Dot11ExtSendPacket** function.

- When the WLAN adapter has sent the packet, the operating system calls the *Dot11ExtIhvSendPacketCompletion* function. The operating system passes the packet's handle value to the *hSendCompletion* parameter of the function. This handle value will be the same value used by the IHV Extensions DLL in its call to **Dot11ExtSendPacket**.

  When *Dot11ExtIhvSendPacketCompletion* is called, the IHV Extensions DLL must release the memory it allocated for the packet.

  Note The IHV Extensions DLL must not free the resources allocated for a packet sent through **Dot11ExtSendPacket** until the corresponding call to *Dot11ExtIhvSendPacketCompletion* is made.

Receive Operations

Article • 12/15/2021

When performing a post-association operation, initiated through a call to *Dot11ExtIhvPerformPostAssociate*, the operating system calls the *Dot11ExtIhvReceivePacket* function to forward packets to the HV Extensions DLL received through the wireless LAN (WLAN) adapter. For more information about the post-association operation, see Post-Association Operations.

In order to receive packets, the IHV Extensions DLL must call **Dot11ExtSetEtherTypeHandling** to register a list of one or more IEEE EtherTypes. When a packet is received with an EtherType that matches an entry in this list, the operating system calls the *Dot11ExtIhvReceivePacket* function and passes the packet buffer through the function's *pvInBuffer* parameter.

Note The IHV Extensions DLL must call **Dot11ExtSetEtherTypeHandling** before the DLL completes a pre-association operation. For more information about this operation, see Pre-Association Operations.

When *Dot11ExtIhvReceivePacket* is called, the *pvInBuffer* parameter points to a buffer allocated by the operating system that contains the entire 802.11 packet, including media access control (MAC) header, LLC encapsulation (if necessary), and payload data.

The IHV Extensions DLL can send a response to the received packet from within the call to *Dot11ExtIhvReceivePacket*. In this situation, the DLL must follow the guidelines described in Send Operations.

Notification Operations

Article • 12/15/2021

This section describes the types of notifications the IHV Extensions DLL sends or receives and contains the following topics:

Sending Notifications

Receiving Notifications

Sending Notifications

Article • 12/06/2022

The IHV Extensions DLL calls the Dot11ExtSendNotification function to send notifications to any service or application that has registered for the notification. In order to receive the notification, the service or application must register with the Auto Configuration Manager (ACM) by calling the WlanRegisterNotification function.

Note The service or application must register for notifications with a source value of L2_NOTIFICATION_SOURCE_WLAN_IHV in order to receive notifications through calls to the Dot11ExtSendNotification function.

When calling Dot11ExtSendNotification, the IHV Extensions DLL passes a pointer to a L2_NOTIFICATION_DATA structure to the *pNotificationData* parameter. The L2_NOTIFICATION_DATA defines the type of the notification and can provide additional data about the notification to the IHV Extensions DLL.

Receiving Notifications

Article • 12/15/2021

The operating system forwards IHV-specific indications from the Native 802.11 miniport driver by calling the *Dot11ExtIhvReceiveIndication* function. For more information about how the driver makes this type of indication, see IHV-Specific Indications.

When the *Dot11ExtIhvReceiveIndication* function is called, the *pvBuffer* parameter is passed a pointer to a buffer that contains data in a format defined by the IHV.

Virtual Station

Article • 12/15/2021

Beginning with NDIS 6.20 (Windows 7), the operating system provides a virtual station (VSTA) that can interact with the 802.11 miniport driver.

An independent hardware vendor (IHV) uses VSTA functionality through the IHV Extensibility framework rather than through Win32 application programming interfaces (APIs).

The creation of the virtual station is initiated when the IHV Extensions DLL calls the **Dot11ExtRequestVirtualStation** function. The operating system creates only one virtual station on the computer at a time, and only if an IHV Extensions DLL issues a **Dot11ExtRequestVirtualStation** request.

The operating system calls the *Dot11ExtIhvInitVirtualStation* function to initialize the IHV Extensions DLL for virtual station operations. This call also initializes the user-mode API interface between the operating system and the DLL.

Note To ensure that a virtual station is created in a consistent fashion, a computer should have only one installation of the IHV Extensions DLL that attempts to use Virtual Station functionality. Even if more than one DLL is installed, only one virtual station can be created. The operating system cannot guarantee which DLL will have access to a virtual station after the computer is restarted. Note that if a virtual station already has been created at the request of one DLL and a second DLL then calls **Dot11ExtRequestVirtualStation**, a success code might be returned but a second virtual station will not be created. An IHV Extensions DLL should set a two-minute timer after it calls the **Dot11ExtRequestVirtualStation** function. If the timer expires before the virtual station adapter arrival event, the DLL should assume that the virtual station was not created.

Extensible AP/Virtual Station Interactions

If your driver implements virtual station functionality but cannot sustain both Extensible Access Point (ExtAP) and virtual station connections at the same time on different ports, the driver should perform the following actions.

- Inform the operating system whether a port that is being used for ExtAP can or cannot sustain functionality at all times. In particular, the driver should issue the following status indications on the ExtAP port, using the appropriate status code (**NDIS_STATUS_INDICATION**->**StatusCode**) and reason code:

  NDIS_STATUS_DOT11_STOP_AP
  Indicates that AP functionality cannot be sustained on the ExtAP port. In this case, set **DOT11_STOP_AP_PARAMETERS**-> **ulReason** to a value of DOT11_STOP_AP_REASON_AP_ACTIVE. Issue this status indication in the following situations:
  - Before the virtual station port begins to use the shared resource that would block simultaneous virtual station and ExtAP connections
  - If the ExtAP port transitions to the ExtAP INIT state, and virtual station resource use would prevent successful initialization of the ExtAP port.

  NDIS_STATUS_DOT11_CAN_SUSTAIN_AP
  Indicates that the virtual station port is disconnected, or that use of a virtual station resource will not prevent successful transition of the port to the ExtAP INIT state.

- While connecting to a virtual station port, call the **Dot11ExtSetVirtualStationAPProperties** function to provide information about the AP implementation that is hosted by the virtual station connection.

- Fail the virtual station port connections if the ExtAP port is running in the OP state and one of the following situations occurs:
  - One or more clients is on the ExtAP port.
  - The virtual station attempts to start a connection that duplicates Wireless Hosted Network settings.

Native 802.11 IHV Extensibility Functions That Support a Virtual Station

Dot11ExtQueryVirtualStationProperties

Dot11ExtReleaseVirtualStation

Dot11ExtRequestVirtualStation

Dot11ExtSetVirtualStationAPProperties

Structures That Support a Virtual Station

DOT11EXT_VIRTUAL_STATION_AP_PROPERTY

DOT11EXT_VIRTUAL_STATION_APIS

Native 802.11 IHV UI Extensions DLL Topics

Article • 12/15/2021

This section discusses the architecture of the Native 802.11 IHV UI Extensions DLL, and has the following topics:

Windows SDK References

Native 802.11 IHV UI Extensions DLL Overview

Native 802.11 IHV UI Extensions COM Interfaces

Extending the Properties for Wireless Network Profiles

Extending the Advanced Properties for Wireless Network Adapters

Handling Requests for the Display of a Custom UI

Windows SDK References

Article • 05/29/2024

To understand the material in this section, you should be familiar with standard COM interfaces and methods, and specifically the following:

IWizardExtension COM Interface

IWizardSite COM Interface

IObjectWithSite COM Interface

IPropertyBag COM Interface

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Native 802.11 IHV UI Extensions DLL Overview

Article • 12/15/2021

If the independent hardware vendor (IHV) provides a Native 802.11 IHV Extensions DLL, the IHV can optionally provide a Native 802.11 IHV UI Extensions DLL. Through this DLL, the IHV can do the following:

- Extend the Microsoft Network Configuration user interface (UI) properties, which are used for wireless connection and security configuration settings. In this situation, the Native 802.11 IHV UI Extensions DLL can do the following:
  - Add custom display elements to the standard Microsoft 802.11 properties. For example, the Native 802.11 IHV UI Extensions DLL can add items to a list to provide the end user with selections for proprietary security options.
  - Launch a custom UI that you can use to configure proprietary connection and security settings.

  For more information about extending Microsoft 802.11 properties, see Extending 801.11 Properties.

- Launch a custom UI at the request of the Native 802.11 IHV Extensions DLL. Depending on the current state of the underlying Native 802.11 miniport driver, the operating system displays the UI as either of the following:

  - A set of wizard pages within the flow of the operating system's 802.11 UI. For example, if the Native 802.11 IHV Extensions DLL requires user credentials during a wireless LAN (WLAN) connection operation, the operating system displays the custom UI pages provided by the Native 802.11 UI Extensions DLL as part of the standard UI flow.

    A stand-alone UI launched from a balloon notification. For example, if the Native 802.11 Extensions DLL determines that the connection or authentication state on the WLAN connection has changes, the DLL can request that the Native 802.11 UI Extensions DLL display a balloon notification to alert the end user.

  For more information about launching a UI that results from requests from the Native 802.11 Extensions DLL, see Handling UI Requests from the Native 802.11 IHV Extensions DLL.

For more information about the Native 802.11 IHV Extensions DLL, see Native 802.11 IHV Extensions DLL.

For more information about the Microsoft Network Configuration UI and other Native 802.11 components, see Native 802.11 Software Architecture.

Native 802.11 IHV UI Extensions COM Interfaces

Article • 12/15/2021

The Native 802.11 IHV UI Extensions DLL implements one or more of the following COM interfaces:

IDot11ExtUI
Through the **IDot11ExtUI** COM interface, the operating system's Native 802.11 Network Configuration UI can interact with the Native 802.11 IHV UI Extensions DLL. For example, this COM interface provides the methods used by the Native 802.11 Network Configuration UI to query the DLL for the **IDot11ExtUIProperty** COM interfaces that are used to extend the operating system's 802.11 connection and security properties.

The Native 802.11 IHV UI Extensions DLL must provide an implementation of the **IDot11ExtUI** COM interface.

For more information about this COM interface, see IDot11ExtUI COM Interface.

IDot11ExtUIProperty
Through the **IDot11ExtUIProperty** COM interface, the Native 802.11 IHV UI Extensions DLL can extend the connection and security properties that are displayed by the Native 802.11 Network Configuration UI.

The **IDot11ExtUIProperty** COM interface is optional and is only required if the Native 802.11 IHV UI Extensions DLL supports extensions to the operating system's 802.11 connection and security properties.

The Native 802.11 IHV UI Extensions DLL can provide one or more implementations of the **IDot11ExtUIProperty** COM interface, with each implementation representing an IHV-defined extension to a Native 802.11 property. The DLL can provide one or more property extensions for security settings. For Windows Vista, the DLL can add no more than one property extension for connection settings.

For more information about this COM interface, see IDot11ExtUIProperty COM Interface.

IWizardExtension
The Native 802.11 IHV UI Extensions DLL can provide one or more implementations of the **IWizardExtension** COM interface. Each implementation supports the display of one or more custom UI pages. These UI pages are displayed through one of the following:

- An external request made by the Native 802.11 IHV Extensions DLL. For more information about this process, see Requesting the Display of a Custom UI.

- A query made by the operating system to determine whether the Native 802.11 IHV Extensions DLL has a custom UI to display. For more information about this process, see Querying for the Display of a Custom UI.

- An internal request made by a component of the Native 802.11 IHV UI Extensions DLL.

For more information about the **IWizardExtension** COM interface, see IWizardExtension COM Interface.

For more information about the Native 802.11 Network Configuration UI component, see Native 802.11 Software Architecture.

Extending the Properties for Wireless Network Profiles

Article • 12/15/2021

The end user creates or edits a wireless network connection profile through the Native 802.11 Network Configuration user interface (UI). For more information about this UI, see Native 802.11 Software Architecture.

The independent hardware vendor (IHV) can extend the Network Configuration UI to support proprietary connection and security profile settings through a Native 802.11 UI Extensions DLL. The DLL can extend the following tabs that are displayed in the Network Configuration UI.

Connection tab
This tab displays the UI for the connection settings of a wireless LAN (WLAN) network.

The Native 802.11 IHV UI Extensions DLL can extend this UI by following the procedure described in Extending Wireless Connection Property Pages.

Note For Windows Vista, the Native 802.11 UI Extensions DLL can only support one extension to the **Connection** tab.

Security tab
This tab displays the UI for the security settings of a wireless LAN (WLAN) network.

The Native 802.11 IHV UI Extensions DLL can extend this UI by adding display elements for proprietary security settings. For more information about this process, see Extending Wireless Security Property Pages.

The Native 802.11 IHV UI Extensions DLL can also extend the Microsoft 802.1X settings on the **Security** tab. For more information about this process, see Extending Microsoft 802.1X Security Settings.

Note For Windows Vista, the Native 802.11 IHV UI Extensions DLL can only extend the properties of connection and security profiles for infrastructure wireless networks.

Extending Wireless Connection Properties

Article • 12/15/2021

This topic describes how a Native 802.11 IHV UI Extensions DLL extends the properties on the **Connection** tab that are displayed through the Network Configuration user interface (UI). In this situation, the Native 802.11 IHV UI Extensions DLL adds properties to the **Connection** tab for proprietary connection settings.

For more information about the Network Configuration UI and other Native 802.11 components, see Native 802.11 Software Architecture.

Before it displays the **Connection** tab, the operating system does the following:

1. Queries the Native 802.11 IHV UI Extensions DLL for its connection properties through a call to the **IDot11ExtUI::GetDot11ExtUIProperties** method. The operating system passes a value of **DOT11_EXT_UI_CONNECTION** to the method's *ExtType* parameter.

   If the Native 802.11 IHV UI Extension DLL supports a property of type **DOT11_EXT_UI_CONNECTION**, the DLL returns (through the method's *ppDot11ExtUIProperty* parameter) the address of the IDot11ExtUIProperty COM interface, which implements the connection property extension. For more information about the COM interfaces that are used to extend connection properties, see Native 802.11 IHV UI Extensions COM Interfaces.

   Note For Windows Vista, the Native 802.11 IHV UI Extensions DLL must not return more than one IDot11ExtUI COM Interface for a connection property extension.

2. If the Native 802.11 IHV UI Extensions DLL supports a connection property, the operating system queries the friendly name of the property extension by calling the extension's **IDot11ExtUIProperty::GetDot11ExtUIPropertyFriendlyName** method. The operating system inserts the friendly name within the text "Enable *xxx* connection settings," where "*xxx*" is the friendly name of the property extension. The operating system displays this text along with a check box on the **Connection** tab.

3. Queries the extension to determine whether it has a custom UI property that can be displayed. The operating system does this by calling the extension's IDot11ExtUIProperty::Dot11ExtUIPropertyHasConfigurationUI method. If the connection property extension supports a custom UI property, the operating system adds a **Configure** button below the check box for the property.

If the selected proprietary connection setting supports a configuration UI and the end user clicks the **Configure** button in the**Connection** tab, the operating system calls the connection property extension's IDot11ExtUIProperty::DisplayDot11ExtUIProperty method to launch the custom UI. The operating system passes the current profile data for the extension through the method's *bstrIHVProfile* argument.

The profile data is formatted as an XML fragment bounded by the <IHV> </IHV> XML tags. The XML data within these tags is specific to the IHV's implementation and is opaque to the operating system. For more information about the format of the Native 802.11 profile data, refer to the documentation within the Microsoft Windows SDK.

If the profile data is changed through the custom UI, the extension's IDot11ExtUIProperty::DisplayDot11ExtUIProperty method must do the following before returning:

- Allocate a string buffer for the modified profile data and return a pointer to the buffer through the method's *bstrModifiedIHVProfile* parameter. **Note** The extension's IDot11ExtUIProperty::DisplayDot11ExtUIProperty method must not modify the data referenced by the *bstrIHVProfile* argument.

- Set the *pbIsModified* argument to **TRUE**.

Extending Wireless Security Properties

Article • 12/15/2021

This topic describes how a Native 802.11 IHV UI Extensions DLL extends the properties of the **Security** tab that is displayed through the Network Configuration user interface (UI). In this situation, the Native 802.11 IHV UI Extensions DLL adds properties to the **Security** tab for proprietary security settings that are mutually exclusive from the Native 802.11 802.1X module.

The Native 802.11 IHV UI Extensions DLL can also extend the security and encryption methods that are supported by the Native 802.11 802.1X module. For more information about how the DLL does this, see Extending Microsoft 802.1X Security Settings.

For more information about the Network Configuration UI and other Native 802.11 components, see Native 802.11 Software Architecture.

Before it displays the **Security** tab, the operating system does the following:

1. Queries the Native 802.11 IHV UI Extensions DLL for its security property extensions through a call to the **IDot11ExtUI::GetDot11ExtUIProperties** method. The operating system passes a value of **DOT11_EXT_UI_SECURITY** to the method's *ExtType* parameter.

   If the Native 802.11 IHV UI Extension DLL supports one or more properties of type **DOT11_EXT_UI_SECURITY**, the DLL returns (through the method's *ppDot11ExtUIProperty* parameter) a list of IDot11ExtUIProperty COM interfaces for the security property extensions that are supported by the DLL. For more information about the COM interfaces used to extend security properties, see Native 802.11 IHV UI Extensions COM Interfaces.

2. Queries the friendly name of the security extension by calling the extension's **IDot11ExtUIProperty::GetDot11ExtUIPropertyFriendlyName** method. The operating system adds the friendly name to the list of proprietary security settings at the bottom of the **Security** tab.

3. If the end user selects an item from this list, the operating system will call the **IDot11ExtUIProperty::Dot11ExtUIPropertyGetSelected** method of each security extension's IDot11ExtUIProperty COM interfaces. The first extension that returns with a value of **TRUE** for the method's *pfIsSelected* parameter is determined to be the selected extension. The selected entry in the list will then be highlighted.

4. Queries the selected setting's **IDot11ExtUIProperty::Dot11ExtUIPropertyHasConfigurationUI** method to determine whether it has a custom UI property page that can be displayed. If the method returns with the *fHasConfigurationUI* parameter set to **TRUE**, the operating system will add a **Configure** button next to the list of proprietary security settings.

If the selected proprietary security setting supports a configuration UI and the end user clicks the **Configure** button, the operating system calls the setting's **IDot11ExtUIProperty::DisplayDot11ExtUIProperty** method to launch the custom UI. The operating system passes the current profile data for the setting through the method's *bstrIHVProfile* argument.

The profile data is formatted as an XML fragment bounded by the <IHV> </IHV> XML tags. The XML data within these tags is specific to the IHV's implementation and is opaque to the operating system. For more information about the format of the Native 802.11 profile data, refer to the documentation within the Microsoft Windows SDK.

If the profile data is changed through the custom UI, the setting's **IDot11ExtUIProperty::DisplayDot11ExtUIProperty** method must do the following before returning:

- Allocate a string buffer for the modified profile data and return a pointer to the buffer through the method's *bstrModifiedIHVProfile* parameter. **Note** The setting's **IDot11ExtUIProperty::DisplayDot11ExtUIProperty** method must not modify the data that is referenced by the *bstrIHVProfile* argument.

- Set the *pbIsModified* argument to **TRUE**.

Extending Microsoft 802.1X Security Settings

Article • 12/15/2021

If the Native 802.11 IHV Extensions DLL supports extensions to the Native 802.11 802.1X module, the Native 802.11 IHV UI Extensions DLL can extend the Network Configuration user interface's (UI's) **Security** tab to allow user configuration of the 802.1X extensions. For more information about extending the Native 802.11 802.1X module, see Interface to the Native 802.11 802.1X Module.

This section describes how the Native 802.11 IHV UI Extensions DLL can extend the settings for the operating system's security and encryption methods supported by the Native 802.11 802.1X module. This section includes the following topics:

Extending the UI for Proprietary 802.1X Security Methods

Extending the UI for Standard 802.1X Security Methods

For more information about the Network Configuration UI and other Native 802.11 components, see Native 802.11 Software Architecture.

Extending the UI for Proprietary 802.1X Security Methods

Article • 12/15/2021

If the Native 802.11 IHV Extensions DLL supports proprietary 802.1X-based security extensions, the Native 802.11 IHV UI Extensions DLL can extend the Network Configuration user interface's (UI's) **Security** tab to allow user configuration of the these extensions. For more information about extending the Native 802.11 802.1X module, see Interface to the Native 802.11 802.1X Module.

For more information about the Network Configuration UI and other Native 802.11 components, see Native 802.11 Software Architecture.

Before it displays the **Security** tab, the operating system does the following:

1. Queries the Native 802.11 IHV UI Extensions DLL for its security property extensions through a call to the **IDot11ExtUI::GetDot11ExtUIProperties** method. The operating system passes a value of **DOT11_EXT_UI_KEYEXTENSION** to the method's *ExtType* parameter.

   Property extensions of type **DOT11_EXT_UI_KEYEXTENSION** do not provide security settings that are mutually exclusive to the standard Microsoft security settings. Instead, this type of security property extension provides IHV-defined 802.1X settings that are used together with the Microsoft 802.1X settings.

2. Queries the friendly name of the 802.1X security extension by calling the extension's **IDot11ExtUIProperty::GetDot11ExtUIPropertyFriendlyName** method.

3. Queries the extension's **IDot11ExtUIProperty::Dot11ExtUIPropertyIsStandardSecurity** method to determine whether the extension supports a security type extension. If the method sets the *fIsStandardSecurity* parameter to **FALSE**, the operating system will add the extension's friendly name to the **Security type** list on the **Security** tab.

4. When the end user selects an item from the **Security type** list, the operating system responds by calling the **IDot11ExtUIProperty::Dot11ExtUIPropertyGetSelected** method for each extension to match the selection of the end user. The first extension that returns a value of **TRUE** for the method's *pfIsSelected* parameter is determined to be the selected

extension. After this is confirmed, the operating system highlights the selection made by the end user.

5. Calls the selected property extension's **IDot11ExtUIProperty::Dot11ExtUIPropertyHasConfigurationUI** method to determine whether it has a custom UI property page that can be displayed. If the method returns a value of **TRUE** for the method's *fHasConfigurationUI* parameter, the operating system will display a **Configure** button next to the **Security type** list.

   If the end user clicks the **Configure** button, the operating system will call the selected property extension's **IDot11ExtUIProperty::DisplayDot11ExtUIProperty** method to display the custom configuration UI for the extension.

6. Calls the selected property extension's **IDot11ExtUIProperty::Dot11ExtUIPropertyGetDisplayInfo** method. Through this method, the Native 802.11 IHV UI Extensions DLL can return other property extensions to the **Security** tab of the Native 802.11 Network Configuration UI.

   The **IDot11ExtUIProperty::Dot11ExtUIPropertyGetDisplayInfo** method returns a list of the items that the selected property extension adds to the **Security** tab. Each entry in the list is formatted as a **DOT11_EXT_UI_PROPERTY_DISPLAY_INFO** structure.

   For Windows Vista, the Native 802.11 IHV UI Extensions DLL can only add items to the **Encryption** list on the **Security** tab. As a result, each item in the list of **DOT11_EXT_UI_PROPERTY_DISPLAY_INFO** structures must have a **DOT11_EXT_UI_DISPLAY_INFO_TYPE** of **DOT11_EXT_UI_DISPLAY_INFO_CIPHER** in order to be included in the **Encryption** list.

7. When the end user selects from the **Encryption** list, the operating system will call the selected property extension's **IDot11ExtUIProperty::Dot11ExtUIPropertySetDisplayInfo** method to process the profile data that is associated with the end user's selection.

Extending the UI for Standard 802.1X Security Methods

Article • 12/15/2021

If the Native 802.11 IHV Extensions DLL supports proprietary encryption extensions that can be used with the operating system's 802.1X security methods, the Native 802.11 IHV UI Extensions DLL can extend the Network Configuration user interface's (UI')s **Security** tab to allow user configuration of the these extensions. For more information about extending the Native 802.11 802.1X module, see Interface to the Native 802.11 802.1X Module.

For more information about the Network Configuration UI and other Native 802.11 components, see Native 802.11 Software Architecture.

Before it displays the **Security** tab,the operating system does the following:

1. Queries the Native 802.11 IHV UI Extensions DLL for its security property extensions through a call to the **IDot11ExtUI::GetDot11ExtUIProperties** method. The operating system passes a value of **DOT11_EXT_UI_KEYEXTENSION** to the method's *ExtType* parameter.

   Property extensions of type **DOT11_EXT_UI_KEYEXTENSION** do not provide security settings that are mutually exclusive to the standard Microsoft security settings. Instead, this type of security property extension provides IHV-defined 802.1X settings that are used together with the Microsoft 802.1X settings.

2. Queries the friendly name of the 802.1X security extension by calling the extension's **IDot11ExtUIProperty::GetDot11ExtUIPropertyFriendlyName** method.

3. Queries the extension's **IDot11ExtUIProperty::Dot11ExtUIPropertyIsStandardSecurity** method to determine whether the extension supports a security type extension. If the method sets the *fIsStandardSecurity* parameter to **TRUE**, the operating system will not add the extension's friendly name to the **Security type** list on the **Security** tab.

   In this situation, the extension adds functionality to the security settings that are supported by the operating system. The method specifies the type of security setting it extends through the *dot11ExtUISecurityType* parameter.

4. When the end user selects an item from the **Security type** list, the operating system responds by calling the IDot11ExtUIProperty::Dot11ExtUIPropertyGetSelected method for each extension to match the selection of the end user. The first extension that returns a value of **TRUE** for the method's *pfIsSelected* parameter is determined to be the selected extension. After this is confirmed, the operating system highlights the selection made by the end user.

5. When the end user selects an item for a standard security setting from the **Security type** list, the operating system calls the IDot11ExtUIProperty::Dot11ExtUIPropertyGetDisplayInfo method of the property extension that extends the security method. Through the **IDot11ExtUIProperty::Dot11ExtUIPropertyGetDisplayInfo** method, the Native 802.11 IHV UI Extensions DLL can return other items to be added to the **Security** tab of the Native 802.11 Network Configuration UI.

   The **IDot11ExtUIProperty::Dot11ExtUIPropertyGetDisplayInfo** method returns a list of the extended display properties that are supported by the property extension. Each item in the list is formatted as a DOT11_EXT_UI_PROPERTY_DISPLAY_INFO structure.

   For Windows Vista, the Native 802.11 IHV UI Extensions DLL can only add items to the **Encryption** list on the **Security** tab. As a result, each entry within the list of DOT11_EXT_UI_PROPERTY_DISPLAY_INFO structures must have a **DOT11_EXT_UI_DISPLAY_INFO_TYPE** of **DOT11_EXT_UI_DISPLAY_INFO_CIPHER** in order to be included in the **Encryption** list.

6. When the end user selects from the **Encryption** list, the operating system will call the property extension's IDot11ExtUIProperty::Dot11ExtUIPropertySetDisplayInfo method to process the profile data that is associated with the end user's selection.

Extending the Advanced Properties for Wireless Network Adapters

Article • 12/15/2021

In addition to extending the properties for wireless network profiles, the independent hardware vendor (IHV) can also extend the advanced properties for the configuration of the wireless LAN (WLAN) adapter. For more information, refer to the following topics:

Specifying Configuration Parameters for the Advanced Properties Page

Specifying Custom Property Pages for Network Adapters

Handling Requests for the Display of a Custom UI

Article • 12/15/2021

This section discusses how the Native 802.11 IHV UI Extensions DLL can display a custom user interface (UI) through one of the following:

- A request from the Native 802.11 IHV Extensions DLL. For example, the Native 802.11 IHV Extensions DLL might request a custom UI for user notification of a wireless LAN (WLAN) event.
- A query, made by the operating system, to determine whether the Native 802.11 IHV UI Extensions DLL has a custom UI that can be displayed.

This section has the following topics:

Requesting the Display of a Custom UI

Querying for the Display of a Custom UI

Displaying Custom UI Pages within a Balloon Notification

Displaying Custom UI Pages within the Network Connection Wizard

Accessing Profile and Context Data

For more information about the Native 802.11 IHV Extensions DLL, see Native 802.11 IHV Extensions DLL.

Requesting the Display of a Custom UI

Article • 12/15/2021

The Native 802.11 IHV Extensions DLL can request the display of a custom user interface (UI) through the Native 802.11 IHV UI Extensions DLL. For example, the IHV Extensions DLL could request that a custom UI be displayed to:

- Notify the end user at various stages during the wireless LAN (WLAN) association operation.
- Notify the end user when the WLAN adapter has disassociated for the WLAN network.
- Notify the end user with the results of the authentication to the WLAN network.

To launch a custom UI or display a notification, the Native 802.11 IHV Extensions DLL calls **Dot11ExtSendUIRequest** and passes a pointer to a **DOT11EXT_IHV_UI_REQUEST** structure through the *pIhvUIRequest* parameter of this function.

Through the **DOT11EXT_IHV_UI_REQUEST** structure, the Native 802.11 IHV Extensions DLL specifies the custom UI through the following data:

- The user session identifier (ID), which is used to identify a specific user context.
- A globally unique ID (GUID), which identifies the specific UI request.
- The class ID (CLSID) of **IWizardExtension** COM interface implemented within the Native 802.11 IHV UI Extensions DLL. The CLSID is used to request a specific custom UI supported by the DLL.

  For more information about the **IWizardExtension** COM interface, see IWizardExtension COM Interface.
- A buffer containing data in a proprietary format that is defined by the independent hardware vendor (IHV) and processed by the specified **IWizardExtension** COM interface. For example, the buffer could contain the default values that are displayed within the custom UI.

Depending upon the WLAN connection state for the user session ID, the custom UI request will be displayed as one of the following:

- If the adapter has connected to a WLAN network, the request will be displayed as a standalone UI launched through a clickable balloon notification. For more information about this process, see Displaying a Balloon Notification.

- If the adapter is in the process of connecting to a WLAN network, the request will be displayed as a set of wizard pages within the standard Network Connection UI. For more information about this process, see Displaying Custom UI Pages within the Network Connection Wizard.

Querying for the Display of a Custom UI

Article • 12/15/2021

The operating system can query the Native 802.11 IHV Extensions DLL to determine whether the DLL has a custom UI to display. The operating system queries the DLL whenever the wireless LAN (WLAN) adapter transitions to one of the following phases within the WLAN network connection process.

Pre-association
The connection phase before the IHV Extensions DLL initiates a pre-association operation. For more information about the pre-association operation, see Pre-Association Operations.

Post-association
The connection phase after the IHV Extensions DLL completes a post-association operation. For more information about the post-association operation, see Post-Association Operations.

The operating system calls the Native 802.11 IHV Extensions DLL's *Dot11ExtIhvQueryUIRequest* IHV Handler function to query whether a custom UI can be displayed. The operating system passes the current phase of the connection process through the *connectionPhase* parameter. If a custom UI must be displayed, the DLL returns a **DOT11EXT_IHV_UI_REQUEST** structure through the p *pIhvUIRequest* parameter.

Through the **DOT11EXT_IHV_UI_REQUEST** structure, the Native 802.11 IHV Extensions DLL specifies the custom UI through the following data.

- The user session identifier (ID), which is used to identify a specific user context.

- A globally unique ID (GUID), which identifies the specific UI request.

- The class ID (CLSID) of **IWizardExtension** COM interface that is implemented within the Native 802.11 IHV UI Extensions DLL. The CLSID is used to request a specific custom UI that is supported by the DLL.

  For more information about the **IWizardExtension** COM interface, see IWizardExtension COM Interface.

- A buffer that contains data in a proprietary format that is defined by the independent hardware vendor (IHV) and processed by the specified

IWizardExtension COM interface. For example, the buffer could contain the default values that are displayed within the custom UI.

The custom UI will be displayed as a set of wizard pages within the standard Network Connection UI. For more information about this process, see Displaying Custom UI Pages within the Network Connection Wizard.

Displaying Custom UI Pages within a Balloon Notification

Article • 12/15/2021

If the Native 802.11 IHV Extensions DLL calls **Dot11ExtSendUIRequest** to display a custom user interface (UI), the operating system will display the UI through a clickable balloon notification if the wireless LAN (WLAN) adapter has connected to a wireless network. In this situation, the request for the custom UI is displayed as a balloon notification:

- After the Native 802.11 IHV Extensions DLL calls **Dot11ExtPostAssociateCompletion** to successfully complete the post-association operation.
- Before the operating system calls the DLL's *Dot11ExtIhvAdapterReset* IHV Handler function to reset the WLAN connection.

For more information about how the Native 802.11 IHV Extensions DLL requests the display of a custom UI, see Requesting the Display of a Custom UI.

When processing a custom UI request as a balloon notification, the operating system does the following.

1. Calls the Native 802.11 IHV Extensions DLL's *Dot11ExtIhvIsUIRequestPending* IHV Handler function to determine whether a UI request is still pending. The operating system specifies the UI request using the globally unique identifier (GUID) passed to **Dot11ExtSendUIRequest** by the Native 802.11 IHV Extensions DLL.

2. If *Dot11ExtIhvIsUIRequestPending* returns **TRUE** for the specified UI request, the operating system will call the Native 802.11 IHV UI Extensions DLL's **IDot11ExtUI::GetDot11ExtUIBalloonText** method. Through this method, the DLL returns a string buffer that contains the localized text to be displayed within the balloon notification.

3. Displays the balloon notification that contains the localized text.

4. If the end user clicks the balloon notification, the operating system will launch the custom UI that is supported by the requested **IWizardExtension** COM interface. When it calls **Dot11ExtSendUIRequest**, the Native 802.11 IHV Extensions DLL

specifies the class identifier (CLSID) of the **IWizardExtension** implementation within the Native 802.11 IHV UI Extensions DLL.

When the operating system calls the **IWizardExtension::AddPages** method, the Native 802.11 IHV UI Extensions DLL returns an array of handles for PROPSHEETPAGE structures representing the custom UI pages.

For more information about the **IWizardExtension** COM interface, see IWizardExtension COM Interface. For more information about the PROPSHEETPAGE structure, refer to the documentation in the Microsoft Windows SDK.

5. Navigates through the UI pages as specified by the Native 802.11 IHV UI Extensions DLL through **IWizardSite** COM interface. For more information about this interface, see IWizardSite COM Interface.

While the custom UI is displayed, the Native 802.11 IHV UI Extensions DLL can read or write context-specific data through the IPropertyBag COM interface. For more information about this process, see Accessing Profile and Context Data. After the display of the custom UI has completed, the Native 802.11 IHV UI Extensions DLL can return the user-entered response data to the Native 802.11 IHV Extensions DLL by calling **WlanSendUIResponse** . The DLL passes in the GUID for the UI request as well as a pointer to a buffer that contains the response data.

After the Native 802.11 IHV UI Extensions DLLcalls **WlanSendUIResponse**, the operating system will call the Native 802.11 IHV Extension DLL's *Dot11ExtIhvProcessUIResponse* IHV Handler function to forward the response data for the custom UI.

For more information about the **WlanSendUIResponse** API, refer to the documentation in the Windows SDK.

Displaying Custom UI Pages within the Network Connection Wizard

Article • 12/15/2021

A custom user interface (UI) supported by the Native 802.11 IHV UI Extensions DLL can be displayed within the operating system's Network Connection Wizard when the request for UI is made through either:

- A call to **Dot11ExtSendUIRequest**, made by the Native 802.11 IHV Extensions DLL. For more information about this process, see Requesting the Display of a Custom UI.

- A call to the Native 802.11 IHV Extensions DLL's **Dot11ExtQueryUIRequest** IHV Handler function, made by the operating system. For more information about this process, see Querying for the Display of a Custom UI.

The operating system displays the custom UI within the Network Connection Wizard if the wireless LAN (WLAN) adapter is attempting to connect to a wireless network. In this situation, the request for the custom UI will be displayed as a balloon notification within the period:

- After the operating system calls the Native 802.11 IHV Extensions DLL's *Dot11ExtIhvPerformPreAssociate* IHV Handler function to initiate a pre-association operation with the wireless network.

- Before the Native 802.11 IHV Extensions DLL calls **Dot11ExtPostAssociateCompletion** to successfully complete the post-association operation.

When inserting the custom UI request within the Network Connection Wizard, the operating system does the following:

1. Calls the Native 802.11 IHV Extensions DLL's *Dot11ExtIhvIsUIRequestPending* IHV Handler function to determine whether a UI request is still pending. The operating system specifies the UI request using the globally unique identifier (GUID) that is passed to **Dot11ExtSendUIRequest** by the Native 802.11 IHV Extensions DLL.

2. If *Dot11ExtIhvIsUIRequestPending* returns **TRUE** for the specified UI request, the operating system will instantiate the requested **IWizardExtension** COM interface and bind it into the current UI flow of the Network Connection Wizard. When it

calls **Dot11ExtSendUIRequest**, the Native 802.11 IHV Extensions DLL specifies the class identifier (CLSID) of the **IWizardExtension** implementation within the Native 802.11 IHV UI Extensions DLL.

The operating system also calls the **IWizardExtension::AddPages** method, through which the Native 802.11 IHV UI Extensions DLL returns an array of handles for PROPSHEETPAGE structures representing the custom UI pages.

For more information about the **IWizardExtension** COM interface, see IWizardExtension COM Interface.

3. Navigates through the UI pages as controlled by the Native 802.11 IHV UI Extensions DLL through the **IWizardSite** COM interface. For more information about this interface, see IWizardSite COM Interface.

While the custom UI is displayed, the Native 802.11 IHV UI Extensions DLL can read or write context-specific data through the IPropertyBag COM interface. For more information about this process, see Accessing Profile and Context Data.

After the custom UI is displayed, the Native 802.11 IHV UI Extensions DLL can return the user-entered response data to the Native 802.11 IHV Extensions DLL by calling **WlanSendUIResponse**. The DLL passes in the GUID for the UI request as well as a pointer to a buffer containing the response data.

After the Native 802.11 IHV UI Extensions DLL calls **WlanSendUIResponse**, the operating system calls the Native 802.11 IHV Extension DLL's *Dot11ExtIhvProcessUIResponse* IHV Handler function to forward the response data for the custom UI.

For more information about the **WlanSendUIResponse** API, refer to the documentation in the Microsoft Windows SDK.

Accessing Profile and Context Data

Article • 12/15/2021

A custom user interface (UI) that is supported by the Native 802.11 IHV UI Extensions DLL can be displayed through either:

- A call to **Dot11ExtSendUIRequest** made by the Native 802.11 IHV Extensions DLL. For more information about this process, see Requesting the Display of a Custom UI.
- A call to the Native 802.11 IHV Extensions DLL's *Dot11ExtQueryUIRequest* IHV Handler function made by the operating system. For more information about this process, see Querying for the Display of a Custom UI.

Regardless of whether the UI request is displayed through either a balloon notification or the operating system's Network Connection Wizard, the Native 802.11 IHV UI Extensions DLL can access the following data:

Network connection profile data
If the custom UI is displayed within the Network Connection Wizard, the Native 802.11 IHV UI Extensions DLL can access the IHV-defined portion of the current network connection profile. This data is formatted as an XML fragment bounded by the <IHV> </IHV> XML tags. The XML data within these tags is specific to the IHV's implementation and is opaque to the operating system.

Access to the profile data is through the **Read** and **Write** methods of the IPropertyBag COM interface for a property named **IHV_PROFILE_DATA**.

Context data
The Native 802.11 IHV Extensions DLL specifies a custom UI through a **DOT11EXT_IHV_UI_REQUEST** structure, which is passed as an argument in both the **Dot11ExtSendUIRequest** and *Dot11ExtIhvQueryUIRequest* functions. Within the DOT11EXT_IHV_UI_REQUEST structure, the IHV can provide (through the **pvUIRequest** member) context data specific to the custom UI. Typically, the IHV formats this data with default settings for the custom UI.

Access to the profile data is through the **Read** and **Write** methods of the IPropertyBag COM interface for a property named **IHV_NOTIFICATION_DATA**.

The Native 802.11 IHV UI Extensions DLL accesses the IPropertyBag COM interface through the *IUnknown* pointer returned through the **IObjectWithSite::SetSite** method.

For more information, see IObjectWithSite.

As an alternative to the IPropertyBag COM interface, the Native 802.11 IHV UI Extensions DLL can access the **IHV_PROFILE_DATA** and **IHV_NOTIFICATION_DATA** properties through the GetProp Win32 function. In this situation, the DLL must use the handle of the parent window, as shown in the following example:

C++

```
LPWSTR lpszBuffer = (LPWSTR) GetProp(GetParent(hwndDlg),
L"IHV_PROFILE_DATA");
```

Wi-Fi Hotspot Offloading Overview

Article • 12/15/2021

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

Wi-Fi Hotspot Offloading enables Windows 10 Mobile to automatically identify and connect to local Wi-Fi networks designated as "hotspots." The Wi-Fi Hotspot Offload Framework enables mobile operators to ship phones preconfigured with customized hotspot plugins for Wi-Fi hotspot offloading.

In This Section

- Wi-Fi Hotspot Offloading Architecture
- Wi-Fi Hotspot Offloading Plugin
- Wi-Fi Discovery Service
- Wi-Fi Hotspot Offloading Reference

Wi-Fi Hotspot Offloading Architecture

Article • 12/15/2021

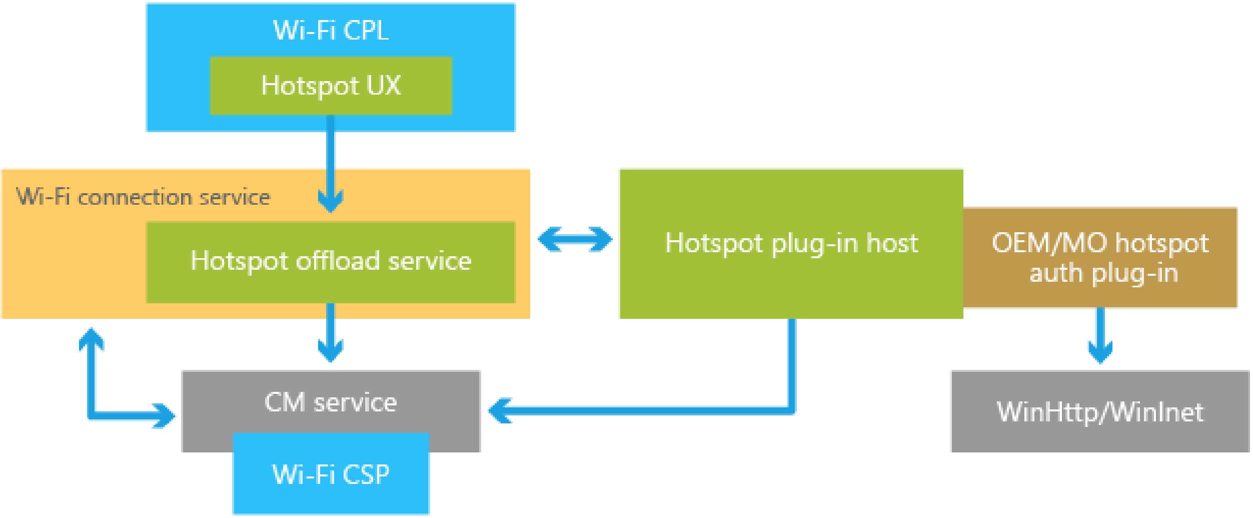
Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The following diagram shows the major components in the Wi-Fi Offload Framework.



Hotspot offload service

The hotspot offload service performs the following functions:

- Identifies Wi-Fi networks that are hotspot networks
- Oversees the creation and maintenance of connections to hotspot networks
- Monitors and responds to connection state changes for hotspot networks
- Monitors and responds to changes in user settings for enabling or disabling Wi-Fi hotspot offloading

The hotspot offload service relies on hotspot plugins created by mobile operators and/or OEMs to identify and authenticate their hotspot networks.

Hotspot plugin host

The hotspot plugin host is the interface between the hotspot offload service and the partner-implemented hotspot plugin. For example, queries to the hotspot plugin to identify hotspot networks from a list of networks are made through the hotspot plugin host. The plugin host also enables the hotspot plugin to, among other things, send and receive HTTP messages via the WinHTTP/WinInet API and send SMS alerts and notifications to the user.

The hotspot offload service is responsible for creating a hotspot plugin host for each hotspot plugin.

Hotspot plugin

The hotspot plugin performs the following functions:

- Identifies hotspot networks from a list of available networks
- Enables automatic connection to networks using EAP-SIM/AKA and HTTP-based authentication, as specified by the OEM or mobile operator
- Sends/receives HTTP messages via the WinHTTP/WinInet API
- Sends SMS notifications to the user
- Selects a bearer token for an HTTP request to send and receive messages over a cellular network

It also interacts directly with the following external components:

- WinInet/WinHTTP

Mobile operators and/or OEMs must implement and install their own hotspot plugins to enable Wi-Fi offloading. The installation package for the plugin includes the following:

- The plugin DLL
- Files containing connection-specific information such as a list of SSIDs, encrypted credentials, etc.
  - **Note:** These files are optional and are not expected in most plugins.
- Registry configuration

Hotspot user interface

The hotspot user interface is displayed in the Wi-Fi control panel. Through the user interface the user can:

- Enable/disable automatic Wi-Fi hotspot offloading.
- View connection status during automatic connection to a hotspot network.
- Manually connect to a hotspot network.
  - If hotspot offload functionality is enabled on the device, user-initiated connections to a network that the hotspot offload service has identified as a hotspot network will be handled as automatic connections to a Wi-Fi hotspot network. Otherwise, the manual connection will be handled as a standard Wi-Fi connection.
- Configure a normal Wi-Fi profile for connection to a hotspot network if the mobile operator hotspot connectivity has been disabled by the user.

The hotspot user interface is only displayed when at least one plugin is configured.

Example: Automatic connection to a hotspot network

The following is a very high-level description of the sequence of component interactions that occur during automatic connection to a hotspot network:

1. The Wi-Fi Connection Service sends to the Hotspot Offload Service a list of networks that are not connected.
2. For each entry in the list of networks, the Hotspot Offload Service queries the hotspot plugins (in the order the plugins were ranked) to determine if it is a hotspot network. The first plugin to identify the network is asked to authenticate that network at connection time.
3. When a hotspot plugin identifies a network as a hotspot network, it returns a priority value associated with that network, the authentication method to be used (whether it is HTTP-based, or EAP-SIM based, or requires no specific SIM) and, optionally, the network display mask. The priority value indicates the order in which a connection should be attempted. Connections to networks with lower priority values will be attempted before connections to networks with higher values.
4. The Hotspot Offload Service creates a Connection Manager profile for the selected network.
5. The Hotspot Offload Service profile may also configure an initial policy setting that will cause the Connection Manager to block applications from connecting to the network until authorized.
6. The Hotspot Offload Service marks the selected network as a hotspot network.

7. The Hotspot Offload Service calls the hotspot plugin, through the hotspot plugin host, to do any pre-connect processing if needed.
8. After the hotspot plugin has completed pre-connect processing, the Hotspot Offload Service waits for the Connection Manager to connect to the hotspot network and provide a connection-completion or failure notification.
9. On connection-completion, the Hotspot Offload Service sends a request to the hotspot plugin to perform any necessary post-connect actions, such as HTTPS exchange.
10. In the meantime, the Hotspot Offload Service does the following:

    - Starts a timer for completion of the post-connect activity (currently set to fire after 5 minutes)
    - Sets proper user interface display state

11. If the hotspot plugin indicates connection success, the Hotspot Offload service calls the Connection Manager to unblock the connection and notify applications.
12. If the post-connection request times out:

    - The Hotspot Offload Service resets the hotspot plugin's state.
    - If retries are not exhausted the hotspot offload service initiates an attempt to reconnect, otherwise it deletes the network's hotspot profile.

13. If the hotspot plugin indicates failure and retries are possible, the hotspot offload service initiates an attempt to reconnect, otherwise it deletes the network's hotspot profile.

Wi-Fi Hotspot Offloading Plugin

Article • 12/15/2021

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

To enable Wi-Fi offloading, create and install a hotspot plugin. This topic discusses a few of the issues to consider when developing a hotspot plugin. It also provides a general description of the plugin APIs to be implemented as part of the plugin package.

Planning the plugin

Before starting plugin development, make sure to address the following issues:

Supported authentication methods

Identify the authentication methods required by the networks that the plugin will support. The hotspot offload framework supports three classes of networks:

- Networks that use WISPr 1.0, or some variant, to authenticate the user and/or device over HTTP. These networks are represented by the following capability:
  - **HS_FLAG_CAPABILITY_NETWORK_AUTH_HTTP**
- Networks that use EAP-SIM/AKA/AKA' to authenticate the device. These networks are represented by the following capabilities:
  - **HS_FLAG_CAPABILITY_NETWORK_AUTH_EAP_SIM**
  - **HS_FLAG_CAPABILITY_NETWORK_AUTH_EAP_AKA**
  - **HS_FLAG_CAPABILITY_NETWORK_AUTH_EAP_AKA_PRIME**

  For EAP-based networks, the plugin can also specify a custom realm by using the **HS_FLAG_CAPABILITY_NETWORK_CUSTOM_REALM** capability.

- Networks that do not require any authentication or networks for which the plugin has an independent authentication mechanism that does not require any device credentials. These networks are represented by the following capability:
  - **HS_FLAG_CAPABILITY_NETWORK_AUTH_NO_SIM**

Hidden networks

Hidden networks must be prespecified at initialization time because the network is not visible in the scan results. Due to the power and privacy implications of hidden networks, the framework supports at most one hidden network globally. Therefore, if another plugin has also requested connectivity to a hidden network, the request of the second plugin will be denied. If the plugin requires a hidden network to be configured, it must specify the **HS_FLAG_CAPABILITY_NETWORK_TYPE_HIDDEN** capability for that network.

For all other networks, the plugin should specify the **HS_FLAG_CAPABILITY_NETWORK_TYPE_VISIBLE** capability.

User interface display strings

Custom UI display strings, used by the plugin to communicate with the user, must be stored in a string table (in an .rc file). The plugin must pass the string IDs to the hotspot offload service to enable it to load the appropriate strings. Currently, the following display strings are supported:

- Provider name (up to **HS_CONST_MAX_PROVIDER_NAME_LENGTH** length)
- Network name (up to **HS_CONST_MAX_NETWORK_DISPLAY_NAME_LENGTH** length)
- Message on the Advanced page (up to **HS_CONST_MAX_ADVANCED_PAGE_STRING_LENGTH** length)
- Any additional strings passed to the user using the HSHostSendUserMessage function (up to **MAX_PATH** length). For more information, see HS_HOST_SEND_USER_MESSAGE.

Note: For more information about Wi-Fi Hotspot Offloading capabilities and constants, see Wi-Fi Hotspot Offloading Constants.

Implementing the plugin

The plugin is implemented as a DLL. The functions HSPluginGetVersion and HSPluginInitPlugin must be exposed either by specifying them in the .def file of the plugin DLL, or by adding the "__declspec(dllexport)" keyword to them in the function implementation.

Initialization

The plugin APIs are invoked in the following order at initialization:

HsPluginGetVersion

The plugin should return its version information, to verify that the plugin version matches the host device version. The current version is stored in the constant **HS_CONST_HOST_CURRENT_API_VERSION**.

HSPluginInitPlugin

This is the main initialization function. It provides the following information to the plugin:

- A context handle for the plugin to use whenever it calls any of the hotspot plugin host (**HS_HOST_****) functions
- The version number currently used by the host (**dwVerNumUsed**)
- Information about the device (**pDeviceIdentity**)
- The OS capabilities available to the plugin, specified as a HS_FLAG_CAPABILITY_NETWORK_** type (**dwHostCapabilities**)
- Handlers for the functions used by the plugin to call back to the host (**pHotspotHostHandlers**)

The plugin returns the following information to the hotspot plugin host:

- A pointer to the structure that contains the list of plugin APIs (**pHotspotPluginAPIs**). For more information, see HOTSPOT_PLUGIN_APIS.
- A pointer to the structure that contains the plugin profile (**pPluginProfile**). For more information, see HS_PLUGIN_PROFILE.

The profile includes all of the capabilities required by the plugin. This is represented by a single value that results from combining the applicable capability flag values (HS_FLAG_CAPABILITY_NETWORK_*) by using a bitwise OR operation. If the plugin specifies the HS_FLAG_CAPABILITY_NETWORK_AUTH_HTTP capability or the HS_FLAG_CAPABILITY_NETWORK_AUTH_EAP_* capabilities, the **dwSupportedSIMCount** member of the **HS_PLUGIN_PROFILE** structure must be set to the number of supported

SIMs. The plugin must also specify the total number of networks that it supports by setting the **dwNumNetworksSupported** member of its **HS_PLUGIN_PROFILE** structure.

HsPluginQueryHiddenNetwork [Optional]

If the plugin specifies the **HS_FLAG_CAPABILITY_NETWORK_TYPE_HIDDEN** capability and the device can support a hidden network, this function is called by the hotspot plugin host to obtain the hidden network information from the plugin. For more information, see HS_PLUGIN_QUERY_HIDDEN_NETWORK.

HsPluginQuerySupportedSIMs [Optional]

The hotspot plugin host calls this function if the plugin specifies a nonzero value for **dwSupportedSIMCount**. When called, the **pNetworkIdentity** argument should be NULL and the plugin is required to provide the list of all SIMs supported by the plugin. This function may also be called later on to identify SIMs that are associated with each hotspot network (at which time, the **pNetworkIdentity** will be non-NULL). The plugin must provide the list of supported SIMs. For more information, see HS_PLUGIN_QUERY_SUPPORTED_SIMS.

Run time

As networks become visible, the hotspot plugin host queries the plugin for each network to determine if it is a hotspot network.

HSPluginIsHotspotNetwork

The hotspot plugin host calls this function to determine if the specified network is a hotspot network. It passes identifying information about the network (SSID, authentication type, cipher) through a HS_NETWORK_IDENTITY structure. The plugin must return an eHS_NETWORK_STATE enumeration value that indicates the type of network. If it is a hotspot network, then information about the network is returned through a HS_NETWORK_PROFILE structure. For more information, see HS_PLUGIN_IS_HOTSPOT_NETWORK.

HsPluginQuerySupportedSIMs [Optional]

The hotspot plugin host calls this function if the plugin specifies the capabilities **HS_FLAG_CAPABILITY_NETWORK_AUTH_HTTP** or **HS_FLAG_CAPABILITIES_NETWORK_AUTH_EAP** in the *HS_NETWORK_PROFILE* argument of the call to HS_PLUGIN_IS_HOTSPOT_NETWORK. When called in this instance, the pNetworkIdentity argument should be non-NULL and the plugin must provide the list of SIMs supported for the network specified in pNetworkIdentity only. For more information, see HS_PLUGIN_QUERY_SUPPORTED_SIMS.

HSPluginQueryCellularExceptionHosts [Optional]

The hotspot plugin host calls this function if the **dwNumCellularExceptions** field of the HS_NETWORK_PROFILE structure, returned by the plugin, is set to a nonzero value. The plugin must return the list of cellular bearer hosts when called. For more information, see HS_PLUGIN_QUERY_CELLULAR_EXCEPTION_HOSTS.

Connect time

When a network is deemed connectable, or the network is selected by the user, the following sequence of calls takes place:

HSPluginPreConnectInit

The hotspot plugin host calls this function to notify the plugin that a connection to the hotspot network specified in the HS_NETWORK_IDENTITY structure, returned by the plugin, is in progress. For more information, see HS_PLUGIN_PRE_CONNECT_INIT.

HSPluginStartPostConnectAuth

When the L2 connection is complete, the hotspot plugin host calls this function to notify the plugin to start authentication. The plugin is provided the *pConnectContext*, *pNetworkIdentity*, and *pNetworkProfile* from the previous call to **HSPluginPreConnectInit**, but it is also provided a *dwConnectionId* and *pSIMData*. The plugin must store the Connection ID and use it when calling back the host's **HSHostPostConnectAuthCompletion** handler to notify the OS of the authentication result, and also in the call to **HSHostSendUserMessage** if a message needs to be conveyed to the user. The **pSIMData** struct contains addition information about the SIM configuration that could be needed by the plugin during authentication. If the plugin returns Success, it must call **HSHostPostConnectAuthCompletion** within 5 minutes or the connection is disconnected.

Disconnect and reset

When a network is disconnected, either explicitly by some user or device action or implicitly as a result of external factors, the following functions are called:

HSPluginStopPostConnectAuth

The hotspot plugin host calls this function to terminate network authentication because the device is about to be disconnected from the network. For more information, see HS_PLUGIN_STOP_POST_CONNECT_AUTH.

HSPluginDisconnectFromNetwork

The hotspot plugin host calls this function to inform the plugin that the device will be disconnected from the network. For more information, see HS_PLUGIN_DISCONNECT_FROM_NETWORK.

HSPluginReset

The hotspot plugin host calls this function to reset the plugin to its initial (just loaded) state. For more information, see HS_PLUGIN_RESET.

Periodic calls

The following functions are called periodically, depending on specific parameters set by the plugin:

HSPluginSendKeepAlive [Optional]

The hotspot plugin host calls this function at the frequency specified in the **dwKeepAliveTimeMins** member of the HS_NETWORK_PROFILE structure returned by the plugin. For more information, see HS_PLUGIN_SEND_KEEP_ALIVE.

HSPluginCheckForUpdates [Optional]

The hotspot plugin host calls this function at the frequency specified in the **dwProfileUpdateTimeDays** member of the HS_PLUGIN_PROFILE structure.

Unloading the plugin

HSPluginDeinit

The hotspot plugin host calls this function to enable the plugin to flush any unsaved information and close any open handles before it is unloaded. The plugin will be provided the reason for the unload in the *UnloadReason* argument. For more information, see HS_PLUGIN_DEINIT.

Plugin Installation package

THe plugin installation package should include the following:

The Plugin DLL File

The DLL file must be signed and placed under **Programs\HotspotHost\** *<ProviderName>*, where *<ProviderName>* is the name of the DLL provider.

For information about signing the DLL, see Sign binaries and packages.

There is no particular convention for naming the DLL file, so making sure the path to the file is correct in the registry is all that is required. For example, the registry information could be specified in the package as:

```XML
<RegKeys>
        <RegKey KeyName="$(hklm.software)\Microsoft\Windows
Phone\HotspotOffload\Plugins\<ProviderName>">
          <RegValue Name="PluginRank" Type="REG_DWORD" Value="00000005" />
          <RegValue Name="PluginPath" Type="REG_SZ"
Value="%SystemDrive%\Programs\HotspotHost\Orange\<ProviderName>\
<HotspotPlugin.dll>" />
        </RegKey>
      </RegKeys>
```

Registry configuration

The required registry settings are saved in a new entry created under: **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Phone\HotspotOffload\Plugins\** *ProviderName*.

ProviderName must be unique to the plugin provider or Mobile Operator.

The following values must be saved under the registry key:

Name	Type	Description

Name	Type	Description
PluginPath	[REG_SZ]	The name and full path to the DLL.
PluginRank	[REG_DWORD]	Any positive value between 1 and 250, inclusive (0 is reserved for Microsoft). A lower value represents a higher priority. If two plugins have the same rank, the hotspot service arbitrarily prioritizes one over another.

Data files that contain connection-specific information such as a list of SSIDs, encrypted credentials, etc. [Optional]

The data files should be saved under: `Data\SharedData\HotspotHost\Plugins\<ProviderName>`.

Wi-Fi Discovery Service Overview

Article • 12/15/2021

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The Wi-Fi discovery service enables users to reduce data costs by offloading cellular traffic to Wi-Fi hotspots. The discovery service aggregates Wi-Fi hotspot data from providers, such as mobile operators, and other sources to produce a directory of known Wi-Fi hotspots. By using this directory, users can obtain information about hotspots near their current position.

Mobile operators can submit hotspot data to the service by sending an HTTP POST request, or by using a command-line tool provided by Microsoft.

Instructions

Wi-Fi hotspot data must be in the format described in Wi-Fi Hotspot Data Submission Format.

The discovery service requires that all of a provider's hotspots are submitted in a single batch. Each batch can contain multiple submission requests with a smaller amount of data. For example, a batch containing 1,000 hotspots can be uploaded to the discovery service by sending 10 submission requests, each containing data for 100 hotspots. Each submission request is assigned the same batch number. The final submission request must include a `X-FinalBatchRequest` header set to the total number of hotspots in the batch. The batch is not processed until a submission request with this header is received. If the header does not match the number of hotspots in the batch, the submission is not processed.

Submitting hotspot data by using HTTP POST

The following example shows a typical submission request. The presence of the `X-FinalBatchRequest` header and the numeric value of "1" indicate that there is only one hotspot in this batch and this is the final submission request. Therefore, this is the only submission request. If the batch contained multiple hotspots, this line should be removed for all submission requests but the last one.

```HTTP
POST https://submitwifiservice.windowsphone.com/v1/SubmitHotspots HTTP/1.1
Content-Type: application/json
X-FinalBatchRequest: 1
 [More headers…]
{
      "Header": {
            "BatchId": "2E20A8DB-9AFA-4A5A-AF6E-5F87DA639C15",
            "TransactionId": 1,
            "ProviderId": "FD9E5EE6-75C7-4A54-8B29-45A3FC83AD63"
      },
      "Hotspots": {
            "add": {
                  "Address": "123 abc street",
                  "City": "Redmond",
                  "CountryOrRegion": "USA",
                  "PostalCode": "98052",
                  "StateOrProvince": "WA"
            },

            "bssids":["00:aa:bb:cc:dd:ee"],
            "free": true,
            "pub": true,
            "loc": {
                  "Latitude": 47.01,
                  "Longitude": 121.1234,
                  "RadialUncertainty": 300,
                  "Altitude": 638.34,
                  "AltitudeUncertainty": 100.0
            },
            "name": "Joes Coffee Shop",
            "phid": "abcdefg",
            "ran": 100,
            "ssid": "JoesCoffee",
            "phone": ["425-882-8080"]
      }
}
```

When the message is received on the destination server, **SubmitHotspots** validates the request and authenticates the sender before sending the hotspot data to the discovery service.

Submitting hotspot data by using the command line tool

WifiProviderExe.exe is a command-line tool, provided by Microsoft, that takes as input a hotspot data file, converts it to the required format, and uploads it to a specified discovery service server.

To run WiFiProvider.exe, use the following syntax:

Windows Command Prompt

```
WifiProviderExe –DataFile filename -ProviderId GUID -ServiceEndpoint URL -
CustomTransformer filename.dll [-MappingFile filename.xml] [-CertFile
filename.pfx] [-CertPassword password] [-CertSubject name]
```

For example:

Windows Command Prompt

```
WifiProviderExe -DataFile "file.txt" -ProviderId 00000000-0000-0000-0000-
000000000000 -ServiceEndpoint
"https://submitwifiservice.windowsphone.com/v1/SubmitHotspots" -
CustomTransformer "transformer.dll"
```

The following table contains the list of possible parameters for WiFiProvider.exe.

Parameter	Description
DataFile	Required. The name of the file that contains the hotspot data.
ProviderId	Required. The Microsoft-assigned provider ID (a GUID).
ServiceEndpoint	Required. The URL of the discovery service server to which the hotspot data will be uploaded.
CustomerTransformer	Required. The name of the assembly that contains the transformer.
MappingFile	Optional. The mapping file that maps the provider's hotspot data to the format required by the discovery service.
CertFile	Optional. A pointer to the actual pfx file that contains the certificate(s) for authentication. The certificate password parameter (**CertPassword**) must be specified when using this authentication method.
CertPassword	Optional. The password to the certificate specified in **CertFile**.

Parameter	Description
CertSubject	Optional. The subject name of the certificate. It is located in current user's My Cert store. When using this authentication mechanism, **CertFile** and **CertPassword** are not required. However, it is required to create a private key for the certificate and, in the access control list, grant access rights for the key to the account that will use the certificate.

Transformers

The hotspot data can be in any format. However, it is required to specify a "data transformer" that the command-line tool can access to convert the hotspot data to the format required by the discovery service.

The following table shows the sample transformers that are provided with the command-line tool. They can be used to convert specific data formats to the required format.

Transformer	Data Format	Description
Microsoft.Wps.WifiService.ProviderSdk.JsonHotspotDataTransformer.dll	JSON	Your data must conform to the JSON format specified in Wi-Fi Hotspot Data Submission Format.
Microsoft.Wps.WifiService.ProviderSdk.JsonHotspotDataTransformer.dll	Simple Excel	You must supply a mapping file.

Mapping Files

If your hotspot data is in simple Excel format, you must supply an XML file that maps columns in the Excel file to corresponding required JSON elements. The following list shows the allowed column names:

- Latitude
- Longitude
- RadialUncertainty
- Altitude

- AltitudeUncertainty
- HotspotName
- SSID
- Range
- Address
- City
- StateOrProvince
- PostalCode
- CountryOrRegion
- Bssids
- ProviderHotspotId
- IsPublic
- IsFree
- PhoneNumbers

The following example shows a portion of a mapping file. Each **MappingRule** element associates an Excel column (**PartnerColumnName** and **PartnerColumnNumber**) with a required JSON element (**MicrosoftColumnName**). The **ContainsHeaderRow** element, located after the closing Rules tag (`</Rules>`), indicates that the file contains a header row, which should be skipped when reading data.

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<MappingRules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Rules>
    <MappingRule>
      <PartnerColumnName>english_hotspot_name</PartnerColumnName>
      <PartnerColumnNumber>3</PartnerColumnNumber>
      <MicrosoftColumnName>HotspotName</MicrosoftColumnName>
    </MappingRule>
    <MappingRule>
      <PartnerColumnName>english_city_name</PartnerColumnName>
      <PartnerColumnNumber>2</PartnerColumnNumber>
      <MicrosoftColumnName>City</MicrosoftColumnName>
    </MappingRule>
    <MappingRule>
      <PartnerColumnName>ssid</PartnerColumnName>
      <PartnerColumnNumber>4</PartnerColumnNumber>
      <MicrosoftColumnName>SSID</MicrosoftColumnName>
    </MappingRule>
    <MappingRule>
      <PartnerColumnName>venue</PartnerColumnName>
      <PartnerColumnNumber>7</PartnerColumnNumber>
      <MicrosoftColumnName>HotspotType</MicrosoftColumnName>
    </MappingRule>
        .
        .
        .
        .
  </Rules>
  <ContainsHeaderRow>true</ContainsHeaderRow>
</MappingRules>
```

Wi-Fi Hotspot Data Submission Format

Article • 12/15/2021

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

Hotspot data submitted to the discovery service must be in JavaScript Object Notation (JSON) and must use the following elements.

Element	Type	Description
BatchId	GUID	A provider can split a single submission that contains multiple hotspots into multiple submissions. Each submission is assigned the same BatchId GUID.
ProviderId	GUID	The provider ID will be assigned to the provider by Microsoft.
TransactionId	int	An incrementing number for each request in the batch. Used for degbugging purposes.
Hotspots		A list of hotspots to upload.
add		The full address property, which includes the following sub-elements: **Address**, **City**, **StateOrProvince**, **PostalCode**, and **CountryOrRegion**.
bssids	List<string>	A list of the BSSIDs that make up the hotspot. Each BSSID consists of eight two digit hexadecimal values in the in the following format: *00:aa:bb:cc:dd:ee*.
free	Boolean	A Boolean value that indicates whether the hotspot is free.
pub	Boolean	A Boolean value that indicates whether the hotspot is public.
loc		The full location property which includes the following sub-elements: **Latitude**, **Longitude**, **RadialUncertainty**, **Altitude**, and **AltitudeUncertainty**.

Element	Type	Description
name	string	The friendly name of the hotspot.
phid	string	The provider's hotspot ID. Used for debugging purposes.
ran	uint	The range of the hotspot.
ssid	string	The hotspot's SSID.
phone	string	A list of all the phone numbers associated with the hotspot.

Note

- The header and its sub-elements (**ProviderId**, **BatchId**, and **TransactionId**) are required.
- The Hotspots list must not be empty.
- If the Location element is specified then **Latitude** and **Longitude** are required.
- If the Location element is not specified then at least one BSSID must be specified. Otherwise, a warning will be returned and the discovery service will not process the hotspot.

The following example shows the complete data for a single hotspot:

JSON

```
{
      "Header": {
            "BatchId": "BA85A383-5943-4D84-8ACB-B113BDEA3783",
            "ProviderId": "AE012377-B0B4-4096-B5D5-D7EFBDC170EC",
            "TransactionId": 1
      },
      "Hotspots":[{
            "add": {
                  "Address": "123 abc street",
                  "City": "Redmond",
                  "CountryOrRegion": "USA",
                  "PostalCode": "98052",
                  "StateOrProvince": "WA"
            },
            "bssids":["00:aa:bb:cc:dd:ee"],
            "free": true,
            "pub": true,
            "loc": {
                  "Latitude": 47.01,
                  "Longitude": 121.1234,
                  "RadialUncertainty": 300,
                  "Altitude": 638.34,
                  "AltitudeUncertainty": 100.0
            },
            "name": "Joes Coffee Shop",
            "phid": "abcdefg",
            "ran": 100,
            "ssid": "JoesCoffee",
            "phone": ["425-882-8080"]
      }
}
```

The discovery service returns a simple response, also in JSON format, that includes the activity ID used for debugging purposes and a list of warnings that were generated when the discovery service validated the hotspot data. All strings returned by the discovery service are encoded in UTF-8.

The following table shows the JSON elements for a discovery service response.

Element	Type	Description
ActivityId	string	Required. This is a string that the provider can use to help debug issues.
Warnings	List<string>	A human-readable list of warnings.

The following example shows a typical response to a hotspot data submission:

```JSON
{
    "ActivityId": "d2856c06-e4a1-4434-90e8-ced0c9ee6e10",
    "Warnings": ["Invalid Latitude: 247.67 – Hotspot Id: abcdefg"]
}
```

Wi-Fi Hotspot Offloading Reference

Article • 12/15/2021

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

This section contains descriptions of the Wi-Fi Hotspot Offloading programming elements and related information.

In this section

Wi-Fi Hotspot Offloading Constants

Wi-Fi Hotspot Offloading Structures

Wi-Fi Hotspot Offloading Functions

Wi-Fi Hotspot Offloading Enumerations

Related topics

Wi-Fi Hotspot Offloading Guide

Wi-Fi Hotspot Offloading Constants

Article • 12/15/2021

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

This section describes the constants that are defined for the Wi-Fi Hotspot Offloading framework.

HS_CONST_HOST_CURRENT_API_VERSION
1
Current API version number.

HS_CONST_MAX_NETWORK_DISPLAY_NAME_LENGTH
32
Maximum length of the network display name string.

HS_CONST_MAX_REALM_LENGTH
255
Maximum length of the realm value string.

HS_CONST_MIN_CONN_KEEPALIVE_TIME_IN_MINS
5
Minimum time between keep-alive message transmissions

HS_CONST_PROFILE_UPDATE_TIME_IN_DAYS
7
Minimum time between checks for profile updates.

HS_CONST_MIN_NETWORK_PRIORITY_VALUE
1
Minimum network priority value.

HS_CONST_MAX_NETWORK_PRIORITY_VALUE

65000

Maximum network priority value.

HS_MAX_PHONE_NUMBER_LENGTH

32

Maximum length of the phone number string.

HS_CONST_MAX_HOST_NAME_LENGTH

255

Maximum length of the host name string.

HS_CONST_PLUGIN_MIN_RANK_VALUE

1

Minimum Rank value. Must be greater than 0.

HS_CONST_PLUGIN_MAX_RANK_VALUE

250

Maximum rank value. Must be less than or equal to 250.

HS_CONST_MAX_PROVIDER_NAME_LENGTH

63

Maximum length of provider name string.

HS_CONST_MAX_ADVANCED_PAGE_STRING_LENGTH

255

Maximum length of the advanced page string.

HS_CONST_MAX_PHONE_NUMBER_LENGTH

32

Maximum length of the phone number string.

HS_CONST_MAX_SUPPORTED_SIMS

1000

Maximum size of the list of supported SIM configurations.

HS_CONST_MAX_CELLULAR_EXCEPTION_HOSTS

5

Maximum size of the list of cellular bearers.

HS_CONST_MAX_AUTH_ERROR_MSG_LENGTH

512

Maximum length of an authentication error message.

HS_CONST_MAX_USER_MESSAGES_IN_MINUTES
7*24*60
Maximum user messages, in minutes (7 days).

The following flags are defined for the plug-in and the host to indicate their requirements and capabilities respectively.

HS_FLAG_CAPABILITY_NETWORK_TYPE_VISIBLE
0x00000001
Specifies visible network.

HS_FLAG_CAPABILITY_NETWORK_TYPE_HIDDEN
0x00000002
Specifies hidden network.

HS_FLAG_CAPABILITY_NETWORK_DISPLAY_NAME
0x00000010
Specifies use of display name for EAP-SIM or EAP-AKA authentication.

HS_FLAG_CAPABILITY_NETWORK_AUTH_NO_SIM
0x00000100
Specifies no-SIM authentication.

HS_FLAG_CAPABILITY_NETWORK_AUTH_HTTP
0x00000200
Specifies HTTP authentication.

HS_FLAG_CAPABILITY_NETWORK_AUTH_EAP_SIM
0x00001000
Specifies EAP-SIM authentication.

HS_FLAG_CAPABILITY_NETWORK_AUTH_EAP_AKA
0x00002000
Specifies EAP-AKA authentication.

HS_FLAG_CAPABILITY_NETWORK_AUTH_EAP_AKA_PRIME
0x00004000
Specifies EAP-AKA’ (AKA Prime) authentication.

HS_FLAG_CAPABILITY_NETWORK_CUSTOM_REALM
0x00010000
Specifies use of custom realm value for network authentication.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

Wi-Fi Hotspot Offloading Reference

HOTSPOT_HOST_HANDLERS structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HOTSPOT_HOST_HANDLERS** structure contains the hotspot host handlers function table. This function table is passed to the plugin when HSPluginInitPlugin is called to initialize it. The table contains functions that are called by the plugin to communicate with the hotspot host.

Syntax

ManagedCPlusPlus

```
typedef struct _HOTSPOT_HOST_HANDLERS {
  HS_HOST_ALLOCATE_MEMORY                            HSHostAllocateMemory;
  HS_HOST_FREE_MEMORY                                HSHostFreeMemory;
  HS_HOST_POST_CONNECT_AUTH_COMPLETION
            HSHostPostConnectAuthCompletion;
  HS_HOST_SEND_KEEP_ALIVE_COMPLETION
              HSHostSendKeepAliveCompletion;
  HS_HOST_UPDATE_CONFIGURATION_COMPLETION
         HSHostUpdateConfigurationCompletion;
  HS_HOST_SEND_USER_MESSAGE                          HSHostSendUserMessage;
} HOTSPOT_HOST_HANDLERS, *PHOTSPOT_HOST_HANDLERS;
```

Members

HSHostAllocateMemory

Optional memory management handler.

Handle to the function that is called by the plugin to allocate any memory needed by the plugin. For more information, see HS_HOST_ALLOCATE_MEMORY.

HSHostFreeMemory
Optional memory management handler.

Handle to the function that is called by the plugin to free any memory that had been allocated earlier by the call to HS_HOST_ALLOCATE_MEMORY. For more information, see HS_HOST_FREE_MEMORY.

HSHostPostConnectAuthCompletion
Required connection-process handler.

Handle to the function that is called by the plugin to indicate the success or failure status resulting from the authentication attempt following a Wi-Fi connection setup at layer 2. For more information, see HS_PLUGIN_START_POST_CONNECT_AUTH.

HSHostSendKeepAliveCompletion
Optional periodic request.

Handle to the function that is called by the plugin to indicate the success or failure status resulting from the Send Keep Alive request. For more information, see HS_PLUGIN_SEND_KEEP_ALIVE.

HSHostUpdateConfigurationCompletion
Optional periodic request.

Handle to the function that is called by the plugin to indicate the success or failure of a call to check for updates. For more information, see HS_PLUGIN_CHECK_FOR_UPDATES.

HSHostSendUserMessage
Optional periodic request.

Handle to the function that is called to communicate with the user. For more information see HS_HOST_SEND_USER_MESSAGE.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HSPluginInitPlugin

HS_HOST_ALLOCATE_MEMORY

HS_HOST_FREE_MEMORY

HS_PLUGIN_START_POST_CONNECT_AUTH

HS_PLUGIN_SEND_KEEP_ALIVE

HS_PLUGIN_CHECK_FOR_UPDATES

HS_HOST_SEND_USER_MESSAGE

HOTSPOT_PLUGIN_APIS structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HOTSPOT_PLUGIN_APIS** structure contains the Hotspot plugin APIs function table. This function table is returned by the plugin when HSPluginInitPlugin is called to initialize the plugin. The table contains functions that are called by the hotspot host to communicate with the plugin.

Syntax

ManagedCPlusPlus

```
typedef struct _HOTSPOT_PLUGIN_APIS {
  HS_PLUGIN_QUERY_SUPPORTED_SIMS     HSPluginQuerySupportedSIMs;
  HS_PLUGIN_QUERY_HIDDEN_NETWORK     HSPluginQueryCellularExceptionHosts;
  HS_PLUGIN_IS_HOTSPOT_NETWORK       HSPluginIsHotspotNetwork;
  HS_PLUGIN_PRE_CONNECT_INIT         HSPluginPreConnectInit;
  HS_PLUGIN_START_POST_CONNECT_AUTH  HSPluginStartPostConnectAuth;
  HS_PLUGIN_STOP_POST_CONNECT_AUTH   HSPluginStopPostConnectAuth;
  HS_PLUGIN_DISCONNECT_FROM_NETWORK  HSPluginDisconnectFromNetwork;
  HS_PLUGIN_RESET                    HSPluginReset;
  HS_PLUGIN_SEND_KEEP_ALIVE          HSPluginSendKeepAlive;
  HS_PLUGIN_CHECK_FOR_UPDATES        HSPluginCheckForUpdates;
  HS_PLUGIN_DEINIT                   HSPluginDeinit;
} HOTSPOT_PLUGIN_APIS, *PHOTSPOT_PLUGIN_APIS;
```

Members

HSPluginQuerySupportedSIMs
API called during plugin initialization.

Called by the hotspot host to retrieve the list of SIMs that the plugin supports. It can be called to retrieve the complete list of supported SIMs, or just the SIMs for a specific network. For more information, see HS_PLUGIN_QUERY_SUPPORTED_SIMS.

HSPluginQueryCellularExceptionHosts
API called during plugin initialization.

Called by the hotspot host if the plugin has specified the **HS_FLAG_CAPABILITY_NETWORK_TYPE_HIDDEN** capability by way of the HS_PLUGIN_PROFILE structure. For more information, see HS_PLUGIN_QUERY_HIDDEN_NETWORK.

HSPluginIsHotspotNetwork
API called while processing scan results.

Called by the hotspot host to request the plugin to identify if the network passed in the *pHiddenNetworkIdentity* parameter is a hotspot network. For more information, see HS_PLUGIN_IS_HOTSPOT_NETWORK.

HSPluginPreConnectInit
Connection-process API.

Called by the hotspot host to notify the plugin to initialize its state when a connection is in progress. For more information, see HS_PLUGIN_PRE_CONNECT_INIT.

HSPluginStartPostConnectAuth
Connection-process API.

Called by the hotspot host to request the plugin to perform any post-connect authentication required to authenticate the device over the network. For more information, see HS_PLUGIN_START_POST_CONNECT_AUTH.

HSPluginStopPostConnectAuth
Connection-process API.

Called by the hotspot host to notify the plugin to stop the authentication process. For more information, see HS_PLUGIN_STOP_POST_CONNECT_AUTH.

HSPluginDisconnectFromNetwork
Connection-process API.

Called by the hotspot host to notify the plugin of disconnection from network. For more information, see HS_PLUGIN_DISCONNECT_FROM_NETWORK.

HSPluginReset
API to reset the plugin. If the plugin does not release any pending calls before returning from this call, the plugin will be unloaded.

Called by the hotspot host to reset the plugin. For more information, see HS_PLUGIN_RESET.

HSPluginSendKeepAlive
API for plugin to do periodic updates.

Called by the hotspot host to send a keep-alive message to the plugin. For more information, see HS_PLUGIN_SEND_KEEP_ALIVE.

HSPluginCheckForUpdates
API for plugin to do periodic updates.

Called by the hotspot host to check for updates. For more information, see HS_PLUGIN_CHECK_FOR_UPDATES.

HSPluginDeinit
API called to de-initialize and clean up the plugin before unloading.

Called by the hotspot host to notify the plugin that it is about to be unloaded. For more information, see HS_PLUGIN_DEINIT.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HSPluginInitPlugin

HS_PLUGIN_QUERY_SUPPORTED_SIMS

HS_PLUGIN_PROFILE

HS_PLUGIN_QUERY_HIDDEN_NETWORK

HS_PLUGIN_IS_HOTSPOT_NETWORK

HS_PLUGIN_PRE_CONNECT_INIT

HS_PLUGIN_START_POST_CONNECT_AUTH

HS_PLUGIN_STOP_POST_CONNECT_AUTH

HS_PLUGIN_DISCONNECT_FROM_NETWORK

HS_PLUGIN_RESET

HS_PLUGIN_SEND_KEEP_ALIVE

HS_PLUGIN_CHECK_FOR_UPDATES

HS_PLUGIN_DEINIT

HS_CONNECTION_CONTEXT structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_CONNECTION_CONTEXT** structure contains the information required by the plugin for post connect authentication.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_CONNECTION_CONTEXT {
  HS_MAC_ADDRESS  MacAddress;
  HS_SIM_IDENTITY SIMIdentity;
  WCHAR           pszPhoneNumber[HS_MAX_PHONE_NUMBER_LENGTH+1];
} HS_CONNECTION_CONTEXT, *PHS_CONNECTION_CONTEXT;
```

Members

MacAddress
The HS_MAC_ADDRESS structure that contains the MAC address.

SIMIdentity
The HS_SIM_IDENTITY structure that contains information required for EAP-SIM/AKA authentication.

pszPhoneNumber
Pointer to the phone number.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_MAC_ADDRESS

HS_SIM_IDENTITY

HS_DEVICE_IDENTITY structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_DEVICE_IDENTITY** structure contains information about the device model and manufacturer.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_DEVICE_IDENTITY {
  DWORD dwSystemType;
  WCHAR wszPhoneManufacturer[HS_CONST_MAX_DEVICE_INFO_LENGTH+1];
  WCHAR wszPhoneModelName[HS_CONST_MAX_DEVICE_INFO_LENGTH+1];
  WCHAR wszPhoneManufacturerModel[HS_CONST_MAX_DEVICE_INFO_LENGTH+1];
  WCHAR wszDeviceModel[HS_CONST_MAX_DEVICE_INFO_LENGTH+1];
} HS_DEVICE_IDENTITY, *PHS_DEVICE_IDENTITY;
```

Members

dwSystemType
The type of SIM, whether GSM or CDMA.

wszPhoneManufacturer
The phone manufacturer name.

wszPhoneModelName
The phone model name.

wszPhoneManufacturerModel

Another name for the phone manufacturer and model.

wszDeviceModel

The device model name.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_MAC_ADDRESS structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_MAC_ADDRESS** structure contains the host Media Access Control (MAC) address.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_MAC_ADDRESS {
  UCHAR ucHSMacAddress[6];
} HS_MAC_ADDRESS, *PHS_MAC_ADDRESS;
```

Members

ucHSMacAddress
The MAC address.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_NETWORK_IDENTITY structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_NETWORK_IDENTITY** structure contains information that uniquely identifies a Wi-Fi network.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_NETWORK_IDENTITY {
  HS_SSID             Ssid;
  HS_AUTH_ALGORITHM   hsAuthAlgo;
  HS_CIPHER_ALGORITHM hsCipherAlgo;
} HS_NETWORK_IDENTITY, *PHS_NETWORK_IDENTITY;
```

Members

Ssid
The network SSID.

hsAuthAlgo
The authentication algorithm used by the wireless network.

hsCipherAlgo
The cipher algorithm used by the wireless network.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_NETWORK_PROFILE structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_NETWORK_PROFILE** structure is provided by the plugin and contains information required for connection to the target network. Each instance of the Network Profile is uniquely associated with a corresponding HS_NETWORK_IDENTITY structure.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_NETWORK_PROFILE {
  DWORD  dwNetworkCapabilities;
  USHORT usPriority;
  DWORD  dwSupportedSIMCount;
  DWORD  dmNumCellularExceptions;
  DWORD  dwNetworkStringID;
  DWORD  dwKeepAliveTimeMins;
  WCHAR  szRealm[HS_CONST_MAX_REALM_LENGTH+1];
} HS_NETWORK_PROFILE, *PHS_NETWORK_PROFILE;
```

Members

dwNetworkCapabilities

A subset of the possible **HS_FLAG_CAPABILITY_NETWORK_*** values. For more information about hotspot host capabilities, see Wi-Fi Hotspot Offloading Constants.

usPriority

A unique priority value assigned to the associated network. It must be a value between 1 and 65000 (a hidden network must have a value of 1). A lower numeric value corresponds to a higher priority.

dwSupportedSIMCount
Supported SIM count. This member is set for HTTP-based and EAP-SIM/AKA/AKA' authentication.

dmNumCellularExceptions
Optional. Number of host connections over cellular only.

dwNetworkStringID
Network name string ID. Maximum string size = MAX_NETWORK_DISPLAY_NAME_LENGTH.

dwKeepAliveTimeMins
Optional. The time interval between network connection keep-alive messages.

szRealm
Network-specific realm value.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_NETWORK_IDENTITY

Wi-Fi Hotspot Offloading Constants

HS_PLUGIN_CELLULAR_EXCEPTION_HOSTS structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_CELLULAR_EXCEPTION_HOSTS** structure contains the list of hosts that the plugin requires to be connected over a cellular bearer only during the authentication process. This is an optional capability that can be requested by the plugin. For more information, see HS_PLUGIN_QUERY_CELLULAR_EXCEPTION_HOSTS.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_PLUGIN_CELLULAR_EXCEPTION_HOSTS {
  DWORD               dwCount;
  HS_PLUGIN_HOST_NAME pExceptions[*];
  HS_PLUGIN_HOST_NAME pExceptions[1];
} HS_PLUGIN_CELLULAR_EXCEPTION_HOSTS, *PHS_PLUGIN_CELLULAR_EXCEPTION_HOSTS;
```

Members

dwCount

The number of host names in the list pointed to by **pExceptions**.

pExceptions

Used if MIDL is utilized. Unique, size is (dwCount).

Pointer to the list of host names.

pExceptions

Used if MIDL is not utilized.

Pointer to the list of host names.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_PLUGIN_QUERY_CELLULAR_EXCEPTION_HOSTS

Microsoft Interface Definition Language

HS_PLUGIN_HOST_NAME structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_HOST_NAME** structure contains the host name.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_PLUGIN_HOST_NAME {
  WHCAR pszHostName[HS_CONST_MAX_HOST_NAME_LENGTH+1];
} HS_PLUGIN_HOST_NAME, *PHS_PLUGIN_HOST_NAME;
```

Members

pszHostName
Pointer to the host name.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_PLUGIN_PROFILE structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_PROFILE** structure provides information about the plugin. The members of this structure are set by the plugin during execution of the HSPluginInitPlugin function that is called by the host.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_PLUGIN_PROFILE {
  DWORD dwPluginCapabilities;
  DWORD dwNumNetworksSupported;
  DWORD dwProviderNameStringID;
  DWORD dwGenericNetworkNameStringID;
  DWORD dwAdvancedPageStringID;
  DWORD dwProfileUpdateTimeDays;
  WCHAR szRealm[HS_CONST_MAX_REALM_LENGTH+1];
  DWORD dwSupportedSIMCount;
} HS_PLUGIN_PROFILE, *PHS_PLUGIN_PROFILE;
```

Members

dwPluginCapabilities
Required.

A subset of the possible **HS_FLAG_CAPABILITY_NETWORK_*** values. For more information about hotspot host capabilities, see Wi-Fi Hotspot Offloading Constants.

dwNumNetworksSupported
Required.

Total number of networks supported by this plugin.

dwProviderNameStringID
Required.

The network provider name which is displayed to the user. Maximum string size = MAX_PROVIDER_NAME_LENGTH.

dwGenericNetworkNameStringID
Optional.

Network name. Maximum string size = MAX_NETWORK_DISPLAY_NAME_LENGTH.

dwAdvancedPageStringID
Optional.

Maximum string size = HS_CONST_MAX_ADVANCED_PAGE_STRING_LENGTH.

dwProfileUpdateTimeDays
Optional.

Must be greater than or equal to HS_CONST_MIN_PROFILE_UPDATE_TIME_IN_DAYS.

szRealm
Required if HS_FLAG_CAPABILITIES_NETWORK_CUSTOM_REALM is set.

Network-specific realm value.

dwSupportedSIMCount
The size of the list pointed to by **pSupported SIMs**.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HSPluginInitPlugin

Wi-Fi Hotspot Offloading Constants

HS_PLUGIN_SUPPORTED_SIMS structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_SUPPORTED_SIMS** structure contains the list of supported SIM configurations. This list must be supplied if the hotspot plugin requires HTTP or EAP authentication for any of its networks.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_PLUGIN_SUPPORTED_SIMS {
  DWORD           dwCount;
  HS_SIM_IDENTITY pSupportedSIMs[*];
  HS_SIM_IDENTITY pSupportedSIMs[1];
} HS_PLUGIN_SUPPORTED_SIMS, *PHS_PLUGIN_SUPPORTED_SIMS;
```

Members

dwCount

The list size.

pSupportedSIMs

Used if MIDL is utilized. Unique, size is (dwCount).

An array of HS_SIM_IDENTITY structures that make up the list of supported SIM configurations.

pSupportedSIMs

Used if MIDL is not utilized.

An array of HS_SIM_IDENTITY structures that make up the list of supported SIM configurations.

Remarks

In the **dwEapMethods** field of the **HS_SIM_IDENTITY** structure for each SIM configuration, you must specify the EAP methods that it supports.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_SIM_IDENTITY

Microsoft Interface Definition Language

HS_PLUGIN_VERSION structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_VERSION** structure contains the minimum and maximum hotspot host versions supported by the plugin.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_PLUGIN_VERSION {
  DWORD dwVerMin;
  DWORD dwVerMax;
} HS_PLUGIN_VERSION, *PHS_PLUGIN_VERSION;
```

Members

dwVerMin
The minimum hotspot host version supported by the plugin.

dwVerMax
The maximum hotspot host version supported by the plugin.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_SIM_DATA structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_SIM_DATA** structure contains information stored in the SIM card.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_SIM_DATA {
  WCHAR wszICCID[HS_CONST_MAX_ICCID_LENGTH+1];
  WCHAR wszIMEI[HS_CONST_MAX_IMEI_LENGTH+1];
  WCHAR wszMEID_ME[HS_CONST_MAX_MEID_ME_LENGTH+1];
  WCHAR wszSF_EUIMID[HS_CONST_MAX_SF_EUIMID_LENGTH+1];
} HS_SIM_DATA, *PHS_SIM_DATA;
```

Members

wszICCID

The Integrated Circuit Card Identifier (ICCID) stored in the SIM card.

wszIMEI

The International Mobile Equipment Identity (IMEI) used to identify 3GPP phones.

wszMEID_ME

The Mobile Equipment Identifier (MEID) defined by 3GPP2.

wszSF_EUIMID

The Short Form Expanded User Identity Module Identifier (EUIMID) for a R-UIM or CSIM

(CDMA SIM application) card.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_SIM_IDENTITY structure

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_SIM_IDENTITY** structure contains SIM identification information required for EAP-SIM or EAP-AKA authentication.

Syntax

ManagedCPlusPlus

```
typedef struct _HS_SIM_IDENTITY {
  eHS_SIM_TYPE SimType;
  DWORD        dwMNC;
  DWORD        dwMCC;
  DWORD        dwNID;
  DWORD        dwSID;
  DWORD        dwEapMethods;
} HS_SIM_IDENTITY, *PHS_SIM_IDENTITY;
```

Members

SimType

The type of SIM, whether GSM or CDMA, or none. If the network is GSM, the **dwMNC** and **dwMCC** pair of fields will be defined, whereas for CDMA the **dwSID** and **dwNID** pair of fields must be defined.

dwMNC

Used if the SIM is GSM type.

The mobile network code (MNC) of the GSM network.

dwMCC
Used if the SIM is GSM type.

The mobile country code (MCC) of the GSM network.

dwNID
Used if the SIM is CDMA type.

The Network Identification Number (NID) of the CDMA network.

dwSID
Used if the SIM is CDMA type.

The System Identification Number (SID) of the CDMA network.

dwEapMethods
The EAP authentication method.

Requirements

Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HSPluginGetVersion function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HSPluginGetVersion** function is exported by the plugin DLL and is called to verify that the plugin version matches the host version.

Syntax

ManagedCPlusPlus

```
DWORD HSPluginGetVersion(
  _Out_ HS_PLUGIN_VERSION *pHotspotPluginVersion
);
```

Parameters

**pHotspotPluginVersion* [out]
A pointer to the HS_PLUGIN_VERSION structure that contains version information for the plugin.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_PLUGIN_VERSION

HSPluginInitPlugin function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HSPluginInitPlugin** function is exported by the plugin DLL and is called to initialize the plugin.

Syntax

ManagedCPlusPlus

```
DWORD HSPluginInitPlugin(
  _In_  HANDLE                hPluginContext,
  _In_  DWORD                 dwVerNumUsed,
  _In_  DWORD                 dwHostCapabilities,
  _In_  HS_DEVICE_IDENTITY    *pDeviceIdentity,
  _In_  HOTSPOT_HOST_HANDLERS *pHotspotHostHandlers,
  _Out_ HOTSPOT_PLUGIN_APIS   *pHotspotPluginAPIs,
  _Out_ HS_PLUGIN_PROFILE     *pPluginProfile
);
```

Parameters

hPluginContext [in]
A handle, provided by the host, that the plugin is required to save and then use when it needs to make a request to the host by way of the host handler functions.

dwVerNumUsed [in]
The host's current version number.

dwHostCapabilities [in]
Value that specifies the list of capabilities that the host can provide to the plugin. This value is the bitwise OR combination of the applicable capability flags. For more information about capability flags, see the **HS_FLAG_CAPABILITY_*** constants in Wi-Fi Hotspot Offloading Constants.

Note If the host does not supply all the capabilities required by the plugin, the plugin will not be initialized.

**pDeviceIdentity* [in]
Pointer to a HS_DEVICE_IDENTITY structure that contains information about the device manufacturer and model.

**pHotspotHostHandlers* [in]
Pointer to a HOTSPOT_HOST_HANDLERS structure that contains the hotspot host handlers function table. This table contains pointers to functions that are called by the plugin to communicate with the hotspot host.

**pHotspotPluginAPIs* [out]
Pointer to the HOTSPOT_PLUGIN_APIS structure that contains the hotspot plugin APIs function table. This table is returned by the plugin and contains pointers to functions that are called by the host to communicate with the plugin.

**pPluginProfile* [out]
Pointer to a HS_PLUGIN_PROFILE structure, returned by the plugin, that provides information about the plugin.

Remarks

During initialization, the host provides the following:

- The plugin context handle
- The current version number
- A list of capabilities that the host can provide to the plugin
- A pointer to the host handler function table through which the plugin can communicate with the host

The plugin returns a pointer to its own function table and a pointer to its HS_PLUGIN_PROFILE structure.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

Wi-Fi Hotspot Offloading Constants

HS_DEVICE_IDENTITY

HOTSPOT_HOST_HANDLERS

HOTSPOT_PLUGIN_APIS

HS_PLUGIN_PROFILE

HS_PLUGIN_CHECK_FOR_UPDATES function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_CHECK_FOR_UPDATES** function checks for configuration updates at the frequency specified in the **dwProfileUpdateTimeDays** member of the plugin's HS_PLUGIN_PROFILE structure.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_CHECK_FOR_UPDATES)(

);
```

Parameters

This function has no parameters.

**

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_PLUGIN_PROFILE

HS_PLUGIN_DEINIT function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_DEINIT** function is called by the host to notify the plugin that it will be unloaded.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_DEINIT)(
  _In_ eHS_UNLOAD_REASON UnloadReason
);
```

Parameters

UnloadReason [in]
An eHS_UNLOAD_REASON enumeration value that indicates the reason for the unload.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Remarks

Upon receiving notification that it will be unloaded, the plugin should complete any current activity and save state, if required.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

eHS_UNLOAD_REASON

HS_PLUGIN_DISCONNECT_FROM_NETWORK function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_DISCONNECT_FROM_NETWORK** function notifies the plugin that the device will be disconnected from the network.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_DISCONNECT_FROM_NETWORK)(
  _In_ HS_NETWORK_IDENTITY *pNetworkIdentity
);
```

Parameters

**pNetworkIdentity* [in]
Pointer to the HS_NETWORK_IDENTITY structure for the network from which the device is to be disconnected.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_NETWORK_IDENTITY

HS_PLUGIN_IS_HOTSPOT_NETWORK function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_IS_HOTSPOT_NETWORK** function is called by the host to determine if a specified network is a hotspot network.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_IS_HOTSPOT_NETWORK)(
  _In_      HS_NETWORK_IDENTITY *pNetworkIdentity,
  _Out_     eHS_NETWORK_STATE   *pNetworkState,
  _Out_opt_ HS_NETWORK_PROFILE  *pNetworkProfile
);
```

Parameters

**pNetworkIdentity* [in]
Pointer to the HS_NETWORK_IDENTITY structure for the network from which the device is to be disconnected.

**pNetworkState* [out]
An eHS_NETWORK_STATE enumeration value that indicates the type of network.

**pNetworkProfile* [out, optional]
Pointer to the HS_NETWORK_PROFILE structure for the network.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_NETWORK_IDENTITY

eHS_NETWORK_STATE

HS_NETWORK_PROFILE

HS_PLUGIN_PRE_CONNECT_INIT function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_PRE_CONNECT_INIT** function is called to notify the plugin to initialize its state when a connection to a hotspot network is in progress.

Syntax

ManagedCPlusPlus

```
typedef DWORD (WINAPI *HS_PLUGIN_PRE_CONNECT_INIT)(
  _In_ HS_NETWORK_IDENTITY *pNetworkIdentity
);
```

Parameters

**pNetworkIdentity* [in]
Pointer to the HS_NETWORK_IDENTITY structure for the target network.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_NETWORK_IDENTITY

HS_PLUGIN_QUERY_CELLULAR_EXCEPTION_HOSTS function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_QUERY_CELLULAR_EXCEPTION_HOSTS** function queries the list of hosts that the plugin will need to connect to over cellular as part of its authentication process.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_QUERY_CELLULAR_EXCEPTION_HOSTS)(
  _Inout_ HS_PLUGIN_CELLULAR_EXCEPTION_HOSTS *pExceptionsList
);
```

Parameters

**pExceptionsList* [in, out]
The HS_PLUGIN_CELLULAR_EXCEPTION_HOSTS structure that contains the list of cellular host names.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Remarks

This function is called only if the plugin sets the **dwNumCellularExceptions** field of its HS_PLUGIN_PROFILE to a value greater than zero.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_PLUGIN_CELLULAR_EXCEPTION_HOSTS

HS_PLUGIN_PROFILE

HS_PLUGIN_QUERY_HIDDEN_NETWORK function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_QUERY_HIDDEN_NETWORK** function returns the network identity and network profile for a hidden network.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_QUERY_HIDDEN_NETWORK)(
  _Out_ HS_NETWORK_IDENTITY *pHiddenNetworkIdentity,
  _Out_ HS_NETWORK_PROFILE  *pHiddenNetworkProfile
);
```

Parameters

**pHiddenNetworkIdentity* [out]
The HS_NETWORK_IDENTITY structure that uniquely identifies the network.

**pHiddenNetworkProfile* [out]
The HS_NETWORK_PROFILE structure that contains the network profile.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Remarks

The host calls this function only if the **dwPluginCapabilities** field of the associated plugin's HS_PLUGIN_PROFILE structure includes the **HS_FLAG_CAPABILITY_NETWORK_TYPE_HIDDEN** capability.

The plugin must provide both the network identity and the network profile for the hidden Wi-Fi network.

This network must have the highest priority (1) among all the hotspot networks.

The hotspot offload service imposes a limitation of one hidden network for the life of the service. Therefore, in the case where there are multiple plugins installed, only the first plugin's request to specify a hidden network will be accepted.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_NETWORK_IDENTITY

HS_NETWORK_PROFILE

HS_PLUGIN_PROFILE

HS_PLUGIN_QUERY_SUPPORTED_SIMS function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_QUERY_SUPPORTED_SIMS** function returns the list of SIMs that the plugin supports.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_QUERY_SUPPORTED_SIMS)(
  _In_opt_ HS_NETWORK_IDENTITY      *pNetworkIdentity,
  _Inout_  HS_PLUGIN_SUPPORTED_SIMS *pSupportedSIMs
);
```

Parameters

**pNetworkIdentity* [in, optional]
The HS_NETWORK_IDENTITY structure that uniquely identifies the network.

**pSupportedSIMs* [in, out]
Pointer to an array of one or more HS_PLUGIN_SUPPORTED_SIMS structures that contains the list of supported SIMs.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Remarks

If the *pNetworkIdentity* parameter exists then only those SIM identities required for connecting to the specified network must be provided, otherwise the entire list of SIMs for connecting to all networks must be provided.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_NETWORK_IDENTITY

HS_PLUGIN_SUPPORTED_SIMS

HS_PLUGIN_RESET function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_RESET** function is called by the host to notify the plugin that it must reset its state.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_RESET)(

);
```

Parameters

This function has no parameters.

**

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Remarks

The plugin should terminate all threads and stop any activities in progress.

The plugin is unloaded if it fails to reset.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_PLUGIN_SEND_KEEP_ALIVE function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_SEND_KEEP_ALIVE** function is called by the host to send a network connection keep-alive message. It will be called at the frequency specified in the **dwKeepAliveTimeMins** member of the plugin's HS_PLUGIN_PROFILE structure.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_SEND_KEEP_ALIVE)(

);
```

Parameters

This function has no parameters.

**

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_PLUGIN_PROFILE

HS_PLUGIN_START_POST_CONNECT_AUTH function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_START_POST_CONNECT_AUTH** function is called to perform any post-connect authentication required to authenticate the device over the network.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_START_POST_CONNECT_AUTH)(
  _In_ DWORD                 dwConnectionId,
  _In_ HS_CONNECTION_CONTEXT *pConnectContext,
  _In_ HS_SIM_DATA           *pSIMData,
  _In_ HS_NETWORK_IDENTITY   *pNetworkIdentity,
  _In_ HS_NETWORK_PROFILE    *pNetworkProfile
);
```

Parameters

dwConnectionId [in]
Unique identifier for the network connection.

**pConnectContext* [in]
Pointer to a HS_CONNECTION_CONTEXT structure that contains the information required by the plugin for post-connect authentication.

**pSIMData* [in]
Pointer to a HS_SIM_DATA structure that contains information from the SIM required by the plugin for post-connect authentication.

**pNetworkIdentity* [in]
Pointer to the HS_NETWORK_IDENTITY structure for the network.

**pNetworkProfile* [in]
Pointer to the HS_NETWORK_PROFILE structure that contains the network profile.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Remarks

After calling this function, the plugin must call the HS_HOST_POST_CONNECT_AUTH_COMPLETION handler to inform the host of the status of the request.

If the network uses EAP-SIM/AKA authentication, the plugin is not expected to perform any activity in this state. However, if the network requires HTTP-based authentication, the plugin must perform the appropriate authentication.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_CONNECTION_CONTEXT

HS_SIM_DATA

HS_NETWORK_IDENTITY

HS_NETWORK_PROFILE

HS_PLUGIN_STOP_POST_CONNECT_AUTH function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_PLUGIN_STOP_POST_CONNECT_AUTH** function is called to notify the plugin to stop the authentication process.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_PLUGIN_STOP_POST_CONNECT_AUTH)(
  _In_ HS_NETWORK_IDENTITY *pNetworkIdentity
);
```

Parameters

**pNetworkIdentity* [in]
The HS_NETWORK_IDENTITY structure that uniquely identifies the network.

Return value

This function is called by the host to communicate with the plugin and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_NETWORK_IDENTITY

HS_HOST_ALLOCATE_MEMORY function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_HOST_ALLOCATE_MEMORY** function returns an amount of memory specified by the caller.

Syntax

ManagedCPlusPlus

```
 (WINAPI *HS_HOST_ALLOCATE_MEMORY)(
  _In_   HANDLE                        hPluginContext,
  _In_   DWORD                         dwByteCount,
  _Out_ _bcount (dwByteCount) LPVOID* ppvBuffer
);
```

Parameters

hPluginContext [in]
Context handle for the plugin making the call to this function.

dwByteCount [in]
The amount of memory to allocate.

ppvBuffer [out]
Pointer to the buffer that contains the allocated memory.

Return value

This function is called by the plugin to communicate with the host and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_HOST_FREE_MEMORY function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_HOST_FREE_MEMORY** function frees any memory that was allocated earlier by a call to HS_HOST_ALLOCATE_MEMORY.

Syntax

ManagedCPlusPlus

```
 typedef VOID (WINAPI *HS_HOST_FREE_MEMORY)(
  _In_     HANDLE hPluginContext,
  _In_opt_ LPVOID pvBuffer
);
```

Parameters

hPluginContext [in]
Context handle for the plugin making the call to this function.

pvBuffer [in, optional]
Pointer to the memory buffer.

Return value

This function is called by the plugin to communicate with the host and does not return a value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_HOST_ALLOCATE_MEMORY

HS_HOST_POST_CONNECT_AUTH_COMPLETION function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_HOST_POST_CONNECT_AUTH_COMPLETION** function indicates the success or failure of an authentication attempt following a Wi-Fi connection setup at layer 2.

Syntax

ManagedCPlusPlus

```
  typedef DWORD (WINAPI *HS_HOST_POST_CONNECT_AUTH_COMPLETION)(
   _In_     HANDLE                     hPluginContext,
   _In_     DWORD                      dwConnectionId,
   _In_     eHS_AUTHENTICATION_RESULT  AuthResult,
   _In_opt_ LPVOID                     pvReserved
);
```

Parameters

hPluginContext [in]
Context handle for the plugin making the call to this function.

dwConnectionId [in]
Unique identifier for the network connection.

AuthResult [in]
The eHS_AUTHENTICATION_RESULT enumeration value that indicates success or failure.

pvReserved [in, optional]
Reserved for future use.

Return value

This function is called by the plugin to communicate with the host and does not return a value.

Remarks

The plugin must call this function to inform the host of the result of a previous call to HS_PLUGIN_START_POST_CONNECT_AUTH.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

eHS_AUTHENTICATION_RESULT

HS_PLUGIN_START_POST_CONNECT_AUTH

HS_HOST_SEND_KEEP_ALIVE_COMPLETION function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_HOST_SEND_KEEP_ALIVE_COMPLETION** function indicates the success or failure of a request for a network send keep-alive message.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_HOST_SEND_KEEP_ALIVE_COMPLETION)(
  _In_ HANDLE hPluginContext,
  _In_ DWORD  dwResult
);
```

Parameters

hPluginContext [in]
Context handle for the plugin making the call to this function.

dwResult [in]
The result code.

Return value

This function is called by the plugin to communicate with the host and does not return a value.

Remarks

The plugin must call this function to inform the host of the result of a previous call to HS_PLUGIN_SEND_KEEP_ALIVE.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_PLUGIN_SEND_KEEP_ALIVE

HS_HOST_SEND_USER_MESSAGE function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_HOST_SEND_USER_MESSAGE** function is called to communicate with the user. The message content is contained in custom UI display strings that are passed to the hotspot offload service.

Syntax

ManagedCPlusPlus

```
 typedef DWORD (WINAPI *HS_HOST_SEND_USER_MESSAGE)(
  _In_ HANDLE hPluginContext,
  _In_ DWORD  dwConnectionId,
  _In_ DWORD  dwStringID
);
```

Parameters

hPluginContext [in]
Context handle for the plugin making the call to this function.

dwConnectionId [in]
Unique identifier for the network connection.

dwStringID [in]
The string ID, used as an index into the string table where the message is stored.

Return value

This function is called by the plugin to communicate with the host and does not return a value.

Remarks

The hotspot plugin stores the messages in a string table. The plugin must pass the string IDs to the hotspot offload service to enable it to load the appropriate strings.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

HS_HOST_UPDATE_CONFIGURATION_COMPLETION function

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **HS_HOST_UPDATE_CONFIGURATION_COMPLETION** function indicates the success or failure of a request to check for updates.

Syntax

ManagedCPlusPlus

```
typedef DWORD (WINAPI *HS_HOST_UPDATE_CONFIGURATION_COMPLETION)(
  _In_ HANDLE            hPluginContext,
  _In_ eHS_UPDATE_RESULT UpdateResult
);
```

Parameters

hPluginContext [in]
Context handle for the plugin making the call to this function.

UpdateResult [in]
The eHS_UPDATE_RESULT enumeration value that indicates the result of the request to check for updates.

Return value

This function is called by the plugin to communicate with the host and does not return a value.

Remarks

The plugin must call this function to inform the host of the result of a previous call to HS_PLUGIN_CHECK_FOR_UPDATES.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

eHS_UPDATE_RESULT

HS_PLUGIN_CHECK_FOR_UPDATES

eHS_AUTHENTICATION_RESULT enumeration

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ Important

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **eHS_AUTHENTICATION_RESULT** enumeration indicates the result of authentication by the plugin after the PostConnectAuth request.

Syntax

ManagedCPlusPlus

```
typedef enum _eHS_AUTHENTICATION_RESULT {
  HS_AUTHENTICATION_RESULT_SUCCESS         = 0,
  HS_AUTHENTICATION_RESULT_FAILED_TIMEOUT  = 100,
  HS_AUTHENTICATION_RESULT_FAILED_AUTH,
  HS_AUTHENTICATION_RESULT_FAILED_CONNECT,
  HS_AUTHENTICATION_RESULT_FAILED_OTHER,
  HS_AUTHENTICATION_RESULT_MAX
} eHS_AUTHENTICATION_RESULT;
```

Constants

HS_AUTHENTICATION_RESULT_SUCCESS
Indicates the authentication was successful.

HS_AUTHENTICATION_RESULT_FAILED_TIMEOUT
Indicates the authentication failed due to a timeout from the server/back end.

HS_AUTHENTICATION_RESULT_FAILED_AUTH

Indicates the authentication failed due to incorrect credentials.

HS_AUTHENTICATION_RESULT_FAILED_CONNECT

Indicates the authentication failed due to an inability to connect to the authentication server

HS_AUTHENTICATION_RESULT_FAILED_OTHER

Indicates the authentication failed for some other reason.

HS_AUTHENTICATION_RESULT_MAX

Indicates an out-of-range value.

Remarks

The plugin passes this enumeration value to the hotspot plugin host through the HS_HOST_POST_CONNECT_AUTH_COMPLETION function, which is used to inform the hotspot plugin host of the results of a call to HS_PLUGIN_START_POST_CONNECT_AUTH.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

eHS_NETWORK_STATE enumeration

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **eHS_NETWORK_STATE** enumeration indicates whether a network is a hotspot network.

Syntax

ManagedCPlusPlus

```
typedef enum _eHS_NETWORK_STATE {
  HS_NETWORK_STATE_NOT_A_HOTSPOT,
  HS_NETWORK_STATE_HOTSPOT_MANUAL_CONNECT,
  HS_NETWORK_STATE_HOTSPOT_AUTO_CONNECT,
  HS_NETWORK_STATE_MAX
} eHS_NETWORK_STATE;
```

Constants

HS_NETWORK_STATE_NOT_A_HOTSPOT
Indicates the network is not a hotspot network.

HS_NETWORK_STATE_HOTSPOT_MANUAL_CONNECT
Indicates the user can manually connect to the hotspot network.

HS_NETWORK_STATE_HOTSPOT_AUTO_CONNECT
Indicates the device can connect automatically to the hotspot network.

HS_NETWORK_STATE_MAX

Indicates an out-of-range value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

eHS_UNLOAD_REASON enumeration

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

ⓘ **Important**

Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).

For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **eHS_UNLOAD_REASON** enumeration indicates the reason for the plugin to get unloaded.

Syntax

ManagedCPlusPlus

```
typedef enum _eHS_UNLOAD_REASON {
  HS_UNLOAD_REASON_NONE,
  HS_UNLOAD_REASON_PLUGN_INIT_FAILED,
  HS_UNLOAD_REASON_NO_NETWORKS_SUPPORTED,
  HS_UNLOAD_REASON_NO_PROVIDE_NAME_ID,
  HS_UNLOAD_REASON_ZERO_SIM_COUNT,
  HS_UNLOAD_REASON_DISPLAY_FLAG_BUT_NO_DISPLAY_STRING_ID,
  HS_UNLOAD_REASON_CUSTOM_REALM_FLAG_BUT_NO_REALM_STRING,
  HS_UNLOAD_REASON_DUPLICATE_PLUGIN_LOADED,
  HS_UNLOAD_REASON_RELOAD_REQUESTED_BY_PLUGIN,
  HS_UNLOAD_REASON_EXCEPTION_DURING_PLUGIN_CALL,
  HS_UNLOAD_REASON_EXCEPTION_IN_PLUGIN_HOST,
  HS_UNLOAD_REASON_ASYNC_INITIALIZATION_FAILED,
  HS_UNLOAD_REASON_UNSUPPORTED_AUTH_CAPABILITY_REQUESTED,
  HS_UNLOAD_REASON_FAILED_TO_LOAD_PROVIDER_NAME_STRING,
  HS_UNLOAD_REASON_FAILED_TO_LOAD_ADVANCED_PAGE_STRING,
  HS_UNLOAD_REASON_FAILED_TO_LOAD_NETWORK_NAME_STRING,
  HS_UNLOAD_REASON_FAILED_TO_CONFIGURE_HIDDEN_NETWORK,
  HS_UNLOAD_REASON_HIDDEN_NETWORK_ALREADY_CONFIGURED,
  HS_UNLOAD_REASON_FAILED_TO_QUERY_SIMS,
  HS_UNLOAD_REASON_PLUGIN_REQUIRED_SIM_NOT_PRESENT,
  HS_UNLOAD_REASON_SIM_CONFIG_CHANGED,
  HS_UNLOAD_REASON_WIFI_SWITCHED_OFF_IN_OS,
  HS_UNLOAD_REASON_MAX
} eHS_UNLOAD_REASON;
```

Constants

HS_UNLOAD_REASON_NONE
No specific reason for the unload operation.

HS_UNLOAD_REASON_PLUGN_INIT_FAILED
The plugin is being unloaded because it failed to initialize successfully.

HS_UNLOAD_REASON_NO_NETWORKS_SUPPORTED
The plugin is being unloaded because the plugin's HS_PLUGIN_PROFILE structure did not indicate a valid value for **dwNumNetworksSupported**.

HS_UNLOAD_REASON_NO_PROVIDE_NAME_ID
The plugin is being unloaded because the plugin's HS_PLUGIN_PROFILE structure did not specify a string ID for **dwProviderNameStringID**.

HS_UNLOAD_REASON_ZERO_SIM_COUNT
The plugin is being unloaded because there are no SIM cards present.

HS_UNLOAD_REASON_DISPLAY_FLAG_BUT_NO_DISPLAY_STRING_ID
The plugin is being unloaded because the plugin's HS_PLUGIN_PROFILE structure requires HTTP or EAP SIM-based authentication but did not specify a value for **dwSupportedSIMCount.**

HS_UNLOAD_REASON_CUSTOM_REALM_FLAG_BUT_NO_REALM_STRING
The plugin is being unloaded because the plugin's HS_PLUGIN_PROFILE structure specified the **HS_FLAG_CAPABILITY_NETWORK_CUSTOM_REALM** capability but did not provide a valid string for **szRealm**.

HS_UNLOAD_REASON_DUPLICATE_PLUGIN_LOADED
The plugin is being unloaded because another plugin is using the same DLL.

HS_UNLOAD_REASON_RELOAD_REQUESTED_BY_PLUGIN
The plugin is being unloaded because the plugin requested to be unloaded and reloaded by specifying the **HS_UPDATE_RESULT_ACTION_RELOAD** action with the eHS_UPDATE_RESULT enumeration.

HS_UNLOAD_REASON_EXCEPTION_DURING_PLUGIN_CALL
The plugin is being unloaded because the host process encountered an exception while in a call to the plugin.

HS_UNLOAD_REASON_EXCEPTION_IN_PLUGIN_HOST
The plugin is being unloaded because the hotspot host encountered an exception.

HS_UNLOAD_REASON_ASYNC_INITIALIZATION_FAILED
The plugin is being unloaded because the hotspot service failed to register for notifications from the plugin.

HS_UNLOAD_REASON_UNSUPPORTED_AUTH_CAPABILITY_REQUESTED
The plugin is being unloaded because none of the authentication capabilities requested by the plugin are available.

HS_UNLOAD_REASON_FAILED_TO_LOAD_PROVIDER_NAME_STRING
The plugin is being unloaded because the hotspot service could not map the **dwProviderNameStringID** string ID provided in the plugin's HS_PLUGIN_PROFILE structure to a valid string.

HS_UNLOAD_REASON_FAILED_TO_LOAD_ADVANCED_PAGE_STRING
The plugin is being unloaded because the plugin's HS_PLUGIN_PROFILE structure specified an (optional) **dwAdvancedPageStringID** string ID but it did not map to a valid string.

HS_UNLOAD_REASON_FAILED_TO_LOAD_NETWORK_NAME_STRING
The plugin is being unloaded because the plugin's HS_PLUGIN_PROFILE structure specified an (optional) **dwGenericNetworkNameStringID** string ID, but it did not map to a valid string.

HS_UNLOAD_REASON_FAILED_TO_CONFIGURE_HIDDEN_NETWORK
The plugin is being unloaded because the plugin specified a hidden network (via the **HS_FLAG_CAPABILITY_NETWORK_TYPE_HIDDEN** capability), but the hotspot service was unable to configure the hidden network.

HS_UNLOAD_REASON_HIDDEN_NETWORK_ALREADY_CONFIGURED
The plugin is being unloaded because the plugin specified a hidden network via the **HS_FLAG_CAPABILITY_NETWORK_TYPE_HIDDEN** capability but another plugin has already claimed the hidden network slot.

HS_UNLOAD_REASON_FAILED_TO_QUERY_SIMS
The plugin is being unloaded because the call to HS_PLUGIN_QUERY_SUPPORTED_SIMS failed.

HS_UNLOAD_REASON_PLUGIN_REQUIRED_SIM_NOT_PRESENT
The plugin is being unloaded because the SIMs required by the plugin are not present in the device.

HS_UNLOAD_REASON_SIM_CONFIG_CHANGED
The plugin is being unloaded because the SIM configuration changed, which requires the plugins to be unloaded and reloaded.

HS_UNLOAD_REASON_WIFI_SWITCHED_OFF_IN_OS
The plugin is being unloaded because Wi-Fi functionality was switched off in the OS.

HS_UNLOAD_REASON_MAX
Indicates an out-of-range value.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_PLUGIN_PROFILE

eHS_UPDATE_RESULT

HS_PLUGIN_QUERY_SUPPORTED_SIMS

eHS_UPDATE_RESULT enumeration

Article • 03/03/2023

Wi-Fi Hotspot Offloading deprecation note

> ⓘ **Important**
>
> Starting in Windows 10, version 1709, the Wi-Fi Hotspot Offloading feature is deprecated and should not be used. Instead, we recommend writing a UWP app and using the Wi-Fi Hotspot Authentication API (Windows.Networking.NetworkOperators).
>
> For a code sample and more info, see the **Wi-Fi hotspot authentication sample**.

The **eHS_UPDATE_RESULT** enumeration indicates the result of a “check for updates” request.

Syntax

ManagedCPlusPlus

```
typedef enum _eHS_UPDATE_RESULT {
  HS_UPDATE_RESULT_SUCCESS,
  HS_UPDATE_RESULT_ACTION_RECONNECT,
  HS_UPDATE_RESULT_ACTION_RELOAD,
  HS_UPDATE_RESULT_MAX
} eHS_UPDATE_RESULT;
```

Constants

HS_UPDATE_RESULT_SUCCESS
Indicates the update was successful.

HS_UPDATE_RESULT_ACTION_RECONNECT
The result of the update request requires the service to disconnect and reconnect.

HS_UPDATE_RESULT_ACTION_RELOAD
The result of the update request requires the service to unload and reload the plugin.

HS_UPDATE_RESULT_MAX

Indicates an out-of-range value.

Remarks

The plugin passes this enumeration value to the hotspot plugin host through the HS_HOST_UPDATE_CONFIGURATION_COMPLETION function, which is used to inform the hotspot plugin host of the results of a call to HS_PLUGIN_CHECK_FOR_UPDATES.

Requirements

Version	Windows 10 Mobile
Header	Hotspotoffloadplugin.h (include Hotspotoffloadplugin.h)

See also

HS_HOST_UPDATE_CONFIGURATION_COMPLETION

HS_PLUGIN_CHECK_FOR_UPDATES

Network Module Registrar Topics

Article • 12/15/2021

This section discusses the Network Module Registrar and includes the following topics:

Introduction to the Network Module Registrar

Network Module

Architecture Overview

Initializing and Registering a Client Module

Provider Module Operations

Programming Considerations

Using the **WskRegister** and **WskDeregister** functions is the preferred method for registering and unregistering WSK applications. The Network Module Registrar remains available for compatibility. For more information, see Registering a Winsock Kernel Application.

Introduction to the Network Module Registrar

Article • 12/15/2021

The Network Module Registrar (NMR) is an operating system module that facilitates the attachment of network modules to each other. Each network module registers itself with the NMR, specifying the characteristics that describe the network module. The NMR initiates attachment between pairs of registered network modules that can be attached to each other. After they are attached, the network modules can interact with each other independent of the NMR. The NMR also facilitates detachment of attached pairs of network modules when one of the network modules deregisters with the NMR. The deregistration of a network module is not complete until the network module is completely detached from all network modules to which it was previously attached.

Although the NMR was developed for use by network modules, the design of the NMR architecture is sufficiently generic so that it can also be used by software modules in other technology areas.

Network Module

Article • 12/15/2021

A *network module* is a software module that implements a specific function in a network stack, such as a data link interface, a transport protocol, or a network application. A network module can be a provider module, a client module, or both, depending on where it is located in the network stack.

Provider Module

Article • 12/15/2021

A *provider module* is a network module that supports and implements the provider side of a Network Programming Interface (NPI). A provider module registers itself with the Network Module Registrar as a *Provider* of the NPI that it supports. A provider module can register itself as a provider of more than one NPI. A network module can be both a provider module and a client module.

Client Module

Article • 12/15/2021

A *client module* is a network module that supports and implements the client side of a Network Programming Interface (NPI). A client module registers itself with the Network Module Registrar as a *Client* of the NPI that it supports. A client module can register itself as a client of more than one NPI. A network module can be both a client module and a provider module.

Network Programming Interface

Article • 12/15/2021

A *Network Programming Interface*, or NPI, defines the interface between network modules that can be attached to one another. A client module that is registered as a client of a particular NPI can only be attached to provider modules that are registered as providers of the same NPI. Likewise, a provider module that is registered as a provider of a particular NPI can only be attached to client modules that are registered as clients of the same NPI.

Each NPI defines the following items:

- An *NPI identifier* that uniquely identifies the NPI. A network module specifies an NPI identifier to indicate the particular NPI that it supports when the network module registers itself with the Network Module Registrar (NMR). A network module can support multiple NPIs by registering itself with the NMR multiple times, once for each NPI that it supports. The NMR will initiate attaching a client module to a provider module only if they both support the same NPI.

- An optional *client characteristics* structure that specifies the NPI-specific characteristics of each client module. Such NPI-specific characteristics might include items such as which version (or versions) of the NPI that a client module supports, or which address family or protocol a client module requires. A provider module can use the information contained in a client module's client characteristics structure to determine if it will attach to the client module. If an NPI does not define any NPI-specific client characteristics, then this structure is not required.

- An optional *provider characteristics* structure that specifies the NPI-specific characteristics of each provider module. Such NPI-specific characteristics might include items such as which version (or versions) of the NPI that a provider module supports, or which address families or protocols a provider module supports. A client module can use the information contained in a provider module's client characteristics structure to determine if it will attach to the provider module. If an NPI does not define any NPI-specific provider characteristics, then this structure is not required.

- Zero or more client module callback functions. After a provider module successfully attaches to a client module, the provider module can access the client module's functionality by calling the client module's callback functions.

- One or more provider module functions. After a client module successfully attaches to a provider module, the client module can access the provider module's functionality by calling the provider module's functions.
- A *client dispatch table* structure that contains function pointers to each of the client module callback functions. If an NPI does not define any client module callback functions, then this structure is not required.
- A *provider dispatch table* structure that contains function pointers to each of the provider module functions.

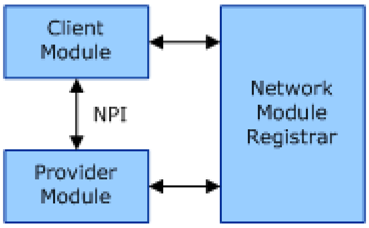
A client module that supports a particular NPI uses the items defined by the NPI to implement the client side of the interface. Similarly, a provider module that supports a particular NPI uses the items defined by the NPI to implement the provider side of the interface.

All of the items defined by an NPI are opaque to the NMR except for the NPI identifier. The NMR uses the NPI identifier to determine which client modules should be attached to which provider modules.

Architecture overview for the Network Module Registrar

Article • 12/15/2021

An overview of the basic architecture of the Network Module Registrar (NMR) is shown in the following diagram:



In this situation, there are two network modules, a client module and a provider module. The client module and the provider module are respectively a client and a provider of the same Network Programming Interface (NPI). Each network module interacts directly with the NMR for the purpose of registration and deregistration, as well as attaching to, and detaching from, other network modules. The NMR will initiate attaching a client module to a provider module only if they both support the same NPI. After the client module and the provider module are attached to each other, they can interact with each other through their NPI functions independent of the NMR.

The following sections provide an overview of the process by which a client module and a provider module that both support a common NPI are attached and detached.

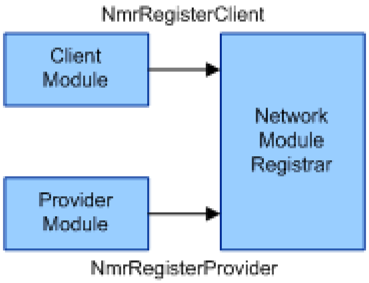
Network Module Attachment

Network Module Detachment

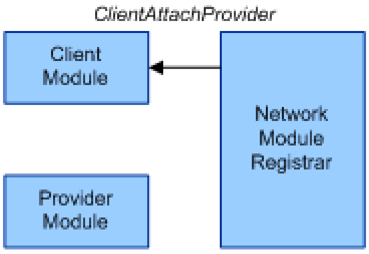
Network Module Attachment

Article • 12/15/2021

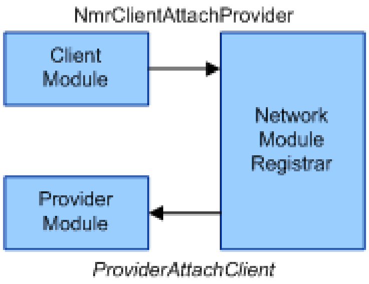
Before a client module and a provider module can be attached to one another, they each must register themselves with the NMR. A client module registers with the NMR by calling the **NmrRegisterClient** function and a provider module registers with the NMR by calling the **NmrRegisterProvider** function. The following diagram illustrates network module registration.



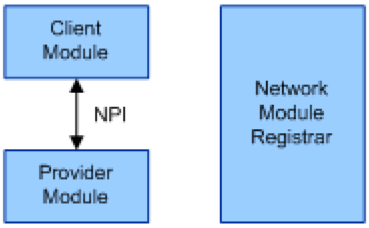
If the client module and the provider module both specify the same Network Programming Interface (NPI) when they register with the NMR, the NMR will initiate attaching the two network modules together. The NMR initiates the attachment process by calling the client module's *ClientAttachProvider* callback function. The following diagram illustrates the Network Module Registrar (NMR) initiating the attachment.



A client module's *ClientAttachProvider* callback function can examine the registration data for the provider module to determine if it will attach to the provider module. If the client module determines that it will attach to the provider module, it continues the attachment process by calling the **NmrClientAttachProvider** function. When a client module calls the **NmrClientAttachProvider** function, the NMR in turn calls the provider module's *ProviderAttachClient* callback function. The following diagram illustrates the client module continuing the attachment.



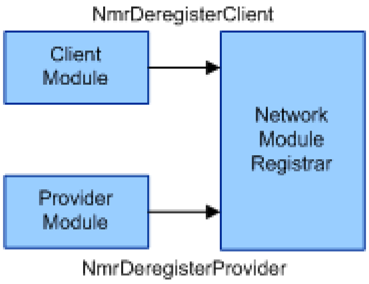
A provider module's *ProviderAttachClient* callback function can examine the registration data for the client module to determine if it will attach to the client module. If the provider module determines that it will attach to the client module, the provider module and client module exchange pointers to their respective NPI dispatch table structures. After the client module and provider module are attached, they can interact with each other through their NPI functions independent of the NMR. The following diagram illustrates the network modules attached.



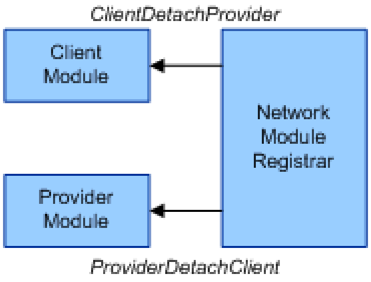
Network Module Detachment

Article • 12/15/2021

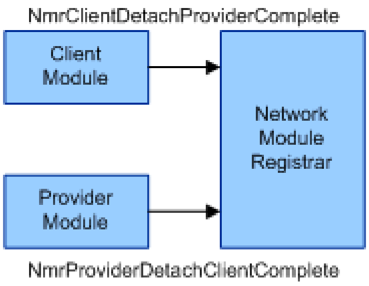
An attached pair of network modules are detached from each other when either the client module or the provider module deregisters with the Network Module Registrar (NMR). A client module deregisters with the NMR by calling the **NmrDeregisterClient** function and a provider module deregisters with the NMR by calling the **NmrDeregisterProvider** function. The following diagram illustrates the network modules initiating deregistration.



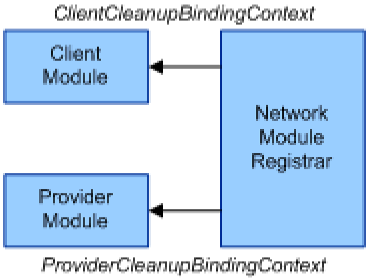
When either network module deregisters with the NMR, the NMR calls both the client module's *ClientDetachProvider* callback function and the provider module's *ProviderDetachClient* callback function to initiate detaching the network module. The following diagram illustrates the NMR initiating the detachment.



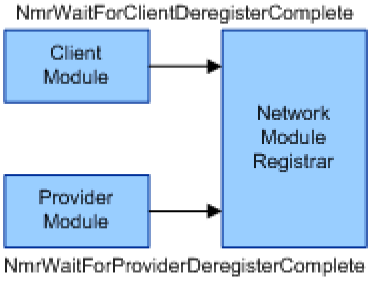
If the client module is unable to detach itself from the provider module immediately, it calls the **NmrClientDetachProviderComplete** function after it completes detaching itself from the provider module. Likewise, if the provider module cannot detach itself from the client module immediately, it calls the **NmrProviderDetachClientComplete** function after it completes detaching itself from the client module. The following diagram illustrates the network modules completing the detachment.


NmrClientDetachProviderComplete
Client Module
Network Module Registrar
Provider Module
NmrProviderDetachClientComplete

After both the client module and the provider module have completed detachment from each other, the NMR calls the client module's *ClientCleanupBindingContext* callback function and the provider module's *ProviderCleanupBindingContext* callback function so that the network modules can clean up their respective binding contexts for the attachment. The following diagram illustrates the NMR initiating cleanup.



If the client module deregistered with the NMR, the deregistration of the client module is not complete until the client module has completely detached from all of the provider modules that it was previously attached to and all of those provider modules have completely detached from the client module. The client module waits for the deregistration to complete by calling the **NmrWaitForClientDeregisterComplete** function. Likewise, if the provider module deregistered with the NMR, the deregistration of the provider module is not complete until the provider module has completely detached from all of the client modules that it was previously attached to and all of those client modules have completely detached from the provider module. The provider module waits for the deregistration to complete by calling the **NmrWaitForProviderDeregisterComplete** function. The following diagram illustrates the network modules waiting for deregistration to complete.


NmrWaitForClientDeregisterComplete
Client Module
Network Module Registrar
Provider Module
NmrWaitForProviderDeregisterComplete

Initializing and Registering a Client Module

Article • 12/15/2021

A client module must initialize a number of data structures before it can register itself with the Network Module Registrar (NMR). These structures include an **NPI_MODULEID** structure, an **NPI_CLIENT_CHARACTERISTICS** structure, an **NPI_REGISTRATION_INSTANCE** structure (contained within the NPI_CLIENT_CHARACTERISTICS structure), and a structure defined by the client module that is used for the client module's registration context.

If a client module registers itself with the NMR as a client of a Network Programming Interface (NPI) that defines NPI-specific client characteristics, the client module must also initialize an instance of the client characteristics structure defined by the NPI.

All of these data structures must remain valid and resident in memory as long as the client module is registered with the NMR.

For example, suppose the "EXNPI" NPI defines the following in header file Exnpi.h:

C++

```
// EXNPI NPI identifier
const NPIID EXNPI_NPIID = { ... };

// EXNPI client characteristics structure
typedef struct EXNPI_CLIENT_CHARACTERISTICS_
{
  .
  . // NPI-specific members
  .
} EXNPI_CLIENT_CHARACTERISTICS, *PEXNPI_CLIENT_CHARACTERISTICS;
```

The following shows how a client module that registers itself as a client of the EXNPI NPI can initialize all of these data structures:

C++

```
// Include the NPI specific header file
#include "exnpi.h"

// Structure for the client module's NPI-specific characteristics
const EXNPI_CLIENT_CHARACTERISTICS NpiSpecificCharacteristics =
{
  .
  . // The NPI-specific characteristics of the client module
  .
};

// Structure for the client module's identification
const NPI_MODULEID ClientModuleId =
{
  sizeof(NPI_MODULEID),
  MIT_GUID,
  { ... }  // A GUID that uniquely identifies the client module
};

// Prototypes for the client module's callback functions
NTSTATUS
  ClientAttachProvider(
    IN HANDLE NmrBindingHandle,
    IN PVOID ClientContext,
    IN PNPI_REGISTRATION_INSTANCE ProviderRegistrationInstance
    );

NTSTATUS
  ClientDetachProvider(
    IN PVOID ClientBindingContext
    );

VOID
  ClientCleanupBindingContext(
    IN PVOID ClientBindingContext
    );

// Structure for the client module's characteristics
const NPI_CLIENT_CHARACTERISTICS ClientCharacteristics =
{
  0,
  sizeof(NPI_CLIENT_CHARACTERISTICS),
  ClientAttachProvider,
  ClientDetachProvider,
  ClientCleanupBindingContext,
  {
    0,
    sizeof(NPI_REGISTRATION_INSTANCE),
    &EXNPI_NPIID,
    &ClientModuleId,
    0,
    &NpiSpecificCharacteristics
  }
};

// Context structure for the client module's registration
typedef struct CLIENT_REGISTRATION_CONTEXT_ {
  .
  . // Client-specific members
  .
} CLIENT_REGISTRATION_CONTEXT, *PCLIENT_REGISTRATION_CONTEXT;
```

```
// Structure for the client's registration context
CLIENT_REGISTRATION_CONTEXT ClientRegistrationContext =
{
  .
  . // Initial values for the registration context
  .
};
```

A client module typically initializes itself within its **DriverEntry** function. The main initialization tasks for a client module are:

- Specify an **Unload** function. The operating system calls this function when the client module is unloaded from the system. If a client module does not provide an unload function, the client module cannot be unloaded from the system.

- Call the **NmrRegisterClient** function to register the client module with the NMR.

For example:

C++

```
// Prototype for the client module's unload function
VOID
  Unload(
    PDRIVER_OBJECT DriverObject
    );

// Variable to contain the handle for the registration
HANDLE ClientHandle;

// DriverEntry function
NTSTATUS
  DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
    )
{
  NTSTATUS Status;

  // Specify the unload function
  DriverObject->DriverUnload = Unload;


  .
  . // Other initialization tasks
  .

  // Register the client module with the NMR
  Status = NmrRegisterClient(
    &ClientCharacteristics,
    &ClientRegistrationContext,
    &ClientHandle
    );
  // Return the result of the registration
  return Status;
}
```

If a client module is a client of more than one NPI, it must initialize an independent set of data structures and call **NmrRegisterClient** for each NPI that it supports. If a network module is both a client module and a provider module (that is, it is a client of one NPI and a provider of another NPI), it must initialize two independent sets of data structures, one for the client interface and one for the provider interface, and call both **NmrRegisterClient** and **NmrRegisterProvider**.

A client module is not required to call **NmrRegisterClient** from within its **DriverEntry** function. For example, in the situation where a client module is a subcomponent of a complex driver, the registration of the client module might occur only when the client module subcomponent is activated.

For more information about implementing a client module's **Unload** function, see Unloading a Client Module.

Attaching a Client Module to a Provider Module

Article • 12/15/2021

After a client module has registered with the Network Module Registrar (NMR), the NMR calls the client module's *ClientAttachProvider* callback function, once for each provider module that is registered as a provider of the same Network Programming Interface (NPI) for which the client module has registered as a client.

The NMR also calls a client module's *ClientAttachProvider* callback function whenever a new provider module registers as a provider of the same NPI for which the client module has registered as a client.

When the NMR calls the client module's *ClientAttachProvider* callback function for a particular provider module, it passes, in the *ProviderRegistrationInstance* parameter, a pointer to the **NPI_REGISTRATION_INSTANCE** structure that is associated with the provider module. The client module's *ClientAttachProvider* callback function can use the data in the provider module's **NPI_REGISTRATION_INSTANCE** structure, as well as the data in the **NPI_MODULEID** structure and the NPI-specific provider characteristics structure pointed to by the **ModuleId** and **NpiSpecificCharacteristics** members of the provider module's **NPI_REGISTRATION_INSTANCE** structure, to determine if it will attach to the provider module.

If the client module determines that it will attach to the provider module, the client module's *ClientAttachProvider* callback function allocates and initializes a binding context structure for the attachment to the provider module and then calls the **NmrClientAttachProvider** function to continue the attachment process. In this situation, the client module's *ClientAttachProvider* callback function must return the status code that is returned by the **NmrClientAttachProvider** function.

If **NmrClientAttachProvider** returns STATUS_SUCCESS, the client module and the provider module have successfully attached to each other. In this situation, the client module's *ClientAttachProvider* callback function must save the binding handle that the NMR passed in the *NmrBindingHandle* parameter when the NMR called the client module's *ClientAttachProvider* callback function. The client module's *ClientAttachProvider* callback function must also save the pointers to the provider binding context and the provider dispatch table that are returned in the variables that the client module passed to the **NmrClientAttachProvider** function in the *ProviderBindingContext* and *ProviderDispatch* parameters. A client module typically saves this data in its binding context for the attachment to the provider module.

If **NmrClientAttachProvider** does not return STATUS_SUCCESS, the client module's *ClientAttachProvider* callback function should clean up and free any resources that it allocated before it called **NmrClientAttachProvider**.

If the client module determines that it will not attach to the provider module, then the client module's *ClientAttachProvider* callback function must return STATUS_NOINTERFACE.

For example, suppose the "EXNPI" Network Programming Interface (NPI) defines the following in header file Exnpi.h:

C++

```
// EXNPI provider characteristics structure
typedef struct EXNPI_PROVIDER_CHARACTERISTICS_
{
  .
  . // NPI-specific members
  .
} EXNPI_PROVIDER_CHARACTERISTICS, *PEXNPI_PROVIDER_CHARACTERISTICS;

// EXNPI client dispatch table
typedef struct EXNPI_CLIENT_DISPATCH_ {
  .
  . // NPI-specific dispatch table of function pointers that
  . // point to a client module's NPI callback functions.
  .
} EXNPI_CLIENT_DISPATCH, *PEXNPI_CLIENT_DISPATCH;

// EXNPI provider dispatch table
typedef struct EXNPI_PROVIDER_DISPATCH_ {
  .
  . // NPI-specific dispatch table of function pointers that
  . // point to a provider module's NPI functions.
  .
} EXNPI_PROVIDER_DISPATCH, *PEXNPI_PROVIDER_DISPATCH;
```

The following code example shows how a client module that is registered as a client of the EXNPI NPI can attach itself to a provider module that is registered as a provider of the EXNPI NPI:

C++

```
// Context structure for the client
// module's binding to a provider module
typedef struct CLIENT_BINDING_CONTEXT_ {
  HANDLE NmrBindingHandle;
  PVOID ProviderBindingContext;
  PEXNPI_PROVIDER_DISPATCH ProviderDispatch;
  .
  . // Other client-specific members
  .
} CLIENT_BINDING_CONTEXT, *PCLIENT_BINDING_CONTEXT;

// Pool tag used for allocating the binding context
#define BINDING_CONTEXT_POOL_TAG 'tpcb'

// Structure for the client's dispatch table
const EXNPI_CLIENT_DISPATCH Dispatch = {
  .
  . // Function pointers to the client module's
  . // NPI callback functions
  .
};

// ClientAttachProvider callback function
NTSTATUS
  ClientAttachProvider(
    IN HANDLE NmrBindingHandle,
    IN PVOID ClientContext,
    IN PNPI_REGISTRATION_INSTANCE ProviderRegistrationInstance
    )
{
  PNPI_MODULEID ProviderModuleId;
  PEXNPI_PROVIDER_CHARACTERISTICS ProviderNpiSpecificCharacteristics;
  PCLIENT_BINDING_CONTEXT BindingContext;
  PVOID ProviderBindingContext;
  PEXNPI_PROVIDER_DISPATCH ProviderDispatch;
  NTSTATUS Status;

  // Get pointers to the provider module's identification structure
  // and the provider module's NPI-specific characteristics structure
  ProviderModuleId = ProviderRegistrationInstance->ModuleId;
  ProviderNpiSpecificCharacteristics =
    (PEXNPI_PROVIDER_CHARACTERISTICS)
      ProviderRegistrationInstance->NpiSpecificCharacteristics;

  //
  // Use the data in the structures pointed to by
  // ProviderRegistrationInstance, ProviderModuleId,
  // and ProviderNpiSpecificCharacteristics to determine
  // whether to attach to the provider module.
  //

  // If the client module determines that it will not attach
  // to the provider module
  if (...)
  {
    // Return status code indicating the modules did not
    // attach to each other
    return STATUS_NOINTERFACE;
  }

  // Allocate memory for the client module's
  // binding context structure
  BindingContext =
    (PCLIENT_BINDING_CONTEXT)
      ExAllocatePoolWithTag(
        NonPagedPool,
        sizeof(CLIENT_BINDING_CONTEXT),
        BINDING_CONTEXT_POOL_TAG
        );

  // Check result of allocation
  if (BindingContext == NULL)
  {
    // Return error status code
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Initialize the client binding context structure
  ...

  // Continue with the attachment to the provider module
  Status = NmrClientAttachProvider(
    NmrBindingHandle,
    BindingContext,
    &Dispatch,
    &ProviderBindingContext,
    &ProviderDispatch
    );

  // Check result of attachment
  if (Status == STATUS_SUCCESS)
  {
    // Save NmrBindingHandle, ProviderBindingContext,
    // and ProviderDispatch for future reference
    BindingContext->NmrBindingHandle =
      NmrBindingHandle;
    BindingContext->ProviderBindingContext =
      ProviderBindingContext;
    BindingContext->ProviderDispatch =
      ProviderDispatch;
  }

  // Attachment did not succeed
  else
  {
    // Free memory for client's binding context structure
    ExFreePoolWithTag(
      BindingContext,
      BINDING_CONTEXT_POOL_TAG
      );
  }

  // Return result of attachment
  return Status;
}
```

Managing Multiple Attached Provider Modules

Article • 05/03/2023

A single client module can attach to more than one provider module. In order to manage multiple attached provider modules, a client module must independently save the binding handle, the provider module's binding context, and the provider module's dispatch table for each provider module to which it is attached. Typically this data is saved in the client module's binding context for each attachment. However, a client module can manage the data for each attached provider module in whatever way it chooses.

A Network Programming Interface (NPI) typically defines the client module callback functions such that they include either a pointer to the client module's binding context or some other NPI-specific identifier as one of the function parameters. As a result, a client module can determine which provider module is the caller when one of its NPI callback functions is called.

Detaching a Client Module from a Provider Module

Article • 12/15/2021

When a client module deregisters with the Network Module Registrar (NMR) by calling the **NmrDeregisterClient** function, the NMR calls the client module's *ClientDetachProvider* callback function, once for each provider module to which it is attached, so that the client module can detach itself from all of the provider modules as part of the client module's deregistration process.

Furthermore, whenever a provider module to which the client module is attached deregisters with the NMR by calling the **NmrDeregisterProvider** function, the NMR also calls the client module's *ClientDetachProvider* callback function so that the client module can detach itself from the provider module as part of the provider module's deregistration process.

After its *ClientDetachProvider* callback function has been called, a client module must not make any further calls to any of the provider module's Network Programming Interface (NPI) functions. If there are no in-progress calls to any of the provider module's NPI functions when the client module's *ClientDetachProvider* callback function is called, then the *ClientDetachProvider* callback function should return STATUS_SUCCESS.

If there are in-progress calls to one or more of the provider module's NPI functions when the client module's *ClientDetachProvider* callback function is called, then the *ClientDetachProvider* callback function should return STATUS_PENDING. In this case, the client module must call the **NmrClientDetachProviderComplete** function after all in-progress calls to the provider module's NPI functions have completed. The call to **NmrClientDetachProviderComplete** notifies the NMR that detachment of the client module from the provider module is complete.

For more information about how to track the number of in-progress calls to a provider module's NPI functions, see Programming Considerations.

If a client module implements a *ClientCleanupBindingContext* callback function, the NMR calls the client module's *ClientCleanupBindingContext* callback function after both the client module and the provider module have completed detachment from each other. A client module's *ClientCleanupBindingContext* callback function should perform any necessary cleanup of the data contained within the client module's binding context structure. It should then free the memory for the binding context structure if the client module dynamically allocated the memory for the structure.

For example:

C++

```cpp
// ClientDetachProvider callback function
NTSTATUS
  ClientDetachProvider(
    IN PVOID ClientBindingContext
    )
{
  PCLIENT_BINDING_CONTEXT BindingContext;

  // Get a pointer to the binding context
  BindingContext = (PCLIENT_BINDING_CONTEXT)ClientBindingContext;

  // Set a flag indicating that the client module is detaching
  // from the provider module so that no more calls are made to
  // the provider module's NPI functions.
  ...

  // Check if there are no in-progress NPI function calls to the
  // provider module
  if (...)
  {
    // Return success status indicating detachment is complete
    return STATUS_SUCCESS;
  }

  // There are one or more in-progress NPI function calls
  // to the provider module
  else
  {
    // Return pending status indicating detachment is pending
    // completion of the in-progress NPI function calls
    return STATUS_PENDING;

    // When the last in-progress call to the provider module's
    // NPI functions completes, the client module must call
    // NmrClientDetachProviderComplete() with the binding handle
    // for the attachment to the provider module.
  }
}

// ClientCleanupBindingContext callback function
VOID
  ClientCleanupBindingContext(
    IN PVOID ClientBindingContext
    )
{
  PCLIENT_BINDING_CONTEXT BindingContext;

  // Get a pointer to the binding context
  BindingContext = (PCLIENT_BINDING_CONTEXT)ClientBindingContext;
  // Clean up the client binding context structure
  ...

  // Free the memory for client's binding context structure
  ExFreePoolWithTag(
    BindingContext,
    BINDING_CONTEXT_POOL_TAG
    );
}
```

Unloading a Client Module

Article • 12/15/2021

To unload a client module, the operating system calls the client module's **Unload** function. See Initializing and Registering a Client Module for more information about how to specify a client module's **Unload** function during initialization.

A client module's **Unload** function ensures that the client module is deregistered from the Network Module Registrar (NMR) before the client module is unloaded from system memory. A client module initiates deregistration from the NMR by calling the NmrDeregisterClient function, which it typically calls from its **Unload** function. A client module must not return from its **Unload** function until after it has been completely deregistered from the NMR. If the call to **NmrDeregisterClient** returns STATUS_PENDING, the client module must call the NmrWaitForClientDeregisterComplete function to wait for the deregistration to complete before it returns from its **Unload** function.

For example:

C++

```
// Variable containing the handle for the registration
HANDLE ClientHandle;

// Unload function
VOID
  Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
  NTSTATUS Status;

  // Deregister the client module from the NMR
  Status =
    NmrDeregisterClient(
      ClientHandle
      );

  // Check if pending
  if (Status == STATUS_PENDING)
  {
    // Wait for the deregistration to be completed
    NmrWaitForClientDeregisterComplete(
      ClientHandle
      );
  }

  // An error occurred
```

```
    else
    {
      // Handle error
      ...
    }
  }
```

If a client module is registered as a client of multiple Network Programming Interfaces (NPIs), it must call **NmrDeregisterClient** for each NPI that it supports. If a network module is registered as both a client module and a provider module (that is, it is a client of one NPI and a provider of another NPI), it must call both **NmrDeregisterClient** and **NmrDeregisterProvider**.

A network module must wait until all of the deregistrations are complete before returning from its **Unload** function.

A client module is not required to call **NmrDeregisterClient** from within its **Unload** function. For example, in the situation where a client module is a subcomponent of a complex driver, the deregistration of the client module might occur when the client module subcomponent is deactivated. However, in such a situation the driver must still ensure that the client module has been completely deregistered from the NMR before returning from its **Unload** function.

Initializing and Registering a Provider Module

Article • 12/15/2021

A provider module must initialize a number of data structures before it can register itself with the Network Module Registrar (NMR). These structures include an NPI_MODULEID structure, an NPI_PROVIDER_CHARACTERISTICS structure, an NPI_REGISTRATION_INSTANCE structure (contained within the NPI_PROVIDER_CHARACTERISTICS structure), and a structure defined by the provider module that is used for the provider module's registration context.

If a provider module registers itself with the NMR as a provider of a Network Programming Interface (NPI) that defines NPI-specific provider characteristics, the provider module must also initialize an instance of the provider characteristics structure that are defined by the NPI.

All of these data structures must remain valid and resident in memory as long as the provider module is registered with the NMR.

For example, suppose the "EXNPI" NPI defines the following in header file Exnpi.h:

C++

```
// EXNPI NPI identifier
const NPIID EXNPI_NPIID = { ... };

// EXNPI provider characteristics structure
typedef struct EXNPI_PROVIDER_CHARACTERISTICS_
{
  .
  . // NPI-specific members
  .
} EXNPI_PROVIDER_CHARACTERISTICS, *PEXNPI_PROVIDER_CHARACTERISTICS;
```

The following shows how a provider module that registers itself as a provider of the EXNPI NPI can initialize all of these data structures:

C++

```
// Include the NPI specific header file
#include "exnpi.h"

// Structure for the provider module's NPI-specific characteristics
const EXNPI_PROVIDER_CHARACTERISTICS NpiSpecificCharacteristics =
{
  .
  . // The NPI-specific characteristics of the provider module
  .
};

// Structure for the provider module's identification
const NPI_MODULEID ProviderModuleId =
{
  sizeof(NPI_MODULEID),
  MIT_GUID,
  { ... }  // A GUID that uniquely identifies the provider module
};

// Prototypes for the provider module's callback functions
NTSTATUS
  ProviderAttachClient(
    IN HANDLE NmrBindingHandle,
    IN PVOID ProviderContext,
    IN PNPI_REGISTRATION_INSTANCE ClientRegistrationInstance,
    IN PVOID ClientBindingContext,
    IN CONST VOID *ClientDispatch,
    OUT PVOID *ProviderBindingContext,
    OUT PVOID *ProviderDispatch
    );

NTSTATUS
  ProviderDetachClient(
    IN PVOID ProviderBindingContext
    );

VOID
  ProviderCleanupBindingContext(
    IN PVOID ProviderBindingContext
    );

// Structure for the provider module's characteristics
const NPI_PROVIDER_CHARACTERISTICS ProviderCharacteristics =
{
  0,
  sizeof(NPI_PROVIDER_CHARACTERISTICS),
  ProviderAttachClient,
  ProviderDetachClient,
  ProviderCleanupBindingContext,
  {
    0,
    sizeof(NPI_REGISTRATION_INSTANCE),
    &EXNPI_NPIID,
    &ProviderModuleId,
    0,
    &NpiSpecificCharacteristics
  }
};

// Context structure for the provider module's registration
typedef struct PROVIDER_REGISTRATION_CONTEXT_ {
  .
  . // Provider-specific members
  .
} PROVIDER_REGISTRATION_CONTEXT, *PPROVIDER_REGISTRATION_CONTEXT;

// Structure for the provider's registration context
PROVIDER_REGISTRATION_CONTEXT ProviderRegistrationContext =
{
  .
  . // Initial values for the registration context
  .
};
```

A provider module typically initializes itself within its DriverEntry function. The main initialization tasks for a provider module are:

- Specify an Unload function. The operating system calls this function when the provider module is unloaded from the system. If a provider module does not provide an unload function, the provider module cannot be unloaded from the system.
- Call the NmrRegisterProvider function to register the provider module with the NMR.

For example:

C++

```
// Prototype for the provider module's unload function
VOID
  Unload(
    PDRIVER_OBJECT DriverObject
   );

// Variable to contain the handle for the registration
HANDLE ProviderHandle;

// DriverEntry function
NTSTATUS
  DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
    )
{
  NTSTATUS Status;

  // Specify the unload function
  DriverObject->DriverUnload = Unload;

  .
  . // Other initialization tasks
```

```
    .

    // Register the provider module with the NMR
    Status = NmrRegisterProvider(
      &ProviderCharacteristics,
      &ProviderRegistrationContext,
      &ProviderHandle
      );

    // Return the result of the registration
    return Status;
  }
```

If a provider module is a provider of more than one NPI, it must initialize an independent set of data structures and call **NmrRegisterProvider** for each NPI that it supports. If a network module is both a provider module and a client module (that is, it is a provider of one NPI and a client of another NPI), it must initialize two independent sets of data structures, one for the provider interface and one for the client interface, and call both **NmrRegisterProvider** and **NmrRegisterClient**.

A provider module is not required to call **NmrRegisterProvider** from within its **DriverEntry** function. For example, in the situation where a provider module is a subcomponent of a complex driver, the registration of the provider module might occur only when the provider module subcomponent is activated.

For more information about implementing a provider module's **Unload** function, see Unloading a Provider Module.

Attaching a Provider Module to a Client Module

Article • 12/15/2021

A client module calls the **NmrClientAttachProvider** function to attach itself to a provider module. For more information about how a client module attaches to a provider module, see Attaching a Client Module to a Provider Module.

When a client module calls **NmrClientAttachProvider**, the NMR calls the provider module's *ProviderAttachClient* callback function. When the NMR calls a provider module's *ProviderAttachClient* callback function, it passes, in the *ClientRegistrationInstance* parameter, a pointer to the **NPI_REGISTRATION_INSTANCE** structure that is associated with the client module that called **NmrClientAttachProvider**. The provider module's *ProviderAttachClient* callback function can use the data in the client module's **NPI_REGISTRATION_INSTANCE** structure, the data in the **NPI_MODULEID** structure and the Network Programming Interface (NPI)-specific client characteristics structure pointed to by the **ModuleId** and **NpiSpecificCharacteristics** members of the client module's **NPI_REGISTRATION_INSTANCE** structure, to determine if it will attach to the client module.

If the provider module determines that it will attach to the client module, then the provider module's *ProviderAttachClient* callback function must do the following:

- Allocate and initialize a binding context structure for the attachment to the client module.

- Save the binding handle passed in the *NmrBindingHandle* parameter.

- Save the pointers passed in the *ClientBindingContext* and *ClientDispatch* parameters so that the provider module can make calls to the client module's NPI callback functions.

- Set the variable pointed to by the *ProviderBindingContext* parameter to point to the provider module's binding context structure.

- Set the variable pointed to by the *ProviderDispatch* parameter to point to a structure that contains the provider module's dispatch table of NPI functions.

- Return STATUS_SUCCESS.

A provider module typically saves the binding handle, the pointer to the client binding context, and the pointer to the client dispatch structure in its binding context for the attachment to the client module.

If a provider module's *ProviderAttachClient* callback function returns STATUS_SUCCESS, the client module and the provider module have successfully attached to each other.

If the provider module determines that it will not attach to the client module, then the provider module's *ProviderAttachClient* callback function must return STATUS_NOINTERFACE.

For example, suppose the "EXNPI" Network Programming Interface (NPI) defines the following in header file Exnpi.h:

C++

```cpp
// EXNPI client characteristics structure
typedef struct EXNPI_CLIENT_CHARACTERISTICS_
{
  .
  . // NPI-specific members
  .
} EXNPI_CLIENT_CHARACTERISTICS, *PEXNPI_CLIENT_CHARACTERISTICS;

// EXNPI client dispatch table
typedef struct EXNPI_CLIENT_DISPATCH_ {
  .
  . // NPI-specific dispatch table of function pointers that
  . // point to a client module's NPI callback functions.
  .
} EXNPI_CLIENT_DISPATCH, *PEXNPI_CLIENT_DISPATCH;

// EXNPI provider dispatch table
typedef struct EXNPI_PROVIDER_DISPATCH_ {
  .
  . // NPI-specific dispatch table of function pointers that
  . // point to a provider module's NPI functions.
  .
} EXNPI_PROVIDER_DISPATCH, *PEXNPI_PROVIDER_DISPATCH;
```

The following code example shows how a provider module that is registered as a provider of the EXNPI NPI can attach itself to a client module that is registered as a client of the EXNPI NPI:

C++

```cpp
// Context structure for the provider
// module's binding to a client module
typedef struct PROVIDER_BINDING_CONTEXT_ {
  HANDLE NmrBindingHandle;
  PVOID ClientBindingContext;
  PEXNPI_CLIENT_DISPATCH ClientDispatch;
  .
  . // Other provider-specific members
  .
} PROVIDER_BINDING_CONTEXT, *PPROVIDER_BINDING_CONTEXT;

// Pool tag used for allocating the binding context
#define BINDING_CONTEXT_POOL_TAG 'tpcb'

// Structure for the provider's dispatch table
const EXNPI_PROVIDER_DISPATCH Dispatch = {
  .
  . // Function pointers to the provider
  . // module's NPI functions
  .
};

// ProviderAttachClient callback function
NTSTATUS
  ProviderAttachClient(
    IN HANDLE NmrBindingHandle,
    IN PVOID ProviderContext,
    IN PNPI_REGISTRATION_INSTANCE ClientRegistrationInstance,
    IN PVOID ClientBindingContext,
    IN CONST VOID *ClientDispatch,
    OUT PVOID *ProviderBindingContext,
    OUT PVOID *ProviderDispatch
    )
{
  PNPI_MODULEID ClientModuleId;
  PEXNPI_CLIENT_CHARACTERISTICS ClientNpiSpecificCharacteristics;
  PPROVIDER_BINDING_CONTEXT BindingContext;
  PVOID ClientBindingContext;
  PEXNPI_CLIENT_DISPATCH ClientDispatch;

  // Check parameters
  if ((ProviderBindingContext == NULL) ||
      (ProviderDispatch == NULL))
  {
    // Return error status code
    return STATUS_INVALID_PARAMETER;
  }

  // Get pointers to the client module's identification structure
  // and the client module's NPI-specific characteristics structure
  ClientModuleId = ClientRegistrationInstance->ModuleId;
  ClientNpiSpecificCharacteristics =
    (PEXNPI_CLIENT_CHARACTERISTICS)
      ProviderRegistrationInstance->NpiSpecificCharacteristics;

  //
  // Use the data in the structures pointed to by
  // ClientRegistrationInstance, ClientModuleId,
  // and ClientNpiSpecificCharacteristics to determine
  // whether to attach to the client module.
  //
  // If the provider module determines that it will not attach
  // to the client module
  if (...)
  {
    // Return status code indicating the modules did not
    // attach to each other
    return STATUS_NOINTERFACE;
  }

  // Allocate memory for the provider module's
  // binding context structure
  BindingContext =
    (PPROVIDER_BINDING_CONTEXT)
      ExAllocatePoolWithTag(
        NonPagedPool,
        sizeof(PROVIDER_BINDING_CONTEXT),
        BINDING_CONTEXT_POOL_TAG
        );

  // Check result of allocation
  if (BindingContext == NULL)
  {
    // Return error status code
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Initialize the provider binding context structure
  ...

  // Save NmrBindingHandle, ClientBindingContext,
  // and ClientDispatch for future reference
  BindingContext->NmrBindingHandle =
    NmrBindingHandle;
  BindingContext->ClientBindingContext =
    ClientBindingContext;
  BindingContext->ClientDispatch =
    ClientDispatch;

  // Return a pointer to the provider binding context structure
  // in the ProviderBindingContext parameter
  *ProviderBindingContext = BindingContext;

  // Return a pointer to the provider dispatch structure
  // in the ProviderDispatch parameter
  *ProviderDispatch = &Dispatch;

  // Return success status
  return STATUS_SUCCESS;
}
```

Managing Multiple Attached Client Modules

Article • 12/15/2021

A single provider module can attach to more than one client module. In order to manage multiple attached client modules, a provider module must independently save the binding handle, the client module's binding context, and the client module's dispatch table for each client module to which it is attached. Typically this data is saved in the provider module's binding context for each attachment. However, a provider module can manage the data for each attached client module in whatever way it chooses.

A Network Programming Interface (NPI) typically defines the provider module functions such that they include either a pointer to the provider module's binding context or some other NPI-specific identifier as one of the function parameters. As a result, a provider module can determine which client module is the caller when one of it's NPI functions is called.

Detaching a Provider Module from a Client Module

Article • 12/15/2021

When a provider module deregisters with the Network Module Registrar (NMR) by calling the **NmrDeregisterProvider** function, the NMR calls the provider module's *ProviderDetachClient* callback function, once for each client module to which it is attached, so that the provider module can detach itself from all of the client modules as part of the provider module's deregistration process.

Furthermore, whenever a client module to which the provider module is attached deregisters with the NMR by calling the **NmrDeregisterClient** function, the NMR also calls the provider module's *ProviderDetachClient* callback function so that the provider module can detach itself from the client module as part of the client module's deregistration process.

After its *ProviderDetachClient* callback function has been called, a provider module must not make any further calls to any of the client module's Network Programming Interface (NPI) callback functions. If there are no in-progress calls to any of the client module's NPI callback functions when the provider module's *ProviderDetachClient* callback function is called, then the *ProviderDetachClient* callback function should return STATUS_SUCCESS.

If there are in-progress calls to one or more of the client module's NPI callback functions when the provider module's *ProviderDetachClient* callback function is called, then the *ProviderDetachClient* callback function should return STATUS_PENDING. In this case, the provider module must call the **NmrProviderDetachClientComplete** function after all in-progress calls to the client module's NPI callback functions have completed. The call to **NmrProviderDetachClientComplete** notifies the NMR that detachment of the provider module from the client module is complete.

For more information about how to track the number of in-progress calls to a client module's NPI callback functions, see Programming Considerations.

If a provider module implements a *ProviderCleanupBindingContext* callback function, the NMR calls the provider module's *ProviderCleanupBindingContext* callback function after both the provider module and the client module have completed detachment from each other. A provider module's *ProviderCleanupBindingContext* callback function should perform any necessary cleanup of the data contained within the provider module's binding context structure. It should then free the memory for the binding context structure if the provider module dynamically allocated the memory for the structure.

For example:

C++

```
// ProviderDetachClient callback function
NTSTATUS
  ProviderDetachClient(
    IN PVOID ProviderBindingContext
    )
{
  PPROVIDER_BINDING_CONTEXT BindingContext;

  // Get a pointer to the binding context
  BindingContext = (PPROVIDER_BINDING_CONTEXT)ProviderBindingContext;

  // Set a flag indicating that the provider module is detaching
  // from the client module so that no more calls are made to
  // the client module's NPI callback functions.
  ...

  // Check if there are no in-progress NPI callback function calls
  // to the client module
  if (...)
  {
    // Return success status indicating detachment is complete
    return STATUS_SUCCESS;
  }

  // There are one or more in-progress NPI callback function
  // calls to the client module
  else
  {
    // Return pending status indicating detachment is pending
    // completion of the in-progress NPI callback function calls
    return STATUS_PENDING;

    // When the last in-progress call to the client module's
    // NPI callback functions completes, the provider module
    // must call NmrProviderDetachClientComplete() with the
    // binding handle for the attachment to the client module.
  }
}

// ProviderCleanupBindingContext callback function
VOID
  ProviderCleanupBindingContext(
    IN PVOID ProviderBindingContext
    )
{
  PPROVIDER_BINDING_CONTEXT BindingContext;

  // Get a pointer to the binding context
  BindingContext = (PPROVIDER_BINDING_CONTEXT)ProviderBindingContext;
  // Clean up the provider binding context structure
  ...

  // Free the memory for provider's binding context structure
  ExFreePoolWithTag(
    BindingContext,
    BINDING_CONTEXT_POOL_TAG
    );
}
```

Unloading a Provider Module

Article • 12/15/2021

To unload a provider module, the operating system calls the provider module's **Unload** function. See Initializing and Registering a Provider Module for more information about how to specify a provider module's **Unload** function during initialization.

A provider module's **Unload** function ensures that the provider module is deregistered from the Network Module Registrar (NMR) before the provider module is unloaded from system memory. A provider module initiates deregistration from the NMR by calling the **NmrDeregisterProvider** function, which it typically calls from its **Unload** function. A provider module must not return from its **Unload** function until after it has been completely deregistered from the NMR. If the call to **NmrDeregisterProvider** returns STATUS_PENDING, the provider module must call the **NmrWaitForProviderDeregisterComplete** function to wait for the deregistration to complete before it returns from its **Unload** function.

For example:

```C++
// Variable containing the handle for the registration
HANDLE ProviderHandle;

// Unload function
VOID
  Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
  NTSTATUS Status;

  // Deregister the provider module from the NMR
  Status =
    NmrDeregisterProvider(
      ProviderHandle
      );

  // Check if pending
  if (Status == STATUS_PENDING)
  {
    // Wait for the deregistration to be completed
    NmrWaitForProviderDeregisterComplete(
      ProviderHandle
      );
  }

  // An error occurred
```

```
    else
    {
      // Handle error
      ...
    }
  }
```

If a provider module is registered as a provider of multiple Network Programming Interfaces (NPIs), it must call **NmrDeregisterProvider** for each NPI that it supports. If a network module is registered as both a provider module and a client module (that is, it is a provider of one NPI and a client of another NPI), it must call both **NmrDeregisterProvider** and **NmrDeregisterClient**.

A network module must wait until all of the deregistrations are complete before returning from its **Unload** function.

A provider module is not required to call **NmrDeregisterProvider** from within its **Unload** function. For example, in the situation where a provider module is a subcomponent of a complex driver, the deregistration of the provider module might occur when the provider module subcomponent is deactivated. However, in such a situation the driver must still ensure that the provider module has been completely deregistered from the NMR before returning from its **Unload** function.

Programming Considerations

Article • 12/15/2021

A network module should use some form of reference counting to keep track of the number of in-progress calls to an attached network module's Network Programming Interface (NPI) functions. This will facilitate detaching from an attached network module when one of the two network modules deregisters with the NMR. A network module cannot complete detachment until there are no in-progress calls to the attached network module's NPI functions. A network module must also ensure that no more calls to the previously attached network module will be initiated once the network modules are detached.

For example, a client module might use an implementation similar to the following for tracking the number of in-progress calls to an attached provider module's NPI functions:

C++

```cpp
// Context structure for the client's binding to a provider module
typedef struct CLIENT_BINDING_CONTEXT_ {
  LIST_ENTRY Link;
  HANDLE NmrBindingHandle;
  PVOID ProviderBindingContext;
  PEXNPI_PROVIDER_DISPATCH ProviderDispatch;
  KSPIN_LOCK DetachLock;
  LONG InProgressCallCount;
  LONG Detaching;
  .
  . // Other client-specific members
  .
} CLIENT_BINDING_CONTEXT, *PCLIENT_BINDING_CONTEXT;

// Pool tag used for allocating the binding context
#define BINDING_CONTEXT_POOL_TAG 'tpcb'

// Structure for the client's dispatch table
const EXNPI_CLIENT_DISPATCH Dispatch = {
  .
  . // Function pointers to the client module's
  . // NPI callback functions
  .
};

// Head of linked list of binding context structures
LIST_ENTRY BindingContextList;

// Spin lock for binding context list
KSPIN_LOCK BindingContextListLock;

// Prototype for the client module's unload function
```

```
VOID
  Unload(
    PDRIVER_OBJECT DriverObject
    );

// Variable to contain the handle for the registration
HANDLE ClientHandle;

// DriverEntry function
NTSTATUS
  DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
    )
{
  NTSTATUS Status;

  // Specify the unload function
  DriverObject->DriverUnload = Unload;

  // Initialize the binding context list spin lock
  KeInitializeSpinLock(
    &BindingContextListLock
    );

  // Initialize the binding context list head
  InitializeListHead(
    &BindingContextList
    );

  .
  . // Other initialization tasks
  .

  // Register the client module with the NMR
  Status = NmrRegisterClient(
    &ClientCharacteristics,
    &ClientRegistrationContext,
    &ClientHandle,
    );

  // Return the result of the registration
  return Status;
}

// ClientAttachProvider callback function
NTSTATUS
  ClientAttachProvider(
    IN HANDLE NmrBindingHandle,
    IN PVOID ClientContext,
    IN PNPI_REGISTRATION_INSTANCE ProviderRegistrationInstance
    )
{
  PNPI_MODULEID ProviderModuleId;
  PEXNPI_PROVIDER_CHARACTERISTICS ProviderNpiSpecificCharacteristics;
  PCLIENT_BINDING_CONTEXT BindingContext;
  PVOID ProviderBindingContext;
  PEXNPI_PROVIDER_DISPATCH ProviderDispatch;
  KLOCK_QUEUE_HANDLE BindingContextListLockHandle;
  NTSTATUS Status;

  // Get pointers to the provider module's identification structure
  // and the provider module's NPI-specific characteristics structure
  ProviderModuleId = ProviderRegistrationInstance->ModuleId;
  ProviderNpiSpecificCharacteristics =
    (PEXNPI_PROVIDER_CHARACTERISTICS)
      ProviderRegistrationInstance->NpiSpecificCharacteristics;

  //
  // Use the data in the structures pointed to by
  // ProviderRegistrationInstance, ProviderModuleId,
  // and ProviderNpiSpecificCharacteristics to determine
  // whether to attach to the provider module.
  //

  // If the client module determines that it will not attach
  // to the provider module
  if (...)
  {
    // Return status code indicating the modules did not
    // attach to each other
    return STATUS_NOINTERFACE;
  }

  // Allocate memory for the client's binding context structure
  BindingContext =
    (PCLIENT_BINDING_CONTEXT)
      ExAllocatePoolWithTag(
        NonPagedPool,
        sizeof(CLIENT_BINDING_CONTEXT),
        BINDING_CONTEXT_POOL_TAG
        );

  // Check result of allocation
  if (BindingContext == NULL)
  {
    // Return error status code
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Initialize the client binding context structure
  KeInitializeSpinLock(
    &BindingContext->DetachLock
   );
  BindingContext->InProgressCallCount = 0;
  BindingContext->Detaching = 0;
  ...

  // Continue with the attachment to the provider module
  Status = NmrClientAttachProvider(
    NmrBindingHandle,
    BindingContext,
    &Dispatch,
    &ProviderBindingContext,
    &ProviderDispatch
    );

  // Check result of attachment
  if (Status == STATUS_SUCCESS)
  {
    // Save NmrBindingHandle, ProviderBindingContext,
    // and ProviderDispatch for future reference
    BindingContext->NmrBindingHandle =
      NmrBindingHandle;
    BindingContext->ProviderBindingContext =
      ProviderBindingContext;
    BindingContext->ProviderDispatch =
      ProviderDispatch;

    // Acquire the binding context list spin lock
    KeAcquireInStackQueuedSpinLock(
      &BindingContextListLock,
      &BindingContextListLockHandle
      );

    // Add this binding context to the list of valid
    // binding contexts
    InsertTailList(
      &BindingContextList,
      &BindingContext->Link
      );

    // Release the binding context list spin lock
    KeReleaseInStackQueuedSpinLock(
      &BindingContextListLockHandle
      );
  }

  // Attachment did not succeed
  else
  {
    // Free memory for client's binding context structure
    ExFreePoolWithTag(
      BindingContext,
      BINDING_CONTEXT_POOL_TAG
      );
  }

  // Return result of attachment
  return Status;
}

// Wrapper function around a provider NPI function
//
// Each of the provider NPI functions should be wrapped
// in this manner.
NTSTATUS
  ProviderNpiFunctionXxx(
    ClientBindingContext,
    .
    . // Parameters to the provider NPI function
    .
    )
{
  KLOCK_QUEUE_HANDLE BindingContextListLockHandle;
  KLOCK_QUEUE_HANDLE DetachLockHandle;
  PCLIENT_BINDING_CONTEXT BindingContextListElement;
  PLIST_ENTRY Entry;
  NTSTATUS Status;

  // Acquire the binding context list spin lock
  KeAcquireInStackQueuedSpinLock(
    &BindingContextListLock,
    &BindingContextListLockHandle
    );

  // Search for the binding context in the list of valid
  // binding contexts
  for (Entry = BindingContextList.Flink;
       Entry != &BindingContextList;
       Entry = Entry->Flink)
  {
    // Get the next binding context from the list
    BindingContextListElement =
      CONTAINING_RECORD(
        Entry,
        CLIENT_BINDING_CONTEXT,
        Link
        );

    // Check if this binding context is a match
    if (BindingContextListElement == ClientBindingContext)
    {
      // Break out of the search loop
      break;
    }
  }

  // Check if the binding context was not found
  if (Entry == &BindingContextList)
  {
    // Release the binding context list spin lock
    KeReleaseInStackQueuedSpinLock(
      &BindingContextListLockHandle
      );

    // Return status indicating that the interface is not available
    return STATUS_NOINTERFACE;
  }
```

```
// Acquire the detach spin lock at DPC level
KeAcquireInStackQueuedSpinLockAtDpcLevel(
  &ClientBindingContext->DetachLock,
  &DetachLockHandle
  );

// The modules should not be detaching
ASSERT(ClientBindingContext->Detaching != 1);

// Increment the in-progress call count
ClientBindingContext->InProgressCallCount++;

// Release the detach spin lock from DPC level
KeReleaseInStackQueuedSpinLockFromDpcLevel(
  &DetachLockHandle
  );

// Release the binding context list spin lock
KeReleaseInStackQueuedSpinLock(
  &BindingContextListLockHandle
  );

// Call the provider NPI function
Status =
  ClientBindingContext->ProviderDispatch->ProviderNpiFunctionXxx(
    ClientBindingContext->ProviderBindingContext,
    .
    . // Parameters to the provider NPI function
    .
    );

// Check if pending
if (Status == STATUS_PENDING)
{
  // If completion of the call is pending, then when the call
  // completes, the completion routine (or other callback function
  // that is called upon completion of the call) must include the
  // the same code as below for the non-pending case.

  // Return pending status
  return STATUS_PENDING;
}

// Acquire the detach spin lock
KeAcquireInStackQueuedSpinLock(
  &ClientBindingContext->DetachLock,
  &DetachLockHandle
  );

// Decrement the in-progress call count
ClientBindingContext->InProgressCallCount--;

// Check if the modules are now detaching
if (ClientBindingContext->Detaching == 1)
{
    // Check if this call was the last of the in-progress
    // calls to the provider module to be completed
    if (ClientBindingContext->InProgressCallCount == 0)
    {
      // Release the detach spin lock
      KeReleaseInStackQueuedSpinLock(
        &DetachLockHandle
        );

      // Inform the NMR that detachment is complete
      NmrClientDetachProviderComplete(
        ClientBindingContext->NmrBindingHandle
        );
    }
    else
    {
      // Release the detach spin lock
      KeReleaseInStackQueuedSpinLock(
        &DetachLockHandle
        );
    }
  }
  else
  {
    // Release the detach spin lock
    KeReleaseInStackQueuedSpinLock(
      &DetachLockHandle
      );
  }

  // Return status of the call to the provider NPI function
  return Status;
}

// ClientDetachProvider callback function
NTSTATUS
  ClientDetachProvider(
    IN PVOID ClientBindingContext
    )
{
  PCLIENT_BINDING_CONTEXT BindingContext;
  KLOCK_QUEUE_HANDLE BindingContextListLockHandle;
  KLOCK_QUEUE_HANDLE DetachLockHandle;
  NTSTATUS Status;

  // Get a pointer to the binding context
  BindingContext = (PCLIENT_BINDING_CONTEXT)ClientBindingContext;

  // Acquire the binding context list spin lock
  KeAcquireInStackQueuedSpinLock(
    &BindingContextListLock,
    &BindingContextListLockHandle
    );

  // Remove the binding context from the binding context list
  RemoveEntryList(&BindingContext->Link);

  // Acquire the detach spin lock at DPC level
  KeAcquireInStackQueuedSpinLockAtDpcLevel(
    &BindingContext->DetachLock,
    &DetachLockHandle
    );

  // Set the flag indicating that the client module is detaching
  // from the provider module so that the completion of the final
  // call will complete the detachment
  BindingContext->Detaching = 1;

  // Check if there are no in-progress NPI function calls to the
  // provider module
  if (BindingContext->InProgressCallCount == 0)
  {
    // Set the status to success to indicate detachment is complete
    Status = STATUS_SUCCESS;
  }

  // There are one or more in-progress NPI function calls
  // to the provider module
  else
  {
    // Set the status to pending to indicate that detachment is
    // pending completion of the in-progress NPI function calls
    Status = STATUS_PENDING;
  }

  // Release the detach spin lock from DPC level
  KeReleaseInStackQueuedSpinLockFromDpcLevel(
    &DetachLockHandle
    );

  // Release the binding context list spin lock
  KeReleaseInStackQueuedSpinLock(
    &BindingContextListLockHandle
    );

  // Return the status of the detachment
  return Status;
}
```

Likewise, a provider module might use an implementation along the same lines as the above client module example for tracking the number of in-progress calls to an attached client module's NPI callback functions.

Note The above code example shows one possible method of tracking the number of in-progress calls to an attached network module's NPI functions. A network module might use an alternate method depending on the implementation details of the particular NPI that the network module supports.

Roadmap for Developing Network Drivers with Winsock Kernel

Article • 11/29/2022

To create a networking driver package that uses the kernel-mode socket programming features of Winsock Kernel (WSK), follow these steps:

- **Step 1:** Learn about Windows architecture and drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- **Step 2:** Learn about the Network Driver Interface Specification (NDIS).

  Your driver package will typically use Network Driver Interface Specification (NDIS) interfaces. For more information about NDIS and NDIS miniport drivers, see the following topics:

  Windows Network Architecture and the OSI Model

  NDIS Miniport Drivers

  Writing NDIS Miniport Drivers

  Network Driver Programming Considerations

- **Step 3:** Determine additional network components to use in your driver.

  In addition to the core NDIS features, you can use the following additional Windows network components with kernel-mode drivers, depending on the hardware configuration:

  IP Helper

  Windows Filtering Platform Callout Drivers

  Native 802.11 Wireless LAN

  Mobile Broadband Network Interface

- **Step 4:** Learn the fundamentals of Winsock Kernel.

Winsock Kernel is supported in Windows Vista and later versions of Windows. For information about how to use Winsock Kernel, see Introduction to Winsock Kernel.

A simpler, more generic network programming interface that you can use in network drivers is Network Module Registrar.

- **Step 5:** Determine additional Windows driver design decisions.

  For information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- **Step 6:** Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user-mode application. For information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Lab Kit (HLK) testing, see Building, Debugging, and Testing Drivers. For information about tools for building, testing, verifying, and debugging, see Driver Development Tools.

- **Step 7:** Review the Winsock Kernel (WSK TCP Echo Server) driver sample in the Windows driver samples repository on GitHub.

- **Step 8:** Develop, build, test, and debug your driver.

  For information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process helps ensure that you build a driver that works.

- **Step 9:** Create a driver package for your driver.

  For information about how to install drivers, see Providing a Driver Package.

- **Step 10:** Sign and distribute your driver.

  The final step is to sign (optional) and distribute the driver. If your driver meets the quality standards that are defined for the Windows Hardware Lab Kit (HLK), you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual driver.

Introduction to Winsock Kernel

Article • 12/15/2021

Winsock Kernel (WSK) is a kernel-mode Network Programming Interface (NPI). With WSK, kernel-mode software modules can perform network I/O operations using the same socket programming concepts that are supported by user-mode Winsock2. The WSK NPI supports familiar socket operations such as socket creation, binding, connection establishment, and data transfers (send and receive). However, while the WSK NPI supports most of the same socket programming concepts as user-mode Winsock2, it is a completely new and different interface with unique characteristics such as asynchronous I/O that uses IRPs and event callbacks to enhance performance.

Kernel-mode network modules targeted for Windows Vista and later versions of Microsoft Windows should use WSK instead of TDI because WSK provides improved performance and easier programming. Filter drivers should implement the Windows Filtering Platform on Windows Vista, and TDI clients should implement WSK.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

Winsock Kernel Overview Topics

Article • 12/15/2021

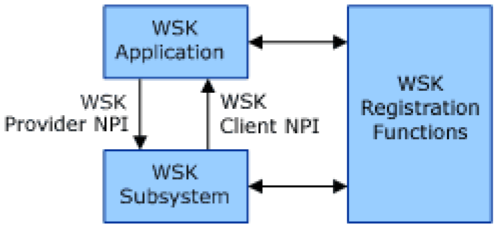
This section provides an overview of Winsock Kernel (WSK) and includes the following topics:

- Winsock Kernel Architecture
- Winsock Kernel Objects
- Winsock Kernel Socket Categories
- Winsock Kernel Events
- Using Winsock Kernel Functions vs. Event Callback Functions
- Winsock Kernel Dispatch Tables
- Winsock Kernel Extension Interfaces
- Using IRPs with Winsock Kernel Functions

Winsock Kernel Architecture

Article • 12/15/2021

The architecture of Winsock Kernel (WSK) is shown in the following diagram.



At the core of the WSK architecture is the WSK subsystem. The WSK subsystem is a network module that implements the provider side of the WSK Network Programming Interface (NPI). The WSK subsystem interfaces with transport providers on its lower edge that provide support for various transport protocols.

Attached to the WSK subsystem are WSK applications. WSK applications are kernel-mode software modules that implement the client side of the WSK NPI in order to perform network I/O operations. (In this context, "client" should not be confused with the term as used in client-server systems). . The WSK subsystem can call the functions in the WSK client NPI to notify the WSK application of asynchronous events.

WSK applications discover and attach to the WSK subsystem by using a set of WSK registration functions. Applications can use these functions to dynamically detect when the WSK subsystem is available and to exchange dispatch tables that constitute the provider and client side implementations of the WSK NPI.

Alternately, WSK applications can attach to the WSK subsystem by using the Network Module Registrar (NMR). For more information, see Using NMR for WSK Registration and Unregistration.

Winsock Kernel Objects

Article • 12/15/2021

The Winsock Kernel (WSK) Network Programming Interface (NPI) is designed around two main object types: *Client* and *Socket* .

Client Object

A *client object* represents the attachment, or binding, between a WSK application and the WSK subsystem. A client object is represented by the **WSK_CLIENT** structure. A pointer to a client object is returned to a WSK application during the process of attachment to the WSK subsystem. A WSK application passes this pointer to all WSK functions that operate at the client object level.

Socket Object

A *socket object* represents a network socket that can be used for network I/O. A socket object is represented by the **WSK_SOCKET** structure. A pointer to a socket object is returned to a WSK application when the application creates a new socket or when the application accepts an incoming connection. A WSK application passes this pointer to all WSK functions that are specific to a particular socket.

Winsock Kernel Socket Categories

Article • 12/15/2021

The Winsock Kernel (WSK) Network Programming Interface (NPI) defines five different categories of sockets: *basic sockets*, *listening sockets*, *datagram sockets*, *connection-oriented sockets*, and *stream sockets*. Each WSK socket category has unique functionality and supports a different set of socket functions. A WSK application must specify which category of WSK socket it is creating whenever it creates a new socket. The purpose for each WSK socket category is as follows:

Basic Sockets
Basic sockets are used only to get and set transport stack socket options or to perform socket I/O control operations. Basic sockets cannot be bound to a local transport address and do not support sending or receiving network data.

Listening Sockets
Listening sockets are used to listen for incoming connections from remote transport addresses. The functionality of a listening socket includes all of the functionality of a basic socket.

Datagram Sockets
Datagram sockets are used to send and receive datagrams. The functionality of a datagram socket includes all of the functionality of a basic socket.

Connection-Oriented Sockets
Connection-oriented sockets are used to send and receive network data over established connections. The functionality of a connection-oriented socket includes all of the functionality of a basic socket.

Stream Sockets
Stream sockets are used to either listen for incoming connections from remote transport addresses (act as a listening socket), or to send and receive network data over established connections (act as a connection-oriented socket). Use a stream socket when you do not know at the time of socket creation if you want a listening socket or a connection-oriented socket. The functionality of a stream socket includes all of the functionality of a basic socket.

Winsock Kernel Events

Article • 12/15/2021

The Winsock Kernel (WSK) subsystem can asynchronously notify a WSK application when certain socket events occur, such as when new data has been received on a socket or when a socket has been disconnected. In order for a WSK application to be asynchronously notified of socket events, the WSK application must implement the appropriate event callback functions and enable those event callback functions on the sockets that it creates.

Note A WSK application is not required to implement or use event callback functions. A WSK application can perform most WSK socket operations by calling the appropriate WSK socket functions. The only WSK feature that requires using event callback functions is conditional-accept mode on listening sockets. For more information about the advantages and disadvantages between using WSK functions versus using event callback functions, see Using Winsock Kernel Functions vs. Event Callback Functions.

Each WSK socket category supports a different set of socket events.

Basic sockets

Basic sockets do not support any socket events.

Listening sockets

Event	Event callback function
An incoming connection has been accepted.	*WskAcceptEvent*
An incoming connection request has arrived.	*WskInspectEvent*
An incoming connection request has been dropped.	*WskAbortEvent*

* Applies only to listening sockets that have conditional-accept mode enabled. For more information about using conditional accept mode with listening sockets, see Listening for and Accepting Incoming Connections.

Datagram sockets

Event	Event callback function
One or more new datagrams have been received.	*WskReceiveFromEvent*

Connection-oriented sockets

Event	Event callback function
New data has been received.	*WskReceiveEvent*
The socket has been disconnected.	*WskDisconnectEvent*
The ideal send backlog size has changed.	*WskSendBacklogEvent*

When a WSK application creates a socket, the socket's event callback functions are disabled by default. A WSK application must enable a socket's event callback functions in order for the WSK subsystem to call the socket's event callback functions when socket events occur. For more information about enabling and disabling a socket's event callback functions, see Enabling and Disabling Event Callback Functions.

If a WSK application registers an extension interface for a socket, the extension interface might support additional events. For more information about registering an extension interface for a socket, see Registering an Extension Interface.

The WSK subsystem can also notify a WSK application of events that are not specific to a particular socket. In order for a WSK application to be notified of these events, the WSK application must implement a *WskClientEvent* event callback function. There are currently no events defined that are not specific to a particular socket. A WSK application's *WskClientEvent* event callback function is always enabled and cannot be disabled.

A WSK application's event callback functions must not wait for completion of other WSK requests in the context of WSK completion or event callback functions. The callback may initiate other WSK requests assuming that it doesn't spend too much time at DISPATCH_LEVEL or exhaust the kernel stack, but it must not wait for their completion even when the callback is called at IRQL = PASSIVE_LEVEL.

Using Winsock Kernel Functions vs. Event Callback Functions

Article • 12/15/2021

For certain socket operations, a Winsock Kernel (WSK) application can either call one of the socket's WSK functions to perform the operation or implement and enable an event callback function on the socket that the WSK subsystem calls when the event associated with the operation occurs. For example, when receiving data on a connection-oriented socket, a WSK application can either make calls to the socket's **WskReceive** function, or implement and enable a *WskReceiveEvent* event callback function on the socket. The requirements of a WSK application dictate which method the application should use. Examples for how to use both methods are provided throughout the WSK documentation.

The following lists summarize some key points for each method.

Using Winsock Kernel Functions

- The WSK application drives the socket operations, meaning the WSK application controls when the socket operations occur. This might simplify the synchronization required by the WSK application.

- The WSK application provides IRPs to the socket functions. These IRPs are queued by the WSK subsystem until the socket operation completes. For more information about using IRPs with WSK functions, see Using IRPs with Winsock Kernel Functions.

- The WSK application can perform blocking socket operations by waiting for the IRP for each operation to be completed by the WSK subsystem.

- The WSK application might need to keep multiple socket operations queued in some situations in order to ensure high performance data transfer on connection-oriented sockets, to prevent incoming datagrams from being dropped on datagram sockets, or to prevent incoming connections being dropped on listening sockets.

- The WSK application provides the data buffers for the data transfer operations. This reduces the number of times the data might need to be copied. However, if a WSK application keeps multiple data transfer operations queued, the application must provide data buffers to the WSK subsystem for each queued data transfer operation. Thus, the WSK application might require additional memory resources.

Using Event Callback Functions

- The WSK subsystem drives the socket operations, meaning the WSK subsystem notifies the WSK application of the socket's events by calling the socket's event callback functions. The WSK application might require more complex synchronization to handle the asynchronous nature of the event callback functions.

- The WSK application does not use IRPs for the socket operations.

- The WSK application does not need to queue socket operations. The WSK subsystem calls the WSK application's event callback functions as soon as the socket's events occur. If the WSK application can keep up with the rate that a socket's event callback functions are called, using event callback functions could provide the highest performance and least chance of dropping datagrams or incoming connections.

- The WSK subsystem supplies the data buffers for data transfer operations. The WSK application must release these data buffers back to the WSK subsystem either immediately, or within a reasonable amount of time, so that the WSK subsystem does not run out of memory resources. Thus, the WSK application might need to copy the data from the data buffers that are owned by the WSK subsystem into its own data buffers.

Note The above lists are not necessarily exhaustive. There might be other points to consider when choosing which method is the best choice for a particular WSK application.

Winsock Kernel Dispatch Tables

Article • 12/15/2021

The socket object for a Winsock Kernel (WSK) socket contains a pointer to a provider dispatch table structure that contains function pointers to the socket functions supported by the socket. A WSK application calls the functions in the provider dispatch table structure to perform network I/O operations on the socket. Because each WSK socket category supports a different set of socket functions, the WSK Network Programming Interface (NPI) defines a different provider dispatch table structure for each category of WSK socket.

Socket category	Dispatch table structure
Basic socket	WSK_PROVIDER_BASIC_DISPATCH
Listening socket	WSK_PROVIDER_LISTEN_DISPATCH
Datagram socket	WSK_PROVIDER_DATAGRAM_DISPATCH
Connection-oriented socket	WSK_PROVIDER_CONNECTION_DISPATCH

If a WSK application uses event callback functions for the sockets that it creates, it must provide a client dispatch table structure that contains function pointers to the socket's event callback functions whenever it creates a new socket. Because each WSK socket category supports a different set of event callback functions, the WSK NPI defines a different client dispatch table structure for each category of WSK socket.

Socket category	Dispatch table structure
Listening socket	WSK_CLIENT_LISTEN_DISPATCH
Datagram socket	WSK_CLIENT_DATAGRAM_DISPATCH
Connection-oriented socket	WSK_CLIENT_CONNECTION_DISPATCH

Note Basic sockets do not support any event callback functions. Therefore, no client dispatch table structure is defined for basic sockets.

Winsock Kernel Extension Interfaces

Article • 12/15/2021

The Winsock Kernel (WSK) Network Programming Interface (NPI) includes support for *extension interfaces*. The WSK subsystem can use extension interfaces to extend the functionality of WSK sockets beyond the set of socket functions and event callback functions currently defined by the WSK NPI. Each extension interface is defined by an NPI that is independent of the WSK NPI. Currently no extension interfaces have been defined.

A WSK application can register for an extension interface that is supported by the WSK subsystem by using the **SIO_WSK_REGISTER_EXTENSION** socket IOCTL operation. A WSK application registers for extension interfaces on a socket-by-socket basis.

For more information about registering an extension interface, see Registering an Extension Interface.

Using IRPs with Winsock Kernel Functions

Article • 12/15/2021

The Winsock Kernel (WSK) Network Programming Interface (NPI) uses IRPs for asynchronous completion of network I/O operations. Each WSK function takes a pointer to an IRP as a parameter. The WSK subsystem completes the IRP after the operation performed by the WSK function is complete.

An IRP that a WSK application uses to pass to a WSK function can originate in one of the following ways.

- The WSK application allocates the IRP by calling the **IoAllocateIrp** function. In this situation, the WSK application must allocate the IRP with at least one I/O stack location.

- The WSK application reuses a completed IRP that it previously allocated. In this situation, the WSK must call the **IoReuseIrp** function to reinitialize the IRP.

- The WSK application uses an IRP that was passed down to it either by a higher level driver or by the I/O manager. In this situation, the IRP must have at least one remaining I/O stack location available for use by the WSK subsystem.

After a WSK application has an IRP to use for calling a WSK function, it can set an **IoCompletion** routine for the IRP to be called when the IRP is completed by the WSK subsystem. A WSK application sets an **IoCompletion** routine for an IRP by calling the **IoSetCompletionRoutine** function. Depending upon how the IRP originated, an **IoCompletion** routine is either required or optional.

- If the WSK application allocated the IRP, or is reusing an IRP that it previously allocated, then it must set an **IoCompletion** routine for the IRP before calling a WSK function. In this situation, the WSK application must specify **TRUE** for the *InvokeOnSuccess*, *InvokeOnError*, and *InvokeOnCancel* parameters that are passed to the **IoSetCompletionRoutine** function to ensure that the **IoCompletion** routine is always called. Furthermore, the **IoCompletion** routine that is set for the IRP must always return STATUS_MORE_PROCESSING_REQUIRED to terminate the completion processing of the IRP. If the WSK application is done using the IRP after the **IoCompletion** routine has been called, then it should call the **IoFreeIrp** function to free the IRP before returning from the **IoCompletion** routine. If the WSK application does not free the IRP then it can reuse the IRP for a call to another WSK function.

- If the WSK application uses an IRP that was passed down to it by a higher level driver or by the I/O manager, it should set an **IoCompletion** routine for the IRP before calling the WSK function only if it must be notified when the operation performed by the WSK function has completed. If the WSK application does not set an **IoCompletion** routine for the IRP, then when the IRP is completed the IRP will be passed back up to the higher level driver or to the I/O manager as per normal IRP completion processing. If the WSK application sets an **IoCompletion** routine for the IRP, the **IoCompletion** routine can either return STATUS_SUCCESS or STATUS_MORE_PROCESSING_REQUIRED. If the **IoCompletion** routine returns STATUS_SUCCESS, the IRP completion processing will continue normally. If the **IoCompletion** routine returns STATUS_MORE_PROCESSING_REQUIRED, the WSK application must complete the IRP by calling **IoCompleteRequest** after it has finished processing the results of the operation that was performed by the WSK function. A WSK application should never free an IRP that was passed down to it by a higher level driver or by the I/O manager.

Note If the WSK application sets an **IoCompletion** routine for an IRP that was passed down to it by a higher level driver or by the I/O manager, then the **IoCompletion** routine must check the **PendingReturned** member of the IRP and call the **IoMarkIrpPending** function if the **PendingReturned** member is **TRUE**. For more information, see Implementing an IoCompletion Routine.

Note A WSK application should not call new WSK functions in the context of the **IoCompletion** routine. Doing so may result in recursive calls and exhaust the kernel mode stack. When executing at IRQL = DISPATCH_LEVEL, this can also lead to starvation of other threads.

A WSK application does not initialize the IRPs that it passes to the WSK functions other than setting an **IoCompletion** routine. When a WSK application passes an IRP to a WSK function, the WSK subsystem sets up the next I/O stack location on behalf of the application.

The following code example shows how a WSK application can allocate and use an IRP when performing a receive operation on a socket.

C++

```
// Prototype for the receive IoCompletion routine
NTSTATUS
  ReceiveComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );
```

```
// Function to receive data
NTSTATUS
  ReceiveData(
    PWSK_SOCKET Socket,
    PWSK_BUF DataBuffer
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    ReceiveComplete,
    DataBuffer,  // Use the data buffer for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the receive operation on the socket
  Status =
    Dispatch->WskReceive(
      Socket,
      DataBuffer,
      0,  // No flags are specified
      Irp
      );

  // Return the status of the call to WskReceive()
  return Status;
}

// Receive IoCompletion routine
NTSTATUS
  ReceiveComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_BUF DataBuffer;
  ULONG ByteCount;

  // Check the result of the receive operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the pointer to the data buffer
    DataBuffer = (PWSK_BUF)Context;

    // Get the number of bytes received
    ByteCount = (ULONG)(Irp->IoStatus.Information);

    // Process the received data
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

The model shown in the previous example, where the WSK application allocates an IRP and then frees it in the completion routine, is the model that is used in the examples throughout the remainder of the WSK documentation.

The following code example shows how a WSK application can use an IRP that has been passed to it by a higher level driver or by the I/O manager when performing a receive operation on a socket.

C++

```
// Prototype for the receive IoCompletion routine
NTSTATUS
  ReceiveComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to receive data
NTSTATUS
  ReceiveData(
    PWSK_SOCKET Socket,
    PWSK_BUF DataBuffer,
    PIRP Irp;  // IRP from a higher level driver or the I/O manager
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  NTSTATUS Status;

  // Get pointer to the provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Set the completion routine for the IRP such that it is
  // only called if the receive operation succeeds.
  IoSetCompletionRoutine(
    Irp,
    ReceiveComplete,
    DataBuffer,  // Use the data buffer for the context
    TRUE,
    FALSE,
    FALSE
    );

  // Initiate the receive operation on the socket
  Status =
    Dispatch->WskReceive(
      Socket,
      DataBuffer,
      0,  // No flags are specified
      Irp
      );

  // Return the status of the call to WskReceive()
  return Status;
}

// Receive IoCompletion routine
NTSTATUS
  ReceiveComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_BUF DataBuffer;
  ULONG ByteCount;
  // Since the completion routine was only specified to
  // be called if the operation succeeds, this should
  // always be true.
  ASSERT(Irp->IoStatus.Status == STATUS_SUCCESS);

  // Check the pending status of the IRP
  if (Irp->PendingReturned == TRUE)
  {
    // Mark the IRP as pending
    IoMarkIrpPending(Irp);
  }

  // Get the pointer to the data buffer
  DataBuffer = (PWSK_BUF)Context;

  // Get the number of bytes received
  ByteCount = (ULONG)(Irp->IoStatus.Information);

  // Process the received data
  ...

  // Return STATUS_SUCCESS to continue the
  // completion processing of the IRP.
  return STATUS_SUCCESS;
}
```

For more information about using IRPs, see Handling IRPs.

Winsock Kernel Operations Topics

Article • 12/15/2021

This section discusses Winsock Kernel (WSK) operations and includes the following topics:

- Registering a Winsock Kernel Application
- Performing Control Operations on a Client Object
- Creating Sockets
- Performing Control Operations on a Socket
- Enabling and Disabling Event Callback Functions
- Binding a Socket to a Transport Address
- Listening for and Accepting Incoming Connections
- Establishing a Connection with a Destination
- Sending and Receiving Data
- Disconnecting a Socket from a Destination
- Closing a Socket
- Registering an Extension Interface
- Unregistering a Winsock Kernel Application
- Resolving Host Names and IP Addresses
- WSK Client Control Operations
- WSK Socket IOCTL Operations
- WSK Socket Options
- WSK data types

Registering a Winsock Kernel Application

Article • 12/15/2021

WSK Client Object Registration

A Winsock Kernel (WSK) application must register as a WSK client by calling the **WskRegister** function. **WskRegister** requires the WSK application to initialize and pass a pointer to its WSK client's Network Programming Interface (NPI)(a **WSK_CLIENT_NPI** structure) and a WSK registration object (a **WSK_REGISTRATION** structure) that will be initialized by **WskRegister** upon successful return.

The following code example shows how a WSK application can register as a WSK client.

C++

```cpp
// Include the WSK header file
#include "wsk.h"

// WSK Client Dispatch table that denotes the WSK version
// that the WSK application wants to use and optionally a pointer
// to the WskClientEvent callback function
const WSK_CLIENT_DISPATCH WskAppDispatch = {
  MAKE_WSK_VERSION(1,0), // Use WSK version 1.0
  0,    // Reserved
  NULL  // WskClientEvent callback not required for WSK version 1.0
};

// WSK Registration object
WSK_REGISTRATION WskRegistration;

// DriverEntry function
NTSTATUS
  DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
    )
{
  NTSTATUS Status;
  WSK_CLIENT_NPI wskClientNpi;

  .
  .
  .

  // Register the WSK application
  wskClientNpi.ClientContext = NULL;
  wskClientNpi.Dispatch = &WskAppDispatch;
  Status = WskRegister(&wskClientNpi, &WskRegistration);

  if(!NT_SUCCESS(Status)) {
      .
      .
      .
      return Status;
  }

  .
  .
  .
}
```

A WSK application is not required to call **WskRegister** from within its **DriverEntry** function. For example, if a WSK application is a subcomponent of a complex driver, the registration of the application might occur only when the WSK application subcomponent is activated.

A WSK application must keep the **WSK_CLIENT_DISPATCH** structure passed to **WskRegister** valid and resident in memory until **WskDeregister** is called and the registration is no longer valid. The **WSK_REGISTRATION** structure must also be kept valid and resident in memory until the WSK application stops making calls to the other WSK registration functions. The previous code example keeps these two structures in the global data section of the driver, thereby keeping the structure data resident in memory until the driver is unloaded.

WSK Provider NPI Capture

After a WSK application has registered as a WSK client with **WskRegister**, it must use the **WskCaptureProviderNPI** function to capture the WSK provider NPI from the WSK subsystem in order to start using the WSK interface.

Because the WSK subsystem might not yet be ready when a WSK application attempts to capture the WSK provider NPI, the **WskCaptureProviderNPI** function allows the WSK application to poll or wait for the WSK subsystem to become ready as follows:

- If the *WaitTimeout* parameter is WSK_NO_WAIT, the function will always return immediately without waiting.
- If *WaitTimeout* is WSK_INFINITE_WAIT, the function will wait until the WSK subsystem becomes ready.
- If *WaitTimeout* is any other value, the function will return either when the WSK subsystem becomes ready or when the wait time, in milliseconds, reaches the value

of *WaitTimeout*, whichever occurs first.

Important To avoid adversely affecting the start of other drivers and services, a WSK application that calls **WskCaptureProviderNPI** from its **DriverEntry** function should not set the *WaitTimeout* parameter to WSK_INFINITE_WAIT or an excessive wait time. Also, if a WSK application starts very early in the system startup phase, it should wait for the WSK subsystem to become ready in a different worker thread than the one in which **DriverEntry** runs.

If the call to **WskCaptureProviderNPI** fails with STATUS_NOINTERFACE, the WSK application can use the WskQueryProviderCharacteristics function to discover the range of WSK NPI versions supported by the WSK subsystem. The WSK application can call **WskDeregister** to unregister its current registration instance, and then register again by using a different WSK_CLIENT_DISPATCH instance that uses a supported WSK NPI version.

When **WskCaptureProviderNPI** returns successfully, its *WskProviderNpi* parameter points to a WSK provider NPI (WSK_PROVIDER_NPI) ready for use by the WSK application. The WSK_PROVIDER_NPI structure contains pointers to the WSK client object (WSK_CLIENT) and the WSK_PROVIDER_DISPATCH dispatch table of WSK functions that the WSK application can use to create WSK sockets and perform other operations on the WSK client object. After the WSK application is finished using the WSK_PROVIDER_DISPATCH functions, it must release the WSK provider NPI by calling WskReleaseProviderNPI.

The following code example shows how a WSK application can capture the WSK provider NPI, use it to create a socket, and then release it.

C++

```
// WSK application routine that waits for WSK subsystem
// to become ready and captures the WSK Provider NPI
NTSTATUS
  WskAppWorkerRoutine(
    )
{
  NTSTATUS Status;
  WSK_PROVIDER_NPI wskProviderNpi;

  // Capture the WSK Provider NPI. If WSK subsystem is not ready yet,
  // wait until it becomes ready.
  Status = WskCaptureProviderNPI(
    &WskRegistration, // must have been initialized with WskRegister
    WSK_INFINITE_WAIT,
    &wskProviderNpi
    );
  if(!NT_SUCCESS(Status))
  {
    // The WSK Provider NPI could not be captured.
    if( Status == STATUS_NOINTERFACE ) {
      // WSK application's requested version is not supported
    }
    else if( status == STATUS_DEVICE_NOT_READY ) {
      // WskDeregister was invoked in another thread thereby causing
      // WskCaptureProviderNPI to be canceled.
    }
    else {
      // Some other unexpected failure has occurred
    }

    return Status;
  }

  // The WSK Provider NPI has been captured.
  // Create and set up a listening socket that accepts
   // incoming connections.
  Status = CreateListeningSocket(&wskProviderNpi, ...);

  // The WSK Provider NPI will not be used any more.
  // So, release it here immediately.
  WskReleaseProviderNPI(&WskRegistration);

  // Return result of socket creation routine
  return Status;

}
```

A WSK application can call WskCaptureProviderNPI more than once. For each call to **WskCaptureProviderNPI** that returns successfully, there must be a corresponding call to WskReleaseProviderNPI. A WSK application must not make any further calls to the functions in WSK_PROVIDER_DISPATCH after calling **WskReleaseProviderNPI**.

Performing Control Operations on a Client Object

Article • 12/15/2021

After a Winsock Kernel (WSK) application has successfully attached to the WSK subsystem, it can perform control operations on the client object (**WSK_CLIENT**) that was returned by the WSK subsystem during attachment. These control operations are not specific to a particular socket, but instead have a more general scope. For more information about each of the control operations that can be performed on a client object, see WSK Client Control Operations.

A WSK application performs client control operations by calling the **WskControlClient** function. The **WskControlClient** function is pointed to by the **WskControlClient** member of the **WSK_PROVIDER_DISPATCH** structure that was returned by the WSK subsystem during attachment.

The following code example shows how a WSK application can use the **WSK_TRANSPORT_LIST_QUERY** client control operation to retrieve a list of available network transports that can be specified when creating a new socket.

```C++
// Function to retrieve a list of available network transports
NTSTATUS
  GetTransportList(
    PWSK_PROVIDER_NPI WskProviderNpi,
    PWSK_TRANSPORT TransportList,
    ULONG MaxTransports,
    PULONG TransportsRetrieved
    )
{
  SIZE_T BytesRetrieved;
  NTSTATUS Status;

  // Perform client control operation
  Status =
    WskProviderNpi->Dispatch->
        WskControlClient(
          WskProviderNpi->Client,
          WSK_TRANSPORT_LIST_QUERY,
          0,
          NULL,
          MaxTransports * sizeof(WSK_TRANSPORT),
          TransportList,
          &BytesRetrieved,
          NULL  // No IRP for this control operation
          );

  // Convert bytes retrieved to transports retrieved
  TransportsRetrieved = BytesRetrieved / sizeof(WSK_TRANSPORT);

  // Return status of client control operation
  return Status;
}
```

Creating Sockets

Article • 12/15/2021

After a Winsock Kernel (WSK) application has successfully attached to the WSK subsystem, it can create sockets that can be used for network I/O operations. A WSK application creates sockets by calling the **WskSocket** function. The **WskSocket** function is pointed to by the **WskSocket** member of the **WSK_PROVIDER_DISPATCH** structure that was returned by the WSK subsystem during attachment.

A WSK application must specify which category of WSK socket it is creating whenever it creates a new socket. For more information about WSK socket categories, see Winsock Kernel Socket Categories.

A WSK application must also specify the address family, socket type, and protocol whenever it creates a new socket. For more information about the address families supported by WSK, see WSK Address Families.

When creating a new socket, a WSK application must provide a socket context value and a pointer to a client dispatch table structure if the application will be enabling any event callback functions on the socket. For more information about enabling event callback functions on a socket, see Enabling and Disabling Event Callback Functions.

If the socket is created successfully, the **IoStatus.Information** field of the IRP contains a pointer to a socket object structure (**WSK_SOCKET**) for the new socket. For more information about using IRPs with WSK functions, see Using IRPs with Winsock Kernel Functions.

The following code example shows how a WSK application can create a listening socket.

C++

```
// Context structure for each socket
typedef struct _WSK_APP_SOCKET_CONTEXT {
  PWSK_SOCKET Socket;
  .
  .  // Other application-specific members
  .
} WSK_APP_SOCKET_CONTEXT, *PWSK_APP_SOCKET_CONTEXT;

// Prototype for the socket creation IoCompletion routine
NTSTATUS
  CreateListeningSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );
```

```
// Function to create a new listening socket
NTSTATUS
  CreateListeningSocket(
    PWSK_PROVIDER_NPI WskProviderNpi,
    PWSK_APP_SOCKET_CONTEXT SocketContext,
    PWSK_CLIENT_LISTEN_DISPATCH Dispatch,
    )
{
  PIRP Irp;
  NTSTATUS Status;

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    CreateListeningSocketComplete,
    SocketContext,
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the creation of the socket
  Status =
    WskProviderNpi->Dispatch->
        WskSocket(
          WskProviderNpi->Client,
          AF_INET,
          SOCK_STREAM,
          IPPROTO_TCP,
          WSK_FLAG_LISTEN_SOCKET,
          SocketContext,
          Dispatch,
          NULL,
          NULL,
          NULL,
          Irp
          );

  // Return the status of the call to WskSocket()
  return Status;
}

// Socket creation IoCompletion routine
NTSTATUS
  CreateListeningSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_APP_SOCKET_CONTEXT SocketContext;

  // Check the result of the socket creation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the pointer to the socket context
    SocketContext =
      (PWSK_APP_SOCKET_CONTEXT)Context;

    // Save the socket object for the new socket
    SocketContext->Socket =
      (PWSK_SOCKET)(Irp->IoStatus.Information);

    // Set any socket options for the new socket
    ...

    // Enable any event callback functions on the new socket
    ...

    // Perform any other initializations
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

For connection-oriented sockets, a WSK application can call the WskSocketConnect function to create, bind, and connect a socket in a single function call.

Performing Control Operations on a Socket

Article • 12/15/2021

After a Winsock Kernel (WSK) application has successfully created a socket, it can perform control operations on the socket. The control operations that can be performed on a socket include setting and retrieving socket options and executing socket IOCTL operations.

A WSK application performs control operations on a socket by calling the WskControlSocket function. The **WskControlSocket** function is pointed to by the **WskControlSocket** member of the socket's provider dispatch structure. A socket's provider dispatch structure is pointed to by the **Dispatch** member of the socket object structure (WSK_SOCKET) that was returned by the WSK subsystem during the creation of the socket.

The following code example shows how a WSK application can set the SO_EXCLUSIVEADDRUSE socket option on a datagram socket.

```C++
// Prototype for the control socket IoCompletion routine
NTSTATUS
  ControlSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to set the SO_EXCLUSIVEADDRUSE socket option
// on a datagram socket
NTSTATUS
  SetExclusiveAddrUse(
    PWSK_SOCKET Socket
    )
{
  PWSK_PROVIDER_DATAGRAM_DISPATCH Dispatch;
  PIRP Irp;
  ULONG SocketOptionState;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_DATAGRAM_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    ControlSocketComplete,
    Socket,  // Use the socket object for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Set the socket option state to 1 to set the socket option
  SocketOptionState = 1;

  // Initiate the control operation on the socket
  Status =
    Dispatch->WskControlSocket(
      Socket,
      WskSetOption,
      SO_EXCLUSIVEADDRUSE,
      SOL_SOCKET,
      sizeof(ULONG),
      &SocketOptionState,
      0,
      NULL,
      NULL,
      Irp
      );

  // Return the status of the call to WskControlSocket()
  return Status;
}

// Control socket IoCompletion routine
NTSTATUS
  ControlSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_SOCKET Socket;
  // Check the result of the control operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the socket object from the context
    Socket = (PWSK_SOCKET)Context;

    // Perform the next operation on the socket
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

For more information about each of the supported socket options, see WSK Socket Options.

The following code example shows how a WSK application can execute the SIO_WSK_SET_REMOTE_ADDRESS socket IOCTL operation on a datagram socket.

C++

```
// Prototype for the control socket IoCompletion routine
NTSTATUS
  ControlSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to perform the SIO_WSK_SET_REMOTE_ADDRESS socket
// IOCTL operation on a datagram socket
NTSTATUS
  SetRemoteAddress(
    PWSK_SOCKET Socket,
    PSOCKADDR RemoteAddress
    )
{
  PWSK_PROVIDER_DATAGRAM_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_DATAGRAM_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    ControlSocketComplete,
    Socket,  // Use the socket object for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the IOCTL operation on the socket
  Status =
    Dispatch->WskControlSocket(
      Socket,
      WskIoctl,
 SIO_WSK_SET_REMOTE_ADDRESS,
      0,
      sizeof(SOCKADDR_IN),  // AF_INET
      RemoteAddress,
      0,
      NULL,
      NULL,
      Irp
      );

  // Return the status of the call to WskControlSocket()
  return Status;
}

// Control socket IoCompletion routine
NTSTATUS
  ControlSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_SOCKET Socket;

  // Check the result of the control operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the socket object from the context
    Socket = (PWSK_SOCKET)Context;

    // Perform the next operation on the socket
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

For more information about each of the supported socket IOCTL operations, see WSK Socket IOCTL Operations.

Enabling and Disabling Event Callback Functions

Article • 12/15/2021

A Winsock Kernel (WSK) application can implement event callback functions that the WSK subsystem calls asynchronously to notify the application when certain events occur on a socket. A WSK application can provide a client dispatch table structure to the WSK subsystem whenever it creates a socket or accepts a socket on a listening socket. This dispatch table contains pointers to the WSK application's event callback functions for the new socket. If a WSK application does not implement any event callback functions for a particular socket, then it does not need to provide a client dispatch table structure to the WSK subsystem for that socket.

All of a socket's event callback functions, except for a listening socket's *WskInspectEvent* and *WskAbortEvent* event callback functions, can be enabled or disabled by using the **SO_WSK_EVENT_CALLBACK** socket option. A WSK application can enable multiple event callback functions on a socket at the same time. However, a WSK application must disable each event callback function individually.

The following code example shows how a WSK application can use the SO_WSK_EVENT_CALLBACK socket option to enable the *WskDisconnectEvent* and *WskReceiveEvent* event callback functions on a connection-oriented socket.

```C++
// Function to enable the WskDisconnectEvent and WskReceiveEvent
// event callback functions on a connection-oriented socket
NTSTATUS
  EnableDisconnectAndRecieveCallbacks(
    PWSK_SOCKET Socket
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  WSK_EVENT_CALLBACK_CONTROL EventCallbackControl;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Specify the WSK NPI identifier
  EventCallbackControl.NpiId = &NPI_WSK_INTERFACE_ID;

  // Set the event flags for the event callback functions that
  // are to be enabled on the socket
  EventCallbackControl.EventMask =
    WSK_EVENT_DISCONNECT | WSK_EVENT_RECEIVE;

  // Initiate the control operation on the socket
  Status =
    Dispatch->WskControlSocket(
      Socket,
      WskSetOption,
      SO_WSK_EVENT_CALLBACK,
      SOL_SOCKET,
      sizeof(WSK_EVENT_CALLBACK_CONTROL),
      &EventCallbackControl,
      0,
      NULL,
      NULL,
      NULL  // No IRP for this control operation
      );

  // Return the status of the call to WskControlSocket()
  return Status;
}
```

The following code example shows how a WSK application can use the **SO_WSK_EVENT_CALLBACK** socket option to disable the *WskReceiveEvent* event callback functions on a connection-oriented socket.

C++

```
// Prototype for the disable disconnect IoCompletion routine
NTSTATUS
  DisableDisconnectComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to disable the WskDisconnectEvent event
// callback functions on a connection-oriented socket
NTSTATUS
  DisableDisconnectCallback(
    PWSK_SOCKET Socket
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  PIRP Irp;
  WSK_EVENT_CALLBACK_CONTROL EventCallbackControl;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);
  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
 DisableDisconnectComplete,
    Socket,  // Use the socket object for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Specify the WSK NPI identifier
  EventCallbackControl.NpiId = &NPI_WSK_INTERFACE_ID;

  // Set the event flag for the event callback function that
  // is to be disabled on the socket along with the disable flag
  EventCallbackControl.EventMask =
    WSK_EVENT_DISCONNECT | WSK_EVENT_DISABLE;

  // Initiate the control operation on the socket
  Status =
    Dispatch->WskControlSocket(
      Socket,
      WskSetOption,
      SO_WSK_EVENT_CALLBACK,
      SOL_SOCKET,
      sizeof(WSK_EVENT_CALLBACK_CONTROL),
      &EventCallbackControl,
      0,
      NULL,
      NULL,
      Irp
      );

  // Return the status of the call to WskControlSocket()
  return Status;
}

// Disable disconnect IoCompletion routine
NTSTATUS
  DisableDisconnectComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_SOCKET Socket;

  // Check the result of the control operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // The WskDisconnectEvent event callback
    // function is now disabled

    // Get the socket object from the context
    Socket = (PWSK_SOCKET)Context;

    // Perform the next operation on the socket
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

For listening sockets, the *WskInspectEvent* and *WskAbortEvent* event callback functions are enabled only if the WSK application enables conditional accept mode on the socket. A WSK application enables conditional accept mode on a listening socket by setting the **SO_CONDITIONAL_ACCEPT** socket option for the socket prior to binding the socket to a local transport address. For more information about how to set socket options, see Performing Control Operations on a Socket.

After conditional accept mode has been enabled on a listening socket, the socket's *WskInspectEvent* and *WskAbortEvent* event callback functions cannot be disabled. For more information about conditionally accepting incoming connections on listening sockets, see Listening for and Accepting Incoming Connections.

A listening socket can automatically enable event callback functions on connection-oriented sockets that are accepted by the listening socket's *WskAcceptEvent* event callback function. A WSK application automatically enables these callback functions by enabling the connection-oriented socket event callback functions on the listening socket. For more information about this process, see SO_WSK_EVENT_CALLBACK.

If a WSK application always enables certain event callback functions on every socket that it creates, the application can configure the WSK subsystem to automatically enable those event callback functions by using the WSK_SET_STATIC_EVENT_CALLBACKS client control operation. The event callback functions that are enabled in this manner are always enabled and cannot be disabled or re-enabled later by the WSK application. If a WSK application always enables certain event callback functions on every socket that it creates, the application should use this method to automatically enable those event callback functions because it will yield much better performance.

The following code example shows how a WSK application can use the WSK_SET_STATIC_EVENT_CALLBACKS client control operation to automatically enable the *WskReceiveFromEvent* event callback function on datagram sockets and the *WskReceiveEvent* event callback function on connection-oriented sockets.

C++

```
// Function to set static event callbacks
NTSTATUS
  SetStaticEventCallbacks(
    PWSK_APP_BINDING_CONTEXT BindingContext,
    )
{
  WSK_EVENT_CALLBACK_CONTROL EventCallbackControl;
  NTSTATUS Status;

  // Specify the WSK NPI identifier
  EventCallbackControl.NpiId = &NPI_WSK_INTERFACE_ID;

  // Set the event flags for the event callback functions that
  // are to be automatically enabled on every new socket
  EventCallbackControl.EventMask =
    WSK_EVENT_RECEIVE_FROM | WSK_EVENT_RECEIVE;

  // Perform client control operation
  Status =
    BindingContext->
      WskProviderDispatch->
        WskControlClient(
          BindingContext->WskClient,
          WSK_SET_STATIC_EVENT_CALLBACKS,
          sizeof(WSK_EVENT_CALLBACK_CONTROL),
          &EventCallbackControl,
          0,
          NULL,
          NULL,
          NULL  // No IRP for this control operation
          );

  // Return status of client control operation
  return Status;
}
```

If a WSK application uses the WSK_SET_STATIC_EVENT_CALLBACKS client control operation to automatically enable certain event callback functions, it must do so before it creates any sockets.

Binding a Socket to a Transport Address

Article • 12/15/2021

After a Winsock Kernel (WSK) application has successfully created a socket, it can bind that socket to a local transport address. A listening socket must be bound to a local transport address before it can accept incoming connections. A datagram socket must be bound to a local transport address before it can send or receive datagrams. A connection-oriented socket must be bound to a local transport address before it can connect to a remote transport address.

Note Basic sockets do not support sending or receiving network data. Therefore, a WSK application cannot bind a basic socket to a local transport address.

A WSK application binds a socket to a local transport address by calling the WskBind function. The **WskBind** function is pointed to by the **WskBind** member of the socket's provider dispatch structure. A socket's provider dispatch structure is pointed to by the **Dispatch** member of the socket object structure (WSK_SOCKET) that was returned by the WSK subsystem during the creation of the socket.

A socket can be bound to a local wildcard address. For more information about the behavior of a socket that has been bound to a local wildcard address, see **WskBind**.

The following code example shows how a WSK application can bind a listening socket to a local transport address.

```
C++

// Prototype for the bind IoCompletion routine
NTSTATUS
  BindComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to bind a listening socket to a local transport address
NTSTATUS
  BindListeningSocket(
    PWSK_SOCKET Socket,
    PSOCKADDR LocalAddress
    )
{
  PWSK_PROVIDER_LISTEN_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_LISTEN_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    BindComplete,
    Socket,  // Use the socket object for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the bind operation on the socket
  Status =
    Dispatch->WskBind(
      Socket,
      LocalAddress,
      0,  // No flags
      Irp
      );

  // Return the status of the call to WskBind()
  return Status;
}

// Bind IoCompletion routine
NTSTATUS
  BindComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_SOCKET Socket;

  // Check the result of the bind operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the socket object from the context
    Socket = (PWSK_SOCKET)Context;

    // Perform the next operation on the socket
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

For connection-oriented sockets, a WSK application can call the **WskSocketConnect** function to create, bind, and connect a socket in a single function call.

Listening for and Accepting Incoming Connections

Article • 12/15/2021

After a Winsock Kernel (WSK) application binds a listening socket to a local transport address, the socket begins listening for incoming connections from remote transport addresses. A WSK application can accept an incoming connection on a listening socket by calling the WskAccept function. The IRP that the application passes to the **WskAccept** function is queued until an incoming connection arrives.

The following code example shows how a WSK application can accept an incoming connection by calling the **WskAccept** function.

```C++
// Prototype for the accept IoCompletion routine
NTSTATUS
  AcceptComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to accept an incoming connection
NTSTATUS
  AcceptConnection(
    PWSK_SOCKET Socket,
    PVOID AcceptSocketContext,
    PWSK_CLIENT_CONNECTION_DISPATCH AcceptSocketDispatch
    )
{
  PWSK_PROVIDER_LISTEN_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_LISTEN_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    AcceptComplete,
    AcceptSocketContext,
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the accept operation on the socket
  Status =
    Dispatch->WskAccept(
      Socket,
      0,  // No flags
      AcceptSocketContext,
      AcceptSocketDispatch,
      NULL,
      NULL,
      Irp
      );

  // Return the status of the call to WskAccept()
  return Status;
}

// The accept IoCompletion routine
NTSTATUS
  AcceptComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_SOCKET Socket;
  PVOID AcceptSocketContext;

  // Check the result of the accept operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the accepted socket object from the IRP
    Socket = (PWSK_SOCKET)(Irp->IoStatus.Information);

    // Get the accepted socket's context
    AcceptSocketContext = Context;

    // Perform the next operation on the accepted socket
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

As an alternative to calling the **WskAccept** function to accept incoming connections on a listening socket, a WSK application can enable the *WskAcceptEvent* event callback function on the socket. If a WSK application enables the *WskAcceptEvent* event callback function on a listening socket, the WSK subsystem calls the socket's *WskAcceptEvent* event callback function whenever a new incoming connection is accepted on the socket. For more information about enabling a listening socket's *WskAcceptEvent* event callback function, see Enabling and Disabling Event Callback Functions.

The following code example shows how a WSK application can accept an incoming connection by the WSK subsystem calling a listening socket's *WskAcceptEvent* event callback function.

C++

```
// Dispatch table of event callback functions for accepted sockets
const WSK_CLIENT_CONNECTION_DISPATCH ConnectionDispatch =
{
  .
  . // Function pointers for the event callback functions
  .
};

// Pool tag used for allocating the socket context
#define SOCKET_CONTEXT_POOL_TAG 'tpcs'

// A listening socket's WskAcceptEvent event callback function
NTSTATUS WSKAPI
  WskAcceptEvent(
    PVOID SocketContext,
    ULONG Flags,
    PSOCKADDR LocalAddress,
    PSOCKADDR RemoteAddress,
    PWSK_SOCKET AcceptSocket,
    PVOID *AcceptSocketContext,
    CONST WSK_CLIENT_CONNECTION_DISPATCH **AcceptSocketDispatch
    )
{
  PWSK_APP_SOCKET_CONTEXT SocketContext;

  // Check for a valid new socket
  if (AcceptSocket != NULL)
  {
    // Allocate the socket context
    SocketContext =
      (PWSK_APP_SOCKET_CONTEXT)
        ExAllocatePoolWithTag(
          NonPagedPool,
          sizeof(WSK_APP_SOCKET_CONTEXT),
          SOCKET_CONTEXT_POOL_TAG
          );

    // Check result of allocation
    if (SocketContext == NULL)
    {
      // Reject the socket
      return STATUS_REQUEST_NOT_ACCEPTED;
    }

    // Initialize the socket context
    SocketContext->Socket = AcceptSocket;
    ...

    // Set the accepted socket's client context
    *AcceptSocketContext = SocketContext;

    // Set the accepted socket's dispatch table of callback functions
    *AcceptSocketDispatch = ConnectionDispatch;

    // Perform additional operations on the accepted socket
    ...

    // Return status indicating that the socket was accepted
    return STATUS_SUCCESS:
  }

  // Error with listening socket
  else
  {
    // Handle error
    ...

    // Return status indicating that no socket was accepted
    return STATUS_REQUEST_NOT_ACCEPTED;
  }
}
```

A WSK application can configure a listening socket to conditionally accept incoming connections that are received on the socket. A WSK application enables conditional accept mode on a listening socket by setting the **SO_CONDITIONAL_ACCEPT** socket option for the socket prior to binding the socket to a local transport address. For more information about how to set socket options, see Performing Control Operations on a Socket.

If conditional accept mode is enabled on a listening socket, the WSK subsystem first calls the socket's *WskInspectEvent* event callback function whenever a new incoming connection request is received on the socket. A socket's *WskInspectEvent* event callback function can inspect the incoming connection request to determine if the request should be accepted or rejected. To accept the request, the socket's *WskInspectEvent* event callback function returns **InspectAccept**. To reject the request, the socket's *WskInspectEvent* event callback function returns **InspectReject**. If a socket's *WskInspectEvent* event callback function cannot immediately determine if the request should be accepted or rejected, it returns **InspectPend**. In this situation, a WSK application must call the **WskInspectComplete** function after completing the inspection process for the incoming connection request. If an incoming connection request is dropped before the socket connection is fully established, the WSK subsystem calls the WSK application's *WskAbortEvent* event callback function.

The following code example shows how a WSK application can inspect an incoming connection request by the WSK subsystem calling the listening socket's *WskInspectEvent* event callback function.

C++

```
// Inspect ID for a pending inspection
WSK_INSPECT_ID PendingInspectID

// A listening socket's WskInspectEvent event callback function
WSK_INSPECT_ACTION WSKAPI
  WskInspectEvent(
    PVOID SocketContext,
    PSOCKADDR LocalAddress,
    PSOCKADDR RemoteAddress,
    PWSK_INSPECT_ID InspectID
    )
{
  // Check for a valid inspect ID
  if (InspectID != NULL)
  {
    // Inspect local and/or remote address of the incoming
    // connection request to determine if the connection should
    // be accepted or rejected.
    ...
    // If the incoming connection should be accepted
    if (...)
    {
      // Return status indicating that the incoming
      // connection request was accepted
      return InspectAccept;
    }

    // If the incoming connection should be rejected
    else if (...)
    {
      // Return status indicating that the incoming
      // connection request was rejected
      return InspectReject;
    }

    // Cannot determine immediately
    else
    {
      // Save the inspect ID while the inspection is pending.
      // This will be passed to WskInspectComplete when the
      // inspection process is completed.
      PendingInspectID = *InspectID;

      // Return status indicating that the result of the
      // inspection process for the incoming connection
      // request is pending
      return InspectPend;
    }
  }

  // Error with listening socket
  else
  {
    // Handle error
    ...

    // Return status indicating that a socket was not accepted
    return InspectReject;
  }
}

// A listening socket's WskAbortEvent event callback function
NTSTATUS WSKAPI
  WskAbortEvent(
    PVOID SocketContext,
    PWSK_INSPECT_ID InspectID
    )
{
  // Terminate the inspection for the incoming connection
  // request with a matching inspect ID. To test for a matching
  // inspect ID, the contents of the WSK_INSPECT_ID structures
  // must be compared, not the pointers to the structures.
  ...
}
```

If a WSK application determines that it will accept an incoming connection request on a listening socket that has conditional accept mode enabled, the incoming connection will be established and it can be accepted normally by either the application calling to the **WskAccept** function or the WSK subsystem calling the socket's *WskAcceptEvent* event callback function as described previously.

Establishing a Connection with a Destination

Article • 12/15/2021

After a Winsock Kernel (WSK) application has bound a connection-oriented socket to a local transport address, it can connect the socket to a remote transport address in order to establish a connection with the remote system. A WSK application must connect a connection-oriented socket to a remote transport address before it can send or receive data over the socket.

A WSK application connects a socket to a remote transport address by calling the WskConnect function. The **WskConnect** function is pointed to by the **WskConnect** member of the socket's provider dispatch structure. A socket's provider dispatch structure is pointed to by the **Dispatch** member of the socket object structure (WSK_SOCKET) that was returned by the WSK subsystem during the creation of the socket.

The following code example shows how a WSK application can connect a connection-oriented socket to a remote transport address.

C++

```
// Prototype for the connect IoCompletion routine
NTSTATUS
  ConnectComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to connect a socket to a remote transport address
NTSTATUS
  ConnectSocket(
    PWSK_SOCKET Socket,
    PSOCKADDR RemoteAddress
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    ConnectComplete,
    Socket,  // Use the socket object for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the connect operation on the socket
  Status =
    Dispatch->WskConnect(
      Socket,
      RemoteAddress,
      0,  // No flags
      Irp
      );

  // Return the status of the call to WskConnect()
  return Status;
}

// Connect IoCompletion routine
NTSTATUS
  ConnectComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_SOCKET Socket;

  // Check the result of the connect operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the socket object from the context
    Socket = (PWSK_SOCKET)Context;

    // Perform the next operation on the socket
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

A WSK application can call the **WskSocketConnect** function to create, bind, and connect a connection-oriented socket in a single function call.

Sending and receiving data over WSK sockets

Article • 12/15/2021

This section discusses sending and receiving data over Winsock Kernel (WSK) sockets and includes the following sections:

- Sending Data over a Datagram Socket
- Receiving Data over a Datagram Socket
- Sending Data over a Connection-Oriented Socket
- Receiving Data over a Connection-Oriented Socket

Sending Data over a Datagram Socket

Article • 12/15/2021

After a Winsock Kernel (WSK) application has bound a datagram socket to a local transport address it can send datagrams over the socket. A WSK application sends a datagram over a datagram socket by calling the **WskSendTo** function.

The following code example shows how a WSK application can send a datagram over a datagram socket.

```C++
// Prototype for the send datagram IoCompletion routine
NTSTATUS
  SendDatagramComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to send a datagram
NTSTATUS
  SendDatagram(
    PWSK_SOCKET Socket,
    PWSK_BUF DatagramBuffer,
    PSOCKADDR RemoteAddress
    )
{
  PWSK_PROVIDER_DATAGRAM_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_DATAGRAM_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }
  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    SendDatagramComplete,
    DatagramBuffer,  // Use the datagram buffer for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the send operation on the socket
  Status =
    Dispatch->WskSendTo(
      Socket,
      DatagramBuffer,
      0,  // No flags
      RemoteAddress,
      0,
      NULL,  // No associated control info
      Irp
      );

  // Return the status of the call to WskSendTo()
  return Status;
}

// Send datagram IoCompletion routine
NTSTATUS
  SendDatagramComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_BUF DatagramBuffer;
  ULONG ByteCount;

  // Check the result of the send operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the pointer to the datagram buffer
    DatagramBuffer = (PWSK_BUF)Context;

    // Get the number of bytes sent
    ByteCount = (ULONG)(Irp->IoStatus.Information);

    // Re-use or free the datagram buffer
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

If the WSK application has set either a fixed remote transport address or a fixed destination transport address for the datagram socket, the *RemoteAddress* parameter passed to the **WskSendTo** function is optional and can be **NULL**. If **NULL**, the datagram is sent to the fixed remote transport address or the fixed destination transport address. If non-**NULL**, the datagram is sent to the specified remote transport address.

For more information about setting a fixed remote transport address for a datagram socket, see SIO_WSK_SET_REMOTE_ADDRESS.

For more information about setting a fixed destination transport address for a datagram socket, see SIO_WSK_SET_SENDTO_ADDRESS.

Receiving Data over a Datagram Socket

Article • 12/15/2021

After a Winsock Kernel (WSK) application has bound a datagram socket to a local transport address it can receive datagrams over the socket. A WSK application receives a datagram over a datagram socket by calling the **WskReceiveFrom** function.

The following code example shows how a WSK application can receive a datagram over a datagram socket.

C++

```C++
// Prototype for the receive datagram IoCompletion routine
NTSTATUS
  ReceiveDatagramComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to receive a datagram
NTSTATUS
  ReceiveDatagram(
    PWSK_SOCKET Socket,
    PWSK_BUF DatagramBuffer
    )
{
  PWSK_PROVIDER_DATAGRAM_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_DATAGRAM_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    ReceiveDatagramComplete,
    DatagramBuffer,  // Use the datagram buffer for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the receive operation on the socket
  Status =
    Dispatch->WskReceiveFrom(
      Socket,
      DatagramBuffer,
      0,  // No flags are specified
      NULL,  // Not interested in the remote address
      NULL,  // Not interested in any associated control information
      NULL,
      NULL,
      Irp
      );

  // Return the status of the call to WskReceiveFrom()
  return Status;
}

// Receive datagram IoCompletion routine
NTSTATUS
  ReceiveDatagramComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_BUF DataBuffer;
  ULONG ByteCount;

  // Check the result of the receive operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the pointer to the data buffer
    DataBuffer = (PWSK_BUF)Context;

    // Get the number of bytes received
    ByteCount = (ULONG)(Irp->IoStatus.Information);

    // Process the received datagram
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

As an alternative to calling the **WskReceiveFrom** function to receive each datagram over a datagram socket, a WSK application can enable the *WskReceiveFromEvent* event callback function on the socket. If a WSK application enables the *WskReceiveFromEvent* event callback function on a datagram socket, the WSK subsystem calls the socket's *WskReceiveFromEvent* event callback function whenever new datagrams are received on the socket. For more information about enabling a datagram socket's *WskReceiveFromEvent* event callback function, see Enabling and Disabling Event Callback Functions.

The following code example shows how a WSK application can receive datagrams by the WSK subsystem by calling a datagram socket's *WskReceiveFromEvent* event callback function.

C++

```
// A datagram socket's WskReceiveFromEvent
// event callback function
NTSTATUS WSKAPI
  WskReceiveFromEvent(
    PVOID SocketContext,
    ULONG Flags,
    PWSK_DATAGRAM_INDICATION DataIndication
    )
{
  // Check for a valid data indication
  if (DataIndication != NULL)
  {
    // Loop through the list of data indication structures
    while (DataIndication != NULL)
    {
      // Process the data in the data indication structure
      ...

      // Move to the next data indication structure
      DataIndication = DataIndication->Next;
    }

    // Return status indicating the datagrams were received
    return STATUS_SUCCESS;
  }

  // Error
  else
  {
    // Close the socket
    ...

    // Return success since no datagrams were indicated
    return STATUS_SUCCESS;
  }
}
```

If a datagram socket's *WskReceiveFromEvent* event callback function does not retrieve all of the datagrams from the list of **WSK_DATAGRAM_INDICATION** structures pointed to by the *DataIndication* parameter, it can retain the list for further processing by returning STATUS_PENDING. In this situation, the WSK application must call the **WskRelease** function to release the list of WSK_DATAGRAM_INDICATION structures back to the WSK subsystem after it has completed retrieving all of the datagrams from the structures in the list.

Sending Data over a Connection-Oriented Socket

Article • 12/15/2021

After a Winsock Kernel (WSK) application has connected a connection-oriented socket to a remote transport address it can send data over the socket. A WSK application can also send data over a connection-oriented socket that it accepted on a listening socket. A WSK application sends data over a connection-oriented socket by calling the **WskSend** function.

The following code example shows how a WSK application can send data over a connection-oriented socket.

```C++
// Prototype for the send IoCompletion routine
NTSTATUS
  SendComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to send data
NTSTATUS
  SendData(
    PWSK_SOCKET Socket,
    PWSK_BUF DataBuffer
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    SendComplete,
    DataBuffer,  // Use the data buffer for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the send operation on the socket
  Status =
    Dispatch->WskSend(
      Socket,
      DataBuffer,
      0,  // No flags
      Irp
      );

  // Return the status of the call to WskSend()
  return Status;
}

// Send IoCompletion routine
NTSTATUS
  SendComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_BUF DataBuffer;
  ULONG ByteCount;

  // Check the result of the send operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the pointer to the data buffer
    DataBuffer = (PWSK_BUF)Context;

    // Get the number of bytes sent
    ByteCount = (ULONG)(Irp->IoStatus.Information);

    // Re-use or free the data buffer
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Receiving Data over a Connection-Oriented Socket

Article • 12/15/2021

After a Winsock Kernel (WSK) application has connected a connection-oriented socket to a remote transport address it can receive data over the socket. A WSK application can also receive data over a connection-oriented socket that it accepted on a listening socket. A WSK application receives data over a connection-oriented socket by calling the **WskReceive** function.

The following code example shows how a WSK application can receive data over a connection-oriented socket.

C++

```cpp
// Prototype for the receive IoCompletion routine
NTSTATUS
  ReceiveComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to receive data
NTSTATUS
  ReceiveData(
    PWSK_SOCKET Socket,
    PWSK_BUF DataBuffer
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    ReceiveComplete,
    DataBuffer,  // Use the data buffer for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the receive operation on the socket
  Status =
    Dispatch->WskReceive(
      Socket,
      DataBuffer,
      0,  // No flags are specified
      Irp
      );

  // Return the status of the call to WskReceive()
  return Status;
}

// Receive IoCompletion routine
NTSTATUS
  ReceiveComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_BUF DataBuffer;
  ULONG ByteCount;

  // Check the result of the receive operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the pointer to the data buffer
    DataBuffer = (PWSK_BUF)Context;

    // Get the number of bytes received
    ByteCount = (ULONG)(Irp->IoStatus.Information);

    // Process the received data
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

As an alternative to calling the **WskReceive** function to receive data over a connection-oriented socket, a WSK application can enable the *WskReceiveEvent* event callback function on the socket. If a WSK application enables the *WskReceiveEvent* event callback function on a connection-oriented socket, the WSK subsystem calls the socket's *WskReceiveEvent* event callback function whenever new data is received on the socket. For more information about enabling a connection-oriented socket's *WskReceiveEvent* event callback function, see Enabling and Disabling Event Callback Functions.

The following code example shows how a WSK application can receive data by the WSK subsystem calling a connection-oriented socket's *WskReceiveEvent* event callback function.

C++

```
// A connection-oriented socket's WskReceiveEvent
// event callback function
NTSTATUS WSKAPI
  WskReceiveEvent(
    PVOID SocketContext,
    ULONG Flags,
    PWSK_DATA_INDICATION DataIndication,
    SIZE_T BytesIndicated,
    SIZE_T *BytesAccepted
    )
{
  // Check for a valid data indication
  if (DataIndication != NULL)
  {
    // Loop through the list of data indication structures
    while (DataIndication != NULL)
    {
      // Process the data in the data indication structure
      ...

      // Move to the next data indication structure
      DataIndication = DataIndication->Next;
    }
    // Return status indicating the data was received
    return STATUS_SUCCESS;
  }

  // Error
  else
  {
    // Close the socket
    ...

    // Return success since no data was indicated
    return STATUS_SUCCESS;
  }
}
```

If a connection-oriented socket's *WskReceiveEvent* event callback function does not retrieve all of the data contained in the list of **WSK_DATA_INDICATION** structures pointed to by the *DataIndication* parameter, it can retain the list for further processing by returning STATUS_PENDING. In this situation, the WSK application must call the **WskRelease** function to release the list of WSK_DATA_INDICATION structures back to the WSK subsystem after it has completed retrieving all of the data from the structures in the list.

If a connection-oriented socket's *WskReceiveEvent* event callback function only accepts a portion of the total number of bytes of received data, it must set the variable pointed to by the *BytesAccepted* parameter to the number of bytes of data that were actually accepted. However, if the socket's *WskReceiveEvent* event callback function accepts all of the received data, it does not need to set the variable pointed to by the *BytesAccepted* parameter.

Disconnecting a Socket from a Destination

Article • 12/15/2021

When a Winsock Kernel (WSK) application has finished sending and receiving data over an established socket connection, it can disconnect the connection-oriented socket from the remote transport address to which it is connected. A WSK application disconnects a socket from a remote transport address by calling the **WskDisconnect** function. A WSK application can perform either an *abortive disconnect* or a *graceful disconnect* of the socket. For more information about the difference between an abortive disconnect and a graceful disconnect, see **WskDisconnect**.

The following code example shows how a WSK application can gracefully disconnect a connection-oriented socket from a remote transport address.

C++

```
// Prototype for the disconnect IoCompletion routine
NTSTATUS
  DisconnectComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to disconnect a socket from a remote transport address
NTSTATUS
  DisconnectSocket(
    PWSK_SOCKET Socket
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    DisconnectComplete,
    Socket,  // Use the socket object for the context
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the disconnect operation on the socket
  Status =
    Dispatch->WskDisconnect(
      Socket,
      NULL,  // No final data to be transmitted
      0,     // No flags (graceful disconnect)
      Irp
      );

  // Return the status of the call to WskDisconnect()
  return Status;
}

// Disconnect IoCompletion routine
NTSTATUS
  DisconnectComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_SOCKET Socket;

  // Check the result of the disconnect operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the socket object from the context
    Socket = (PWSK_SOCKET)Context;

    // Perform the next operation on the socket
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

If a WSK application performs a graceful disconnect of a socket, the application can send a final buffer of data to the remote transport address before the socket is disconnected by passing a pointer to a **WSK_BUF** structure to the **WskDisconnect** function.

If a WSK application closes a connection-oriented socket without first disconnecting the socket from the remote transport address to which it is connected, the WSK subsystem automatically performs an abortive disconnect of the socket prior to closing the socket. For more information about closing a socket, see Closing a Socket.

Closing a Socket

Article • 12/15/2021

When a Winsock Kernel (WSK) application has finished using a socket, it should close the socket and free up any associated resources. A WSK application must close all open sockets before the application can detach itself from the WSK subsystem. For more information about detaching a WSK application from the WSK subsystem, see Unregistering a Winsock Kernel Application.

A WSK application closes a socket by calling the **WskCloseSocket** function. Before calling the **WskCloseSocket** function, a WSK application must ensure that there are no other function calls in progress to any of the socket's functions, including any extension functions in any of the application's other threads. However, a WSK application can call **WskCloseSocket** if there are pending IRPs from prior calls to the socket's functions that have not yet completed.

The method that a WSK application uses to ensure that there are no other function calls in progress to any of the socket's functions before it calls the **WskCloseSocket** function is dependent upon the design of the application. For example, if a WSK application might need to close a socket in one thread while there could be calls in progress to that socket's other functions in one or more other threads, then the application would typically use a reference counter to track the number of function calls that are currently in progress on the socket. In this situation, the WSK application atomically tests and increments a socket's reference counter before it calls one of the socket's functions, and then atomically decrements the socket's reference counter when the function returns. When the reference counter is zero, the WSK application can safely call the **WskCloseSocket** function to close the socket.

On the other hand, if the design of the WSK application guarantees that there will not be any calls in progress to a particular socket's functions in any other threads when the application calls the **WskCloseSocket** function to close the socket, then the WSK application does not need to use a reference counter to track the number of function calls that are currently in progress on the socket. For example, if the WSK application performs all of its socket operations for a particular socket from a single thread, then the application can safely call the **WskCloseSocket** function from within that thread without the need for a reference counter.

Calling the **WskCloseSocket** function causes the WSK subsystem to cancel and complete all pending IRPs from prior calls to the socket's functions. The WSK subsystem also ensures that any event callback functions in progress have returned control back to the WSK subsystem before the close operation is completed.

The following code example shows how a WSK application can close a socket.

```C++
// Prototype for the socket close IoCompletion routine
NTSTATUS
  CloseSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    );

// Function to close a socket
NTSTATUS
  CloseSocket(
    PWSK_SOCKET Socket,
    PWSK_APP_SOCKET_CONTEXT SocketContext
    )
{
  PWSK_PROVIDER_BASIC_DISPATCH Dispatch;
  PIRP Irp;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_BASIC_DISPATCH)(Socket->Dispatch);

  // Allocate an IRP
  Irp =
    IoAllocateIrp(
      1,
      FALSE
      );

  // Check result
  if (!Irp)
  {
    // Return error
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Set the completion routine for the IRP
  IoSetCompletionRoutine(
    Irp,
    CloseSocketComplete,
    SocketContext,
    TRUE,
    TRUE,
    TRUE
    );

  // Initiate the close operation on the socket
  Status =
    Dispatch->WskCloseSocket(
      Socket,
      Irp
      );

  // Return the status of the call to WskCloseSocket()
  return Status;
}

// Socket close IoCompletion routine
NTSTATUS
  CloseSocketComplete(
    PDEVICE_OBJECT DeviceObject,
    PIRP Irp,
    PVOID Context
    )
{
  UNREFERENCED_PARAMETER(DeviceObject);

  PWSK_APP_SOCKET_CONTEXT SocketContext;

  // Check the result of the socket close operation
  if (Irp->IoStatus.Status == STATUS_SUCCESS)
  {
    // Get the pointer to the socket context
    SocketContext =
      (PWSK_APP_SOCKET_CONTEXT)Context;

    // Perform any cleanup and/or deallocation of the socket context
    ...
  }

  // Error status
  else
  {
    // Handle error
    ...
  }

  // Free the IRP
  IoFreeIrp(Irp);

  // Always return STATUS_MORE_PROCESSING_REQUIRED to
  // terminate the completion processing of the IRP.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

After a WSK application has called WskCloseSocket, it should not make any further calls to any of the socket's functions.

If a WSK application closes a connection-oriented socket that has not been previously disconnected in both directions, the WSK subsystem automatically performs an abortive disconnect of the socket before closing the socket. For more information about disconnecting a socket, see Disconnecting a Socket from a Destination.

Registering an Extension Interface

Article • 12/15/2021

After a Winsock Kernel (WSK) application has successfully created a socket, it can register that socket for one or more of the extension interfaces that are supported by the WSK subsystem. A WSK application determines which set of extension interfaces are supported by the WSK subsystem by examining the **Version** member of the **WSK_PROVIDER_DISPATCH** structure that the WSK subsystem returned to the application during attachment.

Each extension interface is defined by an NPI that is independent of the WSK NPI. Note, however, that the NPIs for extension interfaces do not support NPI-specific characteristics.

A WSK application registers for an extension interface by executing the **SIO_WSK_REGISTER_EXTENSION** socket IOCTL operation on the socket. For more information about executing socket IOCTL operations, see Performing Control Operations on a Socket.

If a WSK application attempts to register a socket for an extension interface that is not supported by the WSK subsystem, the SIO_WSK_REGISTER_EXTENSION socket IOCTL operation will return STATUS_NOT_SUPPORTED.

For example, suppose that an extension interface is defined as in the following code example.

```C++
const NPIID EXAMPLE_EXTIF_NPIID = {...};

typedef struct _EXAMPLE_EXTIF_PROVIDER_DISPATCH {
  .
  . // Function pointers for the functions that are
  . // defined by the extension interface.
  .
} EXAMPLE_EXTIF_PROVIDER_DISPATCH, *PEXAMPLE_EXTIF_PROVIDER_DISPATCH;

typedef struct _EXAMPLE_EXTIF_CLIENT_DISPATCH {
  .
  . // Function pointers for the callback functions
  . // that are defined by the extension interface.
  .
} EXAMPLE_EXTIF_CLIENT_DISPATCH, *PEXAMPLE_EXTIF_CLIENT_DISPATCH;
```

The following shows how a WSK application can register for this extension interface for a connection-oriented socket.

C++

```
// Client dispatch structure for the extension interface
const EXAMPLE_EXTIF_CLIENT_DISPATCH ExtIfClientDispatch = {
  .
  . // The WSK application's callback functions
  . // for the extension interface
  .
};

// Context structure type for the example extension interface
typedef struct _EXAMPLE_EXTIF_CLIENT_CONTEXT
{
  const EXAMPLE_EXTIF_PROVIDER_DISPATCH *ExtIfProviderDispatch;
  PVOID ExtIfProviderContext;
    .
    .  // Other application-specific members
    .
} EXAMPLE_EXTIF_CLIENT_CONTEXT, *PEXAMPLE_EXTIF_CLIENT_CONTEXT;

// Function to register the example extension interface
NTSTATUS
  RegisterExampleExtIf(
    PWSK_SOCKET Socket,
    PEXAMPLE_EXTIF_CLIENT_CONTEXT ExtIfClientContext
    )
{
  PWSK_PROVIDER_CONNECTION_DISPATCH Dispatch;
  WSK_EXTENSION_CONTROL_IN ExtensionControlIn;
  WSK_EXTENSION_CONTROL_OUT ExtensionControlOut;
  NTSTATUS Status;

  // Get pointer to the socket's provider dispatch structure
  Dispatch =
    (PWSK_PROVIDER_CONNECTION_DISPATCH)(Socket->Dispatch);

  // Fill in the WSK_EXTENSION_CONTROL_IN structure
  ExtensionControlIn.NpiId = &EXAMPLE_EXTIF_NPIID;
  ExtensionControlIn.ClientContext = ExtIfClientContext;
  ExtensionControlIn.ClientDispatch = &ExtIfClientDispatch;

  // Initiate the IOCTL operation on the socket
  Status =
    Dispatch->WskControlSocket(
      Socket,
      WskIoctl,
      SIO_WSK_REGISTER_EXTENSION,
      0,
      sizeof(WSK_EXTENSION_CONTROL_IN),
      &ExtensionControlIn,
      sizeof(WSK_EXTENSION_CONTROL_OUT),
      &ExtensionControlOut,
      NULL,
      NULL  // No IRP used for this IOCTL operation
      );

  // Check result
  if (Status == STATUS_SUCCESS)
  {
    // Save provider dispatch table and provider context
    ExtIfClientContext->ExtIfProviderDispatch =
      (const EXAMPLE_EXTIF_PROVIDER_DISPATCH *)
        ExtensionControlOut.ProviderDispatch;
    ExtIfClientContext->ExtIfProviderContext =
      ExtensionControlOut.ProviderContext;
  }

  // Return the status of the call to WskControlSocket()
  return Status;
}
```

A WSK application registers for extension interfaces on a socket-by-socket basis.

Unregistering a Winsock Kernel Application

Article • 12/15/2021

A Winsock Kernel (WSK) application that has successfully registered as a WSK client with the WskRegister function must ensure that WskDeregister has been called, and the call has returned, before the driver's Unload function returns. A WSK application that has called **WskRegister** successfully should never unload without calling **WskDeregister**, which will wait to unregister the WSK client object until all captured instances of the WSK provider NPI are released with WskReleaseProviderNPI and all sockets are closed by the WSK application. If there are pending IRPs that were passed to the functions in WSK_PROVIDER_DISPATCH, **WskDeregister** will also wait until those pending IRPs complete. A WSK application should never call the functions in WSK_PROVIDER_DISPATCH after **WskReleaseProviderNPI** is called.

For example:

```C++
// Unload function
VOID
  Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
  // Unregister the WSK application
  WskDeregister(
    &WskRegistration
    );

}
```

A WSK application is not necessarily required to always call **WskDeregister** from within its *Unload* function. For example, if a WSK application is a subcomponent of a complex driver, the WSK application's call to **WskDeregister** might occur when the WSK application subcomponent is deactivated. In such a scenario, before the driver returns from its *Unload* function, it must still ensure that the WSK application has been successfully unregistered with a call to **WskDeregister**.

Resolving Host Names and IP Addresses

Article • 12/15/2021

Beginning with Windows 7, a *kernel name resolution* feature allows kernel-mode components to perform protocol-independent translation between Unicode host names and transport addresses. You can use the following Winsock Kernel (WSK) client-level functions to perform this name resolution:

- **WskFreeAddressInfo**
- **WskGetAddressInfo**
- **WskGetNameInfo**

These functions perform name-address translation similarly to the user-mode functions **FreeAddrInfoW**, **GetAddrInfoW**, and **GetNameInfoW**, respectively.

To take advantage of this feature, you must compile or recompile your driver with the NTDDI_VERSION macro set to NTDDI_WIN7 or greater.

In order for your driver to use kernel name resolution functionality, it must perform the following calling sequence:

1. Call **WskRegister** to register with WSK.
2. Call **WskCaptureProviderNPI** to capture the WSK provider Network Programming Interface (NPI).
3. When you are done using the WSK provider NPI, call **WskReleaseProviderNPI** to release the WSK provider NPI.
4. Call **WskDeregister** to deregister the WSK application.

The following code example uses the above calling sequence to show how a WSK application can call the **WskGetAddressInfo** function to translate a host name to a transport address.

C++

```
NTSTATUS
SyncIrpCompletionRoutine(
    __in PDEVICE_OBJECT Reserved,
    __in PIRP Irp,
    __in PVOID Context
    )
{
    PKEVENT compEvent = (PKEVENT)Context;
    UNREFERENCED_PARAMETER(Reserved);
    UNREFERENCED_PARAMETER(Irp);
    KeSetEvent(compEvent, 2, FALSE);
    return STATUS_MORE_PROCESSING_REQUIRED;
}

NTSTATUS
KernelNameResolutionSample(
    __in PCWSTR NodeName,
    __in_opt PCWSTR ServiceName,
    __in_opt PADDRINFOEXW Hints,
    __in PWSK_PROVIDER_NPI WskProviderNpi
    )
{
    NTSTATUS status;
    PIRP irp;
    KEVENT completionEvent;
    UNICODE_STRING uniNodeName, uniServiceName, *uniServiceNamePtr;
    PADDRINFOEXW results;

    PAGED_CODE();
    //
    // Initialize UNICODE_STRING structures for NodeName and ServiceName
    //

    RtlInitUnicodeString(&uniNodeName, NodeName);

    if(ServiceName == NULL) {
        uniServiceNamePtr = NULL;
    }
    else {
        RtlInitUnicodeString(&uniServiceName, ServiceName);
        uniServiceNamePtr = &uniServiceName;
    }

    //
    // Use an event object to synchronously wait for the
    // WskGetAddressInfo request to be completed.
    //

    KeInitializeEvent(&completionEvent, SynchronizationEvent, FALSE);

    //
    // Allocate an IRP for the WskGetAddressInfo request, and set the
    // IRP completion routine, which will signal the completionEvent
    // when the request is completed.
    //

    irp = IoAllocateIrp(1, FALSE);
    if(irp == NULL) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    IoSetCompletionRoutine(irp, SyncIrpCompletionRoutine,
  &completionEvent, TRUE, TRUE, TRUE);

    //
    // Make the WskGetAddressInfo request.
    //

    WskProviderNpi->Dispatch->WskGetAddressInfo (
        WskProviderNpi->Client,
        &uniNodeName,
        uniServiceNamePtr,
        NS_ALL,
        NULL, // Provider
        Hints,
        &results,
        NULL, // OwningProcess
        NULL, // OwningThread
        irp);

    //
    // Wait for completion. Note that processing of name resolution results
    // can also be handled directly within the IRP completion routine, but
    // for simplicity, this example shows how to wait synchronously for
    // completion.
    //

    KeWaitForSingleObject(&completionEvent, Executive,
                        KernelMode, FALSE, NULL);

    status = irp->IoStatus.Status;

    IoFreeIrp(irp);

    if(!NT_SUCCESS(status)) {
        return status;
    }

    //
    // Process the name resolution results by iterating through the
addresses
    // within the returned ADDRINFOEXW structure.
    //

   results; // your code here

    //
    // Release the returned ADDRINFOEXW structure when no longer needed.
    //

    WskProviderNpi->Dispatch->WskFreeAddressInfo(
        WskProviderNpi->Client,
        results);

    return status;
}
```

WSK_CACHE_SD

Article • 03/03/2023

A WSK application uses the WSK_CACHE_SD client control operation to obtain a cached copy of a security descriptor that can be passed to the WskSocket, WskSocketConnect, and WskControlSocket functions.

To obtain a cached copy of a security descriptor, a WSK application calls the WskControlClient function with the following parameters.

Parameter	Value
ControlCode	WSK_CACHE_SD
InputSize	sizeof(PSECURITY_DESCRIPTOR)
InputBuffer	A pointer to a PSECURITY_DESCRIPTOR-typed variable. This variable contains a pointer to the SECURITY_DESCRIPTOR structure that defines the uncached security descriptor that is being cached.
OutputSize	sizeof(PSECURITY_DESCRIPTOR)
OutputBuffer	A pointer to a PSECURITY_DESCRIPTOR-typed variable. This variable receives a pointer to a SECURITY_DESCRIPTOR structure that describes the cached security descriptor.
OutputSizeReturned	**NULL**
Irp	**NULL**

A WSK application must release the cached copy of the security descriptor by using the WSK_RELEASE_SD client control operation when the security descriptor is no longer needed.

See the reference page for the SECURITY_DESCRIPTOR structure for more information.

The *Irp* parameter must be **NULL** for this client control operation.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Wsk.h (include Wsk.h)

WSK_RELEASE_SD

Article • 03/03/2023

A WSK application uses the WSK_RELEASE_SD client control operation to release a cached copy of a security descriptor that either was previously obtained by using the WSK_CACHE_SD client control operation or was retrieved by using the SO_WSK_SECURITY socket option.

To release a cached copy of a security descriptor, a WSK application calls the WskControlClient function with the following parameters.

Parameter	Value
ControlCode	WSK_RELEASE_SD
InputSize	sizeof(PSECURITY_DESCRIPTOR)
InputBuffer	A pointer to a PSECURITY_DESCRIPTOR-typed variable. This variable contains a pointer to the SECURITY_DESCRIPTOR structure that defines the cached security descriptor that is being released.
OutputSize	0
OutputBuffer	**NULL**
OutputSizeReturned	**NULL**
Irp	**NULL**

See the reference page for the SECURITY_DESCRIPTOR structure for more information.

The *Irp* parameter must be **NULL** for this client control operation.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

WSK_SET_STATIC_EVENT_CALLBACKS

Article • 03/03/2023

A WSK application uses the WSK_SET_STATIC_EVENT_CALLBACKS client control operation to automatically enable certain event callback functions on every socket that it creates. The event callback functions that are enabled in this manner are always enabled and cannot be disabled or re-enabled later by the WSK application. However, if a WSK application always enables certain event callback functions on every socket that it creates, the application should use this method to automatically enable those event callback functions because it will yield much better performance.

If a WSK application uses the WSK_SET_STATIC_EVENT_CALLBACKS client control operation, it must do so before it creates any sockets.

To automatically enable certain event callback functions on every socket it creates, a WSK application calls the **WskControlClient** function with the following parameters.

Parameter	Value
ControlCode	WSK_SET_STATIC_EVENT_CALLBACKS
InputSize	sizeof(WSK_EVENT_CALLBACK_CONTROL)
InputBuffer	A pointer to a **WSK_EVENT_CALLBACK_CONTROL** structure that specifies the desired event callback functions to be automatically enabled
OutputSize	0
OutputBuffer	**NULL**
OutputSizeReturned	**NULL**
Irp	**NULL**

A WSK application can specify a combination of event flags for different socket types in the **EventMask** member of the **WSK_EVENT_CALLBACK_CONTROL** structure. When the WSK application creates a new socket, the WSK subsystem will automatically enable the appropriate event callback functions for the specific category of WSK socket that is being created.

For more information about the event flags for the standard WSK event callback functions, see **SO_WSK_EVENT_CALLBACK**.

For more information about enabling and disabling a socket's event callback functions, see Enabling and Disabling Event Callback Functions.

The *Irp* parameter must be **NULL** for this client control operation.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

WSK_TDI_BEHAVIOR

Article • 03/03/2023

Note The TDI feature is deprecated and will be removed in future versions of Microsoft Windows.

A WSK application uses the WSK_TDI_BEHAVIOR client control operation to control whether the WSK subsystem will divert network I/O to TDI transports. A WSK application uses this client control operation only if it needs to override the default behavior of the WSK subsystem.

If a WSK application uses the WSK_TDI_BEHAVIOR client control operation, it must do so before it creates any sockets.

To control whether the WSK subsystem will divert network I/O to TDI transports, a WSK application calls the **WskControlClient** function with the following parameters.

Parameter	Value
ControlCode	WSK_TDI_BEHAVIOR
InputSize	sizeof(ULONG)
InputBuffer	A pointer to a ULONG typed variable that contains flags that control the behavior of the WSK subsystem.
OutputSize	0
OutputBuffer	**NULL**
OutputSizeReturned	**NULL**
Irp	**NULL**

The following flags are defined for the WSK_TDI_BEHAVIOR client control operation.

WSK_TDI_BEHAVIOR_BYPASS_TDI
If a native WSK transport exists for the address family, socket type, and protocol that are specified when the WSK application creates a socket, then, if this flag is set, the WSK subsystem ignores any TDI filter drivers and always uses the native WSK transport.

The default behavior is that if a TDI filter driver is detected for the address family, socket type, and protocol that are specified when the WSK application creates a new socket, the WSK subsystem diverts the network I/O for the new socket to the TDI transport so that the network traffic and other socket operations pass through the TDI filter driver.

The *Irp* parameter must be **NULL** for this client control operation.

Note TDI is not supported in Microsoft Windows versions after Windows Vista.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

WSK_TDI_DEVICENAME_MAPPING

Article • 03/03/2023

A WSK application uses the WSK_TDI_DEVICENAME_MAPPING client control operation to map combinations of address family, socket type, and protocol to device names of TDI transports. A WSK application uses this client control operation only if it requires support for TDI transports. When a WSK application creates a socket, the WSK subsystem refer to the list of mappings only if there is no native support for the combination of address family, socket type, and protocol specified by the WSK application.

If a WSK application uses the WSK_TDI_DEVICENAME_MAPPING client control operation to map combinations of address family, socket type, and protocol to device names of TDI transports, it must do so before it creates any sockets.

To map combinations of address family, socket type, and protocol to device names of TDI transports, a WSK application calls the **WskControlClient** function with the following parameters.

Parameter	Value
ControlCode	WSK_TDI_DEVICENAME_MAPPING
InputSize	sizeof(WSK_TDI_MAP_INFO)
InputBuffer	A pointer to a **WSK_TDI_MAP_INFO** structure that contains a list of mappings of combinations of address family, socket type, and protocol to TDI device names.
OutputSize	0
OutputBuffer	**NULL**
OutputSizeReturned	**NULL**
Irp	**NULL**

For more information about using TDI transports, see Using TDI Transports.

The *Irp* parameter must be **NULL** for this client control operation.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

WSK_TRANSPORT_LIST_CHANGE

Article • 03/03/2023

A WSK application uses the WSK_TRANSPORT_LIST_CHANGE client control operation to receive notification if the list of available network transports changes.

To receive notification of when the list of available network transports changes, a WSK application calls the WskControlClient function with the following parameters.

Parameter	Value
ControlCode	WSK_TRANSPORT_LIST_CHANGE
InputSize	0
InputBuffer	NULL
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL
Irp	A pointer to an IRP that is queued by the WSK subsystem until the list of available network transports changes. The WSK subsystem will complete the IRP after either a new network transport is added or an existing network transport is removed.

An IRP is required for this client control operation.

The WSK subsystem will cancel any pending IRPs if the WSK application calls WskDeregister to detach itself from the WSK subsystem.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

WSK_TRANSPORT_LIST_QUERY

Article • 03/03/2023

A WSK application uses the WSK_TRANSPORT_LIST_QUERY client control operation to retrieve a list of available network transports that can be specified when creating a new socket.

To retrieve a list of available network transports, a WSK application calls the WskControlClient function with the following parameters.

Parameter	Value
ControlCode	WSK_TRANSPORT_LIST_QUERY
InputSize	0
InputBuffer	**NULL**
OutputSize	The size, in bytes, of the array of structures that is pointed to by the *OutputBuffer* parameter
OutputBuffer	A pointer to an array of WSK_TRANSPORT structures that receives the list of available network transports
OutputSizeReturned	A pointer to a SIZE_T-typed variable that receives the number of bytes of data that are copied into the array of structures that is pointed to by the *OutputBuffer* parameter
Irp	**NULL**

A WSK application can specify zero in the *OutputSize* parameter and **NULL** in the *OutputBuffer* parameter to determine the size of the array of WSK_TRANSPORT structures, in bytes, that is required to contain the complete list of available network transports. In such a situation, the call to the WskControlClient function returns STATUS_BUFFER_OVERFLOW, and the variable that is pointed to by the *OutputSizeReturned* parameter contains the required buffer size. The application can then allocate a buffer that is large enough to contain the complete list of available network transports and can call the **WskControlClient** function a second time, specifying the parameters that are shown in the preceding table.

The *Irp* parameter must be **NULL** for this client control operation.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SIO_WSK_QUERY_IDEAL_SEND_BACKLOG

Article • 03/03/2023

The SIO_WSK_QUERY_IDEAL_SEND_BACKLOG socket I/O control operation allows a WSK application to query the ideal send backlog size for a connection-oriented socket. This socket I/O control operation applies only to connection-oriented sockets.

The ideal send backlog size for a connection-oriented socket is the optimal amount of send data that needs to be kept outstanding (that is, passed to the WSK subsystem but not yet completed) to keep the socket's data stream full at all times. A WSK application can use this size to incrementally probe and lock the buffers of data to be sent based on the underlying connection's flow control state.

If a WSK application uses this socket I/O control operation to query the ideal send backlog size, it must do so after the connection-oriented socket has been connected to a remote transport address.

To query the ideal send backlog size for a connection-oriented socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_QUERY_IDEAL_SEND_BACKLOG
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(SIZE_T)
OutputBuffer	A pointer to a SIZE_T-typed variable that receives the current ideal send backlog size
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to query the ideal send backlog size for a connection-oriented socket.

A connection-oriented socket can be notified of changes to the ideal send backlog size by enabling its *WskSendBacklogEvent* event callback function.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SIO_WSK_QUERY_INSPECT_ID

Article • 03/03/2023

The SIO_WSK_QUERY_INSPECT_ID socket I/O control operation allows a WSK application to query the inspection identification data for a connection-oriented socket that has been successfully accepted on a listening socket that has conditional accept mode enabled. This socket I/O control operation applies only to connection-oriented sockets that have been accepted on a listening socket that has conditional accept mode enabled.

To query the inspection identification data for a connection-oriented socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_QUERY_INSPECT_ID
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(WSK_INSPECT_ID)
OutputBuffer	A pointer to a WSK_INSPECT_ID structure that receives the inspection identification data
OutputSizeReturned	A pointer to a ULONG-typed variable that receives the number of bytes of data that is copied into the buffer that is pointed to by the *OutputBuffer* parameter.
Irp	NULL

If a WSK application calls the **WskControlSocket** function to query the inspection identification data for any socket other than a connection-oriented socket that was accepted on a listening socket that has conditional accept mode enabled, the **WskControlSocket** function returns STATUS_INVALID_DEVICE_REQUEST.

For more information about conditionally accepting connections, see Listening for and Accepting Incoming Connections.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SIO_WSK_QUERY_RECEIVE_BACKLOG

Article • 03/03/2023

The SIO_WSK_QUERY_RECEIVE_BACKLOG socket I/O control operation allows a WSK application to query the current backlog of received data for a connection-oriented socket. This socket I/O control operation applies only to connection-oriented sockets.

If a WSK application uses this socket I/O control operation to query the receive backlog, it must do so after the connection-oriented socket has been connected to a remote transport address.

To query the current backlog of received data for a connection-oriented socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_QUERY_RECEIVE_BACKLOG
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(SIZE_T)
OutputBuffer	A pointer to a SIZE_T-typed variable that receives the current backlog of received data
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to query the current backlog of received data for a connection-oriented socket.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SIO_WSK_REGISTER_EXTENSION

Article • 03/03/2023

The SIO_WSK_REGISTER_EXTENSION socket I/O control operation allows a WSK application to register for an extension interface that is supported by the WSK subsystem. This socket I/O control operation applies to all socket types.

To register an extension interface, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_REGISTER_EXTENSION
Level	0
InputSize	sizeof(WSK_EXTENSION_CONTROL_IN)
InputBuffer	A pointer to a **WSK_EXTENSION_CONTROL_IN** structure. This structure contains a pointer to the Network Programming Interface (NPI) identifier for the extension interface and pointers to the dispatch table and to the context for the WSK application's implementation of the extension interface.
OutputSize	sizeof(WSK_EXTENSION_CONTROL_OUT)
OutputBuffer	A pointer to a **WSK_EXTENSION_CONTROL_OUT** structure. This structure receives a pointer to the dispatch table and a pointer to the context for the WSK subsystem's implementation of the extension interface.
OutputSizeReturned	NULL

A WSK application does not specify a pointer to an IRP when calling the **WskControlSocket** function to register an extension interface.

The contents of the dispatch table structures are extension interface-specific.

For more information about registering an extension interface, see Registering an Extension Interface.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SIO_WSK_SET_REMOTE_ADDRESS

Article • 03/03/2023

The SIO_WSK_SET_REMOTE_ADDRESS socket I/O control operation allows a WSK application to specify a fixed remote transport address for a datagram socket. This socket I/O control operation applies only to datagram sockets.

If a WSK application sets a fixed remote transport address for a datagram socket, all datagrams that are sent over the socket are sent to the fixed remote transport address, and only datagrams that are received from the fixed remote transport address are accepted.

A WSK application can override a fixed remote transport address when it sends a datagram over the socket by specifying an alternative remote transport address in the *RemoteAddress* parameter when calling the WskSendTo function. In this situation, the datagram is sent to the alternative remote transport address instead of the fixed remote transport address. However, any responses that are sent back from an alternative remote transport address will not be accepted.

If a WSK application uses this socket I/O control operation to specify a fixed remote transport address, it must do so after the datagram socket has been bound to a local transport address.

To set a fixed remote transport address for a datagram socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_SET_REMOTE_ADDRESS
Level	0
InputSize	The size of the SOCKADDR structure pointed to by the *InputBuffer* parameter.
InputBuffer	A pointer to a structure that specifies a fixed remote transport address for the datagram socket. The pointer must be a pointer to the specific SOCKADDR structure type that corresponds to the address family that the WSK application specified when it created the datagram socket.

Parameter	Value
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

To clear a fixed remote transport address for a datagram socket, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_SET_REMOTE_ADDRESS
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or clear a fixed remote transport address for a datagram socket.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SIO_WSK_SET_SENDTO_ADDRESS

Article • 03/03/2023

The SIO_WSK_SET_SENDTO_ADDRESS socket I/O control operation allows a WSK application to specify a fixed destination transport address for a datagram socket. This socket I/O control operation applies only to datagram sockets.

If a WSK application sets a fixed destination transport address for a datagram socket, all datagrams that are sent over the socket are sent to the fixed destination transport address. However, datagrams that are received on the socket will be accepted from any transport address.

A WSK application can override a fixed destination transport address when it sends a datagram over the socket by specifying an alternative remote transport address in the *RemoteAddress* parameter when calling the WskSendTo function. In this situation, the datagram is sent to the alternative remote transport address instead of the fixed destination transport address.

If a WSK application uses this socket I/O control operation to specify a fixed destination transport address, it must do so after the datagram socket has been bound to a local transport address.

To set a fixed destination transport address for a datagram socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_SET_SENDTO_ADDRESS
Level	0
InputSize	The size of the SOCKADDR structure that is pointed to by the *InputBuffer* parameter.
InputBuffer	A pointer to a structure that specifies a fixed destination transport address for the datagram socket. The pointer must be a pointer to the specific SOCKADDR structure type that corresponds to the address family that the WSK application specified when it created the datagram socket.
OutputSize	0

Parameter	Value
OutputBuffer	NULL
OutputSizeReturned	NULL

To clear a fixed destination transport address for a datagram socket, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_WSK_SET_SENDTO_ADDRESS
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or clear a fixed destination transport address for a datagram socket.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SIO_WSK_SET_TCP_SILENT_MODE control code

Article • 03/03/2023

The **SIO_WSK_SET_TCP_SILENT_MODE** socket I/O control operation allows a WSK client to enable silent mode on the TCP connection.

A TCP connection in silent mode will not send any data or control packets on the wire. This socket I/O control operation applies only to connected TCP sockets. It is not supported on loopback.

To perform this operation, call the WskControlSocket function with the following parameters.

Parameters

RequestType [in]
Use **WskIoctl** for this operation.

ControlCode [in]
The control code for the operation. Use **SIO_WSK_SET_TCP_SILENT_MODE** for this operation.

Level
Use zero for this operation.

InputSize [in]
Use zero for this operation.

InputBuffer [in]
Use **NULL** for this operation.

OutputSize [out]
Use zero for this operation.

OutputBuffer [in]
Use **NULL** for this operation.

OutputSizeReturned [out]
Use **NULL** for this operation.

Remarks

A WSK application must specify a pointer to an IRP when calling the WskControlSocket function to enable silent mode.

The WSK application before calling WskControlSocket to enable silent mode must ensure that there are no pending send or disconnect requests.

WskControlSocket will return **STATUS_SUCCESS** when silent mode is enabled. Once silent mode is enabled, send and disconnect requests will be failed with **STATUS_INVALID_DEVICE_STATE** and all received control or data packets will be discarded silently.

The only valid operation on this socket is WskCloseSocket.

Requirements

Version	Available in Windows 8, Windows Server 2012, and later.
Header	Wsk.h (include Wsk.h)

See also

WskCloseSocket

WskControlSocket

SO_WSK_EVENT_CALLBACK

Article • 03/03/2023

The SO_WSK_EVENT_CALLBACK socket option allows a WSK application to enable and disable a socket's event callback functions. This socket option applies only to listening sockets, datagram sockets, connection-oriented sockets, and basic sockets that have registered an extension interface for which at least one event callback function is defined.

If a WSK application uses this socket option to enable or disable event callback functions on either a listening socket or a datagram socket, it must do so after the socket has been bound to a local transport address.

If a WSK application uses this socket option to enable or disable event callback functions on a connection-oriented socket, it must do so after the socket has been connected to a remote transport address.

To enable or disable event callback functions on a socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_WSK_EVENT_CALLBACK
Level	SOL_SOCKET
InputSize	sizeof(WSK_EVENT_CALLBACK_CONTROL)
InputBuffer	A pointer to a WSK_EVENT_CALLBACK_CONTROL structure
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

A WSK application does not specify a pointer to an IRP when calling the **WskControlSocket** function to enable event callback functions on a socket.

A WSK application can optionally specify a pointer to an IRP when calling the **WskControlSocket** function to disable an event callback function on a socket.

When a WSK application calls **WskControlSocket** to disable an event callback function, the WSK subsystem behaves as follows:

- If there are no in-progress calls to the event callback function that is being disabled when the WSK application calls the **WskControlSocket** function, the event callback function is disabled and the **WskControlSocket** function returns STATUS_SUCCESS. If the WSK application specifies an IRP, the IRP is completed with success status.

- If there are in-progress calls to the event callback function that is being disabled when the WSK application calls the **WskControlSocket** function and the WSK application specified an IRP, the **WskControlSocket** function returns STATUS_PENDING. The WSK subsystem disables the event callback function and completes the IRP after all in-progress calls to the event callback function have returned.

- If there are in-progress calls to the event callback function that is being disabled when the WSK application calls the **WskControlSocket** function and the WSK application did not specify an IRP, the **WskControlSocket** function returns STATUS_EVENT_PENDING. The WSK subsystem disables the event callback function after all in-progress calls to the event callback function have returned.

When enabling or disabling any of the standard WSK event callback functions, a WSK application sets the **NpiId** member of the **WSK_EVENT_CALLBACK_CONTROL** structure to a pointer to the WSK Network Programming Interface (NPI) identifier, NPI_WSK_INTERFACE_ID.

When enabling or disabling any callback functions for an extension interface, a WSK application sets the **NpiId** member of the WSK_EVENT_CALLBACK_CONTROL structure to a pointer to the NPI identifier for that extension interface.

When enabling event callback functions, a WSK application can simultaneously enable any combination of the event callback functions that are valid for a particular category of WSK socket. The WSK application simultaneously enables these combinations by setting the **EventMask** member of the WSK_EVENT_CALLBACK_CONTROL structure to a bitwise OR of the event flags for all of the event callback functions that are being enabled.

When disabling event callback functions, a WSK application must disable each event callback function independently. A WSK application independently disables an event callback function by setting the **EventMask** member of the WSK_EVENT_CALLBACK_CONTROL structure to a bitwise OR of the event flag for the event callback function that is being disabled and the WSK_EVENT_DISABLE flag.

The following table shows the valid event flags for a listening socket.

Event flag	Event callback function
WSK_EVENT_ACCEPT	*WskAcceptEvent*

The following table shows the valid event flags for a datagram socket.

Event flag	Event callback function
WSK_EVENT_RECEIVE_FROM	*WskReceiveFromEvent*

The following table shows the valid event flags for a connection-oriented socket.

Event flag	Event callback function
WSK_EVENT_DISCONNECT	*WskDisconnectEvent*
WSK_EVENT_RECEIVE	*WskReceiveEvent*
WSK_EVENT_SEND_BACKLOG	*WskSendBacklogEvent*

A listening socket can automatically enable event callback functions on connection-oriented sockets that are accepted by the listening socket. A WSK application automatically enables these callback functions by enabling the connection-oriented socket event callback functions on the listening socket. The event callback functions are automatically enabled on an accepted connection-oriented socket only if the socket is accepted by the listening socket's *WskAcceptEvent* event callback function. If the connection-oriented socket is accepted by the listening socket's **WskAccept** function, the accepted socket's event callback functions are not automatically enabled.

After any connection-oriented event callback functions are enabled on a listening socket, they cannot be disabled on the listening socket. If the *WskAcceptEvent* event callback function is disabled and then re-enabled on a listening socket, any connection-oriented event callback functions that were originally enabled on that listening socket will continue to be applied to all connection-oriented sockets that are accepted by the *WskAcceptEvent* event callback function.

For more information about enabling and disabling a socket's event callback functions, see Enabling and Disabling Event Callback Functions.

Requirements

Version	Available in Windows Vista and later versions of

	the Windows operating systems.
Header	Wsk.h (include Wsk.h)

SO_WSK_SECURITY

Article • 03/03/2023

The SO_WSK_SECURITY socket option allows a WSK application to either apply a security descriptor to a socket or retrieve a cached copy of a socket's security descriptor from a socket. The security descriptor controls the sharing of the local transport address to which the socket is bound.

This socket option applies only to listening sockets, datagram sockets, and connection-oriented sockets.

If a WSK application uses this socket option to apply a security descriptor to a socket, it must do so before the socket is bound to a local transport address.

To apply a security descriptor to a socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_WSK_SECURITY
Level	SOL_SOCKET
InputSize	sizeof(PSECURITY_DESCRIPTOR)
InputBuffer	A pointer to a PSECURITY_DESCRIPTOR-typed variable. This variable must contain a pointer to a cached copy of a security descriptor that was obtained by calling the WskControlClient function with the WSK_CACHE_SD control code.
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to apply a security descriptor to a socket.

If a WSK application uses this socket option to apply a security descriptor to a socket, the new security descriptor replaces any security descriptor that was previously applied to the socket.

A WSK application must not release the cached copy of the security descriptor until after the IRP is completed.

A WSK application can also apply a security descriptor to a socket when the socket is initially created by specifying a pointer to a cached copy of a security descriptor in the *SecurityDescriptor* parameter when it calls the WskSocket or WskSocketConnect function.

If a WSK application does not apply a security descriptor to a socket, the WSK subsystem uses a default security descriptor that does not allow sharing of the local transport address.

To retrieve a cached copy of a socket's security descriptor from a socket, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskGetOption**
ControlCode	SO_WSK_SECURITY
Level	SOL_SOCKET
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(PSECURITY_DESCRIPTOR)
OutputBuffer	A pointer to a PSECURITY_DESCRIPTOR-typed variable. This variable receives a pointer to a cached copy of the socket's security descriptor.
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to retrieve a cached copy of a socket's security descriptor from a socket.

A WSK application must call the WskControlClient function with the WSK_RELEASE_SD control code to release the cached copy of the security descriptor when it is no longer needed.

See the reference page for the SECURITY_DESCRIPTOR structure for more information.

Requirements

Version	Available in Windows Vista and later versions of

	the Windows operating systems.
Header	Wsk.h (include Wsk.h)

WSK_CLIENT

Article • 03/03/2023

The WSK_CLIENT data type defines the WSK subsystem's binding context for its attachment to a WSK application.

```c++
typedef PVOID PWSK_CLIENT;
```

PWSK_CLIENT
The contents of the **WSK_CLIENT** structure are opaque to a WSK application.

Remarks

When a WSK application calls the WskCaptureProviderNPI function, the WSK subsystem returns a pointer to a WSK_CLIENT structure to the WSK application by means of the *WskProviderNpi* parameter. The WSK subsystem uses this structure to track the state of the binding between the WSK application and the WSK subsystem. A WSK application passes this pointer as a parameter to all the functions in WSK_PROVIDER_DISPATCH (WskControlClient, WskSocket, and WskSocketConnect).

For more information, see Registering a Winsock Kernel Application.

Requirements

Version: Available in Windows Vista and later versions of the Windows operating systems.

Header: Wsk.h (include Wsk.h)

See also

WskCaptureProviderNPI
WskControlClient
WskSocket
WskSocketConnect
WSK_PROVIDER_DISPATCH

Porting TDI Drivers to Winsock Kernel

Article • 12/15/2021

To port your TDI driver to Winsock Kernel (WSK), you'll need to convert TDI tasks to their WSK equivalents as shown in the following table.

Tasks	TDI	Winsock Kernel (WSK)
Register and Deregister	N/A	WskRegister and WskDeregister
Capture and Release Provider NPI	N/A	WskCaptureProviderNPI and WskReleaseProviderNPI
Create Address File Object	Create *EaBuffer*, then call ZwCreateFile	No longer necessary. See WskSocket.
Create Connection File Object	Create connection *EaBuffer*, then call ZwCreateFile	No longer necessary. See WskSocket and *WskAcceptEvent*.
Associate Address	TDI_ASSOCIATE_ADDRESS	WskBind
Set Event Handlers	TDI_SET_EVENT_HANDLER	WskControlSocket or static variation using WskControlClient
Clear Event Handlers	TDI_SET_EVENT_HANDLER	WskControlSocket
Connect	TDI_CONNECT	WskConnect
Disconnect	TDI_DISCONNECT	WskDisconnect
Send	TDI_SEND	WskSend
Receive	TDI_RECEIVE	WskReceive
Disassociate Address	TDI_DISASSOCIATE_ADDRESS	N/A
Receive Handler	ClientEventReceive, TDI_RECEIVE	*WskReceiveEvent*
Connect Handler	ClientEventConnect, TDI_CONNECT	WskAccept
Close Socket or Connection	ObDereferenceObject or ZwClose	WskCloseSocket

Sharing Transport Addresses

Article • 12/15/2021

In most situations, a Winsock Kernel (WSK) application cannot bind a socket to a local transport address that is already in use by another socket. WSK applications can use the SO_EXCLUSIVEADDRUSE and SO_REUSEADDR socket options to control the sharing of the local transport address to which a socket is bound. Neither of these socket options are set for a socket by default. For more information about setting socket options, see Performing Control Operations on a Socket.

The following table shows the result of binding a second socket to a local transport address that is already in use by another socket. The *Wildcard* and *Specific* cases specify whether the socket is bound to a wildcard local transport address or to a specific local transport address.

Second bind		**First bind**					
		No socket options (default)		SO_REUSEADDR		SO_EXCLUSIVEADDRUSE	
		Wildcard	Specific	Wildcard	Specific	Wildcard	Specific
No socket options (default)	Wildcard	INUSE	SUCCESS	INUSE	SUCCESS	INUSE	SUCCESS
	Specific	CHECK	INUSE	CHECK	DENIED	DENIED	INUSE
SO_REUSEADDR	Wildcard	DENIED	SUCCESS	SUCCESS	SUCCESS	DENIED	SUCCESS
	Specific	CHECK	DENIED	SUCCESS	SUCCESS	DENIED	DENIED
SO_EXCLUSIVEADDRUSE	Wildcard	INUSE	INUSE	INUSE	INUSE	INUSE	INUSE
	Specific	CHECK	INUSE	CHECK	INUSE	DENIED	INUSE

The results are defined as follows:

SUCCESS
The bind operation for the second socket succeeds. The WSK subsystem returns a status of STATUS_SUCCESS.

INUSE
The bind operation on the second socket fails. The WSK subsystem returns a status of STATUS_ADDRESS_ALREADY_EXISTS.

DENIED
The bind operation on the second socket fails. The WSK subsystem returns a status of STATUS_ACCESS_DENIED.

CHECK
An access check is performed to determine if the bind operation on the second socket succeeds or fails. If access is granted, the bind succeeds and the WSK subsystem returns a status of STATUS_SUCCESS. If access is denied, the bind fails and the WSK subsystem returns a status of STATUS_ACCESS_DENIED.

In the cases defined in the previous table where an access check is performed, the second socket's security context is checked against the first socket's security descriptor.

- A socket's security context is determined by the *OwningProcess* and *OwningThread* parameters that are passed to either the WskSocket function or the WskSocketConnect function when the socket is created. If no specific process or thread is specified when the socket is created, the security context of the process that created the socket is used.

- A socket's security descriptor is specified by the *SecurityDescriptor* parameter that is passed to either the WskSocket function or the WskSocketConnect function when the socket is created. If no specific security descriptor is specified, the WSK subsystem uses a default security descriptor that does not permit sharing of transport addresses. A security descriptor can also be applied to a socket after the socket has been created by using the SO_WSK_SECURITY socket option.

If the two sockets are bound to two different specific local transport addresses, there is no sharing of either transport address. In this situation the second bind operation will always complete successfully.

Using TDI Transports

Article • 12/15/2021

The Winsock Kernel (WSK) subsystem provides support for using TDI transports. In order to use TDI transports via the WSK Network Programming Interface (NPI), a WSK application must map the combination of address family, socket type, and protocol for each of the TDI transports it uses to the associated device name of each of those TDI transports. A WSK application maps combinations of address family, socket type, and protocol to device names of TDI transports by using the **WSK_TDI_DEVICENAME_MAPPING** client control operation.

The following code example shows how a WSK application can map combinations of address family, socket type, and protocol to device names of TDI transports.

C++

```cpp
// Number of TDI mappings
#define MAPCOUNT 2

// Array of TDI mappings
const WSK_TDI_MAP TdiMap[MAPCOUNT] =
{
  {SOCK_STREAM, ..., ..., ...},
  {SOCK_DGRAM, ..., ..., ...}
};

// TDI map info structure
const WSK_TDI_MAP_INFO TdiMapInfo =
{
  MAPCOUNT,
  TdiMap
}

// Function to set the TDI map
NTSTATUS
  SetTdiMap(
    PWSK_APP_BINDING_CONTEXT BindingContext
  )
{
  NTSTATUS Status;

  // Perform client control operation
  Status =
    BindingContext->
      WskProviderDispatch->
        WskControlClient(
          BindingContext->WskClient,
          WSK_TDI_DEVICENAME_MAPPING,
          sizeof(WSK_TDI_MAP_INFO),
          &TdiMapInfo,
          0,
          NULL,
          NULL,
          NULL  // No IRP for this control operation
          );

  // Return status of client control operation
  return Status;
}
```

A WSK application must map combinations of address family, socket type, and protocol to device names of TDI transports before it creates any sockets. After the WSK application has successfully mapped the combinations of address family, socket type, and protocol to device names of TDI transports, the application can then create new sockets that use the mapped TDI transports.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

Using NMR for WSK Registration and Unregistration

Article • 12/15/2021

The Registering a Winsock Kernel Application and Unregistering a Winsock Kernel Application sections describe how a WSK application can attach to and detach from the WSK subsystem by using the WSK registration functions. However, WSK can also attach to the WSK subsystem by using the Network Module Registrar (NMR).

A WSK application can register itself with the NMR as a client of the WSK Network Programming Interface (NPI) by using the procedures in the following sections:

- Initializing NMR Data Structures
- Attaching the WSK Client to the WSK Subsystem
- Unregistering and Unloading the WSK Client

Using the **WskRegister** and **WskDeregister** functions is the preferred method for registering and unregistering WSK applications. The Network Module Registrar remains available for compatibility.

Initializing NMR Data Structures

Article • 12/15/2021

Before a Winsock Kernel (WSK) application can register with the Network Module Registrar (NMR), the application must first initialize the following structures.

- NPI_MODULEID
- NPI_CLIENT_CHARACTERISTICS
- NPI_REGISTRATION_INSTANCE contained within the NPI_CLIENT_CHARACTERISTICS structure

All of these data structures must remain valid and resident in memory as long as the WSK application is registered with the NMR.

The following code example shows how a WSK application can initialize all of the data structures listed previously.

C++

```
// Include the WSK header file
#include "wsk.h"

// Structure for the WSK application's network module identification
const NPI_MODULEID ModuleId =
{
  sizeof(NPI_MODULEID),
  MIT_GUID,
  { ... }  // A GUID that uniquely identifies the WSK application
};

// Prototypes for the WSK application's NMR API callback functions
NTSTATUS
  ClientAttachProvider(
    IN HANDLE NmrBindingHandle,
    IN PVOID ClientContext,
    IN PNPI_REGISTRATION_INSTANCE ProviderRegistrationInstance
    );

NTSTATUS
  ClientDetachProvider(
    IN PVOID ClientBindingContext
    );

VOID
  ClientCleanupBindingContext(
    IN PVOID ClientBindingContext
    );
```

```cpp
// Structure for the WSK application's characteristics
const NPI_CLIENT_CHARACTERISTICS Characteristics =
{
  0,
  sizeof(NPI_CLIENT_CHARACTERISTICS),
  ClientAttachProvider,
  ClientDetachProvider,
  ClientCleanupBindingContext,
  {
    0,
    sizeof(NPI_REGISTRATION_INSTANCE),
    &NPI_WSK_INTERFACE_ID,
    &ModuleId,
    0,
    NULL
  }
};
```

A WSK application calls the **NmrRegisterClient** function to register the application with the NMR.

For example:

C++

```cpp
// Variable to contain the handle for the registration with the NMR
HANDLE RegistrationHandle;

// DriverEntry function
NTSTATUS
  DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
    )
{
  NTSTATUS Status;

  .
  .
  .

  // Register the WSK application with the NMR
  Status = NmrRegisterClient(
    &Characteristics,
    NULL,
    &RegistrationHandle
    );

  if(!NT_SUCCESS(Status)) {
      .
      .
      .
      return Status;
  }

  .
  .
  .
}
```

A WSK application is not required to call **NmrRegisterClient** from within its **DriverEntry** function. For example, if a WSK application is a subcomponent of a complex driver, the registration of the application might occur only when the WSK application subcomponent is activated.

Attaching the WSK Client to the WSK Subsystem

Article • 12/15/2021

After a Winsock Kernel (WSK) application has registered with the Network Module Registrar (NMR) as a client of the WSK NPI, the NMR immediately calls the application's *ClientAttachProvider* callback function if the WSK subsystem is loaded and has registered itself with the NMR. If the WSK subsystem is not registered with the NMR, the NMR does not call the application's *ClientAttachProvider* callback function until the WSK subsystem registers with the NMR.

The WSK application should make the following sequence of calls to complete the attachment procedure.

1. When the NMR calls the WSK application's *ClientAttachProvider* callback function, it passes a pointer to the **NPI_REGISTRATION_INSTANCE** structure associated with the WSK subsystem. The WSK application's *ClientAttachProvider* callback function can use the data passed to it by the NMR to determine if it can attach to the WSK subsystem. Typically, a WSK application only needs the version information contained within a **WSK_PROVIDER_CHARACTERISTICS** structure that is pointed to by the **NpiSpecificCharacteristics** member of the WSK subsystem's NPI_REGISTRATION_INSTANCE structure.

2. If the WSK application determines that it can attach to the WSK subsystem, the WSK application's *ClientAttachProvider* callback function allocates and initializes a binding context structure for the attachment to the WSK subsystem. The application then calls the **NmrClientAttachProvider** function to continue the attachment process.

   If **NmrClientAttachProvider** returns STATUS_SUCCESS, the WSK application has successfully attached to the WSK subsystem. In this situation, the WSK application's *ClientAttachProvider* callback function must save the binding handle that the NMR passed in the *NmrBindingHandle* parameter when the NMR called the application's *ClientAttachProvider* callback function. The WSK application's *ClientAttachProvider* callback function must also save the pointers to the client object (**WSK_CLIENT**) and the provider dispatch table (**WSK_PROVIDER_DISPATCH**) that are returned in the variables that the application passed to the **NmrClientAttachProvider** function in the *ProviderBindingContext* and *ProviderDispatch* parameters. A WSK application typically saves this data in its binding context for the attachment to the WSK subsystem. After the WSK application has successfully attached to the WSK subsystem, the WSK application's *ClientAttachProvider* callback function must return STATUS_SUCCESS.

3. If **NmrClientAttachProvider** returns STATUS_NOINTERFACE, the WSK application can make another attempt to attach to the WSK subsystem by calling the **NmrClientAttachProvider** function again, passing a *ClientDispatch* pointer to a different WSK_CLIENT_DISPATCH structure that specifies an alternate version of the WSK NPI that is supported by the application.

4. If a call to the **NmrClientAttachProvider** function does not return STATUS_SUCCESS, and the WSK application does not make any further attempts to attach to the WSK subsystem, the WSK application's *ClientAttachProvider* callback function should clean up and deallocate any resources that it allocated before it called **NmrClientAttachProvider**. In this situation, the WSK application's *ClientAttachProvider* callback function must return the status code that was returned by the last call to the **NmrClientAttachProvider** function.

5. If the WSK application determines that it cannot attach to the provider module, the application's *ClientAttachProvider* callback function must return STATUS_NOINTERFACE.

The following code example shows how a WSK application can attach itself to the WSK subsystem.

C++

```
// Context structure type for the WSK application's
// binding to the WSK subsystem
typedef struct WSK_APP_BINDING_CONTEXT_ {
  HANDLE NmrBindingHandle;
  PWSK_CLIENT WskClient;
  PWSK_PROVIDER_DISPATCH WskProviderDispatch;
  .
  . // Other application-specific members
  .
} WSK_APP_BINDING_CONTEXT, *PWSK_APP_BINDING_CONTEXT;

// Pool tag used for allocating the binding context
#define BINDING_CONTEXT_POOL_TAG 'tpcb'

// The WSK application uses version 1.0 of WSK
#define WSK_APP_WSK_VERSION MAKE_WSK_VERSION(1,0)

// Structure for the WSK application's dispatch table
const WSK_CLIENT_DISPATCH WskAppDispatch = {
  WSK_APP_WSK_VERSION,
  0,
  NULL  // No WskClientEvent callback
```

```
};

// ClientAttachProvider NMR API callback function
NTSTATUS
  ClientAttachProvider(
    IN HANDLE NmrBindingHandle,
    IN PVOID ClientContext,
    IN PNPI_REGISTRATION_INSTANCE ProviderRegistrationInstance
    )
{
  PNPI_MODULEID WskProviderModuleId;
  PWSK_PROVIDER_CHARACTERISTICS WskProviderCharacteristics;
  PWSK_APP_BINDING_CONTEXT BindingContext;
  PWSK_CLIENT WskClient;
  PWSK_PROVIDER_DISPATCH WskProviderDispatch;
  NTSTATUS Status;

  // Get pointers to the WSK subsystem's identification and
  // characteristics structures
  WskProviderModuleId = ProviderRegistrationInstance->ModuleId;
  WskProviderCharacteristics =
    (PWSK_PROVIDER_CHARACTERISTICS)
      ProviderRegistrationInstance->NpiSpecificCharacteristics;

  //
  // The WSK application can use the data in the structures pointed
  // to by ProviderRegistrationInstance, WskProviderModuleId, and
  // WskProviderCharacteristics to determine if the WSK application
  // can attach to the WSK subsystem.
  //
  // In this example, the WSK application does not perform any
  // checks to determine if it can attach to the WSK subsystem.
  //

  // Allocate memory for the WSK application's binding
  // context structure
  BindingContext =
    (PWSK_APP_BINDING_CONTEXT)
      ExAllocatePoolWithTag(
        NonPagedPool,
        sizeof(WSK_APP_BINDING_CONTEXT),
        BINDING_CONTEXT_POOL_TAG
        );

  // Check result of allocation
  if (BindingContext == NULL)
  {
    // Return error status code
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  // Initialize the binding context structure
  ...

  // Continue with the attachment to the WSK subsystem
  Status = NmrClientAttachProvider(
    NmrBindingHandle,
    BindingContext,
    &WskAppDispatch,
    &WskClient,
    &WskProviderDispatch
    );

  // Check result of attachment
  if (Status == STATUS_SUCCESS)
  {
    // Save NmrBindingHandle, WskClient, and
    // WskProviderDispatch for future reference
    BindingContext->NmrBindingHandle = NmrBindingHandle;
    BindingContext->WskClient = WskClient;
    BindingContext->WskProviderDispatch = WskProviderDispatch;
  }

  // Attachment did not succeed
  else
  {
    // Free memory for application's binding context structure
    ExFreePoolWithTag(
      BindingContext,
      BINDING_CONTEXT_POOL_TAG
      );
  }

  // Return result of attachment
  return Status;
}
```

Unregistering and Unloading the WSK Client

Article • 12/15/2021

Any Winsock Kernel (WSK) application that uses the Network Module Registrar (NMR) for attaching to the WSK subsystem must unregister with NMR before unloading. When a WSK application unregisters with the NMR by calling the **NmrDeregisterClient** function, the NMR calls the application's *ClientDetachProvider* callback function so that the application can detach itself from the WSK subsystem as part of the WSK application's unregistration process.

Furthermore, in the unlikely, but possible, case of the WSK subsystem unregistering with the NMR, the NMR also calls the WSK application's *ClientDetachProvider* callback function so that the application can detach itself from the WSK subsystem as part of the WSK subsystem's unregistration process.

The NMR calls a WSK application's *ClientDetachProvider* callback function only once. If both the WSK application and the WSK subsystem unregister with the NMR, the NMR calls the WSK application's *ClientDetachProvider* callback function only after the first unregistration is initiated.

If there are no calls in progress to any of the WSK functions in WSK_PROVIDER_DISPATCH at the time that the NMR calls the WSK application's *ClientDetachProvider* callback function, the WSK application should return STATUS_SUCCESS from its *ClientDetachProvider* callback function. Otherwise, the WSK application must return STATUS_PENDING from its *ClientDetachProvider* callback function, and it must call the **NmrClientDetachProviderComplete** function after all of the calls in progress to the WSK functions in WSK_PROVIDER_DISPATCH have returned. A WSK application calls the **NmrClientDetachProviderComplete** function to notify the NMR that the application has detached from the WSK subsystem. However, the WSK subsystem will not allow the detachment procedure to be completed fully until all open sockets are closed by the WSK application. For more information, see Closing a Socket.

After a WSK application has notified the NMR that detachment is complete, either by returning STATUS_SUCCESS from its *ClientDetachProvider* callback function or by calling the **NmrClientDetachProviderComplete** function, the application must not make any further calls to any of the WSK functions in WSK_PROVIDER_DISPATCH.

If a WSK application implements a *ClientCleanupBindingContext* callback function, the NMR calls the application's *ClientCleanupBindingContext* callback function after both the WSK application and the WSK subsystem have completed detachment from each other.

A WSK application's *ClientCleanupBindingContext* callback function should perform any necessary cleanup of the data contained within the application's binding context structure. It should then free the memory for the binding context structure if the application dynamically allocated memory for the structure.

For example:

```C++
// ClientDetachProvider callback function
NTSTATUS
  ClientDetachProvider(
    IN PVOID ClientBindingContext
    )
{
  PWSK_APP_BINDING_CONTEXT BindingContext;

  // Get a pointer to the binding context
  BindingContext =
    (PWSK_APP_BINDING_CONTEXT)ClientBindingContext;

  // Check if there are no calls in progress to any WSK functions
  // in WSK_PROVIDER_DISPATCH and that there are no open sockets
  if (...)
  {
    // Return success status indicating that detachment is complete
    return STATUS_SUCCESS;
  }

  // There are calls in progress to one or more of the WSK functions
  // in WSK_PROVIDER_DISPATCH and/or one or more open sockets
  else
  {
    // Return pending status, indicating that detachment is pending
    // completion of the calls in progress to the WSK functions in
    // WSK_PROVIDER_DISPATCH and/or closing of the open sockets
    return STATUS_PENDING;

    // When all of the calls in progress to the WSK functions
    // in WSK_PROVIDER_DISPATCH are completed, the WSK application
    // must close all open sockets.
    //
    // After all sockets have been closed, the WSK application must
    // call the NmrClientDetachProviderComplete function with the
    // binding handle for the attachment to the WSK subsystem.
  }
}

// ClientCleanupBindingContext callback function
VOID
  ClientCleanupBindingContext(
    IN PVOID ClientBindingContext
    )
```

```
{
  PWSK_APP_BINDING_CONTEXT BindingContext;

  // Get a pointer to the binding context
  BindingContext =
    (PWSK_APP_BINDING_CONTEXT)ClientBindingContext;

  // Clean up the binding context structure
  ...

  // Free the memory for client's binding context structure
  ExFreePoolWithTag(
    BindingContext,
    BINDING_CONTEXT_POOL_TAG
    );
}
```

A WSK application's **Unload** function must ensure that the application is unregistered from the NMR before the application is unloaded from system memory. A WSK application must not return from its *Unload* function until after it has been completely unregistered from the NMR. If the call to **NmrDeregisterClient** returns STATUS_PENDING, the WSK application must call the **NmrWaitForClientDeregisterComplete** function to wait for the unregistration to complete before it returns from its *Unload* function.

For example:

C++

```
// Variable containing the handle for registration with the NMR
HANDLE RegistrationHandle;

// Unload function
VOID
  Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
  NTSTATUS Status;

  // Unregister the WSK application from the NMR
  Status =
    NmrDeregisterClient(
      RegistrationHandle
      );

  // Check if pending
  if (Status == STATUS_PENDING)
  {
    // Wait for the unregistration to be completed
    NmrWaitForClientDeregisterComplete(
      RegistrationHandle
      );
  }
}
```

A WSK application is not required to call **NmrDeregisterClient** from within its *Unload* function. For example, if a WSK application is a subcomponent of a complex driver, the unregistration of the WSK application might occur when the WSK application subcomponent is deactivated. However, in such a situation the driver must still ensure that the WSK application has been completely unregistered from the NMR before returning from its *Unload* function.

IP Helper Overview

Article • 06/30/2022

Internet Protocol Helper (IP Helper) enables drivers to retrieve information about the network configuration of the local computer and to modify that configuration. IP Helper also provides notification mechanisms to make sure that a driver is notified when certain aspects of the local computer network configuration change. IP Helper is available in Windows Vista and later versions of the Microsoft Windows operating systems.

Many of the IP Helper functions pass structure parameters that represent data types that are associated with the Management Information Base (MIB) technology. The IP Helper functions use these MIB structures to represent various networking information.

The IP Helper documentation uses the terms "adapter" and "interface" extensively. An *adapter* is a legacy term that is an abbreviated form of *network adapter*, which originally referred to some form of network hardware. An adapter is a data link-level abstraction.

An *interface* is described in the IETF RFC documents as an abstract concept that represents a node's attachment to a link. An interface is an IP-level abstraction.

Your driver can use the following kernel-mode functions, MIB structures, and MIB and Network Layer (NL) enumerations to retrieve and modify configuration settings for Transmission Control Protocol/Internet Protocol (TCP/IP) transport on a local computer.

> ⓘ Note
>
> When you are developing driver code, follow the instructions for **including header files.**

Interface Conversion Functions

Function	Description
ConvertInterfaceAliasToLuid	Converts a locally unique identifier (LUID) for a network interface to the Unicode interface name.
ConvertInterfaceGuidToLuid	Converts a globally unique identifier (GUID) for a network interface to the LUID for the interface.

Function	Description
ConvertInterfaceIndexToLuid	Converts a local index for a network interface to the LUID for the interface.
ConvertInterfaceLuidToAlias	Converts a LUID for a network interface to an interface alias.
ConvertInterfaceLuidToGuid	Converts a LUID for a network interface to a GUID for the interface.
ConvertInterfaceLuidToIndex	Converts a LUID for a network interface to the local index for the interface.
ConvertInterfaceLuidToNameA	Converts a LUID for a network interface to the ANSI interface name.
ConvertInterfaceLuidToNameW	Converts a LUID for a network interface to the Unicode interface name.
ConvertInterfaceNameToLuidA	Converts an ANSI network interface name to the LUID for the interface.
ConvertInterfaceNameToLuidW	Converts a Unicode network interface name to the LUID for the interface.
if_indextoname	Converts the local index for a network interface to the ANSI interface name.
if_nametoindex	Converts the ANSI interface name for a network interface to the local index for the interface.

Interface Management Functions

Function	Description
GetIfEntry2	Retrieves information for the specified interface on the local computer.
GetIfStackTable	Retrieves a table of network interface stack row entries that specify the relationship of the network interfaces on an interface stack.
GetIfTable2	Retrieves the MIB-II interface table.
GetIfTable2Ex	Retrieves the MIB-II interface table, given a level of interface information to retrieve.

Function	Description
GetInvertedIfStackTable	Retrieves a table of inverted network interface stack row entries that specify the relationship of the network interfaces on an interface stack.
GetIpInterfaceEntry	Retrieves IP information for the specified interface on the local computer.
GetIpInterfaceTable	Retrieves the IP interface entries on the local computer.
InitializeIpInterfaceEntry	Initializes the members of a MIB_IPINTERFACE_ROW structure entry with default values.
SetIpInterfaceEntry	Sets the properties of an IP interface on the local computer.

IP Address Management Functions

Function	Description
CreateAnycastIpAddressEntry	Adds a new anycast IP address entry on the local computer.
CreateSortedAddressPairs	Pairs a supplied list of destination addresses together with the host machine's local IP addresses and sorts the pairs according to the preferred order of communication.
CreateUnicastIpAddressEntry	Adds a new unicast IP address entry on the local computer.
DeleteAnycastIpAddressEntry	Deletes an existing anycast IP address entry on the local computer.
DeleteUnicastIpAddressEntry	Deletes an existing unicast IP address entry from the local computer.
GetAnycastIpAddressEntry	Retrieves information for an existing anycast IP address entry on the local computer.
GetAnycastIpAddressTable	Retrieves the anycast IP address table on the local computer.
GetMulticastIpAddressEntry	Retrieves information for an existing multicast IP address entry on the local computer.

Function	Description
GetMulticastIpAddressTable	Retrieves the multicast IP address table on the local computer.
GetUnicastIpAddressEntry	Retrieves information for an existing unicast IP address entry on the local computer.
GetUnicastIpAddressTable	Retrieves the unicast IP address table on the local computer.
InitializeUnicastIpAddressEntry	Initializes a MIB_UNICASTIPADDRESS_ROW structure with default values for a unicast IP address entry on the local computer.
NotifyStableUnicastIpAddressTable	Retrieves the stable unicast IP address table on a local computer.
SetUnicastIpAddressEntry	Sets the properties of an existing unicast IP address entry on the local computer.

IP Neighbor Address Management Functions

Function	Description
CreateIpNetEntry2	Creates a new neighbor IP address entry on the local computer.
DeleteIpNetEntry2	Deletes a neighbor IP address entry from the local computer.
FlushIpNetTable2	Flushes the IP neighbor table on the local computer.
GetIpNetEntry2	Retrieves information for a neighbor IP address entry on the local computer.
GetIpNetTable2	Retrieves the IP neighbor table on the local computer.
ResolveIpNetEntry2	Resolves the physical address for a neighbor IP address entry on the local computer.
SetIpNetEntry2	Sets the physical address of an existing neighbor IP address entry on the local computer.

IP Path Management Functions

Function	Description
FlushIpPathTable	Flushes the IP path table on the local computer.
GetIpPathEntry	Retrieves information for an IP path entry on the local computer.
GetIpPathTable	Retrieves information for an IP path entry on the local computer.

IP Route Management Functions

Function	Description
CreateIpForwardEntry2	Creates a new IP route entry on the local computer.
DeleteIpForwardEntry2	Deletes an IP route entry from the local computer.
GetBestRoute2	Retrieves the IP route entry on the local computer for the best route to the specified destination IP address.
GetIpForwardEntry2	Retrieves information for an IP route entry on the local computer.
GetIpForwardTable2	Retrieves the IP route entries on the local computer.
InitializeIpForwardEntry	Initializes a MIB_IPFORWARD_ROW2 structure with default values for an IP route entry on the local computer.
SetIpForwardEntry2	Sets the properties of an IP route entry on the local computer.

IP Table Memory Management Functions

Function	Description
FreeMibTable	Frees the buffer that is allocated by the functions that return tables of network interfaces, addresses, and routes (for example, GetIfTable2 and GetAnycastIpAddressTable).

Notification Functions

Function	Description
CancelMibChangeNotify2	Deregisters the driver for change notifications for IP interface changes, IP address changes, IP route changes, and requests to retrieve the stable unicast IP address table.
NotifyIpInterfaceChange	Registers the driver to be notified for changes to all IP interfaces, IPv4 interfaces, or IPv6 interfaces on a local computer.
NotifyRouteChange2	Registers to be notified for changes to IP route entries on a local computer.
NotifyUnicastIpAddressChange	Registers to be notified for changes to all unicast IP interfaces, unicast IPv4 addresses, or unicast IPv6 addresses on a local computer.

Teredo IPv6 Client Management Functions

Function	Description
GetTeredoPort	Retrieves the dynamic UDP port number that the Teredo client uses on the local computer.
NotifyTeredoPortChange	Registers to be notified for changes to the UDP port number that the Teredo client uses for the Teredo service port on a local computer.
NotifyStableUnicastIpAddressTable	Retrieves the stable unicast IP address table on a local computer.

MIB Structures

Structure	Description
IP_ADDRESS_PREFIX	Stores an IP address prefix.
MIB_ANYCASTIPADDRESS_ROW	Stores information about an anycast IP address.
MIB_ANYCASTIPADDRESS_TABLE	Contains a table of anycast IP address entries.
MIB_IF_ROW2	Stores information about a particular interface.
MIB_IF_TABLE2	Contains a table of logical and physical interface entries.

Structure	Description
MIB_IFSTACK_ROW	Represents the relationship between two network interfaces.
MIB_IFSTACK_TABLE	Contains a table of row entries in the network interface stack. This table specifies the relationship of the network interfaces on an interface stack.
MIB_INVERTEDIFSTACK_ROW	Represents the relationship between two network interfaces.
MIB_INVERTEDIFSTACK_TABLE	Contains a table of inverted network interface stack row entries. This table specifies the relationship of the network interfaces on an interface stack in reverse order.
MIB_IPFORWARD_ROW2	Stores information about an IP route entry.
MIB_IPFORWARD_TABLE2	Contains a table of IP route entries.
MIB_IPINTERFACE_ROW	Stores interface management information for a particular IP address family on a network interface.
MIB_IPINTERFACE_TABLE	Contains a table of IP interface entries.
MIB_IPNET_ROW2	Stores information about a neighbor IP address.
MIB_IPNET_TABLE2	Contains a table of neighbor IP address entries.
MIB_IPPATH_ROW	Stores information about an IP path entry.
MIB_IPPATH_TABLE	Contains a table of IP path entries.
MIB_MULTICASTIPADDRESS_ROW	Stores information about a multicast IP address.
MIB_MULTICASTIPADDRESS_TABLE	Contains a table of multicast IP address entries.
MIB_UNICASTIPADDRESS_ROW	Stores information about a unicast IP address.
MIB_UNICASTIPADDRESS_TABLE	Contains a table of unicast IP address entries.

MIB Enumerations

Enumeration	Description
MIB_IF_TABLE_LEVEL	Defines the level of interface information to retrieve.

Enumeration	Description
MIB_NOTIFICATION_TYPE	Defines the notification type that is passed to a callback function when a notification occurs.

NL Enumerations

Enumeration	Description
NL_ADDRESS_TYPE	Specifies the IP address type of the network layer.
NL_DAD_STATE	Defines the duplicate address detection (DAD) state.
NL_LINK_LOCAL_ADDRESS_BEHAVIOR	Defines the link local address behavior.
NL_NEIGHBOR_STATE	Defines the state of a network layer neighbor IP address, as described in RFC 2461, section 7.3.2.
NL_PREFIX_ORIGIN	Defines the origin of the prefix or network part of the IP address.
NL_ROUTE_ORIGIN	Defines the origin of the IP route.
NL_ROUTE_PROTOCOL	Defines the routing mechanism that an IP route was added with, as described in RFC 4292.
NL_ROUTER_DISCOVERY_BEHAVIOR	Defines the router discovery behavior, as described in RFC 2461.
NL_SUFFIX_ORIGIN	Defines the origin of the suffix or host part of the IP address.

Including Header Files for IP Helper

Article • 06/30/2022

Driver code that uses the kernel-mode IP Helper functions, MIB structures, and enumerations that are declared in Netioapi.h must have **#include** statements in the following sequence.

```C++
#include <ntddk.h>
#include <netioapi.h>
```

Note Do not include Iphlpapi.h in driver code. It is used only for user-mode applications.

When Netioapi.h is used with kernel-mode drivers, it already includes networking header files that define Winsock Kernel, network interface information, the network layer, and Network Driver Interface Specification (NDIS) types.

Therefore, do not include the following header files in your driver code:

- Ifdef.h
- Nldef.h
- Ws2def.h
- Ws2ipdef.h

For information about the user-mode versions of the IP Helper functions and MIB structures, see the Windows SDK IP Helper documentation.

ConvertInterfaceAliasToLuid function

Article • 03/03/2023

The **ConvertInterfaceAliasToLuid** function converts an interface alias name for a network interface to the locally unique identifier (LUID) for the interface.

> ⓘ Note
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

C++

```
NETIOAPI_API ConvertInterfaceAliasToLuid(
  _In_  const WCHAR     *InterfaceAlias,
  _Out_       PNET_LUID InterfaceLuid
);
```

Parameters

- *InterfaceAlias* [in]
  A pointer to a NULL-terminated Unicode string that contains the alias name of the network interface.

- *InterfaceLuid* [out]
  A pointer to the NET_LUID union for the network interface.

Return value

ConvertInterfaceAliasToLuid returns STATUS_SUCCESS if the function succeeds. If the function fails, the *InterfaceLuid* parameter is set to **NULL**, and **ConvertInterfaceAliasToLuid** returns the following error code:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters was invalid. **ConvertInterfaceAliasToLuid** returns this error if either *InterfaceAlias* or *InterfaceLuid* is **NULL**, or if *InterfaceAlias* is invalid.

Remarks

The **ConvertInterfaceAliasToLuid** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceGuidToLuid function

Article • 03/03/2023

The **ConvertInterfaceGuidToLuid** function converts a globally unique identifier (GUID) for a network interface to the locally unique identifier (LUID) for the interface.

> ⓘ Note
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

```C++
NETIOAPI_API ConvertInterfaceGuidToLuid(
  _In_  const GUID      *InterfaceGuid,
  _Out_       PNET_LUID InterfaceLuid
);
```

Parameters

- *InterfaceGuid* [in]

  A pointer to a GUID for the network interface.

- *InterfaceLuid* [out]

  A pointer to the **NET_LUID** union for the network interface.

Return value

ConvertInterfaceGuidToLuid returns STATUS_SUCCESS if the function succeeds. If the function fails, the *InterfaceLuid* parameter is set to **NULL**, and **ConvertInterfaceGuidToLuid** returns the following error code:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters was invalid. **ConvertInterfaceGuidToLuid** returns this error if either *InterfaceAlias* or *InterfaceLuid* is **NULL**, or if *InterfaceGuid* is invalid.

Remarks

The **ConvertInterfaceGuidToLuid** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceIndexToLuid function

Article • 03/03/2023

The **ConvertInterfaceIndexToLuid** function converts a local index for a network interface to the locally unique identifier (LUID) for the interface.

> ⓘ Note
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

```c++
NETIOAPI_API ConvertInterfaceIndexToLuid(
  _In_  NET_IFINDEX InterfaceIndex,
  _Out_ PNET_LUID   InterfaceLuid
);
```

Parameters

- *InterfaceIndex* [in]
  The local index value for the network interface.

- *InterfaceLuid* [out]
  A pointer to the NET_LUID union for the network interface.

Return value

ConvertInterfaceIndexToLuid returns STATUS_SUCCESS if the function succeeds. If the function fails, the *InterfaceLuid* parameter is set to **NULL**, and **ConvertInterfaceIndexToLuid** returns the following error code:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters was invalid. **ConvertInterfaceIndexToLuid** returns this error if the *InterfaceLuid* parameter is **NULL**, or if the *InterfaceIndex* parameter is invalid.

Remarks

The **ConvertInterfaceIndexToLuid** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceLuidToAlias function

Article • 03/03/2023

The **ConvertInterfaceLuidToAlias** function converts a locally unique identifier (LUID) for a network interface to an interface alias.

> **Note**
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

```c++
NETIOAPI_API ConvertInterfaceLuidToAlias(
  _In_  const NET_LUID *InterfaceLuid,
  _Out_       PWSTR    InterfaceAlias,
  _In_        SIZE_T   Length
);
```

Parameters

- *InterfaceLuid* [in]
  A pointer to a **NET_LUID** union for the network interface.

- *InterfaceAlias* [out]
  A pointer to a buffer to hold the NULL-terminated Unicode string. If **ConvertInterfaceLuidToAlias** returns successfully, *InterfaceAlias* contains the alias name of the network interface.

- *Length* [in]
  The length, by character count, of the buffer that the *InterfaceAlias* parameter points to. This value must be large enough to hold the alias name of the network interface and the terminating NULL character. The maximum allowed length is NDIS_IF_MAX_STRING_SIZE + 1. For more information about NDIS_IF_MAX_STRING_SIZE, see the following Remarks section.

Return value

ConvertInterfaceLuidToAlias returns STATUS_SUCCESS if the function succeeds. If the function fails, **ConvertInterfaceLuidToAlias** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters is invalid. **ConvertInterfaceLuidToAlias** returns this error if either *InterfaceLuid* or *InterfaceAlias* is **NULL**, or if *InterfaceLuid* is invalid.
STATUS_NOT_ENOUGH_MEMORY	Not enough storage is available. **ConvertInterfaceLuidToAlias** returns this error if the size of the buffer that the *InterfaceAlias* parameter points to was not as large as specified in the *Length* parameter and, therefore, the buffer could not hold the alias name.

Remarks

The **ConvertInterfaceLuidToAlias** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

The maximum length of the alias name for a network interface, NDIS_IF_MAX_STRING_SIZE, without the terminating NULL character, is declared in the *Ntddndis.h* header file. NDIS_IF_MAX_STRING_SIZE is defined to be the IF_MAX_STRING_SIZE constant, which is defined in the *Ifdef.h* header file.

> ⓘ **Note**
>
> The *Ntddndis.h* and *Ifdef.h* header files are automatically included in the *Netioapi.h* header file. You should never use the *Ntddndis.h* and *Ifdef.h* header files directly.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceLuidToGuid function

Article • 03/03/2023

The **ConvertInterfaceLuidToGuid** function converts a locally unique identifier (LUID) for a network interface to a globally unique identifier (GUID) for the interface.

> Note
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

```c++
NETIOAPI_API ConvertInterfaceLuidToGuid(
  _In_  const NET_LUID *InterfaceLuid,
  _Out_       GUID     *InterfaceGuid
);
```

Parameters

- *InterfaceLuid* [in]

  A pointer to a **NET_LUID** union for the network interface.

- *InterfaceGuid* [out]

  A pointer to the GUID for the network interface.

Return value

ConvertInterfaceLuidToGuid returns STATUS_SUCCESS if the function succeeds. If the function fails, the *InterfaceGuid* parameter is set to **NULL**, and **ConvertInterfaceLuidToGuid** returns the following error code:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters is invalid. **ConvertInterfaceLuidToGuid** returns this error if either *InterfaceLuid* or *InterfaceGuid* is **NULL**, or if *InterfaceLuid* is invalid.

Remarks

The **ConvertInterfaceLuidToGuid** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceLuidToIndex function

Article • 03/03/2023

The **ConvertInterfaceLuidToIndex** function converts a locally unique identifier (LUID) for a network interface to the local index for the interface.

> ⓘ Note
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

C++

```
NETIOAPI_API ConvertInterfaceLuidToIndex(
  _In_  const NET_LUID     *InterfaceLuid,
  _Out_       PNET_IFINDEX InterfaceIndex
);
```

Parameters

- *InterfaceLuid* [in]
  A pointer to a **NET_LUID** union for the network interface.

- *InterfaceIndex* [out]
  The local index value for the network interface.

Return value

ConvertInterfaceLuidToIndex returns STATUS_SUCCESS if the function succeeds. If the function fails, the *InterfaceIndex* parameter is set to NET_IFINDEX_UNSPECIFIED, and **ConvertInterfaceLuidToIndex** returns the following error code:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters is invalid. **ConvertInterfaceLuidToIndex** returns this error if either *InterfaceLuid* or *InterfaceIndex* is **NULL**, or if *InterfaceLuid* is invalid.

Remarks

The **ConvertInterfaceLuidToIndex** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceLuidToNameA function

Article • 03/03/2023

The **ConvertInterfaceLuidToNameA** function converts a locally unique identifier (LUID) for a network interface to the ANSI interface name.

> **Note**
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

C++

```
NETIOAPI_API ConvertInterfaceLuidToNameA(
  _In_  const NET_LUID *InterfaceLuid,
  _Out_       PSTR     InterfaceName,
  _In_        SIZE_T   Length
);
```

Parameters

- *InterfaceLuid* [in]
  A pointer to a NET_LUID union for a network interface.

- *InterfaceName* [out]
  A pointer to a buffer to hold the NULL-terminated ANSI string. If **ConvertInterfaceLuidToNameA** returns successfully, *InterfaceName* contains the ANSI interface name.

- *Length* [in]
  The length, in bytes, of the buffer that the *InterfaceName* parameter points to. This value must be large enough to hold the interface name and the terminating NULL character. The maximum allowed length is NDIS_IF_MAX_STRING_SIZE + 1. For more information about NDIS_IF_MAX_STRING_SIZE, see the following Remarks section.

Return value

ConvertInterfaceLuidToNameA returns STATUS_SUCCESS if the function succeeds. If the function fails, **ConvertInterfaceLuidToNameA** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters is invalid. **ConvertInterfaceLuidToNameA** returns this error if either *InterfaceLuid* or *InterfaceName* is **NULL**, or if *InterfaceLuid* is invalid.
STATUS_NOT_ENOUGH_MEMORY	**ConvertInterfaceLuidToNameA** returns this error if the *InterfaceName* buffer was not as large as specified in the *Length* parameter and, therefore, the buffer could not hold the interface name.

Remarks

The **ConvertInterfaceLuidToNameA** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

The maximum length of the name for a network interface, NDIS_IF_MAX_STRING_SIZE, without the terminating NULL character, is defined in the Ntddndis.h header file. NDIS_IF_MAX_STRING_SIZE is defined to be the IF_MAX_STRING_SIZE constant, which is defined in the Ifdef.h header file.

ⓘ Note

The *Ntddndis.h* and *Ifdef.h* header files are automatically included in the *Netioapi.h* header file. You should never use the *Ntddndis.h* and *Ifdef.h* header files directly.

Use the ConvertInterfaceLuidToNameW function to convert a network interface LUID to a Unicode interface name.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceLuidToNameW function

Article • 03/03/2023

The **ConvertInterfaceLuidToNameW** function converts a locally unique identifier (LUID) for a network interface to the Unicode interface name.

> **Note**
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

```c++
NETIOAPI_API ConvertInterfaceLuidToNameW(
  _In_  const NET_LUID *InterfaceLuid,
  _Out_       PWSTR    InterfaceName,
  _In_        SIZE_T   Length
);
```

Parameters

- *InterfaceLuid* [in]
  A pointer to a **NET_LUID** union for the network interface.

- *InterfaceName* [out]
  A pointer to a buffer to hold the NULL-terminated Unicode string. If **ConvertInterfaceLuidToNameW** returns successfully, *InterfaceName* contains the Unicode interface name.

- *Length* [in]
  The length of the buffer, by character count, that the *InterfaceName* parameter points to. This value must be large enough to hold the interface name and the terminating NULL character. The maximum allowed length is NDIS_IF_MAX_STRING_SIZE + 1. For more information about NDIS_IF_MAX_STRING_SIZE, see the following Remarks section.

Return value

ConvertInterfaceLuidToNameW returns STATUS_SUCCESS if the function succeeds. If the function fails, **ConvertInterfaceLuidToNameW** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	One of the parameters is invalid. **ConvertInterfaceLuidToNameW** returns this error if either *InterfaceLuid* or *InterfaceName* is **NULL**, or if *InterfaceLuid* is invalid.
STATUS_NOT_ENOUGH_MEMORY	**ConvertInterfaceLuidToNameW** returns this error if the *InterfaceName* buffer was not as large as specified in the *Length* parameter and, therefore, the buffer could not hold the interface name.

Remarks

The **ConvertInterfaceLuidToNameW** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

The maximum length of the network interface name, NDIS_IF_MAX_STRING_SIZE, without the terminating NULL character, is defined in the Ntddndis.h header file. The NDIS_IF_MAX_STRING_SIZE is defined to be the IF_MAX_STRING_SIZE constant, which is defined in the Ifdef.h header file.

> ⓘ Note
>
> The *Ntddndis.h* and *Ifdef.h* header files are automatically included in the *Netioapi.h* header file. You should never use the *Ntddndis.h* and *Ifdef.h* header files directly.

Use ConvertInterfaceLuidToNameA to convert a network interface LUID to an ANSI interface name.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceNameToLuidA

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceNameToLuidA function

Article • 03/03/2023

The **ConvertInterfaceNameToLuidA** function converts an ANSI network interface name to the locally unique identifier (LUID) for the interface.

> ⓘ **Note**
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

```C++
NETIOAPI_API ConvertInterfaceNameToLuidA(
  _In_  const CHAR     *InterfaceName,
  _Out_       NET_LUID *InterfaceLuid
);
```

Parameters

- *InterfaceName* [in]
  A pointer to a NULL-terminated ANSI string that contains the network interface name.

- *InterfaceLuid* [out]
  A pointer to the **NET_LUID** union for this interface.

Return value

ConvertInterfaceNameToLuidA returns STATUS_SUCCESS if the function succeeds. If the function fails, **ConvertInterfaceNameToLuidA** returns one of the following error codes:

Return code	Description

Return code	Description
ERROR_BUFFER_OVERFLOW	The length of the ANSI interface name is invalid. **ConvertInterfaceNameToLuidA** returns this error if the *InterfaceName* parameter exceeds the maximum allowed string length for this parameter.
STATUS_INVALID_NAME	The interface name is invalid. **ConvertInterfaceNameToLuidA** returns this error if the *InterfaceName* parameter contains an invalid interface name.
STATUS_INVALID_PARAMETER	One of the parameters is invalid. **ConvertInterfaceNameToLuidA** returns this error if the *InterfaceLuid* parameter is **NULL**.

Remarks

The **ConvertInterfaceNameToLuidA** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

The maximum length of the network interface name, NDIS_IF_MAX_STRING_SIZE, without the terminating **NULL**, is defined in the Ntddndis.h header file. NDIS_IF_MAX_STRING_SIZE is defined to be the IF_MAX_STRING_SIZE constant, which is defined in the Ifdef.h header file.

> ⓘ Note
>
> The *Ntddndis.h* and *Ifdef.h* header files are automatically included in the *Netioapi.h* header file. You should never use the *Ntddndis.h* and *Ifdef.h* header files directly.

Use the ConvertInterfaceNameToLuidW function to convert a Unicode interface name to a LUID.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib

IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidW

NET_LUID

ConvertInterfaceNameToLuidW function

Article • 03/03/2023

The **ConvertInterfaceNameToLuidW** function converts a Unicode network interface name to the locally unique identifier (LUID) for the interface.

> **Note**
>
> The ConvertInterface*Xxx* API family enumerates identifiers over all interfaces bound to TCP/IP, which may include virtual miniports, lightweight filters, tunnel adapters, and physical interfaces.

Syntax

```c++
NETIOAPI_API ConvertInterfaceNameToLuidW(
  _In_  const WCHAR    *InterfaceName,
  _Out_       NET_LUID *InterfaceLuid
);
```

Parameters

- *InterfaceName* [in]
  A pointer to a NULL-terminated Unicode string that contains the network interface name.

- *InterfaceLuid* [out]
  A pointer to the NET_LUID union for this interface.

Return value

ConvertInterfaceNameToLuidW returns STATUS_SUCCESS if the function succeeds. If the function fails, **ConvertInterfaceNameToLuidW** returns one of the following error codes:

Return code	Description

Return code	Description
STATUS_INVALID_NAME	The interface name is invalid. **ConvertInterfaceNameToLuidW** returns this error if the *InterfaceName* parameter contains an invalid name or the length of the *InterfaceName* parameter exceeds the maximum allowed string length for this parameter.
STATUS_INVALID_PARAMETER	One of the parameters is invalid. **ConvertInterfaceNameToLuidW** returns this error if the *InterfaceLuid* parameter is **NULL**.

Remarks

The **ConvertInterfaceNameToLuidW** function is protocol-independent and works with network interfaces for both the IPv6 and IPv4 protocols.

The maximum length of the network interface name, NDIS_IF_MAX_STRING_SIZE, without the terminating NULL character, is defined in the Ntddndis.h header file. NDIS_IF_MAX_STRING_SIZE is defined to be the IF_MAX_STRING_SIZE constant, which is defined in the Ifdef.h header file.

> ⓘ Note
>
> The *Ntddndis.h* and *Ifdef.h* header files are automatically included in the *Netioapi.h* header file. You should never use the *Ntddndis.h* and *Ifdef.h* header files directly.

Use the ConvertInterfaceNameToLuidA function to convert an ANSI interface name to a LUID.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceAliasToLuid

ConvertInterfaceGuidToLuid

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToAlias

ConvertInterfaceLuidToGuid

ConvertInterfaceLuidToIndex

ConvertInterfaceLuidToNameA

ConvertInterfaceLuidToNameW

ConvertInterfaceNameToLuidA

NET_LUID

ConvertIpv4MaskToLength function

Article • 03/03/2023

Reserved for future use. Do not use this function.

Syntax

C++

```
NETIOAPI_API ConvertIpv4MaskToLength(
  _In_  ULONG  Mask,
  _Out_ PUINT8 MaskLength
);
```

Parameters

- *Mask* [in]

  Reserved.

- *MaskLength* [out]

  Reserved.

Requirements

Header	Netioapi.h

ConvertLengthtoIpv4Mask function

Article • 03/03/2023

Reserved for future use. Do not use this function.

Syntax

```c++
NETIOAPI_API ConvertLengthtoIpv4Mask(
  _In_  ULONG  MaskLength,
  _Out_ PULONG Mask
);
```

Parameters

- *MaskLength* [in]

  Reserved.

- *Mask* [out]

  Reserved.

Requirements

Header	Netioapi.h

if_indextoname function

Article • 03/03/2023

The **if_indextoname** function converts the local index for a network interface to the ANSI interface name.

Syntax

```
C++

PCHAR NETIOAPI_API_ if_indextoname(
  _In_  NET_IFINDEX InterfaceIndex,
  _Out_ PCHAR       InterfaceName
);
```

Parameters

- *InterfaceIndex* [in]
  The local index for a network interface.

- *InterfaceName* [out]
  A pointer to a buffer to hold the NULL-terminated ANSI string. If **if_indextoname** succeeds, *InterfaceName* contains the ANSI interface name. The length, in bytes, of the buffer that this parameter points to must be equal to or greater than IF_NAMESIZE. For more information about IF_NAMESIZE, see the following Remarks section.

Return value

If this function succeeds, **if_indextoname** returns a pointer to a NULL-terminated ANSI string that contains the interface name. If this function fails, **if_indextoname** returns a **NULL** pointer

Remarks

The **if_indextoname** function maps an interface index into its corresponding name. This function is designed as part of basic socket extensions for IPv6, as described by the IETF in RFC 2553 .

The **if_indextoname** function is implemented for portability of drivers with Unix environments, but the **ConvertInterface*Xxx*** functions are the preferred method to convert network interface identifiers. You can replace the **if_indextoname** function by a call to the ConvertInterfaceIndexToLuid function to convert an interface index to a NET_LUID union, followed by a call to the ConvertInterfaceLuidToNameA function to convert NET_LUID to the ANSI interface name.

The length, in bytes, of the buffer that the *InterfaceName* parameter points to must be equal or greater than IF_NAMESIZE. The IF_NAMESIZE value is defined in the Netioapi.h header file as equal to NDIS_IF_MAX_STRING_SIZE. The maximum length of an interface name, NDIS_IF_MAX_STRING_SIZE, without the terminating NULL character is declared in the Ntddndis.h header file. The NDIS_IF_MAX_STRING_SIZE is defined to be the IF_MAX_STRING_SIZE constant that is defined in the Ifdef.h header file.

> ⓘ Note
>
> The *Ntddndis.h* and *Ifdef.h* header files are automatically included in the *Netioapi.h* header file. You should never use the *Ntddndis.h* and *Ifdef.h* header files directly.

If the **if_indextoname** function fails and returns a **NULL** pointer, you cannot determine an error code.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceIndexToLuid

ConvertInterfaceLuidToNameA

NET_LUID

if_nametoindex function

Article • 03/03/2023

The **if_nametoindex** function converts the ANSI interface name for a network interface to the local index for the interface.

Syntax

```C++
NET_IFINDEX NETIOAPI_API_ if_nametoindex(
  _In_ PCSTR InterfaceName
);
```

Parameters

- *InterfaceName* [in]
  A pointer to a NULL-terminated ANSI string that contains the interface name.

Return value

If the function succeeds, **if_nametoindex** returns the local interface index. If the function fails, **if_nametoindex** returns zero.

Remarks

The **if_nametoindex** function maps an interface name into its corresponding index. This function is designed as part of basic socket extensions for IPv6 as described by the IETF in RFC 2553 .

The **if_nametoindex** function is implemented for portability of drivers with Unix environments, but the **ConvertInterface*Xxx*** functions are the preferred method to convert network interface identifiers. You can replace the **if_nametoindex** function by a call to the **ConvertInterfaceNameToLuidA** function to convert the ANSI interface name to a **NET_LUID** union, followed by a call to the **ConvertInterfaceLuidToIndex** function to convert NET_LUID to the local interface index.

If the **if_nametoindex** function fails and returns zero, you cannot determine an error code.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

ConvertInterfaceLuidToIndex

ConvertInterfaceNameToLuidA

NET_LUID

GetIfEntry2 function

Article • 04/01/2024

The **GetIfEntry2** function retrieves information for the specified interface on a local computer.

> **Important**
>
> For driver developers, it is recommended to use GetIfEntry2Ex with MibIfEntryNormalWithoutStatistics when possible, in order to avoid a deadlock when servicing NDIS OIDs.

Syntax

C++

```
NETIOAPI_API GetIfEntry2(
  _Inout_ PMIB_IF_ROW2 Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IF_ROW2 structure that, on successful return, receives information for an interface on the local computer. On input, your driver must set the **InterfaceLuid** member or the **InterfaceIndex** member of the MIB_IF_ROW2 structure to the interface to retrieve information for.

Return value

GetIfEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIfEntry2** returns one of the following error codes:

Expand table

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the **MIB_IF_ROW2** structure that the *Row* parameter points to.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

On input, your driver must initialize at least the **InterfaceLuid** or **InterfaceIndex** member in the **MIB_IF_ROW2** structure that is passed in the *Row* parameter. The members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, the remaining fields of the MIB_IF_ROW2 structure that the *Row* parameter points to are filled in.

Requirements

Expand table

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetIfTable2

GetIfTable2Ex

MIB_IF_ROW2

MIB_IF_TABLE2

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

GetIfStackTable function

Article • 03/03/2023

The **GetIfStackTable** function retrieves a table of network interface stack row entries that specify the relationship of the network interfaces on an interface stack.

Syntax

```c++
NETIOAPI_API GetIfStackTable(
  _Out_ PMIB_IFSTACK_TABLE *Table
);
```

Parameters

- *Table* [out]
  A pointer to a buffer that receives the table of interface stack row entries in a MIB_IFSTACK_TABLE structure.

Return value

GetIfStackTable returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIfStackTable** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No interface stack entries were found.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIfStackTable** function enumerates the physical and logical network interfaces on an interface stack on a local computer and returns this information in a MIB_IFSTACK_TABLE structure.

Interface stack entries are returned in a MIB_IFSTACK_TABLE structure in the buffer that the *Table* parameter points to. The MIB_IFSTACK_TABLE structure contains an interface stack entry count and an array of MIB_IFSTACK_ROW structures for each interface stack entry.

The relationship between the interfaces in the interface stack is that the interface with index in the **HigherLayerInterfaceIndex** member of the MIB_IFSTACK_ROW structure is immediately above the interface with index in the **LowerLayerInterfaceIndex** member of the MIB_IFSTACK_ROW structure.

Memory is allocated by the **GetIfStackTable** function for the MIB_IFSTACK_TABLE structure and the MIB_IFSTACK_ROW entries in this structure. When these returned structures are no longer required, your driver should free the memory by calling FreeMibTable.

Note that the returned MIB_IFSTACK_TABLE structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_IFSTACK_ROW array entry in the **Table** member of the MIB_IFSTACK_TABLE structure. Padding for alignment might also be present between the MIB_IFSTACK_ROW array entries. Any access to a MIB_IFSTACK_ROW array entry should assume padding may exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FreeMibTable

GetIfEntry2

GetIfTable2

GetInvertedIfStackTable

GetIpInterfaceEntry

InitializeIpInterfaceEntry

MIB_IF_ROW2

MIB_IF_TABLE2

MIB_IFSTACK_ROW

MIB_IFSTACK_TABLE

MIB_INVERTEDIFSTACK_ROW

MIB_INVERTEDIFSTACK_TABLE

MIB_IPINTERFACE_ROW

NotifyIpInterfaceChange

SetIpInterfaceEntry

GetIfTable2 function

Article • 03/03/2023

The **GetIfTable2** function retrieves the MIB-II interface table.

Syntax

```
C++

NETIOAPI_API GetIfTable2(
  _Out_ PMIB_IF_TABLE2 *Table
);
```

Parameters

- *Table* [out]

  A pointer to a buffer that receives the table of interfaces in a **MIB_IF_TABLE2** structure.

Return value

GetIfTable2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIfTable2** returns one of the following error codes:

Return code	Description
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIfTable2** function enumerates the logical and physical interfaces on a local computer and returns this information in a **MIB_IF_TABLE2** structure.

Your driver can use a similar function, **GetIfTable2Ex**, to specify the level of interfaces to return. A call to the **GetIfTable2Ex** function with the *Level* parameter set to

MibIfTableNormal retrieves the same results as calling the **GetIfTable2** function.

GetIfTable2 returns interfaces in a MIB_IF_TABLE2 structure in the buffer that the *Table* parameter points to. The MIB_IF_TABLE2 structure contains an interface count and an array of **MIB_IF_ROW2** structures for each interface. **GetIfTable2** allocates memory for the MIB_IF_TABLE2 structure and the MIB_IF_ROW2 entries in this structure. When these returned structures are no longer required, your driver should free the memory by calling **FreeMibTable**.

Note that the returned MIB_IF_TABLE2 structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_IF_ROW2 array entry in the **Table** member of the MIB_IF_TABLE2 structure. Padding for alignment might also be present between the MIB_IF_ROW2 array entries. Any access to a MIB_IF_ROW2 array entry should assume padding may exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FreeMibTable

GetIfTable2Ex

MIB_IF_TABLE2

MIB_IF_ROW2

GetIfTable2Ex function

Article • 03/03/2023

The **GetIfTable2Ex** function retrieves the MIB-II interface table, given a level of interface information to retrieve.

Syntax

```c++
NETIOAPI_API GetIfTable2Ex(
  _In_  MIB_IF_TABLE_LEVEL Level,
  _Out_ PMIB_IF_TABLE2     *Table
);
```

Parameters

- *Level* [in]
  The level of interface information to retrieve. This parameter can be one of the values from the MIB_IF_TABLE_LEVEL enumeration.

- *Table* [out]
  A pointer to a buffer that receives the table of interfaces in a MIB_IF_TABLE2 structure.

Return value

GetIfTable2Ex returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIfTable2Ex** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if an illegal value was passed in the *Level* parameter.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIfTable2Ex** function enumerates the logical and physical interfaces on a local computer and returns this information in a **MIB_IF_TABLE2** structure.

Your driver can use a similar function, **GetIfTable2**, to retrieve interfaces, but **GetIfTable2** does not enable your driver tospecify the level of interfaces to return. A call to the **GetIfTable2Ex** function with the *Level* parameter set to **MibIfTableNormal** retrieves the same results as calling the **GetIfTable2** function.

GetIfTable2Ex returns interfaces in a MIB_IF_TABLE2 structure in the buffer that the *Table* parameter points to. The MIB_IF_TABLE2 structure contains an interface count and an array of **MIB_IF_ROW2** structures for each interface. **GetIfTable2** allocates mmory for the MIB_IF_TABLE2 structure and the MIB_IF_ROW2 entries in this structure. When these returned structures are no longer required, your driver should free the memory by calling **FreeMibTable**.

All interfaces, including NDIS intermediate driver interfaces and NDIS filter driver interfaces, are returned for either of the possible values for the *Level* parameter. The setting for the *Level* parameter affects how statistics and state members of the MIB_IF_ROW2 structure in the MIB_IF_TABLE2 structure that is pointed to by the *Table* parameter for the interface are returned. For example, a network interface card (NIC) has an NDIS miniport driver. An NDIS intermediate driver can be installed to interface between upper-level protocol drivers and NDIS miniport drivers. An NDIS filter driver can be attached on top of the NDIS intermediate driver. Assume that the NIC reports the **MediaConnectState** member of the MIB_IF_ROW2 structure as **MediaConnectStateConnected**, but the NDIS filter driver modifies the state and reports the state as **MediaConnectStateDisconnected**. When the interface information is queried with *Level* parameter set to **MibIfTableNormal**, the state at the top of the filter stack (**MediaConnectStateDisconnected**) is reported. When the interface is queried with the *Level* parameter set to **MibIfTableRaw**, the state at the interface level directly (**MediaConnectStateConnected**) is returned.

Note that the returned MIB_IF_TABLE2 structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_IF_ROW2 array entry in the **Table** member of the MIB_IF_TABLE2 structure. Padding for alignment might also be present between the MIB_IF_ROW2 array entries. Any access to a MIB_IF_ROW2 array entry should assume padding may exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FreeMibTable

GetIfTable2

MIB_IF_TABLE_LEVEL

MIB_IF_TABLE2

MIB_IF_ROW2

GetInvertedIfStackTable function

Article • 03/03/2023

The **GetInvertedIfStackTable** function retrieves a table of inverted network interface stack row entries that specify the relationship of the network interfaces on an interface stack.

Syntax

```c++
NETIOAPI_API GetInvertedIfStackTable(
  _Out_ PMIB_INVERTEDIFSTACK_TABLE *Table
);
```

Parameters

- *Table* [out]
  A pointer to a buffer that receives the table of inverted interface stack row entries in a MIB_INVERTEDIFSTACK_TABLE structure.

Return value

GetInvertedIfStackTable returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetInvertedIfStackTable** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No interface stack entries were found.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetInvertedIfStackTable** function enumerates the physical and logical network interfaces on an interface stack on a local computer and returns this information in an inverted form in the MIB_INVERTEDIFSTACK_TABLE structure.

GetInvertedIfStackTable returns interface stack entries in a MIB_INVERTEDIFSTACK_TABLE structure in the buffer that the *Table* parameter points to. The MIB_INVERTEDIFSTACK_TABLE structure contains an interface stack entry count and an array of MIB_INVERTEDIFSTACK_ROW structures for each interface stack entry.

The relationship between the interfaces in the interface stack is that the interface with index in the **HigherLayerInterfaceIndex** member of the MIB_INVERTEDIFSTACK_ROW structure is immediately above the interface with index in the **LowerLayerInterfaceIndex** member of the MIB_INVERTEDIFSTACK_ROW structure.

GetInvertedIfStackTable allocates memory for the MIB_INVERTEDIFSTACK_TABLE structure and the MIB_INVERTEDIFSTACK_ROW entries in this structure. When these returned structures are no longer required, your driver should free the memory by calling FreeMibTable.

Note that the returned MIB_INVERTEDIFSTACK_TABLE structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_INVERTEDIFSTACK_ROW array entry in the **Table** member of the MIB_INVERTEDIFSTACK_TABLE structure. Padding for alignment might also be present between the MIB_INVERTEDIFSTACK_ROW array entries. Any access to a MIB_INVERTEDIFSTACK_ROW array entry should assume padding might exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FreeMibTable

GetIfEntry2

GetIfStackTable

GetIfTable2

GetIpInterfaceEntry

InitializeIpInterfaceEntry

MIB_IF_ROW2

MIB_IF_TABLE2

MIB_IFSTACK_ROW

MIB_IFSTACK_TABLE

MIB_INVERTEDIFSTACK_ROW

MIB_INVERTEDIFSTACK_TABLE

MIB_IPINTERFACE_ROW

NotifyIpInterfaceChange

SetIpInterfaceEntry

GetIpInterfaceTable function

Article • 03/03/2023

The **GetIpInterfaceTable** function retrieves the IP interface entries on a local computer.

Syntax

C++

```
NETIOAPI_API GetIpInterfaceTable(
  _In_  ADDRESS_FAMILY         Family,
  _Out_ PMIB_IPINTERFACE_TABLE *Table
);
```

Parameters

- *Family* [in]
  The address family of IP interfaces to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family.

  - AF_INET6
    The IPv6 address family.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, the **GetIpInterfaceTable** function returns the IP interface table that contains both IPv4 and IPv6 entries.

- *Table* [out]
  A pointer to a buffer that receives the table of IP interface entries in a MIB_IPINTERFACE_TABLE structure.

Return value

GetIpInterfaceTable returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpInterfaceTable** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter or the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No IP interface entries, as specified in the *Family* parameter, were found.
STATUS_NOT_SUPPORTED	The function is not supported. This error is returned when the IP transport that is specified in the *Address* parameter is not configured on the local computer. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIpInterfaceTable** function enumerates the IP interfaces on a local computer and returns this information in an MIB_IPINTERFACE_TABLE structure.

GetIpInterfaceTable returns IP interface entries in a MIB_IPINTERFACE_TABLE structure in the buffer that the *Table* parameter points to. The MIB_IPINTERFACE_TABLE structure contains an IP interface entry count and an array of MIB_IPINTERFACE_ROW structures for each IP interface entry. When these returned structures are no longer required, your driver should free the memory by calling the FreeMibTable function.

Your driver must initialize the *Family* parameter to either AF_INET or AF_INET6.

Note that the returned MIB_IPINTERFACE_TABLE structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first **MIB_IPINTERFACE_ROW** array entry in the **Table** member of the MIB_IPINTERFACE_TABLE structure. Padding for alignment might also be present between the MIB_IPINTERFACE_ROW array entries. Any access to a MIB_IPINTERFACE_ROW array entry should assume padding might exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FreeMibTable

MIB_IPINTERFACE_ROW

MIB_IPINTERFACE_TABLE

MIB_IPNET_ROW2

MIB_IPNET_TABLE2

InitializeIpInterfaceEntry function

Article • 03/03/2023

The **InitializeIpInterfaceEntry** function initializes the members of an MIB_IPINTERFACE_ROW structure entry with default values.

Syntax

```
C++

VOID NETIOAPI_API_ InitializeIpInterfaceEntry(
  _Inout_ PMIB_IPINTERFACE_ROW Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IPINTERFACE_ROW structure to initialize. On successful return, the fields in this parameter are initialized with default information for an interface on the local computer.

Return value

InitializeIpInterfaceEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **InitializeIpInterfaceEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

On output, the members of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to are initialized as follows.

- **Family**

  Set to AF_UNSPEC.

- **InterfaceLuid**

  Set to an unspecified value.

- All other members

  Set to zero.

Your driver must use the **InitializeIpInterfaceEntry** function to initialize the fields of a MIB_IPINTERFACE_ROW structure entry with default values. A driver can then change the fields in the MIB_IPINTERFACE_ROW entry that it wants to modify, and then call the SetIpInterfaceEntry function.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetIpInterfaceEntry

GetIpInterfaceTable

MIB_IPINTERFACE_ROW

MIB_IPINTERFACE_TABLE

SetIpInterfaceEntry

SetIpInterfaceEntry function

Article • 03/03/2023

The **SetIpInterfaceEntry** function sets the properties of an IP interface on a local computer.

Syntax

```c++
NETIOAPI_API SetIpInterfaceEntry(
  _Inout_ PMIB_IPINTERFACE_ROW Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IPINTERFACE_ROW structure entry for an interface. On input, your driver must set the **Family** member of the MIB_IPINTERFACE_ROW to AF_INET6 or AF_INET and your driver must specify the **InterfaceLuid** member or the **InterfaceIndex** member of MIB_IPINTERFACE_ROW. On a successful return, the **InterfaceLuid** member of the MIB_IPINTERFACE_ROW is filled in if the **InterfaceIndex** member of the MIB_IPINTERFACE_ROW entry was specified.

Return value

SetIpInterfaceEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **SetIpInterfaceEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Family** member of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to was not specified as AF_INET or AF_INET6, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPINTERFACE_ROW structure were unspecified.

Return code	Description
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Your driver must use the **InitializeIpInterfaceEntry** function to initialize the fields of a **MIB_IPINTERFACE_ROW** structure entry with default values. A driver can then change the fields in the MIB_IPINTERFACE_ROW entry that it wants to modify, and then call the **SetIpInterfaceEntry** function.

On input, your driver must initialize the following members of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to.

- **Family**
  Set to either AF_INET or AF_INET6.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, the **InterfaceLuid** member of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to is filled in if the **InterfaceIndex** was specified.

SetIpInterfaceEntry ignores the **MaxReassemblySize**, **MinRouterAdvertisementInterval**, **MaxRouterAdvertisementInterval**, **Connected**, **SupportsWakeUpPatterns**, **SupportsNeighborDiscovery**, **SupportsRouterDiscovery**, **ReachableTime**, **TransmitOffload**, and **ReceiveOffload** members of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to. These members are set by the network stack and cannot be changed by using the **SetIpInterfaceEntry** function.

Unprivileged simultaneous access to multiple networks of different security requirements creates a security hole and enables an unprivileged driver to accidentally relay data between the two networks. A typical example is simultaneous access to a virtual private network (VPN) and the Internet. The Windows Server 2003 and Windows

XP operating systems use a weak host model, where Remote Access Service (RAS) prevents such simultaneous access by increasing the route metric of all default routes over other interfaces. Therefore, all traffic is routed through the VPN interface, disrupting other network connectivity.

On Windows Vista and later versions of the Windows operating systems, by default, a strong host model is used. If a source IP address is specified in the route lookup by using the GetBestRoute2 function, the route lookup is restricted to the interface of the source IP address. The route metric modification by RAS has no effect because the list of potential routes does not even have the route for the VPN interface, which enables traffic to the Internet. Your driver can use the **DisableDefaultRoutes** member of the MIB_IPINTERFACE_ROW structure to disable using the default route on an interface. VPN clients can use this member as a security measure to restrict split tunneling when split tunneling is not required by the VPN client. A VPN client can call the **SetIpInterfaceEntry** function to set the **DisableDefaultRoutes** member to **TRUE** when it is required. A VPN client can query the current state of the **DisableDefaultRoutes** member by calling the **GetIpInterfaceEntry** function.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetBestRoute2

GetIfEntry2

GetIfTable2

GetIfTable2Ex

GetIpInterfaceEntry

GetIpInterfaceTable

InitializeIpInterfaceEntry

MIB_IF_ROW2

MIB_IF_TABLE2

MIB_IPINTERFACE_ROW

MIB_IPINTERFACE_TABLE

NotifyIpInterfaceChange

CreateAnycastIpAddressEntry function

Article • 03/03/2023

The **CreateAnycastIpAddressEntry** function adds a new anycast IP address entry on a local computer.

Syntax

```C++
NETIOAPI_API CreateAnycastIpAddressEntry(
  _In_ const MIB_ANYCASTIPADDRESS_ROW *Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_ANYCASTIPADDRESS_ROW structure entry for an anycast IP address entry.

Return value

CreateAnycastIpAddressEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **CreateAnycastIpAddressEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid unicast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_ANYCASTIPADDRESS_ROW structure were unspecified.

Return code	Description
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
ERROR_OBJECT_ALREADY_EXISTS	The object already exists. This error is returned if the **Address** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to is a duplicate of an existing anycast IP address on the interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_ANYCASTIPADDRESS_ROW structure.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Your driver must initialize the following members of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to.

- **Address**
  Set to a valid unicast IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface to add the unicast IP address to. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

The **ScopeId** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to is ignored when the **CreateAnycastIpAddressEntry** function is called. The **ScopeId** member is automatically determined by the interface that the address is added on.

The **CreateAnycastIpAddressEntry** function fails if the anycast IP address that is passed in the **Address** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to is a duplicate of an existing anycast IP address on the interface.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

DeleteAnycastIpAddressEntry

GetAnycastIpAddressEntry

GetAnycastIpAddressTable

MIB_ANYCASTIPADDRESS_ROW

MIB_ANYCASTIPADDRESS_TABLE

CreateSortedAddressPairs function

Article • 03/03/2023

From a supplied list of potential IP destination addresses, the **CreateSortedAddressPairs** function pairs the destination addresses together with the host machine's local IP addresses and sorts the pairs according to the preferred order of communication.

Syntax

C++

```
NETIOAPI_API CreateSortedAddressPairs(
  _In_opt_ const PSOCKADDR_IN6      SourceAddressList,
  _In_           ULONG              SourceAddressCount,
  _In_     const PSOCKADDR_IN6      DestinationAddressList,
  _In_           ULONG              DestinationAddressCount,
  _In_           ULONG              AddressSortOptions,
  _In_           PSOCKADDR_IN6_PAIR *SortedAddressPairList,
  _Out_          ULONG              *SortedAddressPairCount
);
```

Parameters

- *SourceAddressList* [in, optional]

  Reserved. This parameter must be **NULL**.

- *SourceAddressCount* [in]

  Reserved. This parameter must be zero.

- *DestinationAddressList* [in]

  A pointer to a list of potential destination addresses of type **SOCKADDR_IN6**.

- *DestinationAddressCount* [in]

  The number of addresses in the list that the *DestinationAddressList* parameter points to.

- *AddressSortOptions* [in]

  Reserved. This parameter must be zero.

- *SortedAddressPairList* [in]

  A pointer to a list of pairs of source and destination addresses, sorted in the preferred order of communication. For more information about this parameter, see the following Remarks section.

- *SortedAddressPairCount* [out]
  The number of address pairs in the list that the *SortedAddressPairList* parameter points to.

Return value

CreateSortedAddressPairs returns STATUS_SUCCESS if the function succeeds.

If the function fails, **CreateSortedAddressPairs** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources were available to complete the operation.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **CreateSortedAddressPairs** function automatically pairs the host machine's local addresses together with the supplied list of potential destination addresses that the *DestinationAddressList* parameter points to.

The returned list of pairs of addresses that the *SortedAddressPairList* parameter points to is sorted so that the address pairs that are best suited for communication between two peers occurr earlier in the list.

The *SortedAddressPairList* parameter is of type PSOCKADDR_IN6_PAIR, which is defined in the Ws2ipdef.h header as follows.

```C++

    typedef struct _sockaddr_in6_pair
    {
        PSOCKADDR_IN6  SourceAddress;
        PSOCKADDR_IN6  DestinationAddress;
    } SOCKADDR_IN6_PAIR, *PSOCKADDR_IN6_PAIR;

  - **SourceAddress**
    The IP source address.
  - **DestinationAddress**
    The IP destination address.
```

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FormatMessage

SOCKADDR_IN6

CreateUnicastIpAddressEntry function

Article • 03/03/2023

The **CreateUnicastIpAddressEntry** function adds a new unicast IP address entry on the local computer.

Syntax

```C++
NETIOAPI_API CreateUnicastIpAddressEntry(
  _In_ const MIB_UNICASTIPADDRESS_ROW *Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_UNICASTIPADDRESS_ROW structure entry for a unicast IP address entry.

Return value

CreateUnicastIpAddressEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **CreateUnicastIpAddressEntry** returns one of the following error codes:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid unicast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_UNICASTIPADDRESS_ROW structure were unspecified. This error is also returned for other errors in the values that are set for members in the MIB_UNICASTIPADDRESS_ROW structure. These errors include the following situations: • The **ValidLifetime** member is less than than the **PreferredLifetime** member. • The **PrefixOrigin** member is set to **IpPrefixOriginUnchanged** and the **SuffixOrigin** is not set to IpSuffixOriginUnchanged. • The **PrefixOrigin** member is not set to **IpPrefixOriginUnchanged** and the **SuffixOrigin** is set to IpSuffixOriginUnchanged. • The **PrefixOrigin** member is not set to a value from the NL_PREFIX_ORIGIN enumeration. • The **SuffixOrigin** member is not set to a value from the NL_SUFFIX_ORIGIN enumeration. • The **OnLinkPrefixLength** member is set to a value that is greater than the IP address length, in bits (32 for a unicast IPv4 address or 128 for a unicast IPv6 address). For possible values of the NL_PREFIX_ORIGIN and NL_SUFFIX_ORIGIN enumerations, see MIB_UNICASTIPADDRESS_ROW.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.

Return code	Description
ERROR_OBJECT_ALREADY_EXISTS	The object already exists. This error is returned if the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to is a duplicate of an existing unicast IP address on the interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_UNICASTIPADDRESS_ROW.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Use the InitializeUnicastIpAddressEntry function to initialize the members of a MIB_UNICASTIPADDRESS_ROW structure entry with default values. A driver can then change the members in the MIB_UNICASTIPADDRESS_ROW entry that it wants to modify, and then call the **CreateUnicastIpAddressEntry** function.

On input, your driver must initialize the following members of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.

- **Address**
  Set to a valid unicast IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface to add the unicast IP address to. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

If the **OnLinkPrefixLength** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to is set to 255, **CreateUnicastIpAddressEntry** adds the new unicast IP address with the **OnLinkPrefixLength** member set equal to the length of the IP address. So for a unicast IPv4 address, the **OnLinkPrefixLength** is set to 32 and the **OnLinkPrefixLength** is set to 128 for a unicast IPv6 address. If this setting would result in the incorrect subnet mask for an IPv4 address or the incorrect link prefix for an IPv6 address, the driver should set the **OnLinkPrefixLength** member to the correct value before calling **CreateUnicastIpAddressEntry**.

If a unicast IP address is created with the **OnLinkPrefixLength** member set incorrectly, your driver can change the IP address by calling SetUnicastIpAddressEntry with the **OnLinkPrefixLength** member set to the correct value.

The **DadState**, **ScopeId**, and **CreationTimeStamp** members of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to are ignored when the **CreateUnicastIpAddressEntry** function is called. These members are set by the network stack. The **ScopeId** member is automatically determined by the interface that the address is added on.

The **CreateUnicastIpAddressEntry** function fails if the unicast IP address that is passed in the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to is a duplicate of an existing unicast IP address on the interface. Note that your driver can add a loopback IP address to a loopback interface only by using the **CreateUnicastIpAddressEntry** function.

The unicast IP address that is passed in the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to is not usable immediately. The IP address is usable after the duplicate address detection process has completed successfully. It can take several seconds for the duplicate address detection process to complete because IP packets must be sent and potential responses must be waited for. For IPv6, the duplicate address detection process typically takes about 1 second. For IPv4, the duplicate address detection process typically takes about 3 seconds.

After a driver calls the **CreateUnicastIpAddressEntry** function, it can use the following methods to determine if an IP address is still usable:

- **Use polling and the GetUnicastIpAddressEntry function**
  After the call to the **CreateUnicastIpAddressEntry** function returns successfully, pause for 1 to 3 seconds (depending on whether an IPv6 or IPv4 address is being created) to allow time for the successful completion of the duplication address detection process. Then, call **GetUnicastIpAddressEntry** to retrieve the updated MIB_UNICASTIPADDRESS_ROW structure and examine the value of the **DadState** member. If the value of the **DadState** member is set to IpDadStatePreferred, the IP address is now usable. If the value of the **DadState** member is set to IpDadStateTentative, duplicate address detection has not yet completed. In this case, call the **GetUnicastIpAddressEntry** function again every 0.5 seconds while the **DadState** member is still set to IpDadStateTentative. If the value of the **DadState** member returns with some value other than **IpDadStatePreferred** or IpDadStateTentative, duplicate address detection has failed and the IP address is not usable.

- **Call one of the IP Helper NotifyXxx notification functions to set up an asynchronous notification for when an address changes**
  After the call to the **CreateUnicastIpAddressEntry** function returns successfully, call

the NotifyUnicastIpAddressChange function to register the driver to be notified of changes to either IPv6 or IPv4 unicast IP addresses, depending on the type of IP address that is being created. When a notification is received for the IP address that is being created, call the **GetUnicastIpAddressEntry** function to retrieve the **DadState** member. If the value of the **DadState** member is set to IpDadStatePreferred, the IP address is now usable. If the value of the **DadState** member is set to IpDadStateTentative, duplicate address detection has not yet completed and the driver must wait for future notifications. If the value of the **DadState** member returns with some value other than **IpDadStatePreferred** or IpDadStateTentative, duplicate address detection has failed and the IP address is not usable.

If, during the duplicate address detection process, the media is disconnected and then reconnected, the duplicate address detection process is restarted. So the time to complete the process might increase beyond the typical 1 second value for IPv6 or 3 second value for IPv4.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

DeleteUnicastIpAddressEntry

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

InitializeUnicastIpAddressEntry

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NL_PREFIX_ORIGIN

NL_SUFFIX_ORIGIN

NotifyIpInterfaceChange

NotifyRouteChange2

NotifyStableUnicastIpAddressTable

NotifyTeredoPortChange

NotifyUnicastIpAddressChange

SetUnicastIpAddressEntry

DeleteAnycastIpAddressEntry function

Article • 03/03/2023

The **DeleteAnycastIpAddressEntry** function deletes an existing anycast IP address entry on a local computer.

Syntax

```C++
NETIOAPI_API DeleteAnycastIpAddressEntry(
  _In_ const MIB_ANYCASTIPADDRESS_ROW *Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_ANYCASTIPADDRESS_ROW structure entry for an existing anycast IP address entry to delete from the local computer.

Return value

DeleteAnycastIpAddressEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **DeleteAnycastIpAddressEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid unicast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_ANYCASTIPADDRESS_ROW structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **DeleteAnycastIpAddressEntry** function is used to delete an existing MIB_ANYCASTIPADDRESS_ROW structure entry on the local computer.

On input, your driver must initialize the following members of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to.

- **Address**
  Set to a valid unicast IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

If the function is successful, the existing IP address that the *Row* parameter represents was deleted.

Your driver can call the **GetAnycastIpAddressTable** function to enumerate the anycast IP address entries on a local computer. Your driver can call the **GetAnycastIpAddressEntry** function to retrieve a specific existing anycast IP address entry.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateAnycastIpAddressEntry

GetAnycastIpAddressEntry

GetAnycastIpAddressTable

MIB_ANYCASTIPADDRESS_ROW

MIB_ANYCASTIPADDRESS_TABLE

DeleteUnicastIpAddressEntry function

Article • 03/03/2023

The **DeleteUnicastIpAddressEntry** function deletes an existing unicast IP address entry on a local computer.

Syntax

C++

```
NETIOAPI_API DeleteUnicastIpAddressEntry(
  _In_ const MIB_UNICASTIPADDRESS_ROW *Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_UNICASTIPADDRESS_ROW structure entry for an existing unicast IP address entry to delete from the local computer.

Return value

DeleteUnicastIpAddressEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **DeleteUnicastIpAddressEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid unicast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_UNICASTIPADDRESS_ROW structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **DeleteUnicastIpAddressEntry** function is used to delete an existing **MIB_UNICASTIPADDRESS_ROW** structure entry on the local computer.

On input, your driver must initialize the following members of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.

- **Address**
  Set to a valid IPv4 or IPv6 unicast address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

If the function is successful, the existing IP address that the *Row* parameter represents is deleted.

Your driver can call the **GetUnicastIpAddressTable** function to enumerate the unicast IP address entries on a local computer. Your driver can call the **GetUnicastIpAddressEntry** function to retrieve a specific existing unicast IP address entry.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateUnicastIpAddressEntry

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

InitializeUnicastIpAddressEntry

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NotifyUnicastIpAddressChange

SetUnicastIpAddressEntry

GetAnycastIpAddressEntry function

Article • 03/03/2023

The **GetAnycastIpAddressEntry** function retrieves information for an existing anycast IP address entry on a local computer.

Syntax

```C++
NETIOAPI_API GetAnycastIpAddressEntry(
  _Inout_ PMIB_ANYCASTIPADDRESS_ROW Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_ANYCASTIPADDRESS_ROW structure entry for an anycast IP address entry. On successful return, this structure is updated with the properties for an existing anycast IP address.

Return value

GetAnycastIpAddressEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetAnycastIpAddressEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid anycast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_ANYCASTIPADDRESS_ROW structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetAnycastIpAddressEntry** function is used to retrieve an existing MIB_ANYCASTIPADDRESS_ROW structure entry.

On input, your driver must initialize the following members of the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to.

- **Address**
  Set to a valid IPv4 or IPv6 anycast address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **GetAnycastIpAddressEntry** retrieves the other properties for the anycast IP address and fills out the MIB_ANYCASTIPADDRESS_ROW structure that the *Row* parameter points to.

Your driver can call the GetAnycastIpAddressTable function to enumerate the anycast IP address entries on a local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateAnycastIpAddressEntry

DeleteAnycastIpAddressEntry

GetAnycastIpAddressTable

MIB_ANYCASTIPADDRESS_ROW

MIB_ANYCASTIPADDRESS_TABLE

GetAnycastIpAddressTable function

Article • 03/03/2023

The **GetAnycastIpAddressTable** function retrieves the anycast IP address table on a local computer.

Syntax

```c++
NETIOAPI_API GetAnycastIpAddressTable(
  _In_  ADDRESS_FAMILY             Family,
  _Out_ PMIB_ANYCASTIPADDRESS_TABLE *Table
);
```

Parameters

- *Family* [in]
  The address family to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameterare defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function returns the anycast IP address table that contains only IPv4 entries.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function returns the anycast IP address table that contains only IPv6 entries.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, this function

returns the anycast IP address table that contains both IPv4 and IPv6 entries.

- *Table* [out]
  A pointer to a MIB_ANYCASTIPADDRESS_TABLE structure that contains a table of anycast IP address entries on the local computer.

Return value

GetAnycastIpAddressTable returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetAnycastIpAddressTable** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter or the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No anycast IP address entries, as specified in the *Family* parameter, were found.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetAnycastIpAddressTable** function enumerates the anycast IP addresses on a local computer and returns this information in a MIB_ANYCASTIPADDRESS_TABLE structure.

The anycast IP address entries are returned in a MIB_ANYCASTIPADDRESS_TABLE structure in the buffer that the *Table* parameter points to. The MIB_ANYCASTIPADDRESS_TABLE structure contains an anycast IP address entry count and an array of MIB_ANYCASTIPADDRESS_ROW structures for each anycast IP address entry. When these returned structures are no longer required, your driver should free the memory by calling FreeMibTable.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Note that the returned MIB_ANYCASTIPADDRESS_TABLE structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_ANYCASTIPADDRESS_ROW array entry in the **Table** member of the MIB_ANYCASTIPADDRESS_TABLE structure. Padding for alignment might also be present between the MIB_ANYCASTIPADDRESS_ROW array entries. Any access to a MIB_ANYCASTIPADDRESS_ROW array entry should assume padding may exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FreeMibTable

MIB_ANYCASTIPADDRESS_ROW

MIB_ANYCASTIPADDRESS_TABLE

GetMulticastIpAddressEntry function

Article • 03/03/2023

The **GetMulticastIpAddressEntry** function retrieves information for an existing multicast IP address entry on a local computer.

Syntax

```c++
NETIOAPI_API GetMulticastIpAddressEntry(
  _Inout_ PMIB_MULTICASTIPADDRESS_ROW Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_MULTICASTIPADDRESS_ROW structure entry for a multicast IP address entry. On successful return, this structure is updated with the properties for an existing multicast IP address.

Return value

GetMulticastIpAddressEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetMulticastIpAddressEntry** returns one of the following error codes:

Return code	**Description**
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_MULTICASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid multicast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_MULTICASTIPADDRESS_ROW structure were unspecified.

Return code	Description
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_MULTICASTIPADDRESS_ROW structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_MULTICASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetMulticastIpAddressEntry** function is used to retrieve an existing MIB_MULTICASTIPADDRESS_ROW structure entry.

On input, your driver must initialize the following members of the MIB_MULTICASTIPADDRESS_ROW structure that the *Row* parameter points to.

- **Address**
  Set to a valid IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **GetMulticastIpAddressEntry** retrieves the other properties for the multicast IP address and fills in the MIB_MULTICASTIPADDRESS_ROW structure that the *Row* parameter points to.

Your driver can call the **GetMulticastIpAddressTable** function to enumerate the multicast IP address entries on a local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetMulticastIpAddressTable

MIB_MULTICASTIPADDRESS_ROW

MIB_MULTICASTIPADDRESS_TABLE

GetMulticastIpAddressTable function

Article • 03/03/2023

The **GetMulticastIpAddressTable** function retrieves the multicast IP address table on a local computer.

Syntax

```c++
NETIOAPI_API GetMulticastIpAddressTable(
  _In_  ADDRESS_FAMILY                Family,
  _Out_ PMIB_MULTICASTIPADDRESS_TABLE *Table
);
```

Parameters

- *Family* [in]
  The address family to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function returns the multicast IP address table that contains only IPv4 entries.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function returns the multicast IP address table that contains only IPv6 entries.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, this function

returns the multicast IP address table that contains both IPv4 and IPv6 entries.

- *Table* [out]
  A pointer to a MIB_MULTICASTIPADDRESS_TABLE structure that contains a table of anycast IP address entries on the local computer.

Return value

GetMulticastIpAddressTable returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetMulticastIpAddressTable** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter or the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No anycast IP address entries, as specified in the *Family* parameter, were found.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetMulticastIpAddressTable** function enumerates the multicast IP addresses on a local computer and returns this information in a MIB_MULTICASTIPADDRESS_TABLE structure.

GetMulticastIpAddressTable returns the multicast IP address entries in a MIB_MULTICASTIPADDRESS_TABLE structure in the buffer that the *Table* parameter points to. The MIB_MULTICASTIPADDRESS_TABLE structure contains a multicast IP address entry count and an array of MIB_MULTICASTIPADDRESS_ROW structures for each multicast IP address entry. When these returned structures are no longer required, your driver should free the memory by calling FreeMibTable.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Note that the returned MIB_MULTICASTIPADDRESS_TABLE structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_MULTICASTIPADDRESS_ROW array entry in the **Table** member of the MIB_MULTICASTIPADDRESS_TABLE structure. Padding for alignment might also be present between the MIB_MULTICASTIPADDRESS_ROW array entries. Any access to a MIB_MULTICASTIPADDRESS_ROW array entry should assume padding might exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetMulticastIpAddressEntry

MIB_MULTICASTIPADDRESS_ROW

MIB_MULTICASTIPADDRESS_TABLE

GetUnicastIpAddressEntry function

Article • 03/03/2023

The **GetUnicastIpAddressEntry** function retrieves information for an existing unicast IP address entry on a local computer.

Syntax

```c++
NETIOAPI_API GetUnicastIpAddressEntry(
  _Inout_ PMIB_UNICASTIPADDRESS_ROW Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_UNICASTIPADDRESS_ROW structure entry for a unicast IP address entry. On successful return, this structure is updated with the properties for an existing unicast IP address.

Return value

GetUnicastIpAddressEntry returns **STATUS_SUCCESS** if the function succeeds.

If the function fails, **GetUnicastIpAddressEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid unicast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the **MIB_UNICASTIPADDRESS_ROW** structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetUnicastIpAddressEntry** function is typically used to retrieve an existing **MIB_UNICASTIPADDRESS_ROW** structure entry to be modified. A driver can then change the members in the **MIB_UNICASTIPADDRESS_ROW** entry that it wants to modify, and then call the **SetUnicastIpAddressEntry** function.

On input, your driver must initialize the following members of the **MIB_UNICASTIPADDRESS_ROW** structure that the *Row* parameter points to.

- **Address**
  Set to a valid unicast IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **GetUnicastIpAddressEntry** retrieves the other properties for the unicast IP address and fills in the **MIB_UNICASTIPADDRESS_ROW** structure that the *Row* parameter points to.

Your driver can call the **GetUnicastIpAddressTable** function to enumerate the unicast IP address entries on a local computer.

Requirements

Target platform	Universal

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateUnicastIpAddressEntry

DeleteUnicastIpAddressEntry

GetUnicastIpAddressTable

InitializeUnicastIpAddressEntry

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NotifyUnicastIpAddressChange

SetUnicastIpAddressEntry

GetUnicastIpAddressTable function

Article • 03/03/2023

The **GetUnicastIpAddressTable** function retrieves the unicast IP address table on a local computer.

Syntax

```
C++

NETIOAPI_API GetUnicastIpAddressTable(
  _In_  ADDRESS_FAMILY             Family,
  _Out_ PMIB_UNICASTIPADDRESS_TABLE *Table
);
```

Parameters

- *Family* [in]
  The address family to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function returns the multicast IP address table that contains only IPv4 entries.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function returns the multicast IP address table that contains only IPv6 entries.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, this function

returns the multicast IP address table that contains both IPv4 and IPv6 entries.

- *Table* [out]
  A pointer to a MIB_UNICASTIPADDRESS_TABLE structure that contains a table of unicast IP address entries on the local computer.

Return value

GetUnicastIpAddressTable returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetUnicastIpAddressTable** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter or the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No unicast IP address entries, as specified in the *Family* parameter, were found.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetUnicastIpAddressTable** function enumerates the unicast IP addresses on a local computer and returns this information in an MIB_UNICASTIPADDRESS_TABLE structure.

GetUnicastIpAddressTable returns the unicast IP address entries in a MIB_UNICASTIPADDRESS_TABLE structure in the buffer that the *Table* parameter points to. The MIB_UNICASTIPADDRESS_TABLE structure contains a unicast IP address entry count and an array of MIB_UNICASTIPADDRESS_ROW structures for each unicast IP address entry. When these returned structures are no longer required, your driver should free the memory by calling FreeMibTable.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Note that the returned MIB_UNICASTIPADDRESS_TABLE structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_UNICASTIPADDRESS_ROW array entry in the **Table** member of the MIB_UNICASTIPADDRESS_TABLE structure. Padding for alignment might also be present between the MIB_UNICASTIPADDRESS_ROW array entries. Any access to a MIB_UNICASTIPADDRESS_ROW array entry should assume padding might exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateUnicastIpAddressEntry

DeleteUnicastIpAddressEntry

FreeMibTable

GetUnicastIpAddressEntry

InitializeUnicastIpAddressEntry

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NotifyStableUnicastIpAddressTable

NotifyUnicastIpAddressChange

SetUnicastIpAddressEntry

InitializeUnicastIpAddressEntry function

Article • 03/03/2023

The **InitializeUnicastIpAddressEntry** function initializes a MIB_UNICASTIPADDRESS_ROW structure with default values for a unicast IP address entry on a local computer.

Syntax

```c++
VOID NETIOAPI_API_ InitializeUnicastIpAddressEntry(
  _Out_ PMIB_UNICASTIPADDRESS_ROW Row
);
```

Parameters

- *Row* [out]
  On entry, a pointer to a MIB_UNICASTIPADDRESS_ROW structure entry for a unicast IP address entry. On return, the MIB_UNICASTIPADDRESS_ROW structure that this parameter points to is initialized with default values for a unicast IP address.

Return value

None

Remarks

Your driver must use the **InitializeUnicastIpAddressEntry** function to initialize the members of a MIB_UNICASTIPADDRESS_ROW structure entry with default values for a unicast IP address for later use with the CreateUnicastIpAddressEntry function.

On input, your driver must pass **InitializeUnicastIpAddressEntry** a new MIB_UNICASTIPADDRESS_ROW structure to initialize.

On output, the members of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to are initialized as follows.

- PrefixOrigin

  Set to the **IpPrefixOriginUnchanged** value of the NL_PREFIX_ORIGIN enumeration.

- SuffixOrigin

  Set to the **IpSuffixOriginUnchanged** value of the NL_PREFIX_ORIGIN enumeration.

- OnLinkPrefixLength

  Set to an illegal value.

- **PreferredLifetime** and **ValidLifetime**

  Set to infinite values.

- SkipAsSource

  Set to **FALSE**.

- All other members

  Set to zero.

After a driver calls **InitializeUnicastIpAddressEntry**, the driver can then change the members in the MIB_UNICASTIPADDRESS_ROW entry that it wants to modify, and then call the **CreateUnicastIpAddressEntry** to add the new unicast IP address to the local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateUnicastIpAddressEntry

DeleteUnicastIpAddressEntry

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NotifyUnicastIpAddressChange

SetUnicastIpAddressEntry

SetUnicastIpAddressEntry function

Article • 03/03/2023

The **SetUnicastIpAddressEntry** function sets the properties of an existing unicast IP address entry on a local computer.

Syntax

```c++
NETIOAPI_API SetUnicastIpAddressEntry(
  _In_ const MIB_UNICASTIPADDRESS_ROW *Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_UNICASTIPADDRESS_ROW structure entry for an existing unicast IP address entry.

Return value

SetUnicastIpAddressEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **SetUnicastIpAddressEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to was not set to a valid unicast IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_UNICASTIPADDRESS_ROW structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetUnicastIpAddressEntry** function is typically used to retrieve an existing **MIB_UNICASTIPADDRESS_ROW** structure entry to be modified. A driver can then change the members in the MIB_UNICASTIPADDRESS_ROW entry that it wants to modify, and then call the **SetUnicastIpAddressEntry** function.

A driver can call the **InitializeUnicastIpAddressEntry** function to initialize the members of a MIB_UNICASTIPADDRESS_ROW structure entry with default values before making changes. However, the driver typically saves either the **InterfaceLuid** or **InterfaceIndex** member before calling **InitializeUnicastIpAddressEntry** and restores one of these members after the call.

Your driver must initialize the following members of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to.

- **Address**
  Set to a valid unicast IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

If the **OnLinkPrefixLength** member of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to is set to 255, **SetUnicastIpAddressEntry** sets the unicast IP address properties so that the **OnLinkPrefixLength** member is equal to the length of the IP address. For a unicast IPv4 address, **OnLinkPrefixLength** is set to 32. For a unicast IPv6 address, **OnLinkPrefixLength** is set to 128. If these settings would result in the incorrect subnet mask for an IPv4 address or the incorrect link prefix for an IPv6 address, the driver should set this member to the correct value before calling **SetUnicastIpAddressEntry**.

SetUnicastIpAddressEntry ignores the **DadState**, **ScopeId**, and **CreationTimeStamp** members of the MIB_UNICASTIPADDRESS_ROW structure that the *Row* parameter points to. These members are set by the network stack and cannot be changed by using the **SetUnicastIpAddressEntry** function. The **ScopeId** member is automatically determined by the interface that the address was added on.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateUnicastIpAddressEntry

DeleteUnicastIpAddressEntry

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

InitializeUnicastIpAddressEntry

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NotifyUnicastIpAddressChange

CreateIpNetEntry2 function

Article • 03/03/2023

The **CreateIpNetEntry2** function creates a new neighbor IP address entry on the local computer.

Syntax

```
C++

NETIOAPI_API CreateIpNetEntry2(
  _In_ const MIB_IPNET_ROW2 *Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_IPNET_ROW2 structure entry for an IP route entry.

Return value

CreateIpNetEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **CreateIpNetEntry2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if one of the following items occurs: • A **NULL** pointer is passed in the *Row* parameter. • The **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to was not set to a valid unicast, anycast, or multicast IPv4 or IPv6 address. • The **PhysicalAddress** and **PhysicalAddressLength** members of the MIB_IPNET_ROW2 structure were not set to a valid physical address. • Both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPNET_ROW2 structure were unspecified. • A loopback address was passed in the **Address** member.

Return code	Description
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
ERROR_OBJECT_ALREADY_EXISTS	The object already exists. This error is returned if the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to is a duplicate of an existing neighbor IP address on the interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPNET_ROW2 structure.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Your driver must initialize the following members of the **MIB_IPNET_ROW2** structure that the *Row* parameter points to:

- Set the **Address** member to a valid unicast, anycast, or multicast IPv4 or IPv6 address and family.
- Set the **PhysicalAddress** and **PhysicalAddressLength** members in the MIB_IPNET_ROW2 structure to a valid physical address.
- Set **InterfaceLuid** or **InterfaceIndex** to the LUID or index value of the interface.

The **InterfaceLuid** and **InterfaceIndex** members are used in the order that is listed earlier. So if the **InterfaceLuid** is specified, this member is used to determine the interface to add the unicast IP address on. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

The **CreateIpNetEntry2** function fails if the IP address that is passed in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to is a duplicate of an existing neighbor IP address on the interface.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

DeleteIpNetEntry2

FlushIpNetTable2

GetIpNetEntry2

GetIpNetTable2

MIB_IPNET_ROW2

MIB_IPNET_TABLE2

ResolveIpNetEntry2

SetIpNetEntry2

DeleteIpNetEntry2 function

Article • 03/03/2023

The **DeleteIpNetEntry2** function deletes a neighbor IP address entry on a local computer.

Syntax

C++

```
NETIOAPI_API DeleteIpNetEntry2(
  _In_ const MIB_IPNET_ROW2 *Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_IPNET_ROW2 structure entry for a neighbor IP address entry. On successful return, this entry is deleted.

Return value

DeleteIpNetEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **DeleteIpNetEntry2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to was not set to a valid neighbor IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPNET_ROW2 structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **DeleteIpNetEntry2** function is used to delete a **MIB_IPNET_ROW2** structure entry.

On input, your driver must initialize the following members of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.

- **Address**
  Set to a valid neighbor IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **DeleteIpNetEntry2** deletes the neighbor IP address.

Your driver can call the **GetIpNetTable2** function to enumerate the neighbor IP address entries on a local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib

IRQL	< DISPATCH_LEVEL

See also

CreateIpNetEntry2

FlushIpNetTable2

GetIpNetEntry2

GetIpNetTable2

MIB_IPNET_ROW2

MIB_IPNET_TABLE2

ResolveIpNetEntry2

SetIpNetEntry2

FlushIpNetTable2 function

Article • 03/03/2023

The **FlushIpNetTable2** function flushes the IP neighbor table on a local computer.

Syntax

C++

```
NETIOAPI_API FlushIpNetTable2(
  _In_ ADDRESS_FAMILY Family,
  _In_ NET_IFINDEX    InterfaceIndex
);
```

Parameters

- *Family* [in]

  The address family to flush.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET

    The IPv4 address family. When this value is specified, this function flushes the neighbor IP address table that contains only IPv4 entries.

  - AF_INET6

    The IPv6 address family. When this value is specified, this function flushes the neighbor IP address table that contains only IPv6 entries.

  - AF_UNSPEC

    The address family is unspecified. When this value is specified, this function flushes the neighbor IP address table that contains both IPv4 and IPv6 entries.

- *InterfaceIndex* [in]
  The interface index. If the index is specified, the function flushes the neighbor IP address entries on a specific interface.Ootherwise, the function flushes the neighbor IP address entries on all the interfaces. To ignore the interface, set this parameter to zero.

Return value

FlushIpNetTable2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **FlushIpNetTable2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **FlushIpNetTable2** function flushes or deletes the neighbor IP addresses on a local computer. Your driver can use the *Family* parameter to limit neighbor IP addresses to delete to a particular IP address family. If neighbor IP addresses for both IPv4 and IPv6 should be deleted, your driver should set the *Family* parameter to AF_UNSPEC. Your driver can use the *InterfaceIndex* parameter to limit neighbor IP addresses to delete to a particular interface. If neighbor IP addresses for all interfaces should be deleted, your driver should set the *InterfaceIndex* parameter to zero.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpNetEntry2

DeleteIpNetEntry2

GetIpNetEntry2

GetIpNetTable2

MIB_IPNET_ROW2

MIB_IPNET_TABLE2

ResolveIpNetEntry2

SetIpNetEntry2

GetIpNetEntry2 function

Article • 03/03/2023

The **GetIpNetEntry2** function retrieves information for a neighbor IP address entry on the local computer.

Syntax

```c++
NETIOAPI_API GetIpNetEntry2(
  _Inout_ PMIB_IPNET_ROW2 Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IPNET_ROW2 structure entry for a neighbor IP address entry. On successful return, this structure is updated with the properties for neighbor IP address.

Return value

GetIpNetEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpNetEntry2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to was not set to a valid neighbor IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPNET_ROW2 structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIpNetEntry2** function is used to retrieve a **MIB_IPNET_ROW2** structure entry.

On input, your driver must initialize the following members of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.

- **Address**
  Set to a valid neighbor IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **GetIpNetEntry2** retrieves the other properties for the neighbor IP address and fills in the MIB_IPNET_ROW2 structure that the *Row* parameter points to.

Your driver can call the **GetIpNetTable2** function to enumerate the neighbor IP address entries on a local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetIpNetTable2

MIB_IPNET_ROW2

GetIpNetTable2 function

Article • 03/03/2023

The **GetIpNetTable2** function retrieves the IP neighbor table on a local computer.

Syntax

```c++
NETIOAPI_API GetIpNetTable2(
  _In_  ADDRESS_FAMILY    Family,
  _Out_ PMIB_IPNET_TABLE2 *Table
);
```

Parameters

- *Family* [in]

  The address family to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET

    The IPv4 address family. When this value is specified, this function returns the neighbor IP address table that contains only IPv4 entries.

  - AF_INET6

    The IPv6 address family. When this value is specified, this function returns the neighbor IP address table that contains only IPv6 entries.

  - AF_UNSPEC

    The address family is unspecified. When this value is specified, this function returns the neighbor IP address table that contains both IPv4 and IPv6 entries.

- *Table* [out]
  A pointer to a **MIB_IPNET_TABLE2** structure that contains a table of neighbor IP address entries on the local computer.

Return value

GetIpNetTable2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpNetTable2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter or the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No neighbor IP address entries, as specified in the *Family* parameter, were found.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIpNetTable2** function enumerates the neighbor IP addresses on a local computer and returns this information in a **MIB_IPNET_TABLE2** structure.

GetIpNetTable2 returns the neighbor IP address entries in a MIB_IPNET_TABLE2 structure in the buffer that the *Table* parameter points to. The MIB_IPNET_TABLE2 structure contains a neighbor IP address entry count and an array of **MIB_IPNET_ROW2** structures for each neighbor IP address entry. When these returned structures are no longer required, your driver should free the memory by calling **FreeMibTable**.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Note that the returned MIB_IPNET_TABLE2 structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_IPNET_ROW2 array entry in the **Table** member of the MIB_IPNET_TABLE2 structure. Padding for alignment might also be present between the MIB_IPNET_ROW2 array entries. Any access to a MIB_IPNET_ROW2 array entry should assume padding might exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpNetEntry2

FlushIpNetTable2

FreeMibTable

GetIpNetEntry2

MIB_IPNET_ROW2

MIB_IPNET_TABLE2

ResolveIpNetEntry2

SetIpNetEntry2

ResolveIpNetEntry2 function

Article • 03/03/2023

The **ResolveIpNetEntry2** function resolves the physical address for a neighbor IP address entry on a local computer.

Syntax

```c++
NETIOAPI_API ResolveIpNetEntry2(
  _Inout_        PMIB_IPNET_ROW2 Row,
  _In_opt_ const SOCKADDR_INET   *SourceAddress
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IPNET_ROW2 structure entry for a neighbor IP address entry. On successful return, this structure is updated with the properties for neighbor IP address.

- *SourceAddress* [in, optional]
  A pointer to an optional source IP address that is used to select the interface to send the requests on for the neighbor IP address entry.

Return value

ResolveIpNetEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **ResolveIpNetEntry2** returns one of the following error codes:

Return code	Description
STATUS_BAD_NETWORK_NAME	The network name cannot be found. This error is returned if the network with the neighbor IP address is unreachable.

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to was not set to a valid IPv4 or IPv6 address, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPNET_ROW2 structure were unspecified. This error is also returned if a loopback address was passed in the **Address** member.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **ResolveIpNetEntry2** function is used to resolve the physical address for a neighbor IP address entry on a local computer. This function flushes any existing neighbor entry that matches the IP address on the interface and then resolves the physical address (MAC) address by sending ARP requests for an IPv4 address or Neighbor Solicitation (NS) requests for an IPv6 address. If the *SourceAddress* parameter is specified, **ResolveIpNetEntry2** selects the interface with this source IP address to send the requests on. If the *SourceAddress* parameter is not specified (**NULL** was passed in this parameter), **ResolveIpNetEntry2** automatically selects the best interface to send the requests on.

Your driver must initialize the following members of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.

- **Address**
  Set to a valid IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

If the IP address that is passed in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to is a duplicate of an existing neighbor IP address on the interface, the **ResolveIpNetEntry2** function flushes the existing entry before resolving the IP address.

On output, when the call is successful, **ResolveIpNetEntry2** retrieves the other properties for the neighbor IP address and fills in the MIB_IPNET_ROW2 structure that the *Row* parameter points to. The **PhysicalAddress** and **PhysicalAddressLength** members in the MIB_IPNET_ROW2 structure are initialized to a valid physical address.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpNetEntry2

DeleteIpNetEntry2

FlushIpNetTable2

GetIpNetEntry2

GetIpNetTable2

MIB_IPNET_ROW2

MIB_IPNET_TABLE2

SetIpNetEntry2

SetIpNetEntry2 function

Article • 03/03/2023

The **SetIpNetEntry2** function sets the physical address of an existing neighbor IP address entry on a local computer.

Syntax

```
C++

NETIOAPI_API SetIpNetEntry2(
  _In_ PMIB_IPNET_ROW2 Row
);
```

Parameters

- *Row* [in]
  A pointer to a MIB_IPNET_ROW2 structure entry for a neighbor IP address entry.

Return value

SetIpNetEntry2 return STATUS_SUCCESS if the function succeeds.

If the function fails, **SetIpNetEntry2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned in the following situations. • A **NULL** pointer was passed in the *Row* parameter. • The **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to was not set to a valid unicast, anycast, or multicast IPv4 or IPv6 address. • The **PhysicalAddress** and **PhysicalAddressLength** members of the MIB_IPNET_ROW2 structure were not set to a valid physical address. • Both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPNET_ROW2 structure were unspecified. • A loopback address was passed in the **Address** member.

Return code	Description
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Your driver must initialize the following members of the **MIB_IPNET_ROW2** structure that the *Row* parameter points to.

- **Address**
  Set to a valid unicast, anycast, or multicast IPv4 or IPv6 address and family.

- **PhysicalAddress** and **PhysicalAddressLength**
  Set to a valid physical address.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

The **SetIpNetEntry2** function fails if the IP address that is passed in the **Address** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to is not an existing neighbor IP address on the interface that is specified.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpNetEntry2

DeleteIpNetEntry2

FlushIpNetTable2

GetIpNetEntry2

GetIpNetTable2

MIB_IPNET_ROW2

MIB_IPNET_TABLE2

ResolveIpNetEntry2

FlushIpPathTable function

Article • 03/03/2023

The **FlushIpPathTable** function flushes the IP path table on a local computer.

Syntax

```c++
NETIOAPI_API FlushIpPathTable(
  _In_ ADDRESS_FAMILY Family
);
```

Parameters

- *Family* [in]
  The address family to flush.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function flushes the IP path table that contains only IPv4 entries.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function flushes the IP path table that contains only IPv6 entries.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, this function flushes the neighbor IP address table that contains both IPv4 and IPv6 entries.

Return value

FlushIpPathTable returns STATUS_SUCCESS if the function succeeds.

If the function fails, **FlushIpPathTable** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **FlushIpPathTable** function flushes or deletes the IP path entries on a local computer. Your driver can use the *Family* parameter to limit the IP path entries to delete to a particular IP address family. If IP path entries for both IPv4 and IPv6 should be deleted, your driver should set the *Family* parameter to AF_UNSPEC.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetIpPathEntry

GetIpPathTable

MIB_IPPATH_ROW

MIB_IPPATH_TABLE

GetIpPathEntry function

Article • 03/03/2023

The **GetIpPathEntry** function retrieves information for an IP path entry on a local computer.

Syntax

```
C++

NETIOAPI_API GetIpPathEntry(
  _Inout_ PMIB_IPPATH_ROW Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IPPATH_ROW structure entry for an IP path entry. On successful return, this structure is updated with the properties for IP path entry.

Return value

GetIpPathEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpPathEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **si_family** member in the **Destination** member of the MIB_IPPATH_ROW structure that the *Row* parameter points to was not set to AF_INET or AF_INET6, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPPATH_ROW structure were unspecified. This error is also returned if the **si_family** member in the **Source** member of the MIB_IPPATH_ROW structure did not match the destination IP address family and the **si_family** for the source IP address was not specified as AF_UNSPEC.

Return code	Description
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPPATH_ROW structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Source** and **Destination** members of the MIB_IPPATH_ROW structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Source** and **Destination** members.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIpPathEntry** function is used to retrieve a **MIB_IPPATH_ROW** structure entry.

On input, your driver must initialize the following members of the MIB_IPPATH_ROW structure that the *Row* parameter points to.

- **Destination**
  Set to a valid IPv4 or IPv6 address and family.

- **Source**
  Set the address family that is specified in the **Source** member to the destination IP address family that is specified in the **Destination** member, or to AF_UNSPEC.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **GetIpPathEntry** retrieves the other properties for the IP path entry and fills in the MIB_IPPATH_ROW structure that the *Row* parameter points to.

Your driver can call the **GetIpPathTable** function to enumerate the IP path entries on a local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FlushIpPathTable

GetIpPathTable

MIB_IPPATH_ROW

MIB_IPPATH_TABLE

GetIpPathTable function

Article • 03/03/2023

The **GetIpPathEntry** function retrieves information for an IP path entry on a local computer.

Syntax

```C++
NETIOAPI_API GetIpPathTable(
  _In_  ADDRESS_FAMILY    Family,
  _Out_ PMIB_IPPATH_TABLE *Table
);
```

Parameters

- *Family* [in]
  The address family to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function returns the IP path table that contains only IPv4 entries.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function returns the IP path table that contains only IPv6 entries.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, this function

returns the IP path table that contains both IPv4 and IPv6 entries.

- *Table* [out]
  A pointer to a MIB_IPPATH_TABLE structure that contains a table of IP path entries on the local computer.

Return value

GetIpPathEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpPathEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter or the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No IP path entries, as specified in the *Family* parameter, were found.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIpPathTable** function enumerates the IP path entries on a local computer and returns this information in a MIB_IPPATH_TABLE structure.

GetIpPathTable returns the IP path entries in a MIB_IPPATH_TABLE structure in the buffer that the *Table* parameter points to. The MIB_IPPATH_TABLE structure contains an IP path entry count and an array of MIB_IPPATH_ROW structures for each IP path entry. When these returned structures are no longer required, your driver should free the memory by calling FreeMibTable.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Note that the returned MIB_IPPATH_TABLE structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_IPPATH_ROW array entry in the **Table** member of the MIB_IPPATH_TABLE structure. Padding for alignment might also be present between the MIB_IPPATH_ROW array entries. Any access to a MIB_IPPATH_ROW array entry should assume padding might exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

FreeMibTable

FlushIpPathTable

GetIpPathEntry

MIB_IPPATH_ROW

MIB_IPPATH_TABLE

CreateIpForwardEntry2 function

Article • 03/03/2023

The **CreateIpForwardEntry2** function creates a new IP route entry on a local computer.

Syntax

```
C++
```

```
NETIOAPI_API CreateIpForwardEntry2(
  _In_ const MIB_IPFORWARD_ROW2 *Row
);
```

Parameters

- *Row* [in]

  A pointer to a MIB_IPFORWARD_ROW2 structure entry for an IP route entry.

Return value

CreateIpForwardEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **CreateIpForwardEntry2** returns one of the following error codes:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if one of the following situations occurs: - A **NULL** pointer is passed in the *Row* parameter. - The **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to was not specified. - The **NextHop** member of the MIB_IPFORWARD_ROW2 structure was not specified. - Both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPFORWARD_ROW2 structure were unspecified. - The **PreferredLifetime** member of the MIB_IPFORWARD_ROW2 structure is greater than the **ValidLifetime** member. - The **SitePrefixLength** member of the MIB_IPFORWARD_ROW2 structure is greater than the prefix length that is specified by the **DestinationPrefix** member. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that is pointed to by the *Row* parameter was not specified, the **NextHop** member of the MIB_IPFORWARD_ROW2 structure was not specified, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPFORWARD_ROW2 structure were unspecified. This error is also returned if the **PreferredLifetime** member that is specified in the MIB_IPFORWARD_ROW2 structure is greater than the **ValidLifetime** member, or if the **SitePrefixLength** in the MIB_IPFORWARD_ROW2 structure is greater than the prefix length that is specified in the **DestinationPrefix** member.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPNET_ROW2 structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if the interface that is specified does not support routes. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the address family in the **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and AF_INET6 was specified for the address family in the **DestinationPrefix** member.
ERROR_OBJECT_ALREADY_EXISTS	The object already exists. This error is returned if the **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to is a duplicate of an existing IP route entry on the interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPFORWARD_ROW2 structure.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **CreateIpForwardEntry2** function is used to add a new neighbor IP address entry on a local computer. Use the **InitializeIpForwardEntry** function to initialize the members of a **MIB_IPFORWARD_ROW2** structure entry with default values. A driver can then change the members in the MIB_IPFORWARD_ROW2 entry that it wants to modify and then call **CreateIpForwardEntry2**.

Your driver must initialize the following members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to:

- Set **DestinationPrefix** to a valid IPv4 or IPv6 address prefix.
- Set **NextHop** to a valid IPv4 or IPv6 address and family.
- Set **InterfaceLuid** or **InterfaceIndex** to the LUID or index value of the interface.

The **InterfaceLuid** and **InterfaceIndex** members are used in the order that is listed earlier. So if the **InterfaceLuid** is specified, this member is used to determine the interface to add the IP route entry on. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

The route metric offset that is specified in the **Metric** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to represents only part of the complete route metric. The complete metric is a combination of this route metric offset added to the interface metric that is specified in the **Metric** member of the MIB_IPINTERFACE_ROW structure of the associated interface. A driver can retrieve the interface metric by calling the GetIpInterfaceEntry function.

The **Age** and **Origin** members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to are ignored when the **CreateIpForwardEntry2** function is called. These members are set by the network stack and cannot be set by using the **CreateIpForwardEntry2** function.

The **CreateIpForwardEntry2** function fails if the **DestinationPrefix** and **NextHop** members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to are a duplicate of an existing IP route entry on the interface that is specified in the **InterfaceLuid** or **InterfaceIndex** members.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

DeleteIpForwardEntry2

GetBestRoute2

GetIpForwardEntry2

GetIpForwardTable2

GetIpInterfaceEntry

InitializeIpForwardEntry

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

MIB_IPINTERFACE_ROW

NotifyRouteChange2

SetIpForwardEntry2

DeleteIpForwardEntry2 function

Article • 03/03/2023

The **DeleteIpForwardEntry2** function deletes an IP route entry on a local computer.

Syntax

```
C++

NETIOAPI_API DeleteIpForwardEntry2(
  _In_ const MIB_IPFORWARD_ROW2 *Row
);
```

Parameters

- *Row* [in]

  A pointer to a MIB_IPFORWARD_ROW2 structure entry for an IP route entry. On successful return, this entry is deleted.

Return value

DeleteIpForwardEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **DeleteIpForwardEntry2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to was not specified, the **NextHop** member of the MIB_IPFORWARD_ROW2 structure was not specified, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPFORWARD_ROW2 structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to.

Return code	Description
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address was specified in the **Address** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and an IPv6 address was specified in the **Address** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **DeleteIpForwardEntry2** function is used to delete a **MIB_IPFORWARD_ROW2** structure entry.

On input, your driver must initialize the following members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to.

- **DestinationPrefix**
  Set to a valid IPv4 or IPv6 address prefix and family.

- **NextHop**
  Set to a valid IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **DeleteIpForwardEntry2** deletes the IP route entry.

The **DeleteIpForwardEntry2** function fails if the **DestinationPrefix** and **NextHop** members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to do not match an existing IP route entry on the interface that is specified in the **InterfaceLuid** or **InterfaceIndex** members.

Your driver can call the **GetIpForwardTable2** function to enumerate the IP route entries on a local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpForwardEntry2

GetBestRoute2

GetIpForwardEntry2

GetIpForwardTable2

InitializeIpForwardEntry

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

NotifyRouteChange2

SetIpForwardEntry2

GetBestRoute2 function

Article • 03/03/2023

The **GetBestRoute2** function retrieves the IP route entry on a local computer for the best route to the specified destination IP address.

Syntax

C++

```
NETIOAPI_API GetBestRoute2(
  _In_opt_       NET_LUID            *InterfaceLuid,
  _In_           NET_IFINDEX         InterfaceIndex,
  _In_opt_ const SOCKADDR_INET       *SourceAddress,
  _In_     const SOCKADDR_INET       *DestinationAddress,
  _In_           ULONG               AddressSortOptions,
  _Out_          PMIB_IPFORWARD_ROW2 BestRoute,
  _Out_          SOCKADDR_INET       *BestSourceAddress
);
```

Parameters

- *InterfaceLuid* [in, optional]
  The locally unique identifier (LUID) to specify the network interface that is associated with an IP route entry.

- *InterfaceIndex* [in]
  The local index value to specify the network interface that is associated with an IP route entry. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, so this value does not persistent.

- *SourceAddress* [in, optional]
  The source IP address. Your driver can omit this parameter and pass a **NULL** pointer.

- *DestinationAddress* [in]
  The destination IP address.

- *AddressSortOptions* [in]
  A set of options that affect how IP addresses are sorted. This parameter is currently not used.

- *BestRoute* [out]
  A pointer to the MIB_IPFORWARD_ROW2 structure for the best route from the source IP address to the destination IP address.

- *BestSourceAddress* [out]
  A pointer to the best source IP address.

Return value

GetBestRoute2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetBestRoute2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *DestinationAddress*, *BestSourceAddress*, or *BestRoute* parameters. This error is also returned if both *InterfaceLuid* and *InterfaceIndex* parameters were unspecified. This error is also returned if the *DestinationAddress* parameter does not specify an IPv4 or IPv6 address and family
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the network interface that the *InterfaceLuid* or *InterfaceIndex* parameter specifies could not be found.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and an IPv4 address and family was specified in the *DestinationAddress* parameter, or if no IPv6 stack is located on the local computer and an IPv4 address and family was specified in the *DestinationAddress* parameter.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetBestRoute2** function is used to retrieve a MIB_IPFORWARD_ROW2 structure entry for the best route from a source IP address to a destination IP address.

On input, your driver must initialize the following parameters.

- *DestinationAddress*
  Set to a valid IPv4 or IPv6 address and family.

- *InterfaceLuid* or *InterfaceIndex*
  These parameters are used in the order that is listed earlier. So if *InterfaceLuid* is specified, this parameter is used to determine the interface. If no value was set for the *InterfaceLuid* member (the value of this parameter was set to zero), the *InterfaceIndex* parameter is next used to determine the interface.

In addition, on input, your driver can initialize the *SourceAddress* parameter to the preferred IPv4 or IPv6 address and family.

On output, when the call is successful, **GetBestRoute2** retrieves an MIB_IPFORWARD_ROW2 structure for the best route from the source IP address the destination IP address.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpForwardEntry2

DeleteIpForwardEntry2

GetIpForwardEntry2

GetIpForwardTable2

InitializeIpForwardEntry

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

NotifyRouteChange2

SetIpForwardEntry2

GetIpForwardEntry2 function

Article • 03/03/2023

The **GetIpForwardEntry2** function retrieves information for an IP route entry on a local computer.

Syntax

```
C++
```

```
NETIOAPI_API GetIpForwardEntry2(
  _Inout_ PMIB_IPFORWARD_ROW2 Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IPFORWARD_ROW2 structure entry for an IP route entry. On successful return, this structure is updated with the properties for the IP route entry.

Return value

GetIpForwardEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpForwardEntry2** returns one of the following error codes:

Return code	Description

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if one of the following situations occurs: - A **NULL** pointer is passed in the *Row* parameter. - The **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to was not specified. - The **NextHop** member of the MIB_IPFORWARD_ROW2 structure was not specified. - Both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPFORWARD_ROW2 structure were unspecified. - The **PreferredLifetime** member of the MIB_IPFORWARD_ROW2 structure is greater than the **ValidLifetime** member. - The **SitePrefixLength** member of the MIB_IPFORWARD_ROW2 structure is greater than the prefix length that is specified by the **DestinationPrefix** member.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the address family in the **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to, or if no IPv6 stack is located on the local computer and AF_INET6 was specified for the address family in the **DestinationPrefix** member.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIpForwardEntry2** function is used to retrieve a MIB_IPFORWARD_ROW2 structure entry.

On input, your driver must initialize the following members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to.

- **DestinationPrefix**
  Set to a valid IPv4 or IPv6 address prefix and family.

- **NextHop**
  Set to a valid IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, when the call is successful, **GetIpForwardEntry2** retrieves the other properties for the IP route entry and fills out the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to.

The route metric offset that is specified in the **Metric** member of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to represents only part of the complete route metric. The complete metric is a combination of this route metric added to the interface metric that is specified in the **Metric** member of the MIB_IPINTERFACE_ROW structure of the associated interface. A driver can retrieve the interface metric by calling the GetIpInterfaceEntry function.

Your driver can call the GetIpForwardTable2 function to enumerate the IP route entries on a local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpForwardEntry2

DeleteIpForwardEntry2

GetBestRoute2

GetIpForwardTable2

GetIpInterfaceEntry

InitializeIpForwardEntry

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

MIB_IPINTERFACE_ROW

NotifyRouteChange2

SetIpForwardEntry2

GetIpForwardTable2 function

Article • 03/03/2023

The **GetIpForwardTable2** function retrieves the IP route entries on a local computer.

Syntax

```
C++

NETIOAPI_API GetIpForwardTable2(
  _In_  ADDRESS_FAMILY        Family,
  _Out_ PMIB_IPFORWARD_TABLE2 *Table
);
```

Parameters

- *Family* [in]
  The address family to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function returns the IP routing table that contains only IPv4 entries.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function returns the IP routing table that contains only IPv6 entries.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, this function returns the IP routing table that contains both IPv4 and IPv6 entries.

- *Table* [out]

  A pointer to a MIB_IPFORWARD_TABLE2 structure that contains a table of IP route entries on the local computer.

Return value

GetIpForwardTable2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpForwardTable2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Table* parameter or the *Family* parameter was not specified as AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
STATUS_NOT_FOUND	No IP route entries, as specified in the *Family* parameter, were found.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv4 stack is located on the local computer and AF_INET was specified in the *Family* parameter, or if no IPv6 stack is located on the local computer and AF_INET6 was specified in the *Family* parameter. This error is also returned on versions of Windows where this function is not supported.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetIpForwardTable2** function enumerates the IP route entries on a local computer and returns this information in a MIB_IPFORWARD_TABLE2 structure.

The IP route entries are returned in a MIB_IPFORWARD_TABLE2 structure in the buffer that the *Table* parameter points to. The MIB_IPFORWARD_TABLE2 structure contains an IP route entry count and an array of MIB_IPFORWARD_ROW2 structures for each IP route entry. When these returned structures are no longer required, your driver should free the memory by calling FreeMibTable.

Your driver must initialize the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

Note that the returned MIB_IPFORWARD_TABLE2 structure that the *Table* parameter points to might contain padding for alignment between the **NumEntries** member and the first MIB_IPFORWARD_ROW2 array entry in the **Table** member of the MIB_IPFORWARD_TABLE2 structure. Padding for alignment might also be present between the MIB_IPFORWARD_ROW2 array entries. Any access to a MIB_IPFORWARD_ROW2 array entry should assume padding might exist.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpForwardEntry2

DeleteIpForwardEntry2

FreeMibTable

GetBestRoute2

GetIpForwardEntry2

InitializeIpForwardEntry

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

NotifyRouteChange2

SetIpForwardEntry2

GetIpInterfaceEntry function

Article • 03/03/2023

The **GetIpInterfaceEntry** function retrieves IP information for the specified interface on a local computer.

Syntax

```c++
NETIOAPI_API GetIpInterfaceEntry(
  _Inout_ PMIB_IPINTERFACE_ROW Row
);
```

Parameters

- *Row* [in, out]
  A pointer to a MIB_IPINTERFACE_ROW structure that, on successful return, receives information for an interface on the local computer. On input, your driver must set the **InterfaceLuid** member or the **InterfaceIndex** member of the MIB_IPINTERFACE_ROW to the interface to retrieve information for.

Return value

GetIpInterfaceEntry returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetIpInterfaceEntry** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Row* parameter, the **Family** member of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to was not specified as AF_INET or AF_INET6, or the **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPINTERFACE_ROW structure were unspecified.
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to.

Return code	Description
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

On input, your driver must initialize the following members of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to.

- **Family**
  Set to either AF_INET or AF_INET6.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

On output, **GetIpInterfaceEntry** fills in the remaining members of the MIB_IPINTERFACE_ROW structure that the *Row* parameter points to.

Your driver must use the InitializeIpInterfaceEntry function to initialize the fields of a MIB_IPINTERFACE_ROW structure entry with default values. A driver can then change the fields in the MIB_IPINTERFACE_ROW entry that it wants to modify, and then call the SetIpInterfaceEntry function.

Unprivileged simultaneous access to multiple networks of different security requirements creates a security hole and enables an unprivileged driver to accidentally relay data between the two networks. A typical example is simultaneous access to a virtual private network (VPN) and the Internet. The Windows Server 2003 and Windows XP operating systems use a weak host model, where Remote Access Service (RAS) prevents such simultaneous access by increasing the route metric of all default routes over other interfaces. Therefore, all traffic is routed through the VPN interface, disrupting other network connectivity.

On Windows Vista and later versions of the Windows operating systems, by default, a strong host model is used. If a source IP address is specified in the route lookup by using the GetBestRoute2 function, the route lookup is restricted to the interface of the source IP address. The route metric modification by RAS has no effect because the list of potential routes does not even have the route for the VPN interface, which enables traffic to the Internet. Your driver can use the **DisableDefaultRoutes** member of the MIB_IPINTERFACE_ROW to disable using the default route on an interface. VPN clients can use this member as a security measure to restrict split tunneling when split tunneling is not required by the VPN client. A VPN client can call the SetIpInterfaceEntry function to set the **DisableDefaultRoutes** member to **TRUE** when it is required. A VPN client can query the current state of the **DisableDefaultRoutes** member by calling the **GetIpInterfaceEntry** function.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetBestRoute2

GetIfEntry2

GetIfTable2

GetIfTable2Ex

GetIpInterfaceTable

MIB_IF_ROW2

MIB_IF_TABLE2

MIB_IPINTERFACE_ROW

MIB_IPINTERFACE_TABLE

SetIpInterfaceEntry

InitializeIpForwardEntry function

Article • 03/03/2023

The **InitializeIpForwardEntry** function initializes a MIB_IPFORWARD_ROW2 structure with default values for an IP route entry on a local computer.

Syntax

```c++
VOID NETIOAPI_API_ InitializeIpForwardEntry(
  _Out_ PMIB_IPFORWARD_ROW2 Row
);
```

Parameters

- *Row* [out]
  On entry, a pointer to a MIB_IPFORWARD_ROW2 structure entry for an IP route entry.

  On return, the MIB_IPFORWARD_ROW2 structure that this parameter points to is initialized with default values for an IP route entry.

Return value

None

Remarks

Your driver must use the **InitializeIpForwardEntry** function to initialize the members of a MIB_IPFORWARD_ROW2 structure entry with default values for an IP route entry for later use with the CreateIpForwardEntry2 function.

On input, your driver must pass **InitializeIpForwardEntry** a new MIB_IPFORWARD_ROW2 structure to initialize.

On output, the members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to are initialized as follows.

- **ValidLifetime** and **PreferredLifetime**
  Set to an infinite value,

- **Loopback**, **AutoconfigureAddress**, **Publish**, and **Immortal**
  Set to **TRUE**.

- **SitePrefixLength**, **Metric**, and **Protocol**
  Set to illegal values.

- All other members
  Set to zero.

After a driver calls **InitializeIpForwardEntry**, the driver can then change the members in the MIB_IPFORWARD_ROW2 entry that it wants to modify, and then call the **CreateIpForwardEntry2** to add the new IP route entry to the local computer.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpForwardEntry2

DeleteIpForwardEntry2

GetBestRoute2

GetIpForwardEntry2

GetIpForwardTable2

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

NotifyRouteChange2

SetIpForwardEntry2

SetIpForwardEntry2 function

Article • 03/03/2023

The **SetIpForwardEntry2** function sets the properties of an IP route entry on a local computer.

Syntax

```
C++

NETIOAPI_API SetIpForwardEntry2(
  _In_ const MIB_IPFORWARD_ROW2 *Route
);
```

Parameters

- *Route* [in]
  A pointer to a MIB_IPFORWARD_ROW2 structure entry for an IP route entry. Your driver must set the **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure to a valid IP destination prefix and family, set the **NextHop** member of MIB_IPFORWARD_ROW2 to a valid IP address and family, and specify the **InterfaceLuid** member or the **InterfaceIndex** member of MIB_IPFORWARD_ROW2.

Return value

SetIpForwardEntry2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **SetIpForwardEntry2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Route* parameter, the **DestinationPrefix** member of the MIB_IPFORWARD_ROW2 structure that the *Route* parameter points to was not specified, the **NextHop** member of the MIB_IPFORWARD_ROW2 structure was not specified, or both **InterfaceLuid** and **InterfaceIndex** members of the MIB_IPFORWARD_ROW2 structure were unspecified.

Return code	Description
STATUS_NOT_FOUND	The specified interface could not be found. This error is returned if the function cannot find the network interface that is specified by the **InterfaceLuid** or **InterfaceIndex** member of the MIB_IPFORWARD_ROW2 structure that the *Route* parameter points to.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **SetIpForwardEntry2** function is used to set the properties for an existing IP route entry on a local computer.

Your driver must initialize the following members of the **MIB_IPFORWARD_ROW2** structure that the *Row* parameter points to.

- **DestinationPrefix**
  Set to a valid IPv4 or IPv6 address prefix and family.

- **NextHop**
  Set to a valid IPv4 or IPv6 address and family.

- **InterfaceLuid** or **InterfaceIndex**
  These members are used in the order that is listed earlier. So if **InterfaceLuid** is specified, this member is used to determine the interface. If no value was set for the **InterfaceLuid** member (the value of this member was set to zero), the **InterfaceIndex** member is next used to determine the interface.

The route metric offset that is specified in the **Metric** member of the MIB_IPFORWARD_ROW2 structure that *Route* parameter points to represents only part of the complete route metric. The complete metric is a combination of this route metric offset added to the interface metric that is specified in the **Metric** member of the **MIB_IPINTERFACE_ROW** structure of the associated interface. A driver can retrieve the interface metric by calling the **GetIpInterfaceEntry** function.

SetIpForwardEntry2 ignores the **Age** and **Origin** members of the MIB_IPFORWARD_ROW2 structure that the *Row* parameter points to. These members are set by the network stack and cannot be changed by using the **SetIpForwardEntry2** function.

The **SetIpForwardEntry2** function fails if the **DestinationPrefix** and **NextHop** members of the MIB_IPFORWARD_ROW2 structure that the *Route* parameter points to do not match an an IP route entry on the specified interface.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CreateIpForwardEntry2

DeleteIpForwardEntry2

GetBestRoute2

GetIpForwardEntry2

GetIpForwardTable2

GetIpInterfaceEntry

InitializeIpForwardEntry

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

MIB_IPINTERFACE_ROW

NotifyRouteChange2

FreeMibTable function

Article • 03/03/2023

The **FreeMibTable** function frees the buffer that is allocated by the functions that return tables of network interfaces, addresses, and routes (for example, **GetIfTable2** and **GetAnycastIpAddressTable**).

Syntax

C++

```
VOID NETIOAPI_API_ FreeMibTable(
  _In_ PVOID Memory
);
```

Parameters

- *Memory* [in]

  A pointer to the buffer to free.

Return value

None

Remarks

The **FreeMibTable** function is used to free the internal buffers that various functions use to retrieve tables of interfaces, addresses, and routes. When these tables are no longer needed, your driver should call **FreeMibTable** to release the memory that these tables use.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

GetAnycastIpAddressTable

GetIfStackTable

GetIfTable2

GetIfTable2Ex

GetInvertedIfStackTable

GetIpForwardTable2

GetIpInterfaceTable

GetIpNetTable2

GetIpPathTable

GetMulticastIpAddressTable

GetUnicastIpAddressTable

MIB_NOTIFICATION_TYPE enumeration

Article • 03/03/2023

The MIB_NOTIFICATION_TYPE enumeration type defines the notification type that is passed to a callback function when a notification occurs.

Syntax

```C++
typedef enum _MIB_NOTIFICATION_TYPE {
  MibParameterNotification  = 0,
  MibAddInstance            = 1,
  MibDeleteInstance         = 2,
  MibInitialNotification    = 3
} MIB_NOTIFICATION_TYPE, *PMIB_NOTIFICATION_TYPE;
```

Constants

- **MibParameterNotification**
  A parameter was changed.

- **MibAddInstance**
  A new MIB instance was added.

- **MibDeleteInstance**
  An existing MIB instance was deleted.

- **MibInitialNotification**
  A notification that is invoked immediately after registration for change notification completes. This initial notification does not indicate that a change occurred to a MIB instance. The purpose of this initial notification type is to provide confirmation that the callback function is properly registered.

Remarks

The MIB_NOTIFICATION_TYPE enumerated type is used with the callback function that is specified in the *Callback* parameter of one of the IP Helper **Notify*Xxx*** functions to specify the notification type.

On Windows Vista and later versions of the Windows operating systems, new functions are provided to register the driver to be notified when an IPv6 or IPv4 interface changes, an IPv6 or IPv4 unicast address changes, or an IPv6 or IPv4 route changes. These registration functions require that a callback function be passed that is called when a change occurs. One of the parameters that is passed to the callback function when a notification occurs is a parameter that contains a MIB_NOTIFICATION_TYPE value that indicates the notification type.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

NotifyIpInterfaceChange

NotifyRouteChange2

NotifyStableUnicastIpAddressTable

NotifyTeredoPortChange

NotifyUnicastIpAddressChange

IP_ADDRESS_PREFIX structure

Article • 03/03/2023

The IP_ADDRESS_PREFIX structure stores an IP address prefix.

Syntax

```c++
typedef struct _IP_ADDRESS_PREFIX {
  SOCKADDR_INET Prefix;
  UINT8         PrefixLength;
} IP_ADDRESS_PREFIX, *PIP_ADDRESS_PREFIX;
```

Members

- **Prefix**
  The prefix or network part of the address represented as an IP address.

- **PrefixLength**
  The length, in bits, of the prefix or network part of the IP address. For a unicast IPv4 address, any value that is greater than 32 is an illegal value. For a unicast IPv6 address, any value that is greater than 128 is an illegal value. A value of 255 is typically used to represent an illegal value.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

MIB_ANYCASTIPADDRESS_ROW structure

Article • 03/03/2023

The MIB_ANYCASTIPADDRESS_ROW structure stores information about an anycast IP address.

Syntax

```c++
typedef struct _MIB_ANYCASTIPADDRESS_ROW {
  SOCKADDR_INET Address;
  NET_LUID      InterfaceLuid;
  NET_IFINDEX   InterfaceIndex;
  SCOPE_ID      ScopeId;
} MIB_ANYCASTIPADDRESS_ROW, *PMIB_ANYCASTIPADDRESS_ROW;
```

Members

- **Address**
  The anycast IP address. This member can be an IPv6 address or an IPv4 address.

- **InterfaceLuid**
  The locally unique identifier (LUID) for the network interface that is associated with this IP address.

- **InterfaceIndex**
  The local index value for the network interface that is associated with this IP address. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

- **ScopeId**
  The scope ID of the anycast IP address. This member is applicable only to an IPv6 address. Your driver cannot set this member. This member is automatically determined by the interface that the address was added on.

Remarks

The GetAnycastIpAddressTable function enumerates the anycast IP addresses on a local computer and returns this information in a MIB_ANYCASTIPADDRESS_TABLE structure.

The MIB_ANYCASTIPADDRESS_TABLE structure might contain padding for alignment between the **NumEntries** member and the first MIB_ANYCASTIPADDRESS_ROW array entry in the **Table** member. Padding for alignment might also be present between the MIB_ANYCASTIPADDRESS_ROW array entries in the **Table** member. Any access to a MIB_ANYCASTIPADDRESS_ROW array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

CreateAnycastIpAddressEntry

DeleteAnycastIpAddressEntry

GetAnycastIpAddressTable

GetAnycastIpAddressEntry

MIB_ANYCASTIPADDRESS_TABLE

MIB_ANYCASTIPADDRESS_TABLE structure

Article • 03/03/2023

The MIB_ANYCASTIPADDRESS_TABLE structure contains a table of anycast IP address entries.

Syntax

```
C++

typedef struct _MIB_ANYCASTIPADDRESS_TABLE {
  ULONG                       NumEntries;
  MIB_ANYCASTIPADDRESS_ROW Table[ANY_SIZE];
} MIB_ANYCASTIPADDRESS_TABLE, *PMIB_ANYCASTIPADDRESS_TABLE;
```

Members

- **NumEntries**
  A value that specifies the number of anycast IP address entries in the array.

- **Table**
  An array of MIB_ANYCASTIPADDRESS_ROW structures that contain anycast IP address entries.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetAnycastIpAddressTable

MIB_ANYCASTIPADDRESS_ROW

MIB_IF_ROW2 structure

Article • 03/03/2023

The MIB_IF_ROW2 structure stores information about a particular interface.

Syntax

```c++
typedef struct _MIB_IF_ROW2 {
  NET_LUID                   InterfaceLuid;
  NET_IFINDEX                InterfaceIndex;
  GUID                       InterfaceGuid;
  WCHAR                      Alias[IF_MAX_STRING_SIZE + 1];
  WCHAR                      Description[IF_MAX_STRING_SIZE + 1];
  ULONG                      PhysicalAddressLength;
  UCHAR                      PhysicalAddress[IF_MAX_PHYS_ADDRESS_LENGTH];
  UCHAR
                     PermanentPhysicalAddress[IF_MAX_PHYS_ADDRESS_LENGTH];
  ULONG                      Mtu;
  IFTYPE                     Type;
  TUNNEL_TYPE                TunnelType;
  NDIS_MEDIUM                MediaType;
  NDIS_PHYSICAL_MEDIUM       PhysicalMediumType;
  NET_IF_ACCESS_TYPE         AccessType;
  NET_IF_DIRECTION_TYPE      DirectionType;
  struct {
    BOOLEAN HardwareInterface  :1;
    BOOLEAN FilterInterface  :1;
    BOOLEAN ConnectorPresent  :1;
    BOOLEAN NotAuthenticated  :1;
    BOOLEAN NotMediaConnected  :1;
    BOOLEAN Paused  :1;
    BOOLEAN LowPower  :1;
    BOOLEAN EndPointInterface  :1;
  } InterfaceAndOperStatusFlags;
  IF_OPER_STATUS             OperStatus;
  NET_IF_ADMIN_STATUS        AdminStatus;
  NET_IF_MEDIA_CONNECT_STATE MediaConnectState;
  NET_IF_NETWORK_GUID        NetworkGuid;
  NET_IF_CONNECTION_TYPE     ConnectionType;
  ULONG64                    TransmitLinkSpeed;
  ULONG64                    ReceiveLinkSpeed;
  ULONG64                    InOctets;
  ULONG64                    InUcastPkts;
  ULONG64                    InNUcastPkts;
  ULONG64                    InDiscards;
  ULONG64                    InErrors;
  ULONG64                    InUnknownProtos;
  ULONG64                    InUcastOctets;
  ULONG64                    InMulticastOctets;
  ULONG64                    InBroadcastOctets;
  ULONG64                    OutOctets;
  ULONG64                    OutUcastPkts;
  ULONG64                    OutNUcastPkts;
  ULONG64                    OutDiscards;
  ULONG64                    OutErrors;
  ULONG64                    OutUcastOctets;
  ULONG64                    OutMulticastOctets;
  ULONG64                    OutBroadcastOctets;
  ULONG64                    OutQLen;
} MIB_IF_ROW2, *PMIB_IF_ROW2;
```

Members

- **InterfaceLuid**
  The locally unique identifier (LUID) for the network interface.

- **InterfaceIndex**
  The index that identifies the network interface. This index value might change when a network adapter is disabled and then enabled, and should not be considered persistent.

- **InterfaceGuid**
  The GUID for the network interface.

- **Alias**
  A NULL-terminated Unicode string that contains the alias name of the network interface.

- **Description**
  A NULL-terminated Unicode string that contains a description of the network interface.

- **PhysicalAddressLength**
  The length, in bytes, of the physical hardware address that the PhysicalAddress member specifies.

- **PhysicalAddress**
  The physical hardware address of the adapter for this network interface.

- **PermanentPhysicalAddress**
  The permanent physical hardware address of the adapter for this network interface.

- **Mtu**
  The maximum transmission unit (MTU) size, in bytes, for this network interface.

- **Type**
  The interface type as defined by the Internet Assigned Names Authority (IANA). For more information, see IANAifType-MIB DEFINITIONS. Possible values for the interface type are listed in the Ipifcons.h header file.

  The following table lists common values for the interface type, although many other values are possible.

Value	Meaning
IF_TYPE_OTHER 1	Some other type of network interface
IF_TYPE_ETHERNET_CSMACD 6	An Ethernet network interface
IF_TYPE_ISO88025_TOKENRING 9	A token ring network interface
IF_TYPE_PPP 23	A PPP network interface
IF_TYPE_SOFTWARE_LOOPBACK 24	A software loopback network interface
IF_TYPE_ATM 37	An ATM network interface
IF_TYPE_IEEE80211 71	An IEEE 802.11 wireless network interface
IF_TYPE_TUNNEL 131	A tunnel type encapsulation network interface
IF_TYPE_IEEE1394 144	An IEEE 1394 (Firewire) high performance serial bus network interface

- **TunnelType**
  If the Type member is IF_TYPE_TUNNEL, a **TUNNEL_TYPE** type that defines the

encapsulation method that a tunnel uses.

- **MediaType**
  The NDIS media type for the interface. This member can be one of the following values from the NDIS_MEDIUM enumeration type that is defined in the Ntddndis.h header file.

Value	Meaning
NdisMedium802_3 0	An Ethernet (802.3) network.
NdisMedium802_5 1	A Token Ring (802.5) network.
NdisMediumFddi 2	A Fiber Distributed Data Interface (FDDI) network.
NdisMediumWan 3	A wide area network (WAN). This type covers various forms of point-to-point and WAN NICs and variant address/header formats that must be negotiated between the protocol driver and the underlying driver after the binding is established.
NdisMediumLocalTalk 4	A LocalTalk network.
NdisMediumDix 5	An Ethernet network for which the drivers use the DIX Ethernet header format.
NdisMediumArcnetRaw 6	An ARCNET network.
NdisMediumArcnet878_2 7	An ARCNET (878.2) network.
NdisMediumAtm 8	An ATM network. Connection-oriented client protocol drivers can bind themselves to an underlying miniport driver that returns this value. Otherwise, legacy protocol drivers bind themselves to the system-supplied LanE intermediate driver, which reports its medium type as either **NdisMedium802_3** or NdisMedium802_5, depending on how the network administrator configures the LanE driver.

Value	Meaning
NdisMediumWirelessWan 9	A wireless network. NDIS 5. x miniport drivers that support wireless LAN (WLAN) or wireless WAN (WWAN) packets declare their medium as **NdisMedium802_3** and emulate Ethernet to higher-level NDIS drivers. Note This media type is not available for use on Windows Vista or later versions of Windows.
NdisMediumIrda 10	An infrared (IrDA) network.
NdisMediumBpc 11	A broadcast computer network.
NdisMediumCoWan 12	A wide area network in a connection-oriented environment.
NdisMedium1394 13	An IEEE 1394 (fire wire) network.
NdisMediumInfiniBand 14	An InfiniBand network.
NdisMediumTunnel 15	A tunnel network.
NdisMediumNative802_11 16	A native IEEE 802.11 network.
NdisMediumLoopback 17	An NDIS loopback network.

- **PhysicalMediumType**
  The NDIS physical medium type. This member can be one of the following values from the NDIS_PHYSICAL_MEDIUM enumeration type that is defined in the Ntddndis.h header file.

Value	Meaning

Value	Meaning
NdisPhysicalMediumUnspecified 0	The physical medium is none of the following values. For example, a one-way satellite feed is an unspecified physical medium.
NdisPhysicalMediumWirelessLan 1	Packets are transferred over a wireless LAN network through a miniport driver that complies with the 802.11 interface.
NdisPhysicalMediumCableModem 2	Packets are transferred over a DOCSIS-based cable network.
NdisPhysicalMediumPhoneLine 3	Packets are transferred over standard telephone lines. This type includes HomePNA media.
NdisPhysicalMediumPowerLine 4	Packets are transferred over wiring that is connected to a power distribution system.
NdisPhysicalMediumDSL 5	Packets are transferred over a Digital Subscriber Line (DSL) network. This type includes ADSL, UADSL (G.Lite), and SDSL.
NdisPhysicalMediumFibreChannel 6	Packets are transferred over a Fibre Channel interconnect.
NdisPhysicalMedium1394 7	Packets are transferred over an IEEE 1394 bus.
NdisPhysicalMediumWirelessWan 8	Packets are transferred over a Wireless WAN link. This type includes CDPD, CDMA and GPRS.
NdisPhysicalMediumNative802_11 9	Packets are transferred over a wireless LAN network through a miniport driver that complies with the Native 802.11 interface. Note The Native 802.11 interface is supported in NDIS 6.0 and later versions.
NdisPhysicalMediumBluetooth 10	Packets are transferred over a Bluetooth network. Bluetooth is a short-range wireless technology that uses the 2.4 GHz spectrum.
NdisPhysicalMediumInfiniband 11	Packets are transferred over an InfiniBand interconnect.

Value	Meaning
NdisPhysicalMediumUWB 13	Packets are transferred over an ultra wide band network.
NdisPhysicalMedium802_3 14	Packets are transferred over an Ethernet (802.3) network.
NdisPhysicalMedium802_5 15	Packets are transferred over a Token Ring (802.5) network.
NdisPhysicalMediumIrda 16	Packets are transferred over an infrared (IrDA) network.
NdisPhysicalMediumWiredWAN 17	Packets are transferred over a wired WAN network.
NdisPhysicalMediumWiredCoWan 18	Packets are transferred over a wide area network in a connection-oriented environment.
NdisPhysicalMediumOther 19	Packets are transferred over a network that is not described by other possible values.

- AccessType
  A NET_IF_ACCESS_TYPE NDIS network interface access type.

- DirectionType
  A NET_IF_DIRECTION_TYPE NDIS network interface direction type.

- InterfaceAndOperStatusFlags
  A set of the following flags that provide information about the interface. These flags are combined with a bitwise OR operation. If none of the flags applies, this member is set to zero.

  - HardwareInterface
    The network interface is for hardware.

  - FilterInterface
    The network interface is for a filter module.

  - ConnectorPresent
    A connector is present on the network interface. This value is set if there is a physical network adapter.

  - NotAuthenticated
    The default port for the network interface is not authenticated. If a network interface is not authenticated by the target, the network interface is not in an operational mode. Although this situation applies to both wired and wireless network connections, authentication is more common for wireless network connections.

  - NotMediaConnected
    The network interface is not in a media-connected state. If a network cable is unplugged for a wired network, this value is set. For a wireless network, this value is set for the network adapter that is not connected to a network.

  - Paused
    The network stack for the network interface is in the paused or pausing state. This value does not mean that the computer is in a hibernated state.

  - LowPower
    The network interface is in a low power state.

  - EndPointInterface
    The network interface is an endpoint device and not a true network interface that connects to a network. This value can be set by devices, such as smartphones, that use networking infrastructure to communicate to the computer but do not provide connectivity to an external network. These types of devices must set this flag.

- OperStatus
  A IF_OPER_STATUS NDIS network interface operational status type.

- AdminStatus
  The NET_IF_ADMIN_STATUS administrative status type.

- MediaConnectState
  The NET_IF_MEDIA_CONNECT_STATE connection state type.

- NetworkGuid
  The GUID that is associated with the network that the interface belongs to.

- ConnectionType
  A NET_IF_CONNECTION_TYPE NDIS network interface connection type.

- TransmitLinkSpeed
  The speed, in bits per second, of the transmit link.

- **ReceiveLinkSpeed**
  The speed, in bits per second, of the receive link.

- **InOctets**
  The number of octets of data that are received without errors through this interface. This value includes octets in unicast, broadcast, and multicast packets.

- **InUcastPkts**
  The number of unicast packets that are received without errors through this interface.

- **InNUcastPkts**
  The number of non-unicast packets that are received without errors through this interface. This value includes broadcast and multicast packets.

- **InDiscards**
  The number of incoming packets that were discarded even though they did not have errors.

- **InErrors**
  The number of incoming packets that were discarded because of errors.

- **InUnknownProtos**
  The number of incoming packets that were discarded because the protocol was unknown.

- **InUcastOctets**
  The number of octets of data that are received without errors in unicast packets through this interface.

- **InMulticastOctets**
  The number of octets of data that are received without errors in multicast packets through this interface.

- **InBroadcastOctets**
  The number of octets of data that are received without errors in broadcast packets through this interface.

- **OutOctets**
  The number of octets of data that are transmitted without errors through this interface. This value includes octets in unicast, broadcast, and multicast packets.

- **OutUcastPkts**
  The number of unicast packets that are transmitted without errors through this interface.

- OutNUcastPkts
  The number of non-unicast packets that are transmitted without errors through this interface. This value includes broadcast and multicast packets.

- OutDiscards
  The number of outgoing packets that were discarded even though they did not have errors.

- OutErrors
  The number of outgoing packets that were discarded because of errors.

- OutUcastOctets
  The number of octets of data that are transmitted without errors in unicast packets through this interface.

- OutMulticastOctets
  The number of octets of data that are transmitted without errors in multicast packets through this interface.

- OutBroadcastOctets
  The number of octets of data that are transmitted without errors in broadcast packets through this interface.

- OutQLen
  The transmit queue length. This field is not currently used.

Remarks

The values for the Type field are defined in the Ipifcons.h header file. Only the possible values that are listed in the description of the Type member are currently supported.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIfEntry2

GetIfTable2

MIB_IF_TABLE2

NET_IF_ACCESS_TYPE

NET_IF_ADMIN_STATUS

NET_IF_CONNECTION_TYPE

NET_IF_DIRECTION_TYPE

NET_IF_MEDIA_CONNECT_STATE

NET_LUID

TUNNEL_TYPE

MIB_IF_TABLE2 structure

Article • 03/03/2023

The MIB_IF_TABLE2 structure contains a table of logical and physical interface entries.

Syntax

C++

```
typedef struct _MIB_IF_TABLE2 {
  ULONG       NumEntries;
  MIB_IF_ROW2 Table[ANY_SIZE];
} MIB_IF_TABLE2, *PMIB_IF_TABLE2;
```

Members

- **NumEntries**

  The number of interface entries in the array.

- **Table**

  An array of MIB_IF_ROW2 structures that contain interface entries.

Remarks

The GetIfTable2 and GetIfTable2Ex functions enumerate the logical and physical interfaces on a local computer and return this information in a MIB_IF_TABLE2 structure.

The MIB_IF_TABLE2 structure might contain padding for alignment between the **NumEntries** member and the first MIB_IF_ROW2 array entry in the **Table** member. Padding for alignment might also be present between the MIB_IF_ROW2 array entries in the **Table** member. Any access to a MIB_IF_ROW2 array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIfTable2

GetIfTable2Ex

MIB_IF_ROW2

MIB_IFSTACK_ROW structure

Article • 03/03/2023

The MIB_IFSTACK_ROW structure represents the relationship between two network interfaces.

Syntax

```
C++

typedef struct _MIB_IFSTACK_ROW {
  NET_IFINDEX HigherLayerInterfaceIndex;
  NET_IFINDEX LowerLayerInterfaceIndex;
} MIB_IFSTACK_ROW, *PMIB_IFSTACK_ROW;
```

Members

- **HigherLayerInterfaceIndex**
  The network interface index for the interface that is higher in the interface stack table.

- **LowerLayerInterfaceIndex**
  The network interface index for the interface that is lower in the interface stack table.

Remarks

The relationship between the interfaces in the interface stack is that the interface with the index in the **HigherLayerInterfaceIndex** member is immediately above the interface with the index in the **LowerLayerInterfaceIndex** member.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIfStackTable

GetInvertedIfStackTable

MIB_IFSTACK_TABLE

MIB_INVERTEDIFSTACK_ROW

MIB_INVERTEDIFSTACK_TABLE

MIB_IFSTACK_TABLE structure

Article • 03/03/2023

The MIB_IFSTACK_TABLE structure contains a table of network interface stack row entries. This table specifies the relationship of the network interfaces on an interface stack.

Syntax

C++

```
typedef struct _MIB_IFSTACK_TABLE {
  ULONG           NumEntries;
  MIB_IFSTACK_ROW Table[ANY_SIZE];
} MIB_IFSTACK_TABLE, *PMIB_IFSTACK_TABLE;
```

Members

- **NumEntries**
  The number of interface stack row entries in the array.

- **Table**
  An array of MIB_IFSTACK_ROW structures that contain interface stack row entries.

Remarks

The relationship between the interfaces in the interface stack is that the interface with the index in the **HigherLayerInterfaceIndex** member of the MIB_IFSTACK_ROW structure is immediately above the interface with the index in the **LowerLayerInterfaceIndex** member of the MIB_IFSTACK_ROW structure.

The GetIfStackTable function enumerates the network interface stack row entries on a local computer and returns this information in a MIB_IFSTACK_TABLE structure.

The MIB_IFSTACK_TABLE structure might contain padding for alignment between the **NumEntries** member and the first MIB_IFSTACK_ROW array entry in the **Table** member. Padding for alignment might also be present between the MIB_IFSTACK_ROW array entries in the **Table** member. Any access to a MIB_IFSTACK_ROW array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIfStackTable

GetInvertedIfStackTable

MIB_IFSTACK_ROW

MIB_INVERTEDIFSTACK_ROW

MIB_INVERTEDIFSTACK_TABLE

MIB_INVERTEDIFSTACK_ROW structure

Article • 03/03/2023

The MIB_INVERTEDIFSTACK_ROW structure represents the relationship between two network interfaces.

Syntax

```c++
typedef struct _MIB_INVERTEDIFSTACK_ROW {
  NET_IFINDEX LowerLayerInterfaceIndex;
  NET_IFINDEX HigherLayerInterfaceIndex;
} MIB_INVERTEDIFSTACK_ROW, *PMIB_INVERTEDIFSTACK_ROW;
```

Members

- **LowerLayerInterfaceIndex**
  The network interface index for the interface that is lower in the interface stack table.

- **HigherLayerInterfaceIndex**
  The network interface index for the interface that is higher in the interface stack table.

Remarks

The relationship between the interfaces in the interface stack is that the interface with the index in the **HigherLayerInterfaceIndex** member is immediately above the interface with the index in the **LowerLayerInterfaceIndex** member.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIfStackTable

GetInvertedIfStackTable

MIB_IFSTACK_ROW

MIB_IFSTACK_TABLE

MIB_INVERTEDIFSTACK_TABLE

MIB_INVERTEDIFSTACK_TABLE structure

Article • 03/03/2023

The MIB_INVERTEDIFSTACK_TABLE structure contains a table of inverted network interface stack row entries. This table specifies the relationship of the network interfaces on an interface stack in reverse order.

Syntax

C++

```
typedef struct _MIB_INVERTEDIFSTACK_TABLE {
  ULONG                   NumEntries;
  MIB_INVERTEDIFSTACK_ROW Table[ANY_SIZE];
} MIB_INVERTEDIFSTACK_TABLE, *PMIB_INVERTEDIFSTACK_TABLE;
```

Members

- **NumEntries**

  The number of inverted interface stack row entries in the array.

- **Table**

  An array of **MIB_INVERTEDIFSTACK_ROW** structures that contain inverted interface stack row entries.

Remarks

The relationship between the interfaces in the interface stack is that the interface with the index in the **HigherLayerInterfaceIndex** member of the MIB_INVERTEDIFSTACK_ROW structure is immediately above the interface with the index in the **LowerLayerInterfaceIndex** member of the MIB_INVERTEDIFSTACK_ROW structure.

The **GetInvertedIfStackTable** function enumerates the inverted network interface stack row entries on a local computer and returns this information in a MIB_INVERTEDIFSTACK_TABLE structure.

The MIB_INVERTEDIFSTACK_TABLE structure might contain padding for alignment between the **NumEntries** member and the first **MIB_INVERTEDIFSTACK_ROW** array entry in the **Table** member. Padding for alignment might also be present between the

MIB_INVERTEDIFSTACK_ROW array entries in the **Table** member. Any access to a MIB_INVERTEDIFSTACK_ROW array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIfStackTable

GetInvertedIfStackTable

MIB_IFSTACK_ROW

MIB_IFSTACK_TABLE

MIB_INVERTEDIFSTACK_ROW

MIB_IPFORWARD_ROW2 structure

Article • 03/03/2023

The MIB_IPFORWARD_ROW2 structure stores information about an IP route entry.

Syntax

```c++
typedef struct _MIB_IPFORWARD_ROW2 {
  NET_LUID          InterfaceLuid;
  NET_IFINDEX       InterfaceIndex;
  IP_ADDRESS_PREFIX DestinationPrefix;
  SOCKADDR_INET     NextHop;
  UCHAR             SitePrefixLength;
  ULONG             ValidLifetime;
  ULONG             PreferredLifetime;
  ULONG             Metric;
  NL_ROUTE_PROTOCOL Protocol;
  BOOLEAN           Loopback;
  BOOLEAN           AutoconfigureAddress;
  BOOLEAN           Publish;
  BOOLEAN           Immortal;
  ULONG             Age;
  NL_ROUTE_ORIGIN   Origin;
} MIB_IPFORWARD_ROW2, *PMIB_IPFORWARD_ROW2;
```

Members

- **InterfaceLuid**

  The locally unique identifier (LUID) for the network interface that is associated with this IP route entry.

- **InterfaceIndex**

  The local index value for the network interface that is associated with this IP route entry. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

- **DestinationPrefix**

  The IP address prefix for the destination IP address for this route.

- **NextHop**

  For a remote route, the IP address of the next system or gateway that is along the route. If the route is to a local loopback address or an IP address on the local link, the next hop is unspecified (all zeros). For a local loopback route, this member should be an IPv4 address of 0.0.0.0 for an IPv4 route entry or an IPv6 address address of 0::0 for an IPv6 route entry.

- **SitePrefixLength**
  The length, in bits, of the site prefix or network part of the IP address for this route. For an IPv4 route entry, any value that is greater than 32 is an illegal value. For an IPv6 route entry, any value that is greater than 128 is an illegal value. A value of 255 is typically used to represent an illegal value.

- **ValidLifetime**
  The maximum time, in seconds, that the IP route entry is valid. A value of 0xffffffff is considered to be infinite.

- **PreferredLifetime**
  The preferred time, in seconds, that the IP route entry is valid. A value of 0xffffffff is considered to be infinite.

- **Metric**
  The route metric offset value for this IP route entry. Note the actual route metric that is used to compute the route preference is the interface metric that is specified in the **Metric** member of the MIB_IPINTERFACE_ROW structure added to the route metric offset that is specified in this **Metric** member. The semantics of this metric are determined by the routing protocol that is specified in the **Protocol** member. If this metric is not used, its value should be set to -1. This value is documented in RFC 4292. For more information, see IP Forwarding Table MIB.

- **Protocol**
  The NL_ROUTE_PROTOCOL routing mechanism type that this IP route was added with.

- **Loopback**
  A value that specifies if the route is a loopback route (the gateway is on the local host).

- **AutoconfigureAddress**
  A value that specifies if the IP address is autoconfigured.

- **Publish**
  A value that specifies if the route is published.

- **Immortal**
  A value that specifies if the route is immortal.

- **Age**
  The number of seconds since the route was added or modified in the network routing table.

- **Origin**
  A NL_ROUTE_ORIGIN IP route origin type.

Remarks

The GetIpForwardTable2 function enumerates the IP route entries on a local computer and returns this information in a MIB_IPFORWARD_TABLE2 structure as an array of MIB_IPFORWARD_ROW2 entries.

The **GetIpForwardEntry2** function retrieves a single IP route entry and returns this information in a MIB_IPFORWARD_ROW2 structure.

An entry with the **Prefix** and the **PrefixLength** members of IP_ADDRESS_PREFIX set to zero in the **DestinationPrefix** member in the MIB_IPFORWARD_ROW2 structure is considered a default route. The MIB_IPFORWARD_TABLE2 might contain multiple MIB_IPFORWARD_ROW2 entries with the **Prefix** and the **PrefixLength** members of the IP_ADDRESS_PREFIX set to zero in the **DestinationPrefix** member when there are multiple network adapters installed.

The **Metric** member of a MIB_IPFORWARD_ROW2 entry is a value that is assigned to an IP route for a particular network interface that identifies the cost that is associated with using that route. For example, the metric can be valued in terms of link speed, hop count, or time delay. Automatic metric is a feature on Windows XP and later versions of the Windows operating systems that automatically configures the metric for the local routes that are based on link speed. By default, the automatic metric feature is enabled (the **UseAutomaticMetric** member of the MIB_IPINTERFACE_ROW structure is set to **TRUE**) on Windows XP and later. You can also manually configure this feature to assign a specific metric to an IP route.

The route metric that is specified in the **Metric** member of the MIB_IPFORWARD_ROW2 structure represents only the route metric offset. The complete metric is a combination of this route metric offset added to the interface metric that is specified in the **Metric** member of the MIB_IPINTERFACE_ROW structure of the associated interface. A driver can retrieve the interface metric by calling the GetIpInterfaceEntry function.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)

See also

CreateIpForwardEntry2

DeleteIpForwardEntry2

GetIpForwardEntry2

GetIpForwardTable2

GetIpInterfaceEntry

IP_ADDRESS_PREFIX

MIB_IPFORWARD_TABLE2

MIB_IPINTERFACE_ROW

NL_ROUTE_ORIGIN

NL_ROUTE_PROTOCOL

SetIpForwardEntry2

MIB_IPFORWARD_TABLE2 structure

Article • 03/03/2023

The MIB_IPFORWARD_TABLE2 structure contains a table of IP route entries.

Syntax

```C++
typedef struct _MIB_IPFORWARD_TABLE2 {
  ULONG              NumEntries;
  MIB_IPFORWARD_ROW2 Table[ANY_SIZE];
} MIB_IPFORWARD_TABLE2, *PMIB_IPFORWARD_TABLE2;
```

Members

- **NumEntries**

  A value that specifies the number of IP route entries in the array.

- **Table**

  An array of MIB_IPFORWARD_ROW2 structures that contain IP route entries.

Remarks

The GetIpForwardEntry2 function enumerates the IP route entries on a local computer and returns this information in a MIB_IPFORWARD_TABLE2 structure.

The **GetIpForwardEntry2** function retrieves a single IP route entry and returns this information in a MIB_IPFORWARD_ROW2 structure.

The MIB_IPFORWARD_TABLE2 structure might contain padding for alignment between the **NumEntries** member and the first MIB_IPFORWARD_ROW2 array entry in the **Table** member. Padding for alignment might also be present between the MIB_IPFORWARD_ROW2 array entries in the **Table** member. Any access to a MIB_IPFORWARD_ROW2 array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)

See also

GetIpForwardEntry2

GetIpForwardTable2

MIB_IPFORWARD_ROW2

MIB_IPINTERFACE_ROW structure

Article • 03/03/2023

The MIB_IPINTERFACE_ROW structure stores interface management information for a particular IP address family on a network interface.

Syntax

```c++
typedef struct _MIB_IPINTERFACE_ROW {
  ADDRESS_FAMILY                 Family;
  NET_LUID                       InterfaceLuid;
  NET_IFINDEX                    InterfaceIndex;
  ULONG                          MaxReassemblySize;
  ULONG64                        InterfaceIdentifier;
  ULONG                          MinRouterAdvertisementInterval;
  ULONG                          MaxRouterAdvertisementInterval;
  BOOLEAN                        AdvertisingEnabled;
  BOOLEAN                        ForwardingEnabled;
  BOOLEAN                        WeakHostSend;
  BOOLEAN                        WeakHostReceive;
  BOOLEAN                        UseAutomaticMetric;
  BOOLEAN                        UseNeighborUnreachabilityDetection;
  BOOLEAN                        ManagedAddressConfigurationSupported;
  BOOLEAN                        OtherStatefulConfigurationSupported;
  BOOLEAN                        AdvertiseDefaultRoute;
  NL_ROUTER_DISCOVERY_BEHAVIOR   RouterDiscoveryBehavior;
  ULONG                          DadTransmits;
  ULONG                          BaseReachableTime;
  ULONG                          RetransmitTime;
  ULONG                          PathMtuDiscoveryTimeout;
  NL_LINK_LOCAL_ADDRESS_BEHAVIOR LinkLocalAddressBehavior;
  ULONG                          LinkLocalAddressTimeout;
  ULONG                          ZoneIndices[ScopeLevelCount];
  ULONG                          SitePrefixLength;
  ULONG                          Metric;
  ULONG                          NlMtu;
  BOOLEAN                        Connected;
  BOOLEAN                        SupportsWakeUpPatterns;
  BOOLEAN                        SupportsNeighborDiscovery;
  BOOLEAN                        SupportsRouterDiscovery;
  ULONG                          ReachableTime;
  NL_INTERFACE_OFFLOAD_ROD       TransmitOffload;
  NL_INTERFACE_OFFLOAD_ROD       ReceiveOffload;
  BOOLEAN                        DisableDefaultRoutes;
} MIB_IPINTERFACE_ROW, *PMIB_IPINTERFACE_ROW;
```

Members

- **Family**
  The address family. Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for this member are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported:

  - AF_INET
    The IPv4 address family.
  - AF_INET6
    The IPv6 address family.
  - AF_UNSPEC
    The address family is unspecified.

- **InterfaceLuid**
  The locally unique identifier (LUID) for the network interface.

- **InterfaceIndex**
  The local index value for the network interface. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

- **MaxReassemblySize**
  The maximum reassembly size, in bytes, of a fragmented IP packet. This member is currently set to zero and reserved for future use.

- **InterfaceIdentifier**
  Reserved for future use. This member is currently set to zero.

- **MinRouterAdvertisementInterval**
  The minimum router advertisement interval, in milliseconds, on this IP interface. This member defaults to 200 for IPv6. This member is applicable only if the **AdvertisingEnabled** member is set to **TRUE**.

- MaxRouterAdvertisementInterval

  The maximum router advertisement interval, in milliseconds, on this IP interface. This member defaults to 600 for IPv6. This member is applicable only if the **AdvertisingEnabled** member is set to **TRUE**.

- AdvertisingEnabled

  A value that indicates if router advertising is enabled on this IP interface. The default for IPv6 is that router advertisement is enabled only if the interface is configured to act as a router. The default for IPv4 is that router advertisement is disabled.

- ForwardingEnabled

  A value that indicates if IP forwarding is enabled on this IP interface.

- WeakHostSend

  A value that indicates if weak host send mode is enabled on this IP interface.

- WeakHostReceive

  A value that indicates if weak host receive mode is enabled on this IP interface.

- UseAutomaticMetric

  A value that indicates if the IP interface uses automatic metric.

- UseNeighborUnreachabilityDetection

  A value that indicates if neighbor unreachability detection is enabled on this IP interface.

- ManagedAddressConfigurationSupported

  A value that indicates if the IP interface supports managed address configuration by using DHCP.

- OtherStatefulConfigurationSupported

  A value that indicates if the IP interface supports other stateful configuration (for example, route configuration).

- AdvertiseDefaultRoute

  A value that indicates if the IP interface advertises the default route. This member is applicable only if the **AdvertisingEnabled** member is set to **TRUE**.

- RouterDiscoveryBehavior

  An NL_ROUTER_DISCOVERY_BEHAVIOR router discovery behavior type.

- DadTransmits

  The number of consecutive messages that are sent while the driver performs duplicate address detection on a tentative IP unicast address. A value of zero indicates that duplicate address detection is not performed on tentative IP addresses. A value of one indicates a single transmission with no follow up retransmissions. For IPv4, the default value for this member is 3. For IPv6, the default value for this member is 1. For IPv6, these messages are sent as IPv6 Neighbor Solicitation (NS) requests. This member is defined as DupAddrDetectTransmits in RFC 2462. For more information, see IPv6 "Stateless Address Autoconfiguration" ↗.

- **BaseReachableTime**
  The base for random reachable time, in milliseconds. The member is described in RFC 2461. For more information, see "Neighbor Discovery for IP Version 6 (IPv6)" ↗.

- **RetransmitTime**
  The IPv6 Neighbor Solicitation (NS) time-out, in milliseconds. The member is described in RFC 2461. For more information, see "Neighbor Discovery for IP Version 6 (IPv6)" ↗.

- **PathMtuDiscoveryTimeout**
  The path MTU discovery time-out, in milliseconds.

- **LinkLocalAddressBehavior**
  A **NL_LINK_LOCAL_ADDRESS_BEHAVIOR** link local address behavior type.

- **LinkLocalAddressTimeout**
  The link local IP address time-out, in milliseconds.

- **ZoneIndices**
  An array that specifies the zone part of scope IDs.

- **SitePrefixLength**
  The site prefix length, in bits, of the IP interface address. The length, in bits, of the site prefix or network part of the IP interface address. For an IPv4 address, any value that is greater than 32 is an illegal value. For an IPv6 address, any value that is greater than 128 is an illegal value. A value of 255 is typically used to represent an illegal value.

- **Metric**
  The interface metric. Note that the actual route metric that is used to compute the route preference is the summation of the route metric offset that is specified in the **Metric** member of the **MIB_IPFORWARD_ROW2** structure and the interface metric that is specified in this member.

- **NlMtu**
  The network layer MTU size, in bytes.

- **Connected**
  A value that indicates if the interface is connected to a network access point.

- **SupportsWakeUpPatterns**
  A value that specifies if the network interface supports Wake on LAN.

- **SupportsNeighborDiscovery**
  A value that specifies if the IP interface support neighbor discovery.

- **SupportsRouterDiscovery**
  A value that specifies if the IP interface support neighbor discovery.

- **ReachableTime**
  The base for random reachable time, in milliseconds. The member is described in RFC 2461. For more information, see Neighbor Discovery for IP Version 6 (IPv6) .

- **TransmitOffload**
  A set of flags that indicate the transmit offload capabilities for the IP interface. The NL_INTERFACE_OFFLOAD_ROD structure is defined in the Nldef.h header file.

- **ReceiveOffload**
  A set of flags that indicate the receive offload capabilities for the IP interface. The NL_INTERFACE_OFFLOAD_ROD structure is defined in the Nldef.h header file.

- **DisableDefaultRoutes**
  A value that indicates if using default route on the interface should be disabled. VPN clients can use this member to restrict split tunneling.

Remarks

The **Family**, **InterfaceLuid**, and **InterfaceIndex** members uniquely identify a MIB_IPINTERFACE_ROW entry.

When a unicast packet arrives at a host, IP must determine whether the packet is locally destined (its destination matches an address that is assigned to an interface of the host). IP implementations that follow a weak host model accept any locally destined packet, regardless of the interface that the packet was received on. IP implementations that follow the strong host model accept only locally destined packets if the destination address in the packet matches an address that is assigned to the interface that the packet was received on. The weak host model provides better network connectivity. However, it also makes hosts susceptible to multihome-based network attacks.

The current IPv4 implementation in the Windows Server 2003 and Windows XP operating systems uses the weak host model. The TCP/IP stack on Windows Vista and later versions of the Windows operating systems supports the strong host model for both IPv4 and IPv6 and is configured to use the strong host mode by default (the **WeakHostReceive** and **WeakHostSend** members are set to **FALSE**). You can configure the TCP/IP stack on Windows Vista and later to use a weak host model.

A metric is a value that is assigned to an IP route for a particular network interface that identifies the cost that is associated with using that route. For example, the metric can be valued in terms of link speed, hop count, or time delay. Automatic metric is a feature on Windows XP and later that automatically configures the metric for the local routes that are based on link speed. By default, the automatic metric feature is enabled (the **UseAutomaticMetric** is set to **TRUE**) on Windows XP and later. You can also manually configure this feature to assign a specific metric to an IP route.

The automatic metric feature can be useful when the routing table contains multiple routes for the same destination. For example, a computer that has a 10 megabit network interface and a 100 megabit network interface has a default gateway that is configured on both network interfaces. When **UseAutomaticMetric** is **TRUE**, this feature can force all traffic that is destined for the Internet, for example, to use the fastest network interface that is available.

The interface metric that is specified in the **Metric** member represents only the metric for the interface. The complete routing metric is a combination of this interface metric added to the route metric offset that is specified in the **Metric** member of the MIB_IPFORWARD_ROW2 structure of a route entry that is specified on this interface.

Unprivileged simultaneous access to multiple networks of different security requirements creates a security hole and enables an unprivileged driver to accidentally relay data between the two networks. A typical example is simultaneous access to a virtual private network (VPN) and the Internet. Windows Server 2003 and Windows XP use a weak host model, where Remote Access Service (RAS) prevents such simultaneous access by increasing the route metric of all default routes over other interfaces. Therefore, all traffic is routed through the VPN interface, disrupting other network connectivity.

On Windows Vista and later, by default, a strong host model is used. If a source IP address is specified in the route lookup by using the GetBestRoute2 function, the route lookup is restricted to the interface of the source IP address. The route metric modification by RAS has no effect because the list of potential routes does not even have the route for the VPN interface, which enables traffic to the Internet. Your driver can use the **DisableDefaultRoutes** member of the MIB_IPINTERFACE_ROW structure to disable using the default route on an interface. VPN clients can use this member as a security measure to restrict split tunneling when split tunneling is not required by the VPN client. A VPN client can call the SetIpInterfaceEntry function to set the **DisableDefaultRoutes** member to **TRUE** when it is required. A VPN client can query the current state of the **DisableDefaultRoutes** member by calling the GetIpInterfaceEntry function.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetBestRoute2

GetIpInterfaceEntry

MIB_IPFORWARD_ROW2

MIB_IPINTERFACE_TABLE

NET_LUID

NL_LINK_LOCAL_ADDRESS_BEHAVIOR

NL_ROUTER_DISCOVERY_BEHAVIOR

SetIpInterfaceEntry

MIB_IPINTERFACE_TABLE structure

Article • 03/03/2023

The MIB_IPINTERFACE_TABLE structure contains a table of IP interface entries.

Syntax

C++

```
typedef struct _MIB_IPINTERFACE_TABLE {
  ULONG               NumEntries;
  MIB_IPINTERFACE_ROW Table[ANY_SIZE];
} MIB_IPINTERFACE_TABLE, *PMIB_IPINTERFACE_TABLE;
```

Members

- **NumEntries**

  The number of IP interface entries in the array.

- **Table**

  An array of MIB_IPINTERFACE_ROW structures that contain IP interface entries.

Remarks

The GetIpInterfaceTable function enumerates the IP interface entries on a local computer and returns this information in a MIB_IPINTERFACE_TABLE structure.

The MIB_IPINTERFACE_TABLE structure might contain padding for alignment between the **NumEntries** member and the first MIB_IPINTERFACE_ROW array entry in the **Table** member. Padding for alignment might also be present between the MIB_IPINTERFACE_ROW array entries in the **Table** member. Any access to a MIB_IPINTERFACE_ROW array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIpInterfaceTable

MIB_IPINTERFACE_ROW

MIB_IPNET_ROW2 structure

Article • 03/03/2023

The MIB_IPNET_ROW2 structure stores information about a neighbor IP address.

Syntax

C++

```
typedef struct _MIB_IPNET_ROW2 {
  SOCKADDR_INET     Address;
  NET_IFINDEX       InterfaceIndex;
  NET_LUID          InterfaceLuid;
  UCHAR             PhysicalAddress[IF_MAX_PHYS_ADDRESS_LENGTH];
  ULONG             PhysicalAddressLength;
  NL_NEIGHBOR_STATE State;
  union {
    struct {
      BOOLEAN IsRouter   :1;
      BOOLEAN IsUnreachable  :1;
    };
    UCHAR  Flags;
  };
  union {
    ULONG LastReachable;
    ULONG LastUnreachable;
  } ReachabilityTime;
} MIB_IPNET_ROW2, *PMIB_IPNET_ROW2;
```

Members

- **Address**

  The neighbor IP address. This member can be an IPv6 address or an IPv4 address.

- **InterfaceIndex**

  The local index value for the network interface that is associated with this IP address. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

- **InterfaceLuid**

  The locally unique identifier (LUID) for the network interface that is associated with this IP address.

- **PhysicalAddress**
  The physical hardware address of the adapter for the network interface that is associated with this IP address.

- **PhysicalAddressLength**
  The length, in bytes, of the physical hardware address that the **PhysicalAddress** member specifies. The maximum value that is supported is 32 bytes.

- **State**
  An NL_NEIGHBOR_STATE network layer neighbor state type.

- **IsRouter**
  A value that indicates if this IP address is a router.

- **IsUnreachable**
  A value that indicates if this IP address is unreachable.

- **Flags**
  A set of flags that indicate whether the IP address is a router and whether the IP address is unreachable.

- **ReachabilityTime**
  The time that the node assumes that the neighbor is reachable or unreachable after the node receives information about the reachability of the neighbor.

  This union contains the following members:

  - **LastReachable**
    The time, in milliseconds, that a node assumes that the neighbor will remain reachable after the node receives a reachability confirmation from the neighbor.

  - **LastUnreachable**
    The time, in milliseconds, that a node assumes that the neighbor will remain unreachable after the node fails to receive a reachability confirmation from the neighbor.

Remarks

The GetIpNetTable2 function enumerates the neighbor IP addresses on a local computer and returns this information in an MIB_IPNET_TABLE2 structure. For IPv4, this information includes addresses determined by using the Address Resolution Protocol (ARP). For IPv6, this information includes addresses determined by using the Neighbor Discovery (ND) protocol for IPv6 as specified in RFC 2461. For more information, see Neighbor Discovery for IP Version 6 (IPv6).

The GetIpNetEntry2 function retrieves a single neighbor IP address and returns this information in a MIB_IPNET_ROW2 structure.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

CreateIpNetEntry2

GetIpNetEntry2

GetIpNetTable2

MIB_IPNET_TABLE2

NL_NEIGHBOR_STATE

MIB_IPNET_TABLE2 structure

Article • 03/03/2023

The MIB_IPNET_TABLE2 structure contains a table of neighbor IP address entries.

Syntax

C++

```
typedef struct _MIB_IPNET_TABLE2 {
  ULONG          NumEntries;
  MIB_IPNET_ROW2 Table[ANY_SIZE];
} MIB_IPNET_TABLE2, *PMIB_IPNET_TABLE2;
```

Members

- **NumEntries**

  A value that specifies the number of IP network neighbor address entries in the array.

- **Table**

  An array of **MIB_IPNET_ROW2** structures that contain IP network neighbor address entries.

Remarks

The **GetIpNetTable2** function enumerates the neighbor IP addresses on a local computer and returns this information in an MIB_IPNET_TABLE2 structure. For IPv4, this information includes addresses determined by using the Address Resolution Protocol (ARP). For IPv6, this information includes addresses determined by using the Neighbor Discovery (ND) protocol for IPv6 as specified in RFC 2461. For more information, see Neighbor Discovery for IP Version 6 (IPv6).

The MIB_IPNET_TABLE2 structure might contain padding for alignment between the **NumEntries** member and the first **MIB_IPNET_ROW2** array entry in the **Table** member. Padding for alignment might also be present between the MIB_IPNET_ROW2 array entries in the **Table** member. Any access to a MIB_IPNET_ROW2 array entry should assume that padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetIpNetTable2

MIB_IPNET_ROW2

MIB_IPPATH_ROW structure

Article • 03/03/2023

The MIB_IPPATH_ROW structure stores information about an IP path entry.

Syntax

C++

```
typedef struct _MIB_IPPATH_ROW {
  SOCKADDR_INET Source;
  SOCKADDR_INET Destination;
  NET_LUID      InterfaceLuid;
  NET_IFINDEX   InterfaceIndex;
  SOCKADDR_INET CurrentNextHop;
  ULONG         PathMtu;
  ULONG         RttMean;
  ULONG         RttDeviation;
  union {
    ULONG LastReachable;
    ULONG LastUnreachable;
  };
  BOOLEAN       IsReachable;
  ULONG64       LinkTransmitSpeed;
  ULONG64       LinkReceiveSpeed;
} MIB_IPPATH_ROW, *PMIB_IPPATH_ROW;
```

Members

- **Source**

  The source IP address for this IP path entry.

- **Destination**

  The destination IP address for this IP path entry.

- **InterfaceLuid**

  The locally unique identifier (LUID) for the network interface that is associated with this IP path entry.

- **InterfaceIndex**

  The local index value for the network interface that is associated with this IP path entry. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

- **CurrentNextHop**
  The current IP address of the next system or gateway that is along the route. This member can change over the lifetime of a path.

- **PathMtu**
  The maximum transmission unit (MTU) size, in bytes, to the destination IP address for this IP path entry.

- **RttMean**
  The estimated mean round-trip time (RTT), in milliseconds, to the destination IP address for this IP path entry.

- **RttDeviation**
  The estimated mean deviation for the round-trip time (RTT), in milliseconds, to the destination IP address for this IP path entry.

- **LastReachable**
  The time, in milliseconds, that a node assumes that the destination IP address is reachable after having received a reachability confirmation.

- **LastUnreachable**
  The time, in milliseconds, that a node assumes that the destination IP address is unreachable after not having received a reachability confirmation.

- **IsReachable**
  A value that indicates if the destination IP address is reachable for this IP path entry.

- **LinkTransmitSpeed**
  The estimated speed, in bits per second, of the transmit link to the destination IP address for this IP path entry.

- **LinkReceiveSpeed**
  The estimated speed, in bits per second, of the receive link from the destination IP address for this IP path entry.

Remarks

The **GetIpPathTable** function enumerates the IP path entries on a local computer and returns this information in a **MIB_IPPATH_TABLE** structure as an array of MIB_IPPATH_ROW entries.

The **GetIpPathTable** function retrieves a single IP path entry and returns this information in a **MIB_IPPATH_TABLE** structure.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

FlushIpPathTable

GetIpPathEntry

GetIpPathTable

MIB_IPPATH_TABLE

MIB_IPPATH_TABLE structure

Article • 03/03/2023

The MIB_IPPATH_TABLE structure contains a table of IP path entries.

Syntax

```
C++

typedef struct _MIB_IPPATH_TABLE {
  ULONG          NumEntries;
  MIB_IPPATH_ROW Table[ANY_SIZE];
} MIB_IPPATH_TABLE, *PMIB_IPPATH_TABLE;
```

Members

- **NumEntries**

  A value that specifies the number of IP path entries in the array.

- **Table**

  An array of MIB_IPPATH_ROW structures that contain IP path entries.

Remarks

The GetIpPathTable function enumerates the IP path entries on a local computer and returns this information in a MIB_IPPATH_TABLE structure. The FlushIpPathTable function flushes the IP path table entries on a local computer.

The GetIpPathEntry function retrieves a single IP path entry and returns this information in a MIB_IPPATH_ROW structure.

The MIB_IPPATH_TABLE structure might contain padding for alignment between the **NumEntries** member and the first MIB_IPPATH_ROW array entry in the **Table** member. Padding for alignment might also be present between the MIB_IPPATH_ROW array entries in the **Table** member. Any access to a MIB_IPPATH_ROW array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.

Header	Netioapi.h (include Netioapi.h)

See also

FlushIpPathTable

GetIpPathEntry

GetIpPathTable

MIB_IPPATH_ROW

MIB_MULTICASTIPADDRESS_ROW structure

Article • 03/03/2023

The MIB_MULTICASTIPADDRESS_ROW structure stores information about a multicast IP address.

Syntax

```c++
typedef struct _MIB_MULTICASTIPADDRESS_ROW {
  SOCKADDR_INET Address;
  NET_IFINDEX   InterfaceIndex;
  NET_LUID      InterfaceLuid;
  SCOPE_ID      ScopeId;
} MIB_MULTICASTIPADDRESS_ROW, *PMIB_MULTICASTIPADDRESS_ROW;
```

Members

- **Address**
  The multicast IP address. This member can be an IPv6 address or an IPv4 address.

- **InterfaceIndex**
  The local index value for the network interface that is associated with this IP address. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

- **InterfaceLuid**
  The locally unique identifier (LUID) for the network interface that is associated with this IP address.

- **ScopeId**
  The scope ID of the multicast IP address. This member is applicable only to an IPv6 address. Your driver cannot set this member. This member is automatically determined by the interface that the address was added on.

Remarks

The GetMulticastIpAddressTable function enumerates the multicast IP addresses on a local computer and returns this information in a MIB_MULTICASTIPADDRESS_TABLE structure. The **GetMulticastIpAddressEntry** function retrieves a single multicast IP address and returns this information in a MIB_MULTICASTIPADDRESS_ROW structure.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

FlushIpPathTable

GetIpPathEntry

GetIpPathTable

GetMulticastIpAddressTable

MIB_IPPATH_ROW

MIB_MULTICASTIPADDRESS_TABLE

MIB_MULTICASTIPADDRESS_TABLE structure

Article • 03/03/2023

The MIB_MULTICASTIPADDRESS_TABLE structure contains a table of multicast IP address entries.

Syntax

```
C++

typedef struct _MIB_MULTICASTIPADDRESS_TABLE {
  ULONG                          NumEntries;
  MIB_MULTICASTIPADDRESS_ROW Table[ANY_SIZE];
} MIB_MULTICASTIPADDRESS_TABLE, *PMIB_MULTICASTIPADDRESS_TABLE;
```

Members

- **NumEntries**
  A value that specifies the number of multicast IP address entries in the array.

- **Table**
  An array of **MIB_MULTICASTIPADDRESS_ROW** structures that contain multicast IP address entries.

Remarks

The **GetMulticastIpAddressTable** function enumerates the multicast IP addresses on a local computer and returns this information in an MIB_MULTICASTIPADDRESS_TABLE structure.

The MIB_MULTICASTIPADDRESS_TABLE structure might contain padding for alignment between the **NumEntries** member and the first **MIB_MULTICASTIPADDRESS_ROW** array entry in the **Table** member. Padding for alignment might also be present between the MIB_MULTICASTIPADDRESS_ROW array entries in the **Table** member. Any access to a MIB_MULTICASTIPADDRESS_ROW array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetMulticastIpAddressTable

MIB_MULTICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_ROW structure

Article • 03/03/2023

The MIB_UNICASTIPADDRESS_ROW structure stores information about a unicast IP address.

Syntax

C++

```
typedef struct _MIB_UNICASTIPADDRESS_ROW {
  SOCKADDR_INET    Address;
  NET_LUID         InterfaceLuid;
  NET_IFINDEX      InterfaceIndex;
  NL_PREFIX_ORIGIN PrefixOrigin;
  NL_SUFFIX_ORIGIN SuffixOrigin;
  ULONG            ValidLifetime;
  ULONG            PreferredLifetime;
  UINT8            OnLinkPrefixLength;
  BOOLEAN          SkipAsSource;
  NL_DAD_STATE     DadState;
  SCOPE_ID         ScopeId;
  LARGE_INTEGER    CreationTimeStamp;
} MIB_UNICASTIPADDRESS_ROW, *PMIB_UNICASTIPADDRESS_ROW;
```

Members

- **Address**
  The unicast IP address. This member can be an IPv6 address or an IPv4 address.

- **InterfaceLuid**
  The locally unique identifier (LUID) for the network interface that is associated with this IP address.

- **InterfaceIndex**
  The local index value for the network interface that is associated with this IP address. This index value might change when a network adapter is disabled and then enabled, or under other circumstances, and should not be considered persistent.

- **PrefixOrigin**
  An NL_PREFIX_ORIGIN type that specifies the origin of the prefix or network part of the IP address.

- **SuffixOrigin**
  An NL_SUFFIX_ORIGIN type that specifies the origin of the suffix or host part of the IP address.

- **ValidLifetime**
  The maximum time, in seconds, that the IP address is valid. A value of 0xffffffff is considered to be infinite.

- **PreferredLifetime**
  The preferred time, in seconds, that the IP address is valid. A value of 0xffffffff is considered to be infinite.

- **OnLinkPrefixLength**
  The length, in bits, of the prefix or network part of the IP address. For a unicast IPv4 address, any value that is greater than 32 is an illegal value. For a unicast IPv6 address, any value that is greater than 128 is an illegal value. A value of 255 is typically used to represent an illegal value.

- **SkipAsSource**
  A value that specifies if the address can be used as an IP source address.

- **DadState**
  A NL_DAD_STATE duplicate address detection (DAD) type.

- **ScopeId**
  The scope ID of the IP address. This member is applicable only to an IPv6 address. Your driver cannot set this member. This member is automatically determined by the interface that the address was added on.

- **CreationTimeStamp**
  The time stamp when the IP address was created.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

CreateUnicastIpAddressEntry

DeleteUnicastIpAddressEntry

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

InitializeUnicastIpAddressEntry

MIB_UNICASTIPADDRESS_TABLE

NL_DAD_STATE

NL_PREFIX_ORIGIN

NL_SUFFIX_ORIGIN

SetUnicastIpAddressEntry

MIB_UNICASTIPADDRESS_TABLE structure

Article • 03/03/2023

The MIB_UNICASTIPADDRESS_TABLE structure contains a table of unicast IP address entries.

Syntax

```C++
typedef struct _MIB_UNICASTIPADDRESS_TABLE {
  ULONG                    NumEntries;
  MIB_UNICASTIPADDRESS_ROW Table[ANY_SIZE];
} MIB_UNICASTIPADDRESS_TABLE, *PMIB_UNICASTIPADDRESS_TABLE;
```

Members

- **NumEntries**
  A value that specifies the number of unicast IP address entries in the array.

- **Table**
  An array of MIB_UNICASTIPADDRESS_ROW structures that contains unicast IP address entries.

Remarks

The GetUnicastIpAddressTable function enumerates the unicast IP addresses on a local computer and returns this information in an MIB_UNICASTIPADDRESS_TABLE structure.

The MIB_UNICASTIPADDRESS_TABLE structure might contain padding for alignment between the **NumEntries** member and the first MIB_UNICASTIPADDRESS_ROW array entry in the **Table** member. Padding for alignment might also be present between the MIB_UNICASTIPADDRESS_ROW array entries in the **Table** member. Any access to a MIB_UNICASTIPADDRESS_ROW array entry should assume padding might exist.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)

See also

GetUnicastIpAddressTable

MIB_UNICASTIPADDRESS_ROW

NL_DAD_STATE enumeration

Article • 03/03/2023

The NL_DAD_STATE enumeration type defines the duplicate address detection (DAD) state.

Syntax

C++

```
typedef enum  {
  NldsInvalid,
  NldsTentative,
  NldsDuplicate,
  NldsDeprecated,
  NldsPreferred,
  IpDadStateInvalid     = 0,
  IpDadStateTentative,
  IpDadStateDuplicate,
  IpDadStateDeprecated,
  IpDadStatePreferred
} NL_DAD_STATE;
```

Constants

- **NldsInvalid**
  Reserved for system use. Do not use this value in your driver.

- **NldsTentative**
  Reserved for system use. Do not use this value in your driver.

- **NldsDuplicate**
  Reserved for system use. Do not use this value in your driver.

- **NldsDeprecated**
  Reserved for system use. Do not use this value in your driver.

- **NldsPreferred**
  Reserved for system use. Do not use this value in your driver.

- **IpDadStateInvalid**
  The DAD state is invalid.

- **IpDadStateTentative**
  The DAD state is tentative.

- **IpDadStateDuplicate**
  A duplicate IP address has been detected.

- **IpDadStateDeprecated**
  The IP address has been deprecated.

- **IpDadStatePreferred**
  The IP address is the preferred address.

Remarks

The DAD state applies to both IPv4 and IPv6 addresses.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Nldef.h (include Netioapi.h)

NL_PREFIX_ORIGIN enumeration

Article • 03/03/2023

The NL_PREFIX_ORIGIN enumeration type defines the origin of the prefix or network part of the IP address.

Syntax

C++

```
typedef enum  {
  IpPrefixOriginOther                 = 0,
  IpPrefixOriginManual,
  IpPrefixOriginWellKnown,
  IpPrefixOriginDhcp,
  IpPrefixOriginRouterAdvertisement,
  IpPrefixOriginUnchanged             = 1 << 4
} NL_PREFIX_ORIGIN;
```

Constants

- **IpPrefixOriginOther**
  The IP address prefix was configured by using a source other than those that are defined in this enumeration. This value applies to an IPv6 or IPv4 address.

- **IpPrefixOriginManual**
  The IP address prefix was configured manually. This value applies to an IPv6 or IPv4 address.

- **IpPrefixOriginWellKnown**
  The IP address prefix was configured by using a well-known address. This value applies to an IPv6 link-local address or an IPv6 loopback address.

- **IpPrefixOriginDhcp**
  The IP address prefix was configured by using DHCP. This value applies to an IPv4 address configured by using DHCP or an IPv6 address configured by using DHCPv6.

- **IpPrefixOriginRouterAdvertisement**
  The IP address prefix was configured by using router advertisement. This value applies to an anonymous IPv6 address that was generated after receiving a router advertisement.

- **IpPrefixOriginUnchanged**
  The IP address prefix should be unchanged. This value is used when setting the properties for a unicast IP interface when the value for the IP prefix origin should be unchanged.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Nldef.h (include Netioapi.h)

NL_SUFFIX_ORIGIN enumeration

Article • 03/03/2023

The NL_SUFFIX_ORIGIN enumeration type defines the origin of the suffix or host part of the IP address.

Syntax

C++

```
typedef enum  {
  NlsoOther,
  NlsoManual,
  NlsoWellKnown,
  NlsoDhcp,
  NlsoLinkLayerAddress,
  NlsoRandom,
  IpSuffixOriginOther             = 0,
  IpSuffixOriginManual,
  IpSuffixOriginWellKnown,
  IpSuffixOriginDhcp,
  IpSuffixOriginLinkLayerAddress,
  IpSuffixOriginRandom,
  IpSuffixOriginUnchanged         = 1 << 4
} NL_SUFFIX_ORIGIN;
```

Constants

- **NlsoOther**
  Reserved for system use. Do not use this value in your driver.

- **NlsoManual**
  Reserved for system use. Do not use this value in your driver.

- **NlsoWellKnown**
  Reserved for system use. Do not use this value in your driver.

- **NlsoDhcp**
  Reserved for system use. Do not use this value in your driver.

- **NlsoLinkLayerAddress**
  Reserved for system use. Do not use this value in your driver.

- **NlsoRandom**
  Reserved for system use. Do not use this value in your driver.

- **IpSuffixOriginOther**
  The IP address suffix was configured by using a source other than those that are defined in this enumeration. This value applies to an IPv6 or IPv4 address.

- **IpSuffixOriginManual**
  The IP address suffix was configured manually. This value applies to an IPv6 or IPv4 address.

- **IpSuffixOriginWellKnown**
  The IP address suffix was configured by using a well-known address. This value applies to an IPv6 link-local address or an IPv6 loopback address.

- **IpSuffixOriginDhcp**
  The IP address suffix was configured by using DHCP. This value applies to an IPv4 address configured by using DHCP or an IPv6 address configured by using DHCPv6.

- **IpSuffixOriginLinkLayerAddress**
  The IP address suffix was the link-local address. This value applies to an IPv6 link-local address or an IPv6 address where the network part was generated based on a router advertisement and the host part was based on the MAC hardware address.

- **IpSuffixOriginRandom**
  The IP address suffix was generated randomly. This value applies to an anonymous IPv6 address where the host part of the address was generated randomly from the MAC hardware address after the host received a router advertisement.

- **IpSuffixOriginUnchanged**
  The IP address suffix should be unchanged. This value is used when setting the properties for a unicast IP interface when the value for the IP suffix origin should be unchanged.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Nldef.h (include Netioapi.h)

CancelMibChangeNotify2 function

Article • 03/03/2023

The **CancelMibChangeNotify2** function deregisters a driver change notification for IP interface changes, IP address changes, IP route changes, and requests to retrieve the stable Unicast IP address table.

Syntax

```c++
NETIOAPI_API CancelMibChangeNotify2(
  _In_ HANDLE NotificationHandle
);
```

Parameters

- *NotificationHandle* [in]
  The handle that is returned from a notification registration or retrieval function to indicate which notification to cancel.

Return value

CancelMibChangeNotify2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **CancelMibChangeNotify2** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. **CancelMibChangeNotify2** returns this error if the *NotificationHandle* parameter was a **NULL** pointer.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **CancelMibChangeNotify2** function deregisters a driver change notification previously requested for IP interface changes, IP address changes, or IP route changes on a local computer. These requests are made by calling NotifyIpInterfaceChange, NotifyRouteChange2, or NotifyUnicastIpAddressChange. The **CancelMibChangeNotify2** function also cancels a previous request to retrieve the stable unicast IP address table on a local computer. This request is made by calling the NotifyStableUnicastIpAddressTable function.

The *NotificationHandle* parameter that is returned to these notification functions is passed to **CancelMibChangeNotify2** to deregister driver change notifications or to cancel a pending request to retrieve the stable unicast IP address table.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	PASSIVE_LEVEL

See also

NotifyIpInterfaceChange

NotifyRouteChange2

NotifyStableUnicastIpAddressTable

NotifyUnicastIpAddressChange

NotifyIpInterfaceChange function

Article • 03/03/2023

The **NotifyIpInterfaceChange** function registers the driver to be notified for changes to all IP interfaces, IPv4 interfaces, or IPv6 interfaces on a local computer.

Syntax

C++

```
NETIOAPI_API NotifyIpInterfaceChange(
  _In_    ADDRESS_FAMILY               Family,
  _In_    PIPINTERFACE_CHANGE_CALLBACK Callback,
  _In_    PVOID                        CallerContext,
  _In_    BOOLEAN                      InitialNotification,
  _Inout_ HANDLE                       *NotificationHandle
);
```

Parameters

- *Family* [in]
  The address family to register the driver for change notifications on.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function registers the driver to be notified only for IPv4 change notifications.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function registers the driver for only IPv6 change notifications.

    - AF_UNSPEC
      The address family is unspecified. When this value is specified, this function registers the driver to be notified for both IPv4 and IPv6 changes.

- *Callback* [in]
  A pointer to the function to call when a change occurs. This function is called when an interface notification is received.

- *CallerContext* [in]
  A user context that is passed to the callback function that is specified in the *Callback* parameter when an interface notification is received.

- *InitialNotification* [in]
  A value that indicates whether the callback should be invoked immediately after registration for change notification completes. This initial notification does not indicate a change occurred to an IP interface. The purpose of this parameter to provide confirmation that the callback is registered.

- *NotificationHandle* [in, out]
  A pointer used to return a handle that can be later used to deregister the change notification. On success, a notification handle is returned in this parameter. If an error occurs, **NULL** is returned.

Return value

NotifyIpInterfaceChange returns STATUS_SUCCESS if the function succeeds.

If the function fails, **NotifyIpInterfaceChange** returns one of the following error codes:

Return code	Description
ERROR_INVALID_HANDLE	An internal error occurred where an invalid handle was encountered.
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if the *Family* parameter was not either AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	There was insufficient memory.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Your driver must set the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

The invocation of the callback function that is specified in the *Callback* parameter is serialized. The callback function should be defined as a function of type **VOID**. The parameters that are passed to the callback function include the following.

Parameter	Description
IN PVOID *CallerContext*	The *CallerContext* parameter that is passed to the **NotifyIpInterfaceChange** function when it is registering the driver for change notifications.
IN PMIB_IPINTERFACE_ROW *Row* OPTIONAL	A pointer to the MIB_IPINTERFACE_ROW entry for the interface that was changed. This parameter is a **NULL** pointer when the MIB_NOTIFICATION_TYPE value that is passed in the *NotificationType* parameter to the callback function is set to **MibInitialNotification**. This situation can occur only if the *InitialNotification* parameter that is passed to **NotifyIpInterfaceChange** was set to **TRUE** when registering the driver for change notifications.
IN MIB_NOTIFICATION_TYPE *NotificationType*	The notification type. This member can be one of the values from the MIB_NOTIFICATION_TYPE enumeration type.

To deregister the driver for change notifications, call the CancelMibChangeNotify2 function, passing the *NotificationHandle* parameter that **NotifyIpInterfaceChange** returns.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CancelMibChangeNotify2

GetIfEntry2

GetIfStackTable

GetIfTable2

GetInvertedIfStackTable

GetIpInterfaceEntry

InitializeIpInterfaceEntry

MIB_IF_ROW2

MIB_IF_TABLE2

MIB_IPINTERFACE_ROW

MIB_NOTIFICATION_TYPE

SetIpInterfaceEntry

NotifyRouteChange2 function

Article • 03/03/2023

The **NotifyRouteChange2** function registers the driver to be notified for changes to IP route entries on a local computer.

Syntax

C++

```
NETIOAPI_API NotifyRouteChange2(
  _In_    ADDRESS_FAMILY            Family,
  _In_    PIPFORWARD_CHANGE_CALLBACK Callback,
  _In_    PVOID                     CallerContext,
  _In_    BOOLEAN                   InitialNotification,
  _Inout_ HANDLE                    *NotificationHandle
);
```

Parameters

- *Family* [in]
  The address family to register the driver for change notifications on.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, this function registers the driver only for IPv4 route change notifications.

  - AF_INET6
    The IPv6 address family. When this value is specified, this function registers the driver only for IPv6 route change notifications.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, this function registers the driver for both IPv4 and IPv6 route change notifications.

- *Callback* [in]
  A pointer to the function to call when a change occurs. This function is called when an interface notification is received.

- *CallerContext* [in]
  A user context that is passed to the callback function that is specified in the *Callback* parameter when an interface notification is received.

- *InitialNotification* [in]
  A value that indicates whether the callback should be invoked immediately after registration for change notification completes. This initial notification does not indicate a change occurred to the IP route. The purpose of this parameter to provide confirmation that the callback is registered.

- *NotificationHandle* [in, out]
  A pointer to a MIB_IPINTERFACE_ROW structure to initialize. On successful return, the members in this structure are initialized with default information for an interface on the local computer.

Return value

NotifyRouteChange2 returns STATUS_SUCCESS if the function succeeds.

If the function fails, **NotifyRouteChange2** returns one of the following error codes.

Return code	Description
ERROR_INVALID_HANDLE	An internal error occurred where an invalid handle was encountered.
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if the *Family* parameter was not either AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	There was insufficient memory.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Your driver must set the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

The invocation of the callback function that is specified in the *Callback* parameter is serialized. The callback function should be defined as a function of type **VOID**. The parameters that are passed to the callback function include the following.

Parameter	Description
IN PVOID *CallerContext*	The *CallerContext* parameter that is passed to the **NotifyRouteChange2** function when it is registering the driver for change notifications.
IN PMIB_IPFORWARD_ROW2 *Row* OPTIONAL	A pointer to the MIB_IPFORWARD_ROW2 entry for the IP route entry that was changed. This parameter is a **NULL** pointer when the MIB_NOTIFICATION_TYPE value that is passed in the *NotificationType* parameter to the callback function is set to MibInitialNotification. This situation can occur only if the *InitialNotification* parameter that is passed to **NotifyRouteChange2** was set to **TRUE** when registering the driver for change notifications.
IN MIB_NOTIFICATION_TYPE *NotificationType*	The notification type. This member can be one of the values from the MIB_NOTIFICATION_TYPE enumeration type.

To deregister the driver for change notifications, call the CancelMibChangeNotify2 function, passing the *NotificationHandle* parameter that **NotifyRouteChange2** returns.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CancelMibChangeNotify2

CreateIpForwardEntry2

DeleteIpForwardEntry2

GetBestRoute2

GetIpForwardEntry2

GetIpForwardTable2

InitializeIpForwardEntry

MIB_IPFORWARD_ROW2

MIB_IPFORWARD_TABLE2

MIB_NOTIFICATION_TYPE

SetIpForwardEntry2

NotifyUnicastIpAddressChange function

Article • 03/03/2023

The **NotifyUnicastIpAddressChange** function registers the driver to be notified for changes to all unicast IP interfaces, unicast IPv4 addresses, or unicast IPv6 addresses on a local computer.

Syntax

C++

```
NETIOAPI_API NotifyUnicastIpAddressChange(
  _In_    ADDRESS_FAMILY                    Family,
  _In_    PUNICAST_IPADDRESS_CHANGE_CALLBACK Callback,
  _In_    PVOID                             CallerContext,
  _In_    BOOLEAN                           InitialNotification,
  _Inout_ HANDLE                            *NotificationHandle
);
```

Parameters

- *Family* [in]
  The address family to register the driver for change notifications on.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, the function registers the driver only for unicast IPv4 address change notifications.

  - AF_INET6
    The IPv6 address family. When this value is specified, the function registers the driver only for unicast IPv6 address change notifications.

- AF_UNSPEC
  The address family is unspecified. When this value is specified, the function registers the driver for both unicast IPv4 and IPv6 address change notifications.

- *Callback* [in]
  A pointer to the function to call when a change occurs. This function is called when a unicast IP address notification is received.

- *CallerContext* [in]
  A user context that is passed to the callback function that is specified in the *Callback* parameter when an interface notification is received.

- *InitialNotification* [in]
  A value that indicates whether the callback should be called immediately after registration for change notification completes. This initial notification does not indicate that a change occurred to a unicast IP address. This parameter provides confirmation that the callback is registered.

- *NotificationHandle* [in, out]
  A pointer that is used to return a handle that your driver can later use to deregister the driver change notification. On success, a notification handle is returned in this parameter. If an error occurs, **NULL** is returned.

Return value

NotifyUnicastIpAddressChange returns STATUS_SUCCESS if the function succeeds.

If the function fails, **NotifyUnicastIpAddressChange** returns one of the following error codes:

Return code	Description
ERROR_INVALID_HANDLE	An internal error occurred where an invalid handle was encountered.
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if the *Family* parameter was not either AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	There was insufficient memory.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

Your driver must set the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

The invocation of the callback function that is specified in the *Callback* parameter is serialized. The callback function should be defined as a function of type **VOID**. The parameters that are passed to the callback function include the following.

Parameter	Description
IN PVOID *CallerContext*	The *CallerContext* parameter that is passed to the **NotifyUnicastIpAddressChange** function when it is registering the driver for change notifications.
IN PMIB_UNICASTIPADDRESS_ROW *Row* OPTIONAL	A pointer to the MIB_UNICASTIPADDRESS_ROW entry for the unicast IP address that was changed. This parameter is a **NULL** pointer when the MIB_NOTIFICATION_TYPE value that is passed in the *NotificationType* parameter to the callback function is set to **MibInitialNotification**. This situation can occur only if the *InitialNotification* parameter that is passed to **NotifyUnicastIpAddressChange** was set to **TRUE** when registering the driver for change notifications.
IN MIB_NOTIFICATION_TYPE *NotificationType*	The notification type. This member can be one of the values from the MIB_NOTIFICATION_TYPE enumeration type.

To deregister the driver for change notifications, call the CancelMibChangeNotify2 function, passing the *NotificationHandle* parameter that **NotifyUnicastIpAddressChange** returns.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CancelMibChangeNotify2

CreateUnicastIpAddressEntry

DeleteUnicastIpAddressEntry

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

InitializeUnicastIpAddressEntry

MIB_NOTIFICATION_TYPE

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NotifyStableUnicastIpAddressTable

SetUnicastIpAddressEntry

GetTeredoPort function

Article • 03/03/2023

The **GetTeredoPort** function retrieves the dynamic UDP port number that the Teredo client uses on a local computer.

Syntax

```
C++

NETIOAPI_API GetTeredoPort(
  _Out_ USHORT *Port
);
```

Parameters

- *Port* [out]
  A pointer to the UDP port number. On successful return, this parameter is filled with the port number that the Teredo client uses.

Return value

GetTeredoPort returns STATUS_SUCCESS if the function succeeds.

If the function fails, **GetTeredoPort** returns one of the following error codes:

Return code	Description
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *Port* parameter.
ERROR_NOT_READY	The device is not ready. This error is returned if the Teredo client is not started on the local computer.
STATUS_NOT_SUPPORTED	The request is not supported. This error is returned if no IPv6 stack is located on the local computer.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The **GetTeredoPort** function retrieves the current UDP port number that the Teredo client usesfor the Teredo service port. The Teredo port is dynamic and can change any time that the Teredo client is restarted on the local computer. A driver can register to be notified when the Teredo service port changes by calling the **NotifyTeredoPortChange** function.

The Teredo client also uses static UDP port 3544 for listening to multicast traffic that is sent on multicast IPv4 address 224.0.0.253 as defined in RFC 4380. For more information, see Teredo: Tunneling IPv6 over UDPthrough Network Address Translations (NATs) .

The **GetTeredoPort** function is used primarily by firewall drivers in order to configure the appropriate exceptions to enable incoming and outgoing Teredo traffic.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

NotifyTeredoPortChange

NotifyStableUnicastIpAddressTable

NotifyStableUnicastIpAddressTable function

Article • 03/03/2023

The **NotifyStableUnicastIpAddressTable** function retrieves the stable unicast IP address table on a local computer.

Syntax

```
C++

NETIOAPI_API NotifyStableUnicastIpAddressTable(
  _In_    ADDRESS_FAMILY                          Family,
  _Out_   PMIB_UNICASTIPADDRESS_TABLE             *Table,
  _In_    PSTABLE_UNICAST_IPADDRESS_TABLE_CALLBACK CallerCallback,
  _In_    PVOID                                   CallerContext,
  _Inout_ HANDLE                                  *NotificationHandle
);
```

Parameters

- *Family* [in]
  The address family to retrieve.

  Possible values for the address family are listed in the Winsock2.h header file. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, AF_INET and PF_INET), so you can use either constant.

  On Windows Vista and later versions of the Windows operating systems, possible values for the *Family* parameter are defined in the Ws2def.h header file. Note that the Ws2def.h header file is automatically included in Netioapi.h and you should never use Ws2def.h directly.

  The following values are currently supported for the address family:

  - AF_INET
    The IPv4 address family. When this value is specified, the function retrieves the stable unicast IP address table that contains only IPv4 entries.

  - AF_INET6
    The IPv6 address family. When this value is specified, the function retrieves the

stable unicast IP address table that contains only IPv6 entries.

  - AF_UNSPEC
    The address family is unspecified. When this value is specified, the function retrieves the stable unicast IP address table that contains both IPv4 and IPv6 entries.

- *Table* [out]
  A pointer to a MIB_UNICASTIPADDRESS_TABLE structure. When **NotifyStableUnicastIpAddressTable** is successful, this parameter returns the stable unicast IP address table on the local computer.

  When **NotifyStableUnicastIpAddressTable** returns ERROR_IO_PENDING, which indicates that the I/O request is pending, the stable unicast IP address table is returned to the function in the *CallerCallback* parameter.

- *CallerCallback* [in]
  A pointer to the function to call with the stable unicast IP address table. This function is called if **NotifyStableUnicastIpAddressTable** returns ERROR_IO_PENDING, which indicates that the I/O request is pending.

- *CallerContext* [in]
  A user context that is passed to the callback function that is specified in the *CallerCallback* parameter when the stable unicast IP address table is available.

- *NotificationHandle* [in, out]
  A pointer that is used to return a handle that your driver can use to cancel the request to retrieve the stable unicast IP address table. This parameter is returned if the return value from **NotifyStableUnicastIpAddressTable** is ERROR_IO_PENDING, which indicates that the I/O request is pending.

Return value

NotifyStableUnicastIpAddressTable returns STATUS_SUCCESS and the stable unicast IP table is returned in the *Table* parameter if the function succeeds immediately.

If the I/O request is pending, the function returns ERROR_IO_PENDING and the function that the *CallerCallback* parameter points to is called when the I/O request has completed with the stable unicast IP address table.

If the function fails, **NotifyStableUnicastIpAddressTable** returns one of the following error codes:

Return code	Description
ERROR_INVALID_HANDLE	An internal error occurred where an invalid handle was encountered.
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if the *Table* parameter was a **NULL** pointer, the *NotificationHandle* parameter was a **NULL** pointer, or the *Family* parameter was not either AF_INET, AF_INET6, or AF_UNSPEC.
STATUS_NOT_ENOUGH_MEMORY	There was insufficient memory.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

All unicast IP addresses, except dial-on-demand addresses, are considered stable only if they are in the preferred state. For a normal unicast IP address entry, this state would correspond to a **DadState** member of the **MIB_UNICASTIPADDRESS_ROW** for the IP address that is set to **IpDadStatePreferred**. Every dial-on-demand address defines its own stability metric. Currently the only dial-on-demand address that the **NotifyStableUnicastIpAddressTable** function considers is the unicast IP address that the Teredo client uses on the local computer.

Your driver must set the *Family* parameter to either AF_INET, AF_INET6, or AF_UNSPEC.

When **NotifyStableUnicastIpAddressTable** is successful and returns STATUS_SUCCESS, the *Table* parameter returns the stable unicast IP address table on the local computer.

When **NotifyStableUnicastIpAddressTable** returns ERROR_IO_PENDING, which indicates that the I/O request is pending, the stable unicast IP address table is returned to the function in the *CallerCallback* parameter.

If the unicast IP address that Teredo uses is available on the local computer but not in the stable (qualified) state, **NotifyStableUnicastIpAddressTable** returns ERROR_IO_PENDING and the stable unicast IP address table is eventually returned by calling the function in the *CallerCallback* parameter. If the Teredo address is not available or is in the stable state and the other unicast IP addresses are in a stable state, the function in the *CallerCallback* parameter is never called.

The callback function that is specified in the *CallerCallback* parameter should be defined as a function of type **VOID**. The parameters that are passed to the callback function include the following.

Parameter	Description
IN PVOID *CallerContext*	The *CallerContext* parameter that is passed to the **NotifyStableUnicastIpAddressTable** function when it is registering the driver for notifications.
IN PMIB_UNICASTIPADDRESS_TABLE *AddressTable*	A pointer to a MIB_UNICASTIPADDRESS_TABLE structure that contains the stable unicast IP address table on the local computer.

The **NotifyStableUnicastIpAddressTable** function is used primarily by drivers that use the Teredo client.

To cancel the notification after the callback is complete, call the CancelMibChangeNotify2 function, passing the *NotificationHandle* parameter that **NotifyStableUnicastIpAddressTable** returns.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CancelMibChangeNotify2

CreateUnicastIpAddressEntry

DeleteUnicastIpAddressEntry

GetTeredoPort

GetUnicastIpAddressEntry

GetUnicastIpAddressTable

InitializeUnicastIpAddressEntry

MIB_NOTIFICATION_TYPE

MIB_UNICASTIPADDRESS_ROW

MIB_UNICASTIPADDRESS_TABLE

NotifyTeredoPortChange

NotifyUnicastIpAddressChange

SetUnicastIpAddressEntry

NotifyTeredoPortChange function

Article • 03/03/2023

The **NotifyTeredoPortChange** function registers the driver to be notified for changes to the UDP port number that the Teredo client uses for the Teredo service port on a local computer.

Syntax

C++

```
NETIOAPI_API NotifyTeredoPortChange(
  _In_    PTEREDO_PORT_CHANGE_CALLBACK Callback,
  _In_    PVOID                        CallerContext,
  _In_    BOOLEAN                      InitialNotification,
  _Inout_ HANDLE *                     NotificationHandle
);
```

Parameters

- *Callback* [in]
  A pointer to the function to call when a Teredo client port change occurs. This function is called when a Teredo port change notification is received.

- *CallerContext* [in]
  A user context that is passed to the callback function that is specified in the Callback parameter when a Teredo port change notification is received.

- *InitialNotification* [in]
  A value that indicates whether the callback should be called immediately after registration for driver change notification completes. This initial notification does not indicate that a change occurred to the Teredo client port. This parameter provides confirmation that the callback is registered.

- *NotificationHandle* [in, out]
  A pointer that is used to return a handle that your driver can later use to deregister the driver change notification. On success, a notification handle is returned in this parameter. If an error occurs, **NULL** is returned.

Return value

NotifyTeredoPortChange returns STATUS_SUCCESS if the function succeeds.

If the function fails, **NotifyTeredoPortChange** returns one of the following error codes:

Return code	Description
ERROR_INVALID_HANDLE	An internal error occurred where an invalid handle was encountered.
STATUS_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if the *Callback* parameter is a **NULL** pointer.
STATUS_NOT_ENOUGH_MEMORY	There was insufficient memory.
Other	Use the FormatMessage function to obtain the message string for the returned error.

Remarks

The invocation of the callback function that is specified in the *Callback* parameter is serialized. The callback function should be defined as a function of type **VOID**. The parameters that are passed to the callback function include the following.

Parameter	Description
IN PVOID *CallerContext*	The *CallerContext* parameter that is passed to the **NotifyTeredoPortChange** function when it is registering the driver for change notifications.
IN USHORT *Port*	The UDP port number that the Teredo client currently uses. This parameter is zero when the **MIB_NOTIFICATION_TYPE** value that is passed in the *NotificationType* parameter to the callback function is set to **MibInitialNotification**. This situation can occur only if the *InitialNotification* parameter that is passed to **NotifyTeredoPortChange** was set to **TRUE** when registering the driver for change notifications.
IN MIB_NOTIFICATION_TYPE *NotificationType*	The notification type. This member can be one of the values from the **MIB_NOTIFICATION_TYPE** enumeration type.

Your driver can use the **GetTeredoPort** function to retrieve the initial UDP port number that the Teredo client used for the Teredo service port.

The Teredo port is dynamic and can change any time that the Teredo client is restarted on the local computer. A driver can register to be notified when the Teredo service port changes by calling the **NotifyTeredoPortChange** function.

The Teredo client also uses static UDP port 3544 for listening to multicast traffic that is sent on multicast IPv4 address 224.0.0.253 as defined in RFC 4380. For more information, see Teredo: Tunneling IPv6 over UDPthrough Network Address Translations (NATs) .

The **NotifyTeredoPortChange** function is used primarily by firewall drivers to configure the appropriate exceptions to enable incoming and outgoing Teredo traffic.

To deregister the driver for change notifications, call the **CancelMibChangeNotify2** function, passing the *NotificationHandle* parameter that the **NotifyTeredoPortChange** function returns.

Requirements

Target platform	Universal
Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Netioapi.h (include Netioapi.h)
Library	Netio.lib
IRQL	< DISPATCH_LEVEL

See also

CancelMibChangeNotify2

GetTeredoPort

NotifyStableUnicastIpAddressTable

Roadmap for Developing WFP Callout Drivers

Article • 11/29/2022

To create a Windows Filtering Platform (WFP) callout driver, follow these steps:

- Step 1: Learn about WFP architecture.

  For information about WFP, see Windows Filtering Platform. You may find that you can develop a WFP user-mode application and avoid writing a WFP callout driver.

- Step 2: Learn about Windows architecture and drivers.

  You must understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and let you streamline your development process. For more information about driver fundamentals, see Concepts for all driver developers.

- Step 3: Determine the Windows driver model for your WFP callout driver.

  WFP callout drivers can be written either by using the Windows Driver Model (WDM) or the Kernel Mode Driver Framework (KMDF). For more information about how to select a driver model, see Choosing a Driver Model. For more information about WDM, see Introduction to Windows Drivers and Writing WDM Drivers. For more information about KMDF, see WDF Driver Development Guide.

- Step 4: Determine additional Windows driver design decisions.

  For information about how to make additional Windows design decisions, see Creating Reliable Kernel-Mode Drivers, Programming Issues for 64-Bit Drivers, and Creating International INF Files.

- Step 5: Learn about the Windows driver build, test, and debug processes and tools.

  Building a driver differs from building a user-mode application. For information about Windows driver build, debug, and test processes, driver signing, and Windows Hardware Lab Kit (HLK) testing, see Building, Debugging, and Testing Drivers. For information about building, testing, verifying, and debugging tools, see Driver Development Tools.

- Step 6: Review the Windows Filtering Platform (WFP) driver samples in the Windows driver samples repository on GitHub.

- Step 7: Make design decisions about your WFP callout driver.

  For information about how to design WFP callout drivers, see Callout Driver Programming Considerations.

- Step 8: Develop, build, test, and debug your WFP callout driver.

  For information about WFP callout driver specifics, see Callout Driver Operations and Callout Driver Installation. For information about functions, structures, enumerations, or constants that are specific to WFP, see Windows Filtering Platform Callout Drivers Reference. For information about iterative building, testing, and debugging, see Overview of Build, Debug, and Test Process. This process will help ensure that you build a driver that works.

- Step 9: Create a driver package for your WFP callout driver.

  For more information, see Providing a Driver Package and Callout Driver Installation.

- Step 10: Sign and distribute your WFP callout driver.

  The final step is to sign (optional) and distribute the driver. If your driver meets the quality standards that are defined for the Windows Hardware Lab Kit (HLK), you can distribute it through the Microsoft Windows Update program. For more information about how to distribute a driver, see Get started with the hardware submission process.

These are the basic steps. Additional steps might be necessary based on the needs of your individual driver.

Introduction to Windows Filtering Platform (WFP) Callout Drivers

Article • 09/27/2024

This section introduces WFP callout drivers.

For more information about the WFP, see the Windows Filtering Platform documentation in the Microsoft Windows SDK.

For WFP reference information, see Windows Filtering Platform Callout Drivers.

Purpose of Callout Drivers

A callout driver implements one or more callouts. Callouts extend the capabilities of the Windows Filtering Platform by processing TCP/IP-based network data in ways that are beyond the scope of the simple filtering functionality. Callouts are typically used to do the following tasks:

Deep Inspection
Perform complex inspection of the network data to determine which data should be blocked, which data should be permitted, and which data should be passed to another filter. An antivirus product, for example, could look for virus signatures.

Packet Modification
Perform modification and reinjection of the network packet headers or data, or both. A network address translation (NAT) product, for example, could modify the headers on IPv4 packets.

Stream Modification
Perform modification and reinjection of the network data in a stream. A parental control product, for example, could remove or replace specific words or phrases in a data stream.

Data Logging
Log of network traffic data. A network monitoring product, for example, could count the number of data packets that are discarded for a specific reason.

In addition to processing network data, callout drivers can perform other Windows Filtering Platform management tasks, such as adding filters to the base filtering engine. For more information about other tasks that a callout driver can perform, see Calling Other Windows Filtering Platform Functions.

Feedback

Was this page helpful? 👍 Yes 👎 No

Provide product feedback | Get help at Microsoft Q&A

WFP Changes for Windows 8

Article • 12/15/2021

Several changes have been made in the available functions and behavior of the Windows Filtering Platform that begin with Windows 8. Frequently, to take advantage of the new features, you must compile or recompile a callout driver that has the NTDDI_VERSION macro set to NTDDI_WIN8.

- New features:
  - Using Layer 2 Filtering
  - Using Proxied Connections Tracking
  - Using Virtual Switch Filtering
- New functions:
  - **FwpsCalloutRegister2**
  - **FwpsFlowAbort0**
  - **FwpsInjectMacReceiveAsync0**
  - **FwpsInjectMacSendAsync0**
  - **FwpsNetBufferListAssociateContext1**
  - **FwpsQueryConnectionRedirectState0**
  - **FwpsRedirectHandleCreate0**
  - **FwpsRedirectHandleDestroy0**
  - **FwpsvSwitchEventsSubscribe0**
  - **FwpsvSwitchEventsUnsubscribe0**
  - **FwpsvSwitchNotifyComplete0**
- New callback functions:
  - *FWPS_CALLOUT_CLASSIFY_FN2*
  - *FWPS_CALLOUT_NOTIFY_FN2*
  - *FWPS_NET_BUFFER_LIST_NOTIFY_FN1*
  - *FWPS_VSWITCH_FILTER_ENGINE_REORDER_CALLBACK0*
  - *FWPS_VSWITCH_INTERFACE_EVENT_CALLBACK0*
  - *FWPS_VSWITCH_LIFETIME_EVENT_CALLBACK0*
  - *FWPS_VSWITCH_POLICY_EVENT_CALLBACK0*
  - *FWPS_VSWITCH_PORT_EVENT_CALLBACK0*
  - *FWPS_VSWITCH_RUNTIME_STATE_RESTORE_CALLBACK0*
  - *FWPS_VSWITCH_RUNTIME_STATE_SAVE_CALLBACK0*
- New structures:
  - **FWPS_FILTER2**
  - **FWPS_VSWITCH_EVENT_DISPATCH_TABLE0**
- New enumerations:
  - **FWPS_CONNECTION_REDIRECT_STATE**

- FWPS_FIELDS_EGRESS_VSWITCH_ETHERNET
- FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V4
- FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V6
- FWPS_FIELDS_INBOUND_MAC_FRAME_NATIVE
- FWPS_FIELDS_INGRESS_VSWITCH_ETHERNET
- FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V4
- FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V6
- FWPS_FIELDS_OUTBOUND_MAC_FRAME_NATIVE
- FWPS_VSWITCH_EVENT_TYPE

- Renamed enumerations:
  - FWPS_FIELDS_INBOUND_MAC_FRAME_ETHERNET (was FWPS_FIELDS_INBOUND_MAC_FRAME_802_3)
  - FWPS_FIELDS_OUTBOUND_MAC_FRAME_ETHERNET (was FWPS_FIELDS_OUTBOUND_MAC_FRAME_802_3)

WFP Changes for Windows 7

Article • 12/15/2021

Several changes have been made in the available functions and behavior of the Windows Filtering Platform that begin with Windows 7. Frequently, to take advantage of the new features, you must compile or recompile a callout driver that has the NTDDI_VERSION macro set to NTDDI_WIN7.

- New functions:
  - FwpsAcquireClassifyHandle0
  - FwpsAcquireWritableLayerDataPointer0
  - FwpsApplyModifiedLayerData0
  - FwpsCalloutRegister1
  - FwpsCompleteClassify0
  - FwpsPendClassify0
  - FwpsReleaseClassifyHandle0
  - *classifyFn1*
  - *notifyFn1*
  - FWPS_NET_BUFFER_LIST_NOTIFY_FN0
  - FwpsInjectTransportSendAsync1
  - FwpsNetBufferListAssociateContext0
  - FwpsNetBufferListGetTagForContext0
  - FwpsNetBufferListRemoveContext0
  - FwpsNetBufferListRetrieveContext0
  - FwpsAleEndpointCreateEnumHandle0
  - FwpsAleEndpointDestroyEnumHandle0
  - FwpsAleEndpointEnum0
  - FwpsAleEndpointGetById0
  - FwpsAleEndpointGetSecurityInfo0
  - FwpsAleEndpointSetSecurityInfo0
- New structures and enumerations:
  - FWPS_ALE_ENDPOINT_ENUM_TEMPLATE0
  - FWPS_ALE_ENDPOINT_PROPERTIES0
  - FWPS_BIND_REQUEST0
  - FWPS_CALLOUT1
  - FWPS_CONNECT_REQUEST0
  - FWPS_FIELDS_ALE_BIND_REDIRECT_V4
  - FWPS_FIELDS_ALE_BIND_REDIRECT_V6
  - FWPS_FIELDS_ALE_CONNECT_REDIRECT_V4
  - FWPS_FIELDS_ALE_CONNECT_REDIRECT_V6

- FWPS_FIELDS_ALE_ENDPOINT_CLOSURE_V4
    - FWPS_FIELDS_ALE_ENDPOINT_CLOSURE_V6
    - FWPS_FIELDS_ALE_RESOURCE_RELEASE_V4
    - FWPS_FIELDS_ALE_RESOURCE_RELEASE_V6
    - FWPS_FIELDS_INBOUND_MAC_FRAME_802_3
    - FWPS_FIELDS_KM_AUTHORIZATION
    - FWPS_FIELDS_NAME_RESOLUTION_CACHE_V4
    - FWPS_FIELDS_NAME_RESOLUTION_CACHE_V6
    - FWPS_FIELDS_OUTBOUND_MAC_FRAME_802_3
    - FWPS_FIELDS_STREAM_PACKET_V4
    - FWPS_FIELDS_STREAM_PACKET_V6
    - FWPS_FILTER1
    - FWPS_NET_BUFFER_LIST_EVENT_TYPE0
    - FWPS_TRANSPORT_SEND_PARAMS1
- New documentation topics:
    - Using Bind or Connect Redirection
    - Using Packet Tagging
    - ALE Endpoint Lifetime Management

WFP Changes for Windows Vista SP1 and Windows Server 2008

Article • 12/15/2021

Several changes have been made in the available functions and behavior of the Windows Filtering Platform that begin with Windows Vista with Service Pack 1 (SP1) and Windows Server 2008. Frequently, to take advantage of the new features, you must compile or recompile a callout driver that has the NTDDI_VERSION macro set to NTDDI_WIN6SP1.

- New functions: **FwpsConstructIpHeaderForTransportPacket0** **FwpsReassembleForwardFragmentGroup0**

- New FWPS_STREAM_FLAG_RECEIVE_PUSH flag option that is described in **FwpsStreamInjectAsync0**

- Updated and renamed filtering conditions, listed in Filtering Conditions Available at Each Filtering Layer

- Updated and renamed data field identifiers that were added to FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_*XXX* and FWPS_LAYER_INBOUND_ICMP_ERROR_*XXX* layers, listed in Data Field Identifiers, together with behavior changes

- Additional metadata field identifiers, listed in Metadata Fields and Metadata Fields at Each Filtering Layer

- The following documentation topics are new:
  - Developing IPsec-Compatible Callout Drivers
  - Processing Classify Callouts Asynchronously

- The following topics contain additional updates: Processing Notify Callouts Stream Inspection **FwpsFlowAssociateContext0** **FwpsFlowRemoveContext0** *classifyFn* *notifyFn* **FWPS_CALLOUT0** **FWPS_INCOMING_METADATA_VALUES0**

Callout

Article • 12/15/2021

A *callout* provides functionality that extends the capabilities of the Windows Filtering Platform. A callout consists of a set of callout functions and a GUID key that uniquely identifies the callout. There are several built-in callouts that are included with the Windows Filtering Platform. Additional callouts can be added by using callout drivers.

Callout Driver

Article • 12/15/2021

A *callout driver* is a kernel-mode driver that implements one or more callouts. A callout driver registers its callouts with the filter engine so that the filter engine can call the callout functions for the callout when the computer processes connections or packets.

Callout Function

Article • 12/15/2021

A *callout function* is a function that is implemented by a callout driver that is one of the functions that defines a callout. A callout consists of the following list of callout functions:

- A *notifyFn* function to process notifications.
- A *classifyFn* function to process classifications.
- A *flowDeleteFn* function to process flow deletions (optional).

The filter engine calls a callout's callout functions so that the callout can process the network data.

Filter

Article • 12/15/2021

A *filter* defines several filtering conditions for filtering TCP/IP network data and an action that is to be taken on the data if all the filtering conditions are true. If a filter requires additional processing of the network data, it can specify a callout for the filter's action. If the filtering conditions for such a filter are all true, the filter engine passes the network data to the specified callout for additional processing.

Filter Engine

Article • 12/15/2021

The *filter engine* is a component of the Windows Filtering Platform that stores filters and performs filter arbitration. Filters are added to the filter engine at designated filtering layers so that the filter engine can perform the desired filtering action (permit, drop, or a callout). If a filter in the filter engine specifies a callout for the filter's action, the filter engine calls the callout's *classifyFn* function so that the callout can process the network data.

Filtering Layer

Article • 12/15/2021

A *filtering layer* is a point in the TCP/IP network stack where network data is passed to the filter engine for matching against the current set of filters. Each filtering layer in the network stack is identified by a unique filtering layer identifier.

When a filter is added to the filter engine, it is added at a designated filtering layer where it will filter the network data. Specific data fields are made available at each filtering layer for processing by the filters that have been added to the filter engine at that layer. If the filter engine passes the network data to a callout for additional processing, it includes these data fields and any metadata that is available at that filtering layer.

Run-time Filtering Layer Identifiers (FWPS_*XXX*) are used by kernel-mode callout drivers. Management Filtering Layer Identifiers (FWPM_*XXX*) are used by **Fwpm*Xxx*** functions that interact with the Base Filtering Engine (BFE) from either user mode or kernel mode (for example, **FwpmFilterAdd0**).

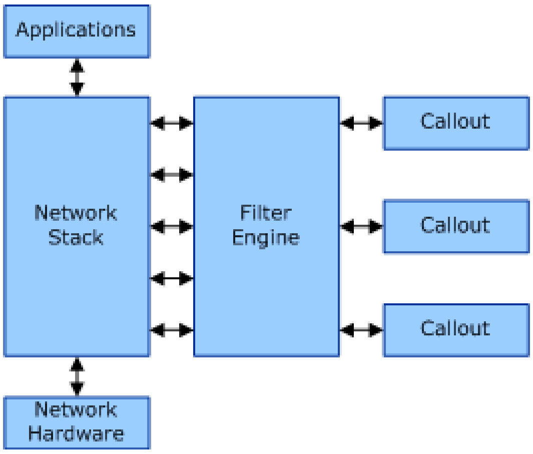
The FWPS data types are smaller than their FWPM counterparts: the FWPM filtering layer identifiers are GUIDs (128 bits), whereas the FWPS filtering layer identifiers are **LUIDs**(64 bits). The smaller size for FWPS data types improves system performance because integer comparisons are faster than GUID comparisons for real-time traffic, and the kernel memory handles FWPS types more efficiently.

Windows Filtering Platform Architecture Overview

Article • 12/15/2021

This section provides a brief overview of the Windows Filtering Platform architecture. For a more thorough discussion of the Windows Filtering Platform architecture, see the Windows Filtering Platform documentation in the Microsoft Windows SDK.

The following figure shows the basic architecture of the Windows Filtering Platform.



The filter engine is the core of the Windows Filtering Platform. The filter engine performs all the filtering operations on the TCP/IP-based network data. At key points in the TCP/IP stack there are filtering layers where network data is passed to the filter engine for processing. If the filtering conditions for a filter of the filtering layer are all true, the filter engine applies the filter's action.

Callout drivers provide additional filtering functionality by registering one or more callouts with the filter engine. Filters in the filter engine can specify a callout for the filter's action. In this case, the filter engine passes the network data to the specified callout for additional processing.

The Windows Filtering Platform includes several built-in callouts. See Built-in Callout Identifiers for a description of each of these callouts.

Callout Driver Operations Topics

Article • 12/15/2021

This section discusses typical callout driver operations and includes the following topics:

Initializing a Callout Driver

Processing Notify Callouts

Processing Classify Callouts

Processing Flow Delete Callouts

Using Packet Tagging

Using Layer 2 Filtering

Using Proxied Connections Tracking

Using Virtual Switch Filtering

Unloading a Callout Driver

In addition to these operations, callout drivers can perform other Windows Filtering Platform management tasks, such as adding filters to the base filtering engine. For more information about the tasks that a callout driver can perform, see Calling Other Windows Filtering Platform Functions.

Initializing a Callout Driver

Article • 12/15/2021

A callout driver initializes itself within its **DriverEntry** function. The main initialization tasks are as follows:

- Specifying an Unload Function
- Creating a Device Object
- Registering Callouts with the Filter Engine

Specifying an Unload Function

Article • 12/15/2021

A callout driver must provide an unload function. The operating system calls this function when the callout driver is unloaded from the system. A callout driver's unload function must guarantee that the callout driver's callouts are unregistered from the filter engine before the callout driver is unloaded from system memory. A callout driver cannot be unloaded from the system if it does not provide an unload function.

How a callout driver specifies an unload function depends on whether the callout driver is based on the Windows Driver Model (WDM) or the Windows Driver Frameworks (WDF).

WDM-Based Callout Drivers

If a callout driver is based on WDM, it specifies an **Unload** function in its **DriverEntry** function. For example:

C++

```
VOID
 Unload(
    IN PDRIVER_OBJECT DriverObject
    );

NTSTATUS
 DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
  ...

  // Specify the callout driver's Unload function
 DriverObject->DriverUnload = Unload;

  ...
}
```

WDF-Based Callout Drivers

If a callout driver is based on WDF, it specifies an *EvtDriverUnload* function in its **DriverEntry** function. For example:

C++

```
VOID
 Unload(
    IN WDFDRIVER Driver
    );

NTSTATUS
 DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
  NTSTATUS status;
  WDF_DRIVER_CONFIG config;
  WDFDRIVER driver;

  ...

  // Initialize the driver config structure
  WDF_DRIVER_CONFIG_INIT(&config, NULL);

  // Indicate that this is a non-PNP driver
 config.DriverInitFlags = WdfDriverInitNonPnpDriver;

  // Specify the callout driver's Unload function
 config.EvtDriverUnload = Unload;

  // Create a WDFDRIVER object
 status =
 WdfDriverCreate(
 DriverObject,
 RegistryPath,
      NULL,
      &config,
      &driver
      );

  ...

 return status;
}
```

For information about how to implement a callout driver's unload function, see Unloading a Callout Driver.

Creating a Device Object (Windows Filtering Platform)

Article • 12/15/2021

A callout driver must create a device object before it can register its callouts with the filter engine. How a callout driver creates a device object depends on whether the callout driver is based on the Windows Driver Model (WDM) or the Windows Driver Frameworks (WDF).

WDM-Based Callout Drivers

If a callout driver is based on WDM, it creates a device object by calling the IoCreateDevice function. For example:

```C++
PDEVICE_OBJECT deviceObject;

NTSTATUS
 DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
  NTSTATUS status;

  ...

  // Create a device object
 status =
 IoCreateDevice(
 DriverObject,
      0,
      NULL,
      FILE_DEVICE_UNKNOWN,
      FILE_DEVICE_SECURE_OPEN,
      FALSE,
      &deviceObject
      );

  ...

 return status;
}
```

WDF-Based Callout Drivers

If a callout driver is based on WDF, it creates a framework device object by calling the WdfDeviceCreate function. To register its callouts with the filter engine, a WDF-based callout driver must obtain a pointer to the WDM device object that is associated with the framework device object. A WDF-based callout driver obtains a pointer to this WDM device object by calling the WdfDeviceWdmGetDeviceObject function. For example:

C++

```
WDFDEVICE wdfDevice;

NTSTATUS
 DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
  WDFDRIVER driver;
  PWDFDEVICE_INIT deviceInit;
  PDEVICE_OBJECT deviceObject;
  NTSTATUS status;

  ...

  // Allocate a device initialization structure
 deviceInit =
 WdfControlDeviceInitAllocate(
    driver;
    &SDDL_DEVOBJ_KERNEL_ONLY
    );

  // Set the device characteristics
 WdfDeviceInitSetCharacteristics(
    deviceInit,
    FILE_DEVICE_SECURE_OPEN,
    FALSE
    );

  // Create a framework device object
 status =
 WdfDeviceCreate(
    &deviceInit,
    WDF_NO_OBJECT_ATTRIBUTES,
    &wdfDevice
    );

  // Check status
 if (status == STATUS_SUCCESS) {

    // Initialization of the framework device object is complete
    WdfControlFinishInitializing(
      wdfDevice
    );

    // Get the associated WDM device object
    deviceObject = WdfDeviceWdmGetDeviceObject(wdfDevice);
  }

  ...

 return status;
}
```

Registering Callouts with the Filter Engine

Article • 12/15/2021

After a callout driver has created a device object, it can then register its callouts with the filter engine. A callout driver can register its callouts with the filter engine at any time, even if the filter engine is currently not running. To register a callout with the filter engine, a callout driver calls the **FwpsCalloutRegister0** function. For example:

```C++
// Prototypes for the callout's callout functions
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
    IN OUT FWPS_CLASSIFY_OUT0  *classifyOut
    );

NTSTATUS NTAPI
 NotifyFn(
 IN FWPS_CALLOUT_NOTIFY_TYPE notifyType,
    IN const GUID  *filterKey,
    IN const FWPS_FILTER0  *filter
    );

VOID NTAPI
 FlowDeleteFn(
    IN UINT16  layerId,
    IN UINT32  calloutId,
    IN UINT64  flowContext
    );

// Callout registration structure
const FWPS_CALLOUT0 Callout =
{
 { ... }, // GUID key identifying the callout
  0,       // Callout-specific flags (none set here)
 ClassifyFn,
 NotifyFn,
 FlowDeleteFn
};

// Variable for the run-time callout identifier
UINT32 CalloutId;
```

```
NTSTATUS
 DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
  PDEVICE_OBJECT deviceObject;
  NTSTATUS status;

  ...

 status =
 FwpsCalloutRegister0(
 deviceObject,
      &Callout,
      &CalloutId
      );

  ...

 return status;
}
```

If the call to the FwpsCalloutRegister0 function is successful, the variable pointed to by the last parameter contains the run-time identifier for the callout. This run-time identifier corresponds to the GUID that was specified for the callout key.

A single callout driver can implement more than one callout. If a callout driver implements more than one callout, it calls the FwpsCalloutRegister0 function one time for each callout that it supports to register each callout with the filter engine.

Related topics

classifyFn

Processing Notify Callouts

Article • 12/15/2021

The filter engine calls a callout's *notifyFn* callout function to notify the callout driver about events that are associated with the callout.

Filter Addition

When a filter that specifies a callout for the filter's action is added to the filter engine, the filter engine calls the callout's *notifyFn* callout function, passing FWPS_CALLOUT_NOTIFY_ADD_FILTER in the *notifyType* parameter.

A callout driver can register a callout with the filter engine after filters that specify the callout for the filter's action have already been added to the filter engine. In this situation, the filter engine does not call the callout's *notifyFn* callout function to notify the callout about any of the existing filters.

The filter engine only calls the callout's *notifyFn* callout function to notify the callout when new filters that specify the callout for the filter's action are added to the filter engine. In this situation, a callout's *notifyFn* callout function might not get called for every filter in the filter engine that specifies the callout for the filter's action.

If a callout driver registers a callout after the filter engine is started and the callout must receive information about every filter in the filter engine that specifies the callout for the filter's action, the callout driver must call the appropriate management functions to enumerate all the filters in the filter engine. The callout driver must sort through the resulting list of all the filters to find those filters that specify the callout for the filter's action. See Calling Other Windows Filtering Platform Functions for more information about calling these functions.

Filter Deletion

When a filter that specifies a callout for the filter's action is deleted from the filter engine, the filter engine calls the callout's *notifyFn* callout function and passes FWPS_CALLOUT_NOTIFY_DELETE_FILTER in the *notifyType* parameter and **NULL** in the *filterKey* parameter. The filter engine calls the callout's *notifyFn* callout function for every deleted filter in the filter engine that specifies the callout for the filter's action. This includes any filters that were added to the filter engine before the callout driver registered the callout with the filter engine. Therefore, a callout might receive filter delete notifications for filters for which it did not receive filter add notifications.

If the callout's *notifyFn* callout function does not recognize the kind of notification that is passed in the *notifyType* parameter, it should ignore the notification and return STATUS_SUCCESS.

A callout driver can specify a context to be associated with a filter when the filter is added to the filter engine. Such a context is opaque to the filter engine. The callout's *classifyFn* callout function can use this context to save state information for the next time that it is called by the filter engine. When the filter is deleted from the filter engine, the callout driver performs any necessary cleanup of the context.

For example:

```C++
// Context structure to be associated with the filters
typedef struct FILTER_CONTEXT_ {
  .
  .  // Driver-specific content
  .
} FILTER_CONTEXT, *PFILTER_CONTEXT;

// Memory pool tag for filter context structures
#define FILTER_CONTEXT_POOL_TAG 'fcpt'

// notifyFn callout function
NTSTATUS NTAPI
 NotifyFn(
    IN FWPS_CALLOUT_NOTIFY_TYPE  notifyType,
    IN const GUID  *filterKey,
    IN const FWPS_FILTER0  *filter
    )
{
  PFILTER_CONTEXT context;

 ASSERT(filter != NULL);

  // Switch on the type of notification
 switch(notifyType) {

    // A filter is being added to the filter engine
 case FWPS_CALLOUT_NOTIFY_ADD_FILTER:

      // Allocate the filter context structure
 context =
        (PFILTER_CONTEXT)ExAllocatePoolWithTag(
 NonPagedPool,
 sizeof(FILTER_CONTEXT),
          FILTER_CONTEXT_POOL_TAG
          );

      // Check the result of the memory allocation
 if (context == NULL) {
        // Return error
 return STATUS_INSUFFICIENT_RESOURCES;
      }

      // Initialize the filter context structure
      ...

      // Associate the filter context structure with the filter
 filter->context = (UINT64)context;

 break;

    // A filter is being removed from the filter engine
 case FWPS_CALLOUT_NOTIFY_DELETE_FILTER:

      // Get the filter context structure from the filter
 context = (PFILTER_CONTEXT)filter->context;

 // Check whether the filter has a context
 if (context) {

        // Cleanup the filter context structure
        ...

        // Free the memory for the filter context structure
 ExFreePoolWithTag(
 context,
         FILTER_CONTEXT_POOL_TAG
         );

      }
 break;

    // Unknown notification
 default:

      // Do nothing
 break;
  }

 return STATUS_SUCCESS;
}
```

Processing Classify Callouts

Article • 12/15/2021

The filter engine calls a callout's *classifyFn* callout function when there is network data to be processed by the callout. This occurs when all the filtering conditions are true for a filter that specifies the callout for the filter's action. If such a filter has no filtering conditions, the filter engine always calls the callout's *classifyFn* callout function.

The filter engine passes several different data items to a callout's *classifyFn* callout function. These data items include fixed data values, metadata values, raw network data, filter information, and any flow context. The particular data items that the filter engine passes to the callout depend on the specific filtering layer and the conditions under which *classifyFn* is called. A *classifyFn* function can use any combination of these data items to make its filtering decisions.

The implementation of a callout's *classifyFn* callout function depends on what the callout is designed to do. The following sections provide examples of some the more typical functions of a callout:

Using a Callout for Deep Inspection

Using a Callout for Deep Inspection of Stream Data

Inspecting Packet and Stream Data

Modifying Stream Data

Data Logging

Associating Context with a Data Flow

Processing Classify Callouts Asynchronously

Using Bind or Connect Redirection

ALE Endpoint Lifetime Management

Using Packet Tagging

The actual implementation of a particular callout's *classifyFn* callout function can be based on a combination of these examples.

For callouts that process data at a filtering layer that supports data flows, the callout's *classifyFn* callout function can associate a context with each of the data flows. The *classifyFn* function can use this context to save state information for the next time that it is called by the filter engine for that data flow. For more information about how a callout function can associate a context with a data flow, see Associating Context with a Data Flow.

WFP supports asynchronous processing of the *classifyFn* callout function. For more information about asynchronous processing, see Processing Classify Callouts Asynchronously.

Using a Callout for Deep Inspection

Article • 12/15/2021

When a callout is performing deep inspection, its classifyFn callout function can inspect any combination of the fixed data fields, the metadata fields, and any raw packet data that is passed to it, and any relevant data that has been stored in a context associated with the filter or the data flow.

For example:

```C++
// classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
    IN OUT FWPS_CLASSIFY_OUT  *classifyOut
    )
{
  PNET_BUFFER_LIST rawData;
  ...

  // Test for the FWPS_RIGHT_ACTION_WRITE flag to check the rights
  // for this callout to return an action. If this flag is not set,
  // a callout can still return a BLOCK action in order to VETO a
  // PERMIT action that was returned by a previous filter. In this
  // example the function just exits if the flag is not set.
 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
  {
    // Return without specifying an action
 return;
  }

  // Get the data fields from inFixedValues
  ...

  // Get any metadata fields from inMetaValues
  ...

  // Get the pointer to the raw data
 rawData = (PNET_BUFFER_LIST)layerData;

  // Get any filter context data from filter->context
  ...

  // Get any flow context data from flowContext
  ...

  // Inspect the various data sources to determine
  // the action to be taken on the data
  ...

  // If the data should be permitted...
 if (...) {

    // Set the action to permit the data
 classifyOut->actionType = FWP_ACTION_PERMIT;

    // Check whether the FWPS_RIGHT_ACTION_WRITE flag should be cleared
 if (filter->flags & FWPS_FILTER_FLAG_CLEAR_ACTION_RIGHT)
    {
       // Clear the FWPS_RIGHT_ACTION_WRITE flag
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
    }

 return;
  }

  ...

  // If the data should be blocked...
 if (...) {

    // Set the action to block the data
 classifyOut->actionType = FWP_ACTION_BLOCK;

    // Clear the FWPS_RIGHT_ACTION_WRITE flag
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

 return;
  }

  ...

  // If the decision to permit or block should be passed
  // to the next filter in the filter engine...
 if (...) {

    // Set the action to continue with the next filter
 classifyOut->actionType = FWP_ACTION_CONTINUE;

 return;
  }

  ...
}
```

The value in *filter->action.type* determines which actions the callout's classifyFn callout function should return in the **actionType** member of the structure pointed to by the *classifyOut* parameter. For more information about these actions, see the **FWPS_ACTION0** structure.

If a callout must perform additional processing of packet data outside its classifyFn callout function before it can determine whether the data should be permitted or blocked, it must pend the packet data until the processing of the data is completed. For information about how to pend packet data, see Types of Callouts and **FwpsPendOperation0**.

At some filtering layers, the *layerData* parameter that is passed by the filter engine to a callout's classifyFn callout function is **NULL**.

For information about how to perform deep inspection of stream data, see Using a Callout for Deep Inspection of Stream Data.

Using a Callout for Deep Inspection of Stream Data

Article • 12/15/2021

When a callout inspects stream data, its classifyFn callout function can inspect any combination of the fixed data fields, the metadata fields, and the raw stream data that is passed to it, and any relevant data that has been stored in a context associated with the filter or the data flow.

For example:

```C++
// classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
    IN OUT FWPS_CLASSIFY_OUT  *classifyOut
    )
{
  FWPS_STREAM_CALLOUT_IO_PACKET0 *ioPacket;
  FWPS_STREAM_BUFFER0 *dataStream;
  UINT32 bytesRequired;
  SIZE_T bytesToPermit;
  SIZE_T bytesToBlock;
  ...

  // Get a pointer to the stream callout I/O packet
 ioPacket = (FWPS_STREAM_CALLOUT_IO_PACKET0 *)layerData;

  // Get the data fields from inFixedValues
  ...

  // Get any metadata fields from inMetaValues
  ...

  // Get the pointer to the data stream
 dataStream = ioPacket->dataStream;

  // Get any filter context data from filter->context
  ...

  // Get any flow context data from flowContext
  ...
  // Inspect the various data sources to determine
  // the action to be taken on the data
  ...

  // If more stream data is required to make a determination...
 if (...) {

    // Let the filter engine know how many more bytes are needed
 ioPacket->streamAction = FWPS_STREAM_ACTION_NEED_MORE_DATA;
 ioPacket->countBytesRequired = bytesRequired;
 ioPacket->countBytesEnforced = 0;

    // Set the action to continue to the next filter
 classifyOut->actionType = FWP_ACTION_CONTINUE;

 return;
  }
  ...

  // If some or all of the data should be permitted...
 if (...) {

    // No stream-specific action is required
 ioPacket->streamAction = FWPS_STREAM_ACTION_NONE;

    // Let the filter engine know how many of the leading bytes
    // in the stream should be permitted
 ioPacket->countBytesRequired = 0;
 ioPacket->countBytesEnforced = bytesToPermit;

    // Set the action to permit the data
 classifyOut->actionType = FWP_ACTION_PERMIT;

 return;
  }

  ...

  // If some or all of the data should be blocked...
 if (...) {

    // No stream-specific action is required
 ioPacket->streamAction = FWPS_STREAM_ACTION_NONE;

    // Let the filter engine know how many of the leading bytes
    // in the stream should be blocked
 ioPacket->countBytesRequired = 0;
 ioPacket->countBytesEnforced = bytesToBlock;

    // Set the action to block the data
 classifyOut->actionType = FWP_ACTION_BLOCK;

 return;
  }
```

```
  ...

  // If the decision to permit or block should be passed
  // to the next filter in the filter engine...
 if (...) {

    // No stream-specific action is required
 ioPacket->streamAction = FWPS_STREAM_ACTION_NONE;

    // No bytes are affected by this callout
 ioPacket->countBytesRequired = 0;
 ioPacket->countBytesEnforced = 0;

 return;
  }

  ...
}
```

The value in *filter->action.type* determines which actions the callout's classifyFn callout function should return in the **actionType** member of the structure pointed to by the *classifyOut* parameter. For more information about these actions, see the **FWPS_ACTION0** structure.

For more information about packet and stream data inspection, see Inspecting Packet and Stream Data.

Packet Inspection Points

Article • 12/15/2021

Incoming packets

Incoming packets that are destined for an address that is assigned to the receiving computer (local host traffic) traverse up WFP layers in the following order:

IP Packet (Network Layer)
All IP packets, including IP packet fragments, are available for inspection at this layer. However, when packets are IPsec-protected, deep content inspection or modification cannot be performed at this layer because the packets are not yet authenticated or decrypted.

Transport Layer
All stand-alone or fully reassembled packets are available for inspection at this layer. IPsec-protected packets have been authenticated or decrypted.

Application Layer Enforcement (ALE) Receive or Accept
The very first packet that arrives at a local endpoint is indicated at this layer. For example, an arriving TCP synchronize (SYN) segment or the first UDP message that is associated with a UDP flow would be indicated. Packets that are required to re-authorize a connection, for example, after a firewall policy change, are also indicated at this layer, and the ALE reauthorization flag will be set.

Datagram Data or Stream
UDP messages and non-ICMP error messages are indicated at the datagram data layer. This layer allows for inspecting network data on a per datagram basis. At the datagram layer, the network data is bidirectional. TCP data flows (data streams only) are available for inspection at the stream layer.

Outgoing packets

Outgoing packets that originate from an address that is assigned to the sending computer (local host sourced traffic) traverse down the following WFP layers:

ALE Connect
TCP connection requests (made before the SYN segment is generated) and the first UDP message that is sent to a remote endpoint are indicated at this layer.

Datagram Data or Stream

UDP messages and non-ICMP error messages are indicated at the datagram data layer. This layer allows for inspecting network data on a per datagram basis. At the datagram layer, the network data is bidirectional. TCP data flows (data streams only) are available for inspection at the stream layer.

Transport and ICMP Error
The transport filtering layer is located in the send path just after a sent packet has been passed to the network layer for processing but before any network layer processing takes place. This filtering layer is located at the top of the network layer instead of at the bottom of the transport layer so that any packets that are sent by third-party transports or as raw packets are filtered at this layer.

The ICMP Error filtering layer is located in the send path for inspecting received ICMP error messages for the transport protocol.

IP Packet
IP packet fragments are not indicated; inspection of outgoing IP fragments is currently unavailable.

IP packets or fragments that do not originate from, or are not destined for, an address that is assigned to the local computer are available for inspection at the forwarding layer. For example, if a packet that is destined for a local client is modified to have a nonlocal destination address and then is injected into the receive path, it will be injected into the forwarding layer. Similarly, if a packet that originates from a local source address is modified to have a nonlocal source address, it will be delivered to the forwarding layer after it is injected into the send path.

WFP Layer Requirements and Restrictions

Article • 12/15/2021

The following requirements and restrictions apply to WFP layers.

Forwarding Layer
An IP packet will be delivered to the forwarding layer if IP forwarding is enabled for a packet that originates from, or is destined for, an address that is assigned to the computer and the packet is sent or received on a different interface than the interface on which the local address is assigned. By default, IP forwarding is disabled and can be enabled by using the **netsh interface ipv4 set interface** command for IPv4 forwarding or the **netsh interface ipv6 set interface** command for IPv6 forwarding.

The forwarding layer can forward each received fragment as it arrives or hold the fragments of an IP payload until all fragments have arrived and then forward them. This is known as *fragment grouping*. When fragment grouping is disabled (it is disabled by default), forwarded IP packet fragments are indicated to WFP one time. When fragment grouping is enabled, a fragment is indicated to WFP two times--first as the fragment itself, and again inside a fragment group that is described by a NET_BUFFER_LIST chain. WFP sets the **FWP_CONDITION_FLAG_IS_FRAGMENT_GROUP** flag when it indicates fragment groups to forwarding layer callouts. You can enable fragment grouping by using the **netsh interface {ipv4|ipv6} set global groupforwardedfragments=enabled** command. Fragment grouping is different than reassembly, which is the reconstruction of the original IP packet at the destination host.

The NET_BUFFER_LIST structure that is indicated at the forwarding layer can describe a full IP packet, an IP packet fragment, or an IP packet fragment group. While an IP packet fragment traverses the forwarding layer, it will be indicated two times to the callout: first as a fragment, and again, as a fragment inside a fragment group.

When a fragment group is indicated, the **FWP_CONDITION_FLAG_IS_FRAGMENT_GROUP** flag is passed as an incoming value to the callout driver's *classifyFn* callout function. In this case, the NET_BUFFER_LIST structure pointed to by the *NetBufferList* parameter is the first node of a **NET_BUFFER_LIST** chain with each **NET_BUFFER_LIST** describing a packet fragment.

A forward injected packet will not be presented to any WFP layer. The injected packet can be indicated to the callout driver again. To prevent infinite looping, the driver should first call the FwpsQueryPacketInjectionState0 function before it continues with a call to the *classifyFn* callout function, and the driver should permit packets that have the injection state FWPS_PACKET_INJECTION_STATE set to **FWPS_PACKET_INJECTED_BY_SELF** or **FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF** to pass through unaltered.

You can use the following command to view the current "Group Forwarded Fragments" setting for the system: **netsh interface {ipv4|ipv6} show global**.

Network Layer
IP packet fragments, which are indicated only for incoming paths, are indicated at three points at this layer: first as an IP packet, again as an IP fragment, and a third time as part of a reassembled IP packet. WFP sets the **FWP_CONDITION_FLAG_IS_FRAGMENT** flag when it indicates fragments to network layer callouts.

For example, if a single IP packet is divided into four fragments, the indications for this packet occur as follows:

1. One indication for each "original" IP packet (4 classifications, or calls to the callout's classify function)
2. One indication for each "original fragment" (4 classifications)
3. One indication for the final reassembled IP packet (1 classification)

When adding filtering conditions, **FWP_MATCH_FLAGS_NONE_SET** can be used together with the **FWP_CONDITION_FLAG_IS_FRAGMENT** flag to avoid the second indication(s). These condition flags are meant to prevent classifications the callout driver does not care about. If the callout has to inspect only full packets (those that have not been fragmented and reassembled), it has to parse the IP header to avoid processing fragments that are indicated as IP packets. A callout might do the following steps to achieve this:

1. Skip the first indication(s) by checking if the More Fragments (MF) flag is set and/or the Fragment Offset field is not 0.
2. Write a filter that allows all classifications where **FWP_CONDITION_FLAG_IS_FRAGMENT** is set.
3. Perform whatever processing is needed on the reassembled packet.

Alternatively, the callout can inspect packets at the transport layer.

Transport Layer and ALE
To be able to coexist with IPsec processing, callouts that inspect packets at the incoming transport layer must also register at the ALE receive and accept layer. Such a callout can inspect/modify most of the traffic at the transport layer, but it must also permit packets that are assigned to the ALE receive/accept layer. Such a callout must also inspect or modify the packets from the ALE layer. WFP sets the

FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED metadata flag when it indicates to the transport layer those packets that require ALE inspection. IPsec processing is deferred until those packets that create the initial "connection" and those that are required to re-authorize the connection reach the ALE layer.

Transport layer and ALE layer callouts must register themselves at a sublayer that is of lower weight than the universal sublayer. The built-in IPsec/ALE enforcement callouts reside at the universal sublayer.

The following table shows packet types that can be indicated at ALE layers. Be aware that some ALE layers do not always have a packet associated with their indication.

ALE layer	TCP packets	UDP packets
Bind (resource assignment)	not applicable	not applicable
Connect	no packet	first UDP packet (outgoing)
Receive/Accept	SYN (incoming)	first UDP packet (incoming)
Flow Established	final ACK (incoming & outgoing)	first UDP packet (incoming & outgoing)

Packet Indication Format

Article • 12/15/2021

Network data is indicated in WFP as NDIS net buffer lists (NET_BUFFER_LIST). The **Next** member of the **NET_BUFFER_LIST** structure can be used to describe a *chain* of net buffer lists. WFP only indicates a single net buffer list to callouts (that is, netBufferList->Next == **NULL**), except for the following cases:

- WFP can indicate net buffer list chains to callouts from the Stream layer.
- WFP indicates net buffer list chains to callouts when it classifies IP packet fragment groups in the forward path to callouts. Each net buffer list inside the chain describes a single fragment.

Although a net buffer list can describe a whole packet, for different types of layers, WFP indicates net buffer lists to callouts at different offsets from the beginning of IP header. For example, at the incoming network layer, the net buffer list starts after the IP header, while at the incoming transport layer, the net buffer list starts after the transport header. IP and transport headers are always described by the first NET_BUFFER structure inside a net buffer list.

Offsets into the net buffer lists are indicated to callouts by using the **ipHeaderSize** and **transportHeaderSize** members of the FWPS_INCOMING_METADATA_VALUES0 structure. Callouts can use the NDIS functions NdisRetreatNetBufferDataStart and NdisAdvanceNetBufferDataStart to adjust the offset of the indicated net buffer lists. However in this case, the callout must undo the offset adjustment before it returns from the *classifyFn* function.

In a call to the *classifyFn* function for outgoing data, a NET_BUFFER_LIST can contain more than one NET_BUFFER structure, each of which describes an IP packet. If some packets (for example, net buffers) in a net buffer list are acceptable, but others are not, a callout driver must do the following:

1. Clone and block the whole net buffer list.
2. Build a new net buffer list that describes the acceptable subset of net buffers.
3. Inject the new net buffer list back into the send path.

Alternatively, the callout can unlink the unwanted net buffers from the net buffer list and inject the altered net buffer list back into the send path. However, in this case the callout driver must undo this modification to the cloned net buffer list before it calls the

FwpsFreeCloneNetBufferList0 function. The callout driver must also save the original net buffer linkage information as part of its state data.

For more information about data offsets that are used by WFP, see Data Offset Positions.

Note Callouts that work with decrypted IPSec ESP packets must use the data length of the **NET_BUFFER** structure instead of MDL data to determine the packet length. To obtain the data length, use the **NET_BUFFER_DATA_LENGTH** macro. For more information, see Developing IPsec-Compatible Callout Drivers.

Types of Callouts

Article • 12/15/2021

The following types of callouts can be used with WFP:

Inline Inspection Callout
This type of callout always returns **FWP_ACTION_CONTINUE** from the *classifyFn* function and does not modify the network traffic in any way. A callout that collects network statistics is an example of this type of callout.

For this type of callout, the filter action type (specified by the **Type** member of the **FWPS_ACTION0** structure) should be set to **FWP_ACTION_CALLOUT_INSPECTION**.

Out-of-band Inspection Callout
This type of callout does not modify network traffic. Instead, it defers any inspection to be done outside the *classifyFn* function by "pending" the indicated data and then reinjecting the pended data back into the TCP/IP stack with one of the packet injection functions. Pending is implemented by first cloning the indicated data, followed by returning **FWP_ACTION_BLOCK** from the *classifyFn* function that has the **FWPS_CLASSIFY_OUT_FLAG_ABSORB** bit set.

Inline Modification Callout
This type of callout modifies network traffic by first making a clone of the indicated data, then modifying the clone, and finally injecting the modified clone back into the TCP/IP stack from the *classifyFn* function. This type of callout also returns **FWP_ACTION_BLOCK** from the *classifyFn* function that has the **FWPS_CLASSIFY_OUT_FLAG_ABSORB** bit set.

The filter action type for this type of callout should be set to **FWP_ACTION_CALLOUT_TERMINATING**.

Out-of-band Modification Callout
This type of callout first references the indicated packet by using the **FwpsReferenceNetBufferList0** function that has the *intentToModify* parameter set to **TRUE**. The callout then returns **FWP_ACTION_BLOCK** with the **FWPS_CLASSIFY_OUT_FLAG_ABSORB** bit set from the *classifyFn* function. When the packet is ready to be modified outside *classifyFn*, the callout clones the referenced packet (as soon as it is cloned, the original packet can then be dereferenced). The callout then modifies the clone and injects the modified packet back into the TCP/IP stack.

The filter action type for this type of callout should be set to **FWP_ACTION_CALLOUT_TERMINATING**.

Redirection Callout

For more information about this type of callout, see Using Bind or Connect Redirection.

There are two types of redirection callouts:

- A bind redirection callout allows the callout driver to modify the local address and local port of a socket.
- A connect redirection callout allows the callout driver to modify the remote address and remote port of a connection.

The filter action type for this type of callout should be set to **FWP_ACTION_PERMIT**.

For more information about **FWPS_CLASSIFY_OUT_FLAG_ABSORB**, see **FWPS_CLASSIFY_OUT0**. This flag is not valid at any WFP discard layer. Returning **FWP_ACTION_BLOCK** with the **FWPS_CLASSIFY_OUT_FLAG_ABSORB** flag set from the *classifyFn* function causes the packet to be silently discarded, in such a way that the packet will not hit any of the WFP discard layers, nor will it cause audit events to be generated.

Although cloned net buffer lists can be modified, for example, by adding or removing net buffers or MDLs, or both, callouts must undo such modifications before they call the **FwpsFreeCloneNetBufferList0** function.

To coexist with other callouts that perform packet inspection, packet modification, or connection redirection, before a packet is pended with the reference/clone-drop-reinject mechanism, a callout must "hard"-drop the original packet by clearing the **FWPS_RIGHT_ACTION_WRITE** flag in the **rights** member of the **FWPS_CLASSIFY_OUT0** structure returned by the *classifyFn* function. If the **FWPS_RIGHT_ACTION_WRITE** flag is set when *classifyFn* is called (which means that the packet could be pended and later reinjected or modified), the callout must not pend the indication and should not change the current action type; and it must wait for a higher-weight callout to inject the clone that might be modified.

The **FWPS_RIGHT_ACTION_WRITE** flag should be set whenever a callout pends a classification. Your callout driver should test for the **FWPS_RIGHT_ACTION_WRITE** flag to check the rights for your callout to return an action. If this flag is not set, your callout can still return a **FWP_ACTION_BLOCK** action in order to veto a **FWP_ACTION_PERMIT** action that was returned by a previous callout. In the example shown in Using a Callout for Deep Inspection, the function just exits if the flag is not set.

The **FwpsPendOperation0** function is used to pend packets that originate from the **FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT**_*XXX*,

FWPM_LAYER_ALE_AUTH_LISTEN_*XXX*, or **FWPM_LAYER_ALE_AUTH_CONNECT_***XXX* management filtering layers.

The **FwpsPendClassify0** function is used to pend packets that originate from the following run-time filtering layers:

FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4 FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6 FWPS_LAYER_ALE_CONNECT_REDIRECT_V4 FWPS_LAYER_ALE_CONNECT_REDIRECT_V6 FWPS_LAYER_ALE_BIND_REDIRECT_V4 FWPS_LAYER_ALE_BIND_REDIRECT_V6

Packet Injection Functions

Article • 12/15/2021

A callout driver can call the following WFP functions to inject pended or modified packet data into the TCP/IP stack. The applicable layers from which data can be injected, together with possible destinations, are listed in the following table.

Injection function	Applicable layer	Destination
FwpsInjectForwardAsync0	network layer	the forwarding data path
FwpsInjectNetworkReceiveAsync0	network layer	the receive data path
FwpsInjectNetworkSendAsync0	network layer	the send data path
FwpsInjectTransportReceiveAsync0	packet data from the transport, datagram data, ICMP error, or ALE layers	the receive data path
FwpsInjectTransportSendAsync0	packet data from the transport, datagram data, ICMP error, or ALE layers	the send data path
FwpsStreamInjectAsync0	TCP data segments	a data stream

In addition, the FwpsQueryPacketInjectionState0 function is used to inspect the injection history of packet data.

Cross-layer injection is enabled if the callout can supply all needed information that is required by the injection function, and the net buffer list has the format expected by the injection function. For example, a callout can capture a packet at the forward path, modify its destination address to that of the local computer, and call **FwpsInjectTransportReceiveAsync0** to redirect the packet into the local computer's TCP/IP stack.

Except for the stream (TCP data) injection, injected incoming packets reenter from the "bottom" of the stack and WFP layers, while injected outgoing packets reenter from the "top" of the stack and WFP layers. For example, a UDP packet injected from the incoming datagram data layer will reenter the stack and traverse the network layer, the transport layer, the ALE receive or accept layer (optional), and back into the datagram data layer. Another UDP packet injected from the outgoing network layer will reenter the stack and traverse the ALE (optional), datagram data, and transport layers, and back to the network layer.

FwpsInjectTransportReceiveAsync0 automatically bypasses IPsec processing for the reinjected packet because it had previously gone through IPsec verification.

A packet injected by a WFP callout driver will be re-indicated to the callout except in the cases in which modification to the packet causes it to miss the original filter conditions. WFP provides the **FwpsQueryPacketInjectionState0** function for callouts to query whether the packet was injected (or injected earlier) by the callout. To prevent infinite looping, callouts should permit self-injected packets.

Callouts must adjust the IP or transport layer checksum, or both, after they modify an IP packet. A callout can set the checksum to 0 for UDP over IPv4 packets. To be compatible with transport layer checksum offload, and to adjust the full checksum versus pseudo checksum calculations accordingly, a callout can use the following logic:

C++

```
NDIS_TCP_IP_CHECKSUM_PACKET_INFO ChecksumInfo;
 ChecksumInfo.Value =
 (ULONG) (ULONG_PTR)NET_BUFFER_LIST_INFO(
 NetBufferList,TcpIpChecksumNetBufferListInfo);
```

If ChecksumInfo.Transmit.NdisPacketTcpChecksum is **TRUE**, the TCP send operation will be offloaded. If ChecksumInfo.Transmit.NdisPacketUdpChecksum is **TRUE**, the UDP send operation will be offloaded.

In Windows Vista with Service Pack 1 (SP1) and Windows Server 2008, if inMetaValues->headerIncludeHeaderLength is greater than 0, the outgoing packet is a RAW send reinjection that includes an IP header. To perform RAW send reinjections that include an IP header for Windows Vista with SP1 and Windows Server 2008, you must retreat the cloned packet by the amount in inMetaValues->headerIncludeHeaderLength and copy the inMetaValues->headerIncludeHeader over the newly extended space. Then, use FwpsInjectTransportSendAsync0 with the net buffer list for the packet and leave the FWPS_TRANSPORT_SEND_PARAMS0 parameter set to **NULL**. For more information about retreat operations for net buffer lists, see Retreat and Advance Operations.

Note For raw send operations, the net buffer list must contain only a single net buffer. If your net buffer list contains more than one net buffer, you have to convert your net buffer list to a series of net buffer lists, and each in the series must contain a single net buffer. For more information about net buffer list management, see NET_BUFFER Architecture.

Packet Modification Examples

Article • 05/09/2022

The following example code shows how to modify and inspect packets with WFP.

Inline packet Modification from Outgoing Transport Layers

```C++
HANDLE gInjectionHandle;

void
NTAPI
InjectionCompletionFn(
   IN void* context,
   IN OUT NET_BUFFER_LIST* netBufferList,
   IN BOOLEAN dispatchLevel
   )
{
   FWPS_TRANSPORT_SEND_PARAMS0* tlSendArgs
     = (FWPS_TRANSPORT_SEND_PARAMS0*)context;

   //
   // TODO: Free tlSendArgs and embedded allocations.
   //

   //
   // TODO: Check netBufferList->Status for injection result
   //

 FwpsFreeCloneNetBufferList0(netBufferList, 0);
}

void
NTAPI
WfpTransportSendClassify(
   IN const FWPS_INCOMING_VALUES0* inFixedValues,
   IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
   IN OUT void* layerData,
   IN const FWPS_FILTER0* filter,
   IN UINT64 flowContext,
   IN OUT FWPS_CLASSIFY_OUT0* classifyOut
   )
{
   NTSTATUS status;

   NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
   NET_BUFFER_LIST* clonedNetBufferList = NULL;
   FWPS_PACKET_INJECTION_STATE injectionState;
```

```
  FWPS_TRANSPORT_SEND_PARAMS0* tlSendArgs = NULL;
  ADDRESS_FAMILY af = AF_UNSPEC;

injectionState = FwpsQueryPacketInjectionState0(
gInjectionHandle,
netBufferList,
                       NULL);
if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
  {
classifyOut->actionType = FWP_ACTION_PERMIT;
goto Exit;
  }

if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
  {
     //
     // Cannot alter the action.
     //
goto Exit;
  }

  //
  // TODO: Allocate and populate tlSendArgs by using information from
  // inFixedValues and inMetaValues.
  // Note: 1) Remote address and controlData (if not NULL) must
  // be deep-copied.
  //       2) IPv4 address must be converted to network order.
  //       3) Handle allocation errors.

ASSERT(tlSendArgs != NULL);

status = FwpsAllocateCloneNetBufferList0(
netBufferList,
              NULL,
              NULL,
              0,
              &clonedNetBufferList);

if (!NT_SUCCESS(status))
  {
classifyOut->actionType = FWP_ACTION_BLOCK;
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

goto Exit;
  }

  //
  // TODO: Perform modification to the cloned net buffer list here.
  //

  //
  // TODO: Set af based on inFixedValues->layerId.
  //
ASSERT(af == AF_INET || af == AF_INET6);
```

```
   //
   // Note: For TCP traffic, FwpsInjectTransportReceiveAsync0 and
   // FwpsInjectTransportSendAsync0 must be queued and run by a DPC.
   //

 status = FwpsInjectTransportSendAsync0(
 gInjectionHandle,
               NULL,
 inMetaValues->transportEndpointHandle,
               0,
 tlSendArgs,
 af,
 inMetaValues->compartmentId,
 clonedNetBufferList,
 InjectionCompletionFn,
 tlSendArgs);

 if (!NT_SUCCESS(status))
   {
 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

 goto Exit;
   }

 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
 classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;

   //
   // Ownership of clonedNetBufferList and tlSendArgs
   // now transferred to InjectionCompletionFn.
   //
 clonedNetBufferList = NULL;
 tlSendArgs = NULL;

Exit:

 if (clonedNetBufferList != NULL)
   {
 FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
   }
 if (tlSendArgs != NULL)
   {
      //
      // TODO: Free tlSendArgs and embedded allocations.
      //
   }

 return;
}
```

Out-of-band Packet Modification from Incoming Datagram Data Layers

C++

```cpp
typedef struct DD_RECV_CLASSIFY_INFO_ {
   NET_BUFFER_LIST* netBufferList;
   UINT32 nblOffset;
   UINT32 ipHeaderSize;
   UINT32 transportHeaderSize;
   ADDRESS_FAMILY af;
   COMPARTMENT_ID compartmentId;
   IF_INDEX interfaceIndex;
   IF_INDEX subInterfaceIndex;
}DD_RECV_CLASSIFY_INFO;

HANDLE gInjectionHandle;

void
NTAPI
InjectionCompletionFn(
   IN void* context,
   IN OUT NET_BUFFER_LIST* netBufferList,
   IN BOOLEAN dispatchLevel
   )
{
   DD_RECV_CLASSIFY_INFO* classifyInfo
 = (DD_RECV_CLASSIFY_INFO*)context;

   //
   // TODO: Remove from queue and free classifyInfo.
   //

   //
   // TODO: Check netBufferList->Status for injection result.
   //

 FwpsFreeCloneNetBufferList0(netBufferList, 0);
}

void
DatagramDataReceiveWorker(
   DD_RECV_CLASSIFY_INFO* classifyInfo
   // ... and other parameters
   )
//
// To prevent WFP from making a deep clone (deep-copying MDLs,
// net buffers, net buffer lists, structures, and data mapped by MDLs,
// DatagramDataReceiveWorker should be run by a DPC targeting the
// processor to which the referenced net buffer list was first
// classified. See KeSetTargetProcessorDpc for DPC targeting.
//
{
   NTSTATUS status;
   NET_BUFFER_LIST* clonedNetBufferList;
   ULONG nblOffset =
      NET_BUFFER_DATA_OFFSET(NET_BUFFER_LIST_FIRST_NB(classifyInfo-
>netBufferList));

   //
   // The TCP/IP stack could have retreated the net buffer list by the
   // transportHeaderSize amount; detect the condition here to avoid
   // retreating two times.
   //
 if (nblOffset != classifyInfo->nblOffset)
   {
 ASSERT(classifyInfo->nblOffset - nblOffset == classifyInfo-
>transportHeaderSize);

 classifyInfo->transportHeaderSize = 0;
   }

   //
   // Adjust the net buffer list offset to start by using the IP header.
   //
 NdisRetreatNetBufferDataStart(
      NET_BUFFER_LIST_FIRST_NB(classifyInfo->netBufferList),
 classifyInfo->ipHeaderSize + classifyInfo->transportHeaderSize,
      0,
      NULL
      );

 status = FwpsAllocateCloneNetBufferList0(
 classifyInfo->netBufferList,
               NULL,
               NULL,
               0,
               &clonedNetBufferList);

 if (!NT_SUCCESS(status))
   {
      // TODO: Handle error condition.
 goto Exit;
   }

   //
   // Undo the adjustment on the original net buffer list.
   //

 NdisAdvanceNetBufferDataStart(
      NET_BUFFER_LIST_FIRST_NB(classifyInfo->netBufferList),
 classifyInfo->ipHeaderSize + classifyInfo->transportHeaderSize,
      FALSE,
      NULL);

   //
   // Because the clone references the original net buffer list,
   // undo the reference that was claimed during classifyFn.
   //
 FwpsDereferenceNetBufferList0(
 classifyInfo->netBufferList,
      FALSE);
 classifyInfo->netBufferList = NULL;

   //
   // TODO: Modify the cloned net buffer list here.
   // Note: 1) The next protocol field of the IP header could be
   // AH/ESP, in which case the IP header must be rebuilt (and
   // the AH/ESP header removed).
   //       2) The callout must re-calculate the IP checksum.
   //

 status = FwpsInjectTransportReceiveAsync0(
 gInjectionHandle,
               NULL,
               NULL,
 0,
 classifyInfo->af,
 classifyInfo->compartmentId,
 classifyInfo->interfaceIndex,
 classifyInfo->subInterfaceIndex,
 clonedNetBufferList,
 InjectionCompletionFn,
 classifyInfo);

 if (!NT_SUCCESS(status))
   {
      // TODO: Handle error condition.
 goto Exit;
   }

   //
   // Ownership of clonedNetBufferList and classifyInfo is
   // now transferred to InjectionCompletionFn.
   //
 clonedNetBufferList = NULL;
 classifyInfo = NULL;

Exit:

 if (clonedNetBufferList != NULL)
   {
 FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
   }
 if (classifyInfo->netBufferList != NULL)
   {
 FwpsDereferenceNetBufferList0(
 classifyInfo->netBufferList,
         FALSE);
   }

   // TODO: Free other resources on error.
}
```

```
void
NTAPI
WfpDatagramDataReceiveClassify(
   IN const FWPS_INCOMING_VALUES0* inFixedValues,
   IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
   IN OUT void* layerData,
   IN const FWPS_FILTER0* filter,
   IN UINT64 flowContext,
   OUT FWPS_CLASSIFY_OUT0* classifyOut
   )
{
   NTSTATUS status;

   NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
   FWPS_PACKET_INJECTION_STATE injectionState;
   DD_RECV_CLASSIFY_INFO* classifyInfo = NULL;

 injectionState = FwpsQueryPacketInjectionState0(
 gInjectionHandle,
 netBufferList,
                        NULL);
 if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
 injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
   {
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
   }

 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
   {
      //
      // Cannot alter the action.
      //
 goto Exit;
   }

   //
   // TODO: Allocate and populate classifyInfo by using information
   // from inFixedValues and inMetaValues.
   //

 classifyInfo->nblOffset =
      NET_BUFFER_DATA_OFFSET(NET_BUFFER_LIST_FIRST_NB(netBufferList));

 ASSERT(classifyInfo != NULL);
 ASSERT(classifyInfo->netBufferList != NULL);

 FwpsReferenceNetBufferList0(
 classifyInfo->netBufferList,
      TRUE // intendToModify
      );

   //
   // TODO: Queue classifyInfo for out-of-band processing.
   //

 classifyInfo = NULL; // Ownership transferred on success.

 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
 classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;

Exit:

 if (classifyInfo)
   {
      // TODO: Free object here.
   }

 return;
}
```

Non-intrusive Out-of-band Inspection from Incoming Transport Layer and ALE Receive/Accept Layers

The following is example code for an inspection procedure that views packet data without changing it.

C++

```
typedef struct TL_ALE_RECV_CLASSIFY_INFO_ {
   BOOLEAN aleInfo;  // TRUE if information is gathered from Ale
receive/accept layer
                     // FALSE if information is gathered from incoming
transport layer

   NET_BUFFER_LIST* netBufferList;
   ADDRESS_FAMILY af;
   COMPARTMENT_ID compartmentId;
   IF_INDEX interfaceIndex;
   IF_INDEX subInterfaceIndex;

   HANDLE aleCompletionCtx;

}TL_ALE_RECV_CLASSIFY_INFO;

HANDLE gInjectionHandle;

void
NTAPI
InjectionCompletionFn(
   IN void* context,
   IN OUT NET_BUFFER_LIST* netBufferList,
   IN BOOLEAN dispatchLevel
   )
```

```
{
   TL_ALE_RECV_CLASSIFY_INFO* classifyInfo =
(TL_ALE_RECV_CLASSIFY_INFO*)context;

   //
   // TODO: Remove from queue and free classifyInfo.
   //

   //
   // TODO: Check netBufferList->Status for injection result.
   //

 FwpsFreeCloneNetBufferList0(netBufferList, 0);
}

void
TlAleReceiveWorker(
   TL_ALE_RECV_CLASSIFY_INFO* classifyInfo
   // ... and other parameters
   )
{
   NTSTATUS status;

 if (classifyInfo->aleInfo)
   {
 FwpsCompleteOperation0(
 classifyInfo->aleCompletionCtx,
 classifyInfo->netBufferList);
   }

 status = FwpsInjectTransportReceiveAsync0(
 gInjectionHandle,
               NULL,
               NULL,
               0,
 classifyInfo->af,
 classifyInfo->compartmentId,
 classifyInfo->interfaceIndex,
 classifyInfo->subInterfaceIndex,
 classifyInfo->netBufferList,
 InjectionCompletionFn,
 classifyInfo);

 if (!NT_SUCCESS(status))
   {
      // TODO: Handle error condition.
 goto Exit;
   }

   //
   // Ownership of classifyInfo now transferred to InjectionCompletionFn.
   //
 classifyInfo = NULL;

Exit:
 if (classifyInfo != NULL)
   {
 FwpsFreeCloneNetBufferList0(classifyInfo->netBufferList, 0);

      // TODO: Remove from queue and free classifyInfo.
   }

   // TODO: Free other resources on error.
}

void
NTAPI
WfpAleReceiveClassify(
   IN const FWPS_INCOMING_VALUES0* inFixedValues,
   IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
   IN OUT void* layerData,
   IN const FWPS_FILTER0* filter,
   IN UINT64 flowContext,
   OUT FWPS_CLASSIFY_OUT0* classifyOut
   )
{
   NTSTATUS status;

   NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
   NET_BUFFER_LIST* clonedNetBufferList = NULL;
   FWPS_PACKET_INJECTION_STATE injectionState;
 TL_ALE_RECV_CLASSIFY_INFO* classifyInfo = NULL;

 injectionState = FwpsQueryPacketInjectionState0(
 gInjectionHandle,
 netBufferList,
                        NULL);
 if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
 injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
   {
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
   }

 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
   {
      //
      // Cannot alter the action.
      //
 goto Exit;
   }
   //
   // Adjust the net buffer list offset so that it starts with the IP
header.
   //
 NdisRetreatNetBufferDataStart(
      NET_BUFFER_LIST_FIRST_NB(netBufferList),
 inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
      0,
      NULL
      );

 status = FwpsAllocateCloneNetBufferList0(
 netBufferList,
               NULL,
               NULL,
               0,
               &clonedNetBufferList);

 if (!NT_SUCCESS(status))
   {
 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

 goto Exit;
   }

   //
   // Undo the adjustment on the original net buffer list.
   //

 NdisAdvanceNetBufferDataStart(
      NET_BUFFER_LIST_FIRST_NB(netBufferList),
 inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
      FALSE,
      NULL);

   //
   // Note: 1) The next protocol field of the IP header in the clone net
   // buffer list could be AH/ESP, in which case the IP header must be
   // rebuilt (and AH/ESP header removed).
   //       2) The callout must re-calculate the IP checksum.

   //
   // TODO: Allocate and populate classifyInfo by using information from
   // inFixedValues and inMetaValues.
   //

 ASSERT(classifyInfo != NULL);

 classifyInfo->aleInfo = TRUE;

 classifyInfo->netBufferList = clonedNetBufferList;
 clonedNetBufferList = NULL; // Ownership transferred.

 status = FwpsPendOperation0(
 inMetaValues->completionHandle,
               &classifyInfo->aleCompletionCtx);

 if (!NT_SUCCESS(status))
   {
 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
```

```
 goto Exit;
   }

   //
   // TODO: Queue classifyInfo for out-of-band processing.
   //

 classifyInfo = NULL; // Ownership transferred on success.

 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
 classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;

Exit:

 if (clonedNetBufferList != NULL)
   {
 FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
   }
 if (classifyInfo)
   {
 if (classifyInfo->netBufferList)
      {
 FwpsFreeCloneNetBufferList0(classifyInfo->netBufferList, 0);
      }
      // TODO: Free object here.
   }

 return;
}

void
NTAPI
WfpTransportReceiveClassify(
   IN const FWPS_INCOMING_VALUES0* inFixedValues,
   IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
   IN OUT void* layerData,
   IN const FWPS_FILTER0* filter,
   IN UINT64 flowContext,
   OUT FWPS_CLASSIFY_OUT0* classifyOut
   )
{
   NTSTATUS status;

   NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
   NET_BUFFER_LIST* clonedNetBufferList = NULL;
   FWPS_PACKET_INJECTION_STATE injectionState;
 TL_ALE_RECV_CLASSIFY_INFO* classifyInfo = NULL;

 injectionState = FwpsQueryPacketInjectionState0(
 gInjectionHandle,
 netBufferList,
                        NULL);
 if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
 injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
```

```
   {
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
   }

 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
   {
      //
      // Cannot alter the action.
      //
 goto Exit;
   }

   //
   // Let go of the packet if it requires ALE classify; the packet can
   // be inspected from the ALE receive/accept layer. Alternatively,
   // the callout can use the combination of
   // FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY and
   // FWP_MATCH_FLAGS_NONE_SET when you set up
   // filter conditions for the incoming transport layer.
   //
   // Beginning with Windows Vista SP1 and Windows Server 2008,
   // do not use FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY.
   // Use FWPS_IS_METADATA_FIELD_PRESENT macro to check for
   // metadata fields.
   //
#if (NTDDI_VERSION >= NTDDI_WIN6SP1)
 if (FWPS_IS_METADATA_FIELD_PRESENT(inMetaValues,

FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED))
#else
 if ((inFixedValues->layerId == FWPS_LAYER_INBOUND_TRANSPORT_V4 &&
         (inFixedValues-
>incomingValue[FWPS_FIELD_INBOUND_TRANSPORT_V4_FLAGS].value.uint32 &
          FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY)) ||
        (inFixedValues->layerId == FWPS_LAYER_INBOUND_TRANSPORT_V6 &&
         (inFixedValues-
>incomingValue[FWPS_FIELD_INBOUND_TRANSPORT_V6_FLAGS].value.uint32 &
          FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY)))
#endif
   {
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
   }
   //
   // Adjust the net buffer list offset so that it starts with the IP
header.
   //
 NdisRetreatNetBufferDataStart(
      NET_BUFFER_LIST_FIRST_NB(netBufferList),
 inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
      0,
      NULL
      );
 status = FwpsAllocateCloneNetBufferList0(
 netBufferList,
               NULL,
               NULL,
               0,
               &clonedNetBufferList);

 if (!NT_SUCCESS(status))
   {
 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

 goto Exit;
   }

   //
   // Undo the adjustment on the original net buffer list.
   //

 NdisAdvanceNetBufferDataStart(
      NET_BUFFER_LIST_FIRST_NB(netBufferList),
 inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
      FALSE,
      NULL);

   //
   // Notes: 1) The next protocol field of the IP header in the clone net
   // buffer list could be AH/ESP, in which case the IP header must be
   // rebuilt (and AH/ESP header removed).
   //        2) The callout must re-calculate the IP checksum.

   //
   // TODO: Allocate and populate classifyInfo by using information from
   // inFixedValues and inMetaValues.
   //

 ASSERT(classifyInfo != NULL);

 classifyInfo->aleInfo = FALSE;

 classifyInfo->netBufferList = clonedNetBufferList;
 clonedNetBufferList = NULL; // ownership transferred

   //
   // TODO: Queue classifyInfo for out-of-band processing.
   //

 classifyInfo = NULL; // Ownership transferred on success.

 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
 classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;

Exit:
 if (clonedNetBufferList != NULL)
   {
 FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
   }
 if (classifyInfo)
   {
 if (classifyInfo->netBufferList)
      {
 FwpsFreeCloneNetBufferList0(classifyInfo->netBufferList, 0);
      }
      // TODO: Free object here.
   }

 return;
}
```

Related topics

classifyFn

Types of Callouts

Stream Inspection

Article • 10/26/2022

Inline Stream Inspection

Inline stream modifiers can edit stream data by permitting or blocking a part of the indicated data by setting the value of the **countBytesEnforced** member of the FWPS_STREAM_CALLOUT_IO_PACKET0 structure as they return **FWP_ACTION_PERMIT** or **FWP_ACTION_BLOCK** from the *classifyFn* callout function. They can also call the FwpsStreamInjectAsync0 function to add new content to the stream. This content can be new or can replace blocked data.

To replace a pattern found in the middle of an indicated segment (for example, *n* bytes followed by a pattern of *p* bytes followed by *m* bytes), the callout would follow these steps:

1. The callout's *classifyFn* function is called by using *n* + *p* + *m* bytes.

2. The callout returns **FWP_ACTION_PERMIT** with the **countBytesEnforced** member set to *n*.

3. The callout's *classifyFn* function is called again with *p* + *m* bytes. WFP will call *classifyFn* again if **countBytesEnforced** is less than the indicated amount.

4. From the *classifyFn* function, the callout calls the *FwpsStreamInjectAsync0* function to inject the replacement pattern *p*'. The callout then returns **FWP_ACTION_BLOCK** with **countBytesEnforced** set to *p*.

5. The callout's *classifyFn* function is called again with *m* bytes.

6. The callout returns **FWP_ACTION_PERMIT** with **countBytesEnforced** set to *m*.

If the indicated data is insufficient for the callout to make an inspection decision, it can set the **streamAction** member of the FWPS_STREAM_CALLOUT_IO_PACKET0 structure to **FWPS_STREAM_ACTION_NEED_MORE_DATA** and set the **countBytesRequired** member to the minimal amount WFP should accumulate before the data is indicated again. When **streamAction** is set, the callout should return **FWP_ACTION_NONE** from the *classifyFn* function.

WFP can accumulate up to 8 MB of stream data when **FWPS_STREAM_ACTION_NEED_MORE_DATA** is set. WFP will set the **FWPS_CLASSIFY_OUT_FLAG_BUFFER_LIMIT_REACHED** flag when it calls the callout's *classifyFn* function and the buffer space is exhausted. When the latter flag is set, the callout must accept the indicated data in full. A callout must not return **FWPS_STREAM_ACTION_NEED_MORE_DATA** when the **FWPS_CLASSIFY_OUT_FLAG_NO_MORE_DATA** flag is set.

For the convenience of being able to scan a stream pattern from a flat buffer, WFP provides the FwpsCopyStreamDataToBuffer0 utility function, which can copy indicated stream data into a contiguous buffer.

Out-of-Band Stream Inspection

For out-of-band inspection or modification, a stream callout would follow the similar pattern as the packet inspection callout: it would first clone all indicated stream segments for deferred processing, and then it would block those segments. The inspected or modified data is later injected back into the data stream. When injecting data out-of-band, the callout must return **FWP_ACTION_BLOCK** on all indicated segments to guarantee integrity of the resulting stream. An out-of-band inspection module must not arbitrarily inject a FIN (which indicates no more data from the sender) into an outgoing data stream. If the module must drop the connection, its *classifyFn* callout function must set the **streamAction** member of the FWPS_STREAM_CALLOUT_IO_PACKET0 structure to **FWPS_STREAM_ACTION_DROP_CONNECTION**.

Note It is a **violation of contract** for callouts to switch from out-of-band to inline, and can cause unexpected behaviors. Ensure out-of-band callouts are meeting each of the specified criteria.

Because stream data can be indicated as a NET_BUFFER_LIST chain, FWP provides the FwpsCloneStreamData0 and FwpsDiscardClonedStreamData0 utility functions that operate on net buffer list chains.

WFP also supports stream data throttling for the incoming direction. If a callout cannot keep pace with the incoming data rate, it can return **FWPS_STREAM_ACTION_DEFER** to "pause" the stream. The stream can then be "resumed" by calling the FwpsStreamContinue0 function. Deferring a stream with this function causes the TCP/IP stack to stop ACK-processing incoming data. This causes the TCP sliding window to decrease toward 0.

For out-of-band stream inspection callouts, FwpsStreamContinue0 must not be called while the **FwpsStreamInjectAsync0** function is called.

Injected stream data will not be re-indicated to the callout, but it will be made available to stream callouts from lower-weight sublayers.

The Windows Filtering Platform Stream Edit Sample in the Windows driver samples repository on GitHub shows how to perform inline and out-of-band editing at the stream layer.

Note Windows Server 2008 and later do not support removal of a stream filter during the following processes:

- The callout is performing out-of-band packet injection.
- The callout is requesting more data by setting the **streamAction** member of the FWPS_STREAM_CALLOUT_IO_PACKET0 structure to **FWPS_STREAM_ACTION_NEED_MORE_DATA**.
- The callout is deferring a stream by setting the **streamAction** member of the FWPS_STREAM_CALLOUT_IO_PACKET0 structure to **FWPS_STREAM_ACTION_DEFER**.

Dynamic Stream Inspection

Windows 7 and later support dynamic stream inspections. A dynamic stream inspection operates on an existing stream data flow, rather than creating and tearing down a new one. A callout driver that can perform dynamic stream inspections should set the **FWP_CALLOUT_FLAG_ALLOW_MID_STREAM_INSPECTION** flag in the **Flags** member of the FWPS_CALLOUT1 or FWPS_CALLOUT2 structure.

Avoiding Unnecessary Inspections

To perform stream inspections only on connections that the driver is interested in, a callout can set the **FWP_CALLOUT_FLAG_CONDITIONAL_ON_FLOW** flag in the **Flags** member of the FWPS_CALLOUT0 structure. This callout will be ignored on all other connections. Performance will be improved and the driver will not have to maintain unnecessary state data.

Stream Layer Waterfall Model

The stream layer in WFP follows a strict waterfall model; that is, a callout in this layer will be allowed to inspect a stream segment only if the previous callout (if any) explicitly permitted it. If a callout blocks an indicated segment, that segment is permanently taken out of the stream and no callouts will be allowed to inspect it.

Moreover:

1. Every non-inspect callout at the stream layer must explicitly assign a value to the **actionType** member of the *classifyOut* parameter regardless of what value may have been previously set in that parameter.
2. The **FWPS_RIGHT_ACTION_WRITE** flag in the **rights** member of the *classifyOut* parameter has no significance in the WFP stream layer. Callouts at this layer should not check for the presence of this flag. Callouts may process the indicated *layerData* parameter regardless of the value of *classifyOut*->**rights**.

Modifying Stream Data

Article • 12/15/2021

When a callout processes data at the stream layer, its *classifyFn* callout function can modify the data in the data stream. The callout's *classifyFn* callout function permits acceptable data in the stream to pass through unaltered, blocks data in the stream that is to be removed, and injects new or altered data into the stream when it is suitable.

A callout can replace data in the stream with other data by blocking the data that is to be replaced, and, at the same time, injecting the new data into the stream. In this situation, the new data is injected into the stream at the same point where the blocked data is removed from the stream.

For a callout driver to inject data into a data stream, it must first create an injection handle. This can be the same injection handle that is created for injecting modified packet data back into the network stack. See Inspecting Packet and Stream Data for information about how to create an injection handle.

For information about how to modify stream data, see the "Windows Filtering Platform Stream Edit Sample" in the Hardware Samples code gallery.

Data Logging

Article • 12/15/2021

To determine what data should be logged, a callout's classifyFn callout function can inspect any combination of the data fields, the metadata fields, and any raw data that is passed to it, as well as any relevant data that has been stored in a context associated with the filter or the data flow.

For example, if a callout keeps track of how many incoming (inbound) IPv4 packets are discarded by a filter at the network layer, the callout is added to the filter engine at the FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD layer. In this situation, the callout's classifyFn callout function might resemble the following example:

```C++
ULONG TotalDiscardCount = 0;
ULONG FilterDiscardCount = 0;

// classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
    IN OUT FWPS_CLASSIFY_OUT  *classifyOut
    )
{
  // Increment the total count of discarded packets
 InterlockedIncrement(&TotalDiscardCount);


  // Check whether a discard reason metadata field is present
 if (FWPS_IS_METADATA_FIELD_PRESENT(
 inMetaValues,
        FWPS_METADATA_FIELD_DISCARD_REASON))
  {
    // Check whether it is a general discard reason
 if (inMetaValues->discardMetadata.discardModule ==
        FWPS_DISCARD_MODULE_GENERAL)
    {
      // Check whether discarded by a filter
 if (inMetaValues->discardMetadata.discardReason ==
          FWPS_DISCARD_FIREWALL_POLICY)
      {
        // Increment the count of packets discarded by a filter
 InterlockedIncrement(&FilterDiscardCount);
      }
    }
  }

  // Take no action on the data
 classifyOut->actionType = FWP_ACTION_CONTINUE;
}
```

Related topics

classifyFn

Associating Context with a Data Flow

Article • 12/15/2021

For callouts that process data at a filtering layer that supports data flows, the callout driver can associate a context with each data flow. Such a context is opaque to the filter engine. The callout's classifyFn callout function can use this context to save state information specific to the data flow for the next time that it is called by the filter engine for that data flow. The filter engine passes this context to the callout's classifyFn callout function through the *flowContext* parameter. If no context is associated with the data flow, the *flowContext* parameter is zero.

To associate a context with a data flow, a callout's classifyFn callout function calls the **FwpsFlowAssociateContext0** function. For example:

C++

```
// Context structure to be associated with data flows
typedef struct FLOW_CONTEXT_ {
  .
  .  // Driver-specific content
  .
} FLOW_CONTEXT, *PFLOW_CONTEXT;

#define FLOW_CONTEXT_POOL_TAG 'fcpt'

// classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
    IN OUT FWPS_CLASSIFY_OUT  *classifyOut
  )
{
  PFLOW_CONTEXT context;
  UINT64 flowHandle;
  NTSTATUS status;

  ...

  // Check for the flow handle in the metadata
  if (FWPS_IS_METADATA_FIELD_PRESENT(
      inMetaValues,
      FWPS_METADATA_FIELD_FLOW_HANDLE))
  {
    // Get the flow handle
    flowHandle = inMetaValues->flowHandle;
    // Allocate the flow context structure
    context =
      (PFLOW_CONTEXT)ExAllocatePoolWithTag(
        NonPagedPool,
        sizeof(FLOW_CONTEXT),
        FLOW_CONTEXT_POOL_TAG
      );

    // Check the result of the memory allocation
    if (context == NULL)
    {

      // Handle memory allocation error
      ...
    }
    else
    {

      // Initialize the flow context structure
      ...

      // Associate the flow context structure with the data flow
      status = FwpsFlowAssociateContext0(
                 flowHandle,
                 FWPS_LAYER_STREAM_V4,
                 calloutId,
                 (UINT64)context
               );

      // Check the result
      if (status != STATUS_SUCCESS)
      {
        // Handle error
        ...
      }
    }
  }

  ...

}
```

If a context is already associated with a data flow, it must first be removed before a new context may be associated with the data flow. To remove a context from a data flow, a callout's classifyFn callout function calls the **FwpsFlowRemoveContext0** function. For example:

C++

```
// Context structure to be associated with data flows
typedef struct FLOW_CONTEXT_ {
  ...
} FLOW_CONTEXT, *PFLOW_CONTEXT;

#define FLOW_CONTEXT_POOL_TAG 'fcpt'

// classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
    OUT FWPS_CLASSIFY_OUT  *classifyOut
  )
{
  PFLOW_CONTEXT context;
  UINT64 flowHandle;
  NTSTATUS status;

  ...

  // Check for the flow handle in the metadata
 if (FWPS_IS_METADATA_FIELD_PRESENT(
    inMetaValues,
    FWPS_METADATA_FIELD_FLOW_HANDLE))
  {
    // Get the flow handle
     flowHandle = inMetaValues->flowHandle;

    // Check whether there is a context associated with the data flow
     if (flowHandle != 0)
     {
        // Get a pointer to the flow context structure
        context = (PFLOW_CONTEXT)flowContext;

        // Remove the flow context structure from the data flow
        status = FwpsFlowRemoveContext0(
                  flowHandle,
                  FWPS_LAYER_STREAM_V4,
                  calloutId
                );

      // Check the result
      if (status != STATUS_SUCCESS)
      {
        // Handle error
        ...
      }

      // Cleanup the flow context structure
      ...

      // Free the memory for the flow context structure
      ExFreePoolWithTag(
        context,
        FLOW_CONTEXT_POOL_TAG
        );
    }
  }

  ...

}
```

In the previous examples, the *calloutId* variable contains the run-time identifier for the callout. The run-time identifier is the same identifier that was returned to the callout driver when the callout driver registered the callout with the filter engine.

Processing Classify Callouts Asynchronously

Article • 12/15/2021

A WFP callout driver can authorize or deny a network operation, or admit or discard a network packet, by returning the action types **FWP_ACTION_PERMIT**, **FWP_ACTION_CONTINUE**, or **FWP_ACTION_BLOCK** from the *classifyFn* callout function. Frequently a callout driver cannot return an inspection decision from its *classifyFn* function until the indicated information, such as classifiable fields, metadata, or packets, can be forwarded for processing to another component, such as a user-mode application. In these cases a decision may have to be made asynchronously at some later time.

General Rules for Asynchronous Processing

WFP supports asynchronous processing of the *classifyFn* callout function. However, the mechanism for doing this differs according to the different layers.

Asynchronous ALE Classify
A callout driver must call the **FwpsPendOperation0** function from *classifyFn*. The asynchronous operation must be completed with a call to the **FwpsCompleteOperation0** function.

Asynchronous Packet Classify
A callout driver should return **FWP_ACTION_BLOCK** from the *classifyFn* function, with the **FWPS_CLASSIFY_OUT_FLAG_ABSORB** flag set. Network packets must be referenced or cloned. The asynchronous operation is completed by either reinjecting the cloned or modified packet or by silently discarding the packet.

Asynchronous ALE Classify That Includes Packets
A combination of the previous two procedures is used: the classify operation is pended and the packet is referenced or cloned, and at some time later the call to *classifyFn* is completed and the cloned packet is reinjected or discarded.

Special Cases and Considerations

ALE Connect vs. Receive/Accept Layers
When **FwpsCompleteOperation0** is called to complete a pended classify operation at an ALE connect layer (**FWPS_LAYER_ALE_AUTH_CONNECT_V4** or **FWPS_LAYER_ALE_AUTH_CONNECT_V6**), an ALE reauthorization classify operation is

triggered at the respective ALE connect layer. The callout driver should return an inspection decision from this reauthorization classify operation. You can detect an ALE reauthorization classify operation by checking whether the **FWP_CONDITION_FLAG_IS_REAUTHORIZE** flag is set.

The callout driver must maintain a unique state for each pended ALE_AUTH_CONNECT classify operation in such a way that the inspection decision for each classify operation can be looked up during a **FwpsCompleteOperation0**-triggered reauthorization. If packets are referenced or cloned during a pended ALE_AUTH_CONNECT classify operation (for example, for non-TCP connections), they can be reinjected after reauthorization occurs.

When **FwpsCompleteOperation0** is called during with a classify operation at an ALE receive/accept layer (**FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4** or **FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6**), **FwpsCompleteOperation0** does not trigger an ALE reauthorization. Instead a new call to *classifyFn* is made again when the cloned packet is reinjected incoming if the modification was not significant enough to bypass the filter. Permitting the self-injected clone from the ALE_RECV_ACCEPT layer effectively authorizes the incoming connection. If the incoming connection is not to be allowed, discard the incoming packet after it calls **FwpsCompleteOperation0**.

ALE Reauthorization
A callout driver can be reclassified at an ALE connect or receive/accept layer for events such as a policy change (for example, adding or removing a filter at the layer), detecting a new arrival interface, and re-keying a connection by using IPsec. Such a reauthorization cannot be pended by calling **FwpsCompleteOperation0**, and it is not necessary to do so. A callout driver should use the rules listed previously to process packets that are indicated during reauthorization.

Be aware that both incoming and outgoing packet can be reauthorized at ALE_AUTH_CONNECT or ALE_RECV_ACCEPT layers. For example, an incoming packet can be reauthorized at the ALE_AUTH_CONNECT layer. A callout driver must not assume that the direction of the packet is the same as the direction of the connection.

ALE_FLOW_ESTABLISHED Layers
Asynchronous processing is not supported at these layers (**FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4** or **FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6**).

INBOUND_TRANSPORT Layers
A callout driver must not perform asynchronous processing of packets that require ALE classify processing at an incoming (inbound) transport layer (**FWPS_LAYER_INBOUND_TRANSPORT_V4** or

FWPS_LAYER_INBOUND_TRANSPORT_V6). Doing this can interfere with flow creation. When WFP calls the *classifyFn* callout function at an incoming transport layer, it sets the **FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED** flag for those packets that require ALE classify processing. A callout driver should permit such packets from an INBOUND_TRANSPORT layer and should defer processing them until they reach an ALE_RECV_ACCEPT layer.

STREAM Layers

At a stream layer (**FWPS_LAYER_STREAM_V4** or **FWPS_LAYER_STREAM_V6**), TCP data segments are indicated instead of an IP or TCP header. The stream layer is also where a chain of net buffer lists can be indicated in one call to the *classifyFn* callout function. WFP makes available specialized clone and injection functions, **FwpsCloneStreamData0** and **FwpsStreamInjectAsync0**, for stream layer callouts to use.

Because of the ordered delivery nature of stream layer data, a callout driver must continue to clone and absorb data as long any stream data is still pending. Mixing asynchronous and synchronous operations for a given stream flow can result in undefined behavior.

Using Bind or Connect Redirection

Article • 09/27/2024

The connect/bind redirection feature of the Windows Filtering Platform (WFP) enables application layer enforcement (ALE) callout drivers to inspect and, if desired, redirect connections.

This feature is available in Windows 7 and later.

Note The ClassifyFunctions_ProxyCallouts.cpp module in the WFP driver sample includes code that demonstrates connect/bind redirection.

A WFP connection redirection callout redirects an application's connection request so that the application connects to a proxy service instead of the original destination. The proxy service has two sockets: one for the redirected original connection and one for the new proxied outbound connection.

A WFP redirect record is a buffer of opaque data that WFP must set on an outbound proxy connection at the **FWPM_LAYER_ALE_AUTH_CONNECT_REDIRECT_V4** and **FWPM_LAYER_ALE_AUTH_CONNECT_REDIRECT_V6** layers, so that the redirected connection and the original connection are logically related.

Changing the local address and port of a flow is only supported in the bind-redirect layer. This functionality isn't supported in the connect-redirect layer.

Layers Used for Redirection

Callout drivers can preform redirection at the following layers, which are called "redirect layers":

- FWPM_LAYER_ALE_BIND_REDIRECT_V4 (FWPS_LAYER_ALE_BIND_REDIRECT_V4)
- FWPM_LAYER_ALE_BIND_REDIRECT_V6 (FWPS_LAYER_ALE_BIND_REDIRECT_V6)
- FWPM_LAYER_ALE_CONNECT_REDIRECT_V4 (FWPS_LAYER_ALE_CONNECT_REDIRECT_V4)
- FWPM_LAYER_ALE_CONNECT_REDIRECT_V6 (FWPS_LAYER_ALE_CONNECT_REDIRECT_V6)

The layer at which redirection is performed determines the effect of the change. Changes at connect layers affect only the flow being connected. Changes at bind layers affect all connections that are using that socket.

The redirect layers are only available for Windows 7 and later versions of Windows. Callout drivers that support classification at these layers must register using **FwpsCalloutRegister1** or higher, not the older **FwpsCalloutRegister0** function.

ⓘ **Important**

Redirection is not available for use with all types of network traffic. The types of packets that are supported for redirection are shown in the following list:

- TCP
- UDP
- Raw UDPv4 without the header include option
- Raw ICMP

Performing Redirection

To redirect a connection, the callout driver must obtain a writable copy of the TCP 4-tuple information, make changes to it as needed, and apply the changes. A set of new functions are provided to obtain writable layer data and to apply it through the engine. Callout drivers have the option of making changes either inline in their classifyFn functions, or asynchronously in another function.

Callout drivers that implement redirection must use *classifyFn1* or later instead of *classifyFn0* as their classification callout function. To use *classifyFn1* or later, the callout must be registered by calling **FwpsCalloutRegister1** or later, not the older **FwpsCalloutRegister0**.

To perform redirection inline a callout driver must perform the following steps in its implementation of classifyFn:

1. Call **FwpsRedirectHandleCreate0** to obtain a handle that can be used to redirect TCP connections. This handle should be cached and used for all redirections. (This step is omitted for Windows 7 and earlier.)

2. In Windows 8 and later, you must query the redirection state of the connection by using the **FwpsQueryConnectionRedirectState0** function in your callout driver. This must be done to prevent infinite redirecting.

3. Call **FwpsAcquireClassifyHandle0** to obtain a handle that will be used for subsequent function calls.

4. Call **FwpsAcquireWritableLayerDataPointer0** to get the writable data structure for the layer in which classifyFn was called. Cast the *writableLayerData* out parameter to the structure corresponding to the layer, either **FWPS_BIND_REQUEST0** or **FWPS_CONNECT_REQUEST0**.

   Starting with Windows 8, if your callout driver is redirecting to a local service, you must call **FwpsRedirectHandleCreate0** to fill in the **localRedirectHandle** member of the **FWPS_CONNECT_REQUEST0** structure in order to make local proxying work.

5. Make changes to the layer data as needed:

   a. Save the original destination in the local redirect context as shown in the following example:

   C++

   ```
   FWPS_CONNECT_REQUEST* connectRequest = redirectContext-
   >connectRequest;
   // Replace "..." with your own redirect context size
   connectRequest->localRedirectContextSize = ...;
   // Store original destination IP/Port information in the
   localRedirectContext member
   connectRequest->localRedirectContext =    ExAllocatePoolWithTag(…);
   ```

   b. Modify the remote address as shown in the following example:

   C++

   ```
   // Ensure we don't need to worry about crossing any of the TCP/IP
   stack's zones
   if(INETADDR_ISANY((PSOCKADDR)&(connectRequest-
   >localAddressAndPort)))
   {
      INETADDR_SETLOOPBACK((PSOCKADDR)&(connectRequest-
   >remoteAddressAndPort));
   }
   else
   {
      INETADDR_SET_ADDRESS((PSOCKADDR)&(connectRequest-
   >remoteAddressAndPort),
                             INETADDR_ADDRESS((PSOCKADDR)&
   (connectRequest->localAddressAndPort)));
   }
   INETADDR_SET_PORT((PSOCKADDR)&connectRequest->remoteAddressAndPort,
                     RtlUshortByteSwap(params->proxyPort));
   ```

   c. If your callout driver is redirecting to a local service, it should set the local proxy PID in the **localRedirectTargetPID** member of the **FWPS_CONNECT_REQUEST0** structure.

d. If your callout driver is redirecting to a local service, it should set the redirect handle returned by FwpsRedirectHandleCreate0 in the **localRedirectHandle** member of the FWPS_CONNECT_REQUEST0 structure.

6. Call FwpsApplyModifiedLayerData0 to apply the changes made to the data.

7. In your proxy service (which could be in user mode or kernel mode), you must query redirect records and contexts as shown in the following example:

   C++

   ```cpp
   BYTE* redirectRecords;
   BYTE redirectContext[CONTEXT_SIZE];
   listenSock = WSASocket(…);
   result = bind(listenSock, …);
   result = listen(listenSock, …);
   clientSock = WSAAccept(listenSock, …);
   // opaque data to be set on proxy connection
   result = WSAIoctl(clientSock,
                     SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS,
                     redirectRecords, …);
   // callout allocated data, contains original destination information
   result = WSAIoctl(clientSock,
                     SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT,
                     redirectContext, …);
   // extract original destination IP and port from above context
   ```

8. In your proxy service (which could be in user mode or kernel mode), you must set redirect records on the proxy connection socket as shown in the following example to create a new outbound socket:

   C++

   ```cpp
   proxySock = WSASocket(…);
   result = WSAIoctl(
                 proxySock,
                 SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS,
                 redirectRecords, …);
   ```

9. Call FwpsReleaseClassifyHandle0 to release the classification handle obtained in step 2.

10. Call FwpsRedirectHandleDestroy0 to destroy the handle that was obtained in step 1.

To perform redirection asynchronously a callout driver must perform the following steps:

1. Call **FwpsRedirectHandleCreate0** to obtain a handle that can be used to redirect TCP connections. (This step is omitted for Windows 7 and earlier.)

2. In Windows 8 and later, you must query the redirection state of the connection by using the **FwpsQueryConnectionRedirectState0** function in your callout driver.

3. Call **FwpsAcquireClassifyHandle0** to obtain a handle that will be used for subsequent function calls. This step and steps 2 and 3 are performed in the callout driver's classifyFn callout function.

4. Call **FwpsPendClassify0** to put the classification in a pending state as shown in the following example:

   C++

   ```
   FwpsPendClassify(
           redirectContext->classifyHandle,
           0,
           &redirectContext->classifyOut);
   classifyOut->actionType = FWP_ACTION_BLOCK;
   classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
   ```

> ⓘ **Note**
>
> If you are targeting Windows 7, you must perform the following steps in a separate worker function. If you are targeting Windows 8 or later, you can perform all steps for asynchronous redirection from within the *classifyFn* and ignore Step 5.

5. Send the classification handle and the writable layer data to another function for asynchronous processing. The remaining steps are performed in that function, not in the callout driver's implementation of classifyFn.

6. Call **FwpsAcquireWritableLayerDataPointer0** to get the writable data structure for the layer in which classifyFn was called. Cast the *writableLayerData* out parameter to the structure corresponding to the layer, either **FWPS_BIND_REQUEST0** or **FWPS_CONNECT_REQUEST0**.

   Starting with Windows 8, if your callout driver is redirecting locally, you must call **FwpsRedirectHandleCreate0** to fill in the **localRedirectHandle** member of the **FWPS_CONNECT_REQUEST0** structure in order to make proxying work.

7. Store any callout-specific context information in a private context structure as shown in the following example:

C++

```
redirectContext->classifyHandle = classifyHandle;
redirectContext->connectRequest = connectRequest;
redirectContext->classifyOut = *classifyOut; // deep copy
// store original destination IP, port
```

8. Make changes to the layer data as needed.

9. Call FwpsApplyModifiedLayerData0 to apply the changes made to the data. Set the **FWPS_CLASSIFY_FLAG_REAUTHORIZE_IF_MODIFIED_BY_OTHERS** flag if you wish to be re-authorized in the event that another callout modifies the data further.

10. Call FwpsCompleteClassify0 to complete the classify operation asynchronously as shown in the following example:

C++

```
FwpsCompleteClassify(
        redirectContext->classifyHandle,
        0,
        &redirectContext->classifyOut);
classifyOut->actionType = FWP_ACTION_PERMIT;
classifyOut->rights |= FWPS_RIGHT_ACTION_WRITE;
```

11. Call FwpsReleaseClassifyHandle0 to release the classification handle obtained in step 1.

Handling Connect Redirection from Multiple Callouts

It's possible that more than one callout driver will initiate connect redirection for the same flow. Callouts that perform connect redirection should be aware of other requests and respond appropriately.

The **FWPS_RIGHT_ACTION_WRITE** flag should be set whenever a callout pends a classification. Your callout should test for the **FWPS_RIGHT_ACTION_WRITE** flag to check the rights for your callout to return an action. If this flag isn't set, your callout can still return a **FWP_ACTION_BLOCK** action in order to veto a **FWP_ACTION_PERMIT** action that was returned by a previous callout.

In Windows 8 and later, your callout driver must query the redirection state of the connection (to see if your callout driver or another callout driver has modified it) by using the FwpsQueryConnectionRedirectState0 function. If the connection is redirected by your callout driver, or if it was previously redirected by your callout driver, the callout driver should do nothing. Otherwise, it should also check for local redirection as shown in the following example:

C++

```cpp
FwpsAcquireWritableLayerDataPointer(...,(PVOID*)&connectRequest), ...);
if(connectRequest->previousVersion->modifierFilterId != filterId)
{
    if(connectRequest->previousVersion->localRedirectHandle)
    {
        classifyOut->actionType = FWP_ACTION_PERMIT;
        classifyOut->rights &= FWPS_RIGHT_ACTION_WRITE;
        FwpsApplyModifiedLayerData(
                classifyHandle,
                (PVOID)connectRequest,
                FWPS_CLASSIFY_FLAG_REAUTHORIZE_IF_MODIFIED_BY_OTHERS);
    }
}
```

If the connection is to a local proxy, your callout driver shouldn't attempt to redirect it.

Callout drivers that use connect redirection should register at the ALE authorization connect layer (**FWPS_LAYER_ALE_AUTH_CONNECT_V4** or **FWPS_LAYER_ALE_AUTH_CONNECT_V6**) and check the following two metadata values for indications where the **FWP_CONDITION_FLAG_IS_CONNECTION_REDIRECTED** flag is set:

- **FWPS_METADATA_FIELD_LOCAL_REDIRECT_TARGET_PID** contains the process identifier for the process that is responsible for the redirected flow.
- **FWPS_METADATA_FIELD_ORIGINAL_DESTINATION** contains the address of the original destination for the flow.

The FWPS_CONNECT_REQUEST0 structure contains a member called **localRedirectTargetPID**. For any loopback connect redirection to be valid, this field must be populated with the PID of the process that will be responsible for the redirected flow. This is the same data that the engine passes at the ALE authorization connect layers as **FWPS_METADATA_FIELD_LOCAL_REDIRECT_TARGET_ID**.

Starting with Windows 8, the proxy service needs to issue the SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS and SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT IOCTLs, using WSAIoctl, against the original endpoint of the proxy service. Additionally, the SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS IOCTL must be issued, using **WSAIoctl**, on the new (proxied) socket.

Related topics

WFP Version-Independent Names and Targeting Specific Versions of Windows

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

ALE Endpoint Lifetime Management

Article • 12/15/2021

A callout driver that supports application layer enforcement (ALE) may need to allocate resources to process indications. This topic describes how to configure a callout driver to release such resources when the associated endpoint is closed. ALE endpoint lifetime management is supported in Windows 7 and later versions of Windows.

To manage resources associated with ALE endpoints, a callout driver can register at the following layers:

- FWPS_LAYER_ALE_RESOURCE_RELEASE_V4 (FWPM_LAYER_ALE_RESOURCE_RELEASE_V4)
- FWPS_LAYER_ALE_RESOURCE_RELEASE_V6 (FWPM_LAYER_ALE_RESOURCE_RELEASE_V6)
- FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4 (FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V4)
- FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6 (FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V6)

An ALE resource release layer is indicated for every indication at the corresponding ALE resource assignment layer (for example, FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4). To ensure that callout drivers can match the release layer to the assignment layer, the FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE metadata field is provided at both layers and each endpoint is assigned a unique handle.

ALE endpoint closure layers are invoked differently depending on the type of endpoint. For TCP connections, an ALE endpoint closure is indicated for every ALE authorize connect layer (for example FWPS_LAYER_ALE_AUTH_CONNECT_V4) or ALE authorize receive accept layer (for example FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4) indication. As with ALE resource release indications, the engine assigns a unique handle for each endpoint and passes it in the FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE metadata field. For non-TCP endpoints, an ALE endpoint closure layer is invoked for each endpoint regardless of the number of unique remote peers the socket communicates with. An ALE endpoint closure layer is also invoked for each TCP listening socket.

Callouts registered for an ALE endpoint closure layer can pend classification. This enables the callout to reinject any packets queued for asynchronous processing before the endpoint is shut down. To pend classification, the callout driver must call **FwpsPendClassify0** followed by a call to **FwpsCompleteClassify0** when processing is complete.

When applicable, the engine will indicate a unique handle for the parent endpoint in the FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE metadata field. This enables the callout driver to track parent/child relationships, if required.

Processing Flow Delete Callouts

Article • 12/15/2021

When a data flow that is being processed by a callout is stopped, the filter engine calls the callout's *flowDeleteFn* callout function if the callout driver previously associated a context with the data flow. A callout's *flowDeleteFn* callout function performs any necessary clean up of the context that the callout driver associated with the data flow before the data flow is stopped.

For example:

```C++
// Context structure to be associated with data flows
typedef struct FLOW_CONTEXT_ {
  ...
} FLOW_CONTEXT, *PFLOW_CONTEXT;

#define FLOW_CONTEXT_POOL_TAG 'fcpt'

// flowDeleteFn callout function
VOID NTAPI
 FlowDeleteFn(
    IN UINT16  layerId,
    IN UINT32  calloutId,
    IN UINT64  flowContext
    )
{
  PFLOW_CONTEXT context;

  // Get the flow context structure
 context = (PFLOW_CONTEXT)flowContext;

  // Cleanup the flow context structure
  ...

  // Free the memory for the flow context structure
 ExFreePoolWithTag(
 context,
    FLOW_CONTEXT_POOL_TAG
    );
}
```

The filter engine automatically removes the context that a callout associated with a data flow when the data flow is stopped. Therefore, a callout is not required to call the **FwpsFlowRemoveContext0** function from its *flowDeleteFn* callout function to remove the context from the data flow.

Using Packet Tagging

Article • 12/15/2021

A callout driver can tag packets of interest and receive notification of events that happen to the tagged packets. Packet tagging is supported in Windows 7 and later versions of Windows.

To use packet tagging, the callout driver must implement the *FWPS_NET_BUFFER_LIST_NOTIFY_FN0* or *FWPS_NET_BUFFER_LIST_NOTIFY_FN1* callback function. This function will receive all of the status notifications for the tagged packets. Before individual packets can be tagged, the callout driver must obtain a special context tag by calling **FwpsNetBufferListGetTagForContext0**. The callout driver can use the same context tag for some or all of the tagged packets. For example, a callout driver might differentiate between types of tagged packets by using different context tags.

To tag packets, the callout driver uses **NET_BUFFER_LIST** structures. The callout driver makes calls to **FwpsNetBufferListAssociateContext0** to tag individual **NET_BUFFER_LIST** structures. The context the callout driver associates with the packet is an arbitrary unsigned 64-bit value. When an event is triggered, the *FWPS_NET_BUFFER_LIST_NOTIFY_FN0* or *FWPS_NET_BUFFER_LIST_NOTIFY_FN1* callback passes the context as an input parameter so that the callout driver can identify individual tagged packets. The context is not used or evaluated by the filtering engine. It is only passed to the callback for use by the callout driver.

Contexts are removed from tagged packets automatically when the packets leave the stack. However, if the packets never enter the TCP/IP stack — for example, in the case of an NDIS filter driver — the contexts will need to be removed manually by calling **FwpsNetBufferListRemoveContext0** with the *netBufferList* parameter set to **NULL**.

If a callout needs to abort tagging operations early, contexts can be removed by calling **FwpsNetBufferListRemoveContext0**. Removing a context generally triggers an **FWPS_NET_BUFFER_LIST_CONTEXT_REMOVED** event. For more information about the events that can be triggered, see the **FWPS_NET_BUFFER_LIST_EVENT_TYPE0** enumeration. In some cases no event will be triggered, such as when the packet never enters the TCP/IP stack for processing.

When a tagged packet is cloned, the callout driver can move or copy the context to the clone packet. To move the context (in the case of a clone), the callout driver must call **FwpsNetBufferListRetrieveContext0** with the *removeContext* parameter set to **TRUE**. Then the context can be associated with the new packet. The process for copying the context (in the case of a duplication) is the same except that the *removeContext* parameter of **FwpsNetBufferListRetrieveContext0** must be set to **FALSE**.

Packets tagged from TCP/IP layers can be retrieved from an NDIS filter driver. The reverse is also true. Packet tagging is not available from stream layers where no packets are indicated except data segments.

A callout driver can retrieve the context for a packet outside of the *FWPS_NET_BUFFER_LIST_NOTIFY_FN0* or *FWPS_NET_BUFFER_LIST_NOTIFY_FN1* function by calling **FwpsNetBufferListRetrieveContext0**. Typically, a callout driver will retrieve the context in its classifyFn callback.

Related topics

classifyFn

FWPS_NET_BUFFER_LIST_EVENT_TYPE0

FWPS_NET_BUFFER_LIST_NOTIFY_FN0

FWPS_NET_BUFFER_LIST_NOTIFY_FN1

FwpsNetBufferListAssociateContext0

FwpsNetBufferListGetTagForContext0

FwpsNetBufferListRemoveContext0

FwpsNetBufferListRetrieveContext0

NET_BUFFER_LIST

NDIS Filter Drivers

Using Layer 2 Filtering

Article • 12/15/2021

Layer 2 filtering is supported in Windows 8 and later versions of Windows.

This WFP feature allows filtering on fields of the layer 2 MAC header. These layers are invoked on a per-packet basis for all packets that are sent or received by the host machine. The layers are invoked prior to packet reassembly on the inbound path and after packet fragmentation on the outbound path. These layers are accessed from an NDIS lightweight filter (LWF) driver.

> ⓘ **Note**
>
> A callout should not inject packets at a layer if it does not already have a corresponding filter at that layer. The injection of the **NET_BUFFER_LIST** structures should be coordinated with the filter addition and removal so that injection is only performed when the filter exists in the corresponding layer. In addition, providers should not remove filters that belong to other providers.

This section includes the following topics:

- Injecting MAC Frames
- Classifying Chained Network Buffer Lists
- WFP Layer 2 Layers and Fields

Injecting MAC Frames

A callback driver calls the **FwpsInjectMacReceiveAsync0** function to reinject a previously absorbed MAC frame (or a clone of the frame) back to the layer 2 inbound data path it was intercepted from, or to inject an invented MAC frame in the inbound data path.

A callback driver calls the **FwpsInjectMacSendAsync0** function to reinject a previously absorbed MAC frame (or a clone of the frame) back to the layer 2 outbound data path it was intercepted from, or to inject an invented MAC frame in the outbound data path.

The *netBufferLists* parameter can be a NET_BUFFER_LIST chain. However the completion function could be invoked multiple times each, completing a segment (or single NET_BUFFER_LIST) of the chain.

Injected frames could get classified again if the packets match the same filter as originally classified. Therefore, as with callouts at IP layers, layer 2 callouts must also protect against infinite packet inspection by calling **FwpsQueryPacketInjectionState0**.

Also, you must have callouts at the layer where you inject. Otherwise, your injected NET_BUFFER_LIST will not be completed to your completion function, and the NET_BUFFER_LIST will go further up the stack. In this case, the behavior is undefined, because NDIS will try to pass the injected NET_BUFFER_LIST to the next component in the stack.

The NET_BUFFER_LIST**Status** member contains the stack injection’s status result. The stack injection’s status result is the status that the stack puts in the NET_BUFFER_LIST after a WFP injection function returns **STATUS_SUCCESS**. You should use the **NT_SUCCESS** macro to check the stack injection's status in the **Status** member. If the **Status** value is **STATUS_SUCCESS**, the injection succeeded with no further information. **Status** member values that are greater than **STATUS_SUCCESS** mean that the injection succeeded, but there might be more information about the injection that should be considered. **Status** member values that are less than **STATUS_SUCCESS** mean that the injection failed for the reason specified in the **Status** member.

Classifying Chained Network Buffer Lists

By default, a callout driver can only classify network buffer lists individually. However, a callout driver can classify NET_BUFFER_LIST chains for better performance, if it does both of the following:

- Specifies the **FWP_CALLOUT_FLAG_ALLOW_L2_BATCH_CLASSIFY** flag in the **Flags** member of the **FWPS_CALLOUT2** structure.
- Registers a *classifyFn2* function that can classify NET_BUFFER_LIST chains.

⚠ Warning

However, if a callout driver does set the **FWP_CALLOUT_FLAG_ALLOW_L2_BATCH_CLASSIFY** flag, it cannot use the following functions to modify NET_BUFFER_LISTs.

- FwpsReferenceNetBufferList0
- FwpsDereferenceNetBufferList0
- FwpsAllocateCloneNetBufferList0
- FwpsFreeCloneNetBufferList0

With this flag set, **FwpsAllocateCloneNetBufferList0** will always return an **INVALID_PARAMETER** error. This may unexpectedly cause a 3rd party callout driver to fail to manage the reference count of NET_BUFFER_LISTs, causing send and receive operations to stop.

WFP Layer 2 Layers and Fields

Run-time Filtering Layer Identifiers for virtual switch filtering include:

FWPS_LAYER_INBOUND_MAC_FRAME_ETHERNET

FWPS_LAYER_OUTBOUND_MAC_FRAME_ETHERNET

FWPS_LAYER_INBOUND_MAC_FRAME_NATIVE

FWPS_LAYER_OUTBOUND_MAC_FRAME_NATIVE

Data Field Identifiers for virtual switch filtering include:

FWPS_FIELDS_INBOUND_MAC_FRAME_ETHERNET

FWPS_FIELDS_OUTBOUND_MAC_FRAME_ETHERNET

FWPS_FIELDS_INBOUND_MAC_FRAME_NATIVE

FWPS_FIELDS_OUTBOUND_MAC_FRAME_NATIVE

Using Proxied Connections Tracking

Article • 12/06/2022

Proxied connections tracking is supported in Windows 8 and later versions of Windows.

This WFP feature facilitates tracking of redirection “records” from the initial redirect of a connection to the final connection to the destination. WFP also allows a callout driver to redirect connections.

Proxied Connections Tracking

With the presence of multiple proxies (for example, developed by different ISVs) the connection used by one party to communicate with the final destination could in turn be redirected by a 2nd party; and that new connection could again be redirected by the original party. Without connection tracking, the original connection might never reach its final destination as it gets stuck in the infinite proxy loop.

Additions to the Data Field Identifiers to support connection tracking include:

FWPS_FIELD_Xxx_ALE_ORIGINAL_APP_ID
The full path of the original application for proxy connections. If the application has not been proxied, this path is identical to the xxx_ALE_APP_ID.

FWPS_FIELD_Xxx_PACKAGE_ID
The package identifier is a security identifier (SID) that identifies the associated AppContainer process.

Redirecting Connections

A callout driver calls the FwpsRedirectHandleCreate0 function to create a handle that can be used to redirect TCP connections.

This section includes the following topics:

Using a Redirection Handle

Querying the Redirect State

Using a Redirection Handle

Before an ALE connect redirection callout can redirect connections to a local process, it must obtain a redirect handle with the FwpsRedirectHandleCreate0 function and put the handle in the FWPS_CONNECT_REQUEST0 structure. The callout modifies the structure in the classifyFn for the ALE connect redirect layers.

The FWPS_CONNECT_REQUEST0 structure contains the following members for redirection:

Term	Description
localRedirectHandle	The redirect handle that the callout driver created by calling the FwpsRedirectHandleCreate0 function.
localRedirectContext	A callout driver context area that the callout driver allocated by calling the ExAllocatePoolWithTag function.
localRedirectContextSize	The size, in bytes, of the callout supplied context area.

After a callout driver has finished using a redirect handle, it must call the FwpsRedirectHandleDestroy0 function to destroy the handle.

Querying the Redirect State

A callout driver calls the FwpsQueryConnectionRedirectState0 function to get the redirect state of a connection. The FWPS_CONNECTION_REDIRECT_STATE enumeration is the return type for a call to the **FwpsQueryConnectionRedirectState0** function.

If the redirect status is FWPS_CONNECTION_NOT_REDIRECTED, the ALE_CONNECT_REDIRECT callout can proceed to proxy the connection.

If the redirect status is FWPS_CONNECTION_REDIRECTED_BY_SELF, the ALE_CONNECT_REDIRECT callout should return FWP_ACTION_PERMIT/FWP_ACTION_CONTINUE.

If the redirect status is FWPS_CONNECTION_REDIRECTED_BY_OTHER, the ALE_CONNECT_REDIRECT callout could proceed to proxy the connection if it does not trust the other inspector’s result.

If the redirect status is FWPS_CONNECTION_PREVIOUSLY_REDIRECTED_BY_SELF, the ALE_CONNECT_REDIRECT callout must not perform redirection even if other inspectors’ results are not acceptable. In this case, it must either permit or block the connection (at the ALE_AUTH_CONNECT layer).

Using Virtual Switch Filtering

Article • 12/15/2021

Overview of Virtual Switch Filtering

Virtual Switch Filtering is supported in Windows 8 and later versions of Windows.

This WFP feature allows filtering on fields of the MAC header, IP header, and upper protocol ports as well as virtual switch specific fields such as virtual port (VPort) and virtual machine identifier (VM ID). These layers are invoked on a per-packet basis for all packets traversing the virtual switch. These layers are accessed from a virtual switch extension filter—a type of NDIS lightweight filter (LWF) driver.

A callout driver calls the **FwpsvSwitchEventsSubscribe0** function to register callback entry points for virtual switch layer events.

The entry points for the callback notification functions are specified in an **FWPS_VSWITCH_EVENT_DISPATCH_TABLE0** structure. The callback functions that are available include:

- *FWPS_VSWITCH_FILTER_ENGINE_REORDER_CALLBACK0*
- *FWPS_VSWITCH_INTERFACE_EVENT_CALLBACK0*
- *FWPS_VSWITCH_LIFETIME_EVENT_CALLBACK0*
- *FWPS_VSWITCH_POLICY_EVENT_CALLBACK0*
- *FWPS_VSWITCH_PORT_EVENT_CALLBACK0*
- *FWPS_VSWITCH_RUNTIME_STATE_RESTORE_CALLBACK0*
- *FWPS_VSWITCH_RUNTIME_STATE_SAVE_CALLBACK0*

The **FWPS_VSWITCH_EVENT_TYPE** enumeration defines the values for the *eventType* parameter of the virtual switch notification functions.

The callout driver must eventually call **FwpsvSwitchEventsUnsubscribe0** to free the system resources.

If a callout driver returns STATUS_PENDING from a WFP notification function, WFP will return STATUS_PENDING to the OID request handler. The callout driver must call the **FwpsvSwitchNotifyComplete0** function to complete the pending operation. After the **FwpsvSwitchNotifyComplete0** call, WFP calls the **NdisFOidRequestComplete** function to complete the OID for the virtual switch.

Callbacks should not add or delete WFP filters synchronously in the context of the notification functions. In addition, if the notification function allows the callback to return STATUS_PENDING, and the callout returns STATUS_PENDING, the callout should not add or delete WFP filters before completing the notification.

WFP Virtual Switch Filter Layer and Fields

Run-time Filtering Layer Identifiers for virtual switch filtering include:

- FWPS_LAYER_INGRESS_VSWITCH_ETHERNET
- FWPS_LAYER_EGRESS_VSWITCH_ETHERNET
- FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V4
- FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V6
- FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V4
- FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V6

Data Field Identifiers for virtual switch filtering include:

- **FWPS_FIELDS_EGRESS_VSWITCH_ETHERNET**
- **FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V4**
- **FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V6**
- **FWPS_FIELDS_INGRESS_VSWITCH_ETHERNET**
- **FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V4**
- **FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V6**

Guidance For WFP Virtual Switch Callout Writers

Port 0 Traffic

For WFP virtual switch callouts, traffic from port 0 (the default port ID) is trusted and should not be filtered. This is because, generally, traffic over port 0 originates from other extensions in the driver stack and is thus treated by the data path as privileged and trusted. Virtual switch extensions will sparingly use port 0 for situations such as originating a control packet, which should not be filtered and rejected by any underlying extensions. For more information about Hyper-V extensible switch source port mofification, see Modifying a Packet's Extensible Switch Source Port Data.

Callout Matching Rules

When defining a matching rule for filtering, virtual switch callouts should not use the MAC address as a basis for comparison. MAC addresses can change at runtime, and some ports may generate traffic from multiple MAC addresses. Instead, callouts should use a more durable matching rule such as NIC ID, which will not change.

I/O Virtualization (IOV) and WFP Coexistence

WFP cannot be enabled on an IOV switch and is blocked by the OS if an attempt is made to enable it.

Enabling or Disabling WFP

Installers for WFP virtual switch callouts should not modify the WFP extension enabled state; that is, they should not enable or disable WFP itself.

Unloading a Callout Driver

Article • 12/15/2021

To unload a callout driver, the operating system calls the callout driver's unload function. For more information about how to specify a callout driver's unload function, see Specifying an Unload Function.

A callout driver's unload function guarantees that the callout driver's callouts are unregistered from the filter engine before the callout driver is unloaded from system memory. A callout driver calls either the **FwpsCalloutUnregisterById0** function or the **FwpsCalloutUnregisterByKey0** function to unregister a callout from the filter engine. A callout driver must not return from its unload function until after it has successfully unregistered all its callouts from the filter engine.

After a callout driver has unregistered all its callouts from the filter engine, it must delete the device object that it created before it originally registered its callouts. A callout driver that is based on the Windows Driver Model (WDM) calls the **IoDeleteDevice** function to delete the device object. A callout driver that is based on the Windows Driver Frameworks (WDF) calls the **WdfObjectDelete** function to delete the framework device object.

A callout driver must also destroy any packet injection handle that it previously created by calling the **FwpsInjectionHandleDestroy0** function before it returns from its unload function.

For example:

```C++
// Device object
PDEVICE_OBJECT deviceObject;

// Variable for the run-time callout identifier
UINT32 CalloutId;

// Injection handle
HANDLE injectionHandle;

// Unload function
VOID
 Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
  NTSTATUS status;
  // Unregister the callout
 status =
 FwpsCalloutUnregisterById0(
 CalloutId
      );

  // Check result
 if (status == STATUS_DEVICE_BUSY)
  {
    // For each data flow that is being processed by the
    // callout that has an associated context, clean up
    // the context and then call FwpsFlowRemoveContext0
    // to remove the context from the data flow.
    ...

    // Finish unregistering the callout
 status =
 FwpsCalloutUnregisterById0(
 CalloutId
        );
  }

  // Check status
 if (status != STATUS_SUCCESS)
  {
    // Handle error
    ...
  }

  // Delete the device object
 IoDeleteDevice(
 deviceObject
    );

  // Destroy the injection handle
 status =
 FwpsInjectionHandleDestroy0(
 injectionHandle
      );

  // Check status
 if (status != STATUS_SUCCESS)
  {
    // Handle error
    ...
  }
}
```

The previous example assumes a WDM-based callout driver. For a WDF-based callout driver, the only difference is the parameter that is passed to the callout driver's unload function and how the callout driver deletes the framework device object.

C++

```
WDFDEVICE wdfDevice;

VOID
 Unload(
    IN WDFDRIVER Driver;
    )
{

  ...

  // Delete the framework device object
 WdfObjectDelete(
 wdfDevice
    );

  ...
}
```

Callout Driver Installation

Article • 12/15/2021

This section discusses callout driver installation and includes the following topics:

INF Files for Callout Drivers

Installation of Callout Drivers

Digital Signatures for Callout Drivers

INF Files for Callout Drivers

Article • 12/15/2021

A Windows Filtering Platform callout driver is installed by a setup information file (INF) file. INF files for callout drivers contain only the following INF file sections:

INF Version Section

INF SourceDisksNames Section

INF SourceDisksFiles Section

INF DestinationDirs Section

INF DefaultInstall Section

INF DefaultInstall.Services Section

INF Strings Section

For example:

INF

```
;
; Example callout driver INF file
;

[Version]
Signature = "$Windows NT$"
Provider = %Msft%
CatalogFile = "ExampleCalloutDriver.cat"
DriverVer = 01/15/05,1.0

[SourceDisksNames]
1 = %DiskName%

[SourceDisksFiles]
ExampleCalloutDriver.sys = 1

[DestinationDirs]
DefaultDestDir = 12 ; %windir%\system32\drivers
ExampleCalloutDriver.DriverFiles = 12 ; %windir%\system32\drivers

[DefaultInstall]
OptionDesc = %Description%
CopyFiles = ExampleCalloutDriver.DriverFiles

[DefaultInstall.Services]
AddService = %ServiceName%,,ExampleCalloutDriver.Service
```

```
[DefaultUninstall]
DelFiles = ExampleCalloutDriver.DriverFiles

[DefaultUninstall.Services]
DelService = ExampleCalloutDriver,0x200 ; SPSVCINST_STOPSERVICE

[ExampleCalloutDriver.DriverFiles]
ExampleCalloutDriver.sys,,,0x00000040 ; COPYFLG_OVERWRITE_OLDER_ONLY

[ExampleCalloutDriver.Service]
DisplayName = %ServiceName%
Description = %ServiceDesc%
ServiceType = 1  ; SERVICE_KERNEL_DRIVER
StartType = 0    ; SERVICE_BOOT_START
ErrorControl = 1 ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\ExampleCalloutDriver.sys

[Strings]
Msft = "Microsoft Corporation"
DiskName = "Example Callout Driver Installation Disk"
Description = "Example Callout Driver"
ServiceName = "ExampleCalloutDriver"
ServiceDesc = "Example Callout Driver"
```

Installation of Callout Drivers

Article • 12/15/2021

A callout driver can be installed by right-clicking the driver's setup information file (INF) file and selecting **Install** from the pop-up menu that appears.

After a callout driver has been successfully installed, it can be loaded (started) by typing the following at the command prompt:

C++

```
net start drivername
```

Depending on the value specified for the **StartType** entry in the [*drivername*.Services] section of the INF file, the callout driver might be automatically loaded the next time that the system is restarted. A callout driver should usually specify zero (SERVICE_BOOT_START) for this value so that the driver is loaded and its callouts are registered before the filter engine is started. See the **INF AddService Directive** for more information.

A callout driver that is currently loaded can be unloaded (stopped) by typing the following at the command prompt:

C++

```
net stop drivername
```

A callout driver can also be installed, loaded (started), unloaded (stopped), and/or uninstalled by writing a user-mode application that calls the Win32 Service Control Manager API. For more information about Win32 service control functions, such as **CreateService**, **OpenService**, **StartService**, **ControlService**, and **DeleteService**, see the Microsoft Windows SDK.

> **Note**
>
> Starting in Windows 8 and later, callout drivers cannot be viewed or managed in the Device Manager because the Plug-and-Play (PnP) manager no longer creates device representations for non-PnP (legacy) devices.

Digital Signatures for Callout Drivers

Article • 09/12/2022

To guarantee the quality and integrity of the driver, all drivers must be digitally signed. This includes Windows Filtering Platform callout drivers.

For more information, see the following topics:

- Driver Signing
- Windows Hardware Lab Kit (HLK)

Callout Driver Programming Considerations

Article • 12/15/2021

Consider the following topics when you program a Windows Filtering Platform callout driver.

User Mode vs. Kernel Mode

If the desired filtering can be done by using the standard filtering functionality that is built in to the Windows Filtering Platform, independent software vendors (ISVs) should write user-mode management applications to configure the filter engine instead of writing kernel-mode callout drivers. A kernel-mode callout driver should only be written when you must process the network data in ways that cannot be handled by the standard, built-in filtering functionality. For information about how to write a user-mode Windows Filtering Platform management application, see the Windows Filtering Platform documentation in the Microsoft Windows SDK.

Choice of Filtering Layer

A callout driver should filter the network data at the highest possible filtering layer in the network stack. For example, if the desired filtering task can be handled at the stream layer, it should not be implemented at the network layer. For more information about recommendations of the filtering layers your driver should use to guarantee compatibility with IPsec in Windows, see Developing IPsec-Compatible Callout Drivers.

Blocking at the Application Layer Enforcement (ALE) Flow Established Layers

Usually, if a callout has been added to the filter engine at one of the *ALE flow established* filtering layers (FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 or FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6), its *classifyFn* callout function should never return FWP_ACTION_BLOCK for the action. A decision to authorize or reject a connection should not be made at one of the ALE flow established filtering layers. Such a decision should always be made at one of the other ALE filtering layers.

The only valid reason for such a *classifyFn* callout function to return FWP_ACTION_BLOCK for the action is if an error occurs that could pose a potential security risk if the established connection is not ended. In this case, returning

FWP_ACTION_BLOCK for the action closes the connection to prevent the potential security risk from being exploited.

Callout Function Execution Time

Because the filter engine typically calls a callout's callout functions at IRQL = DISPATCH_LEVEL, make sure that these functions complete their execution as quickly as possible to keep the system running efficiently. Extended execution at IRQL = DISPATCH_LEVEL can adversely affect the overall performance of the system.

Injecting Into the Receive Data Path

Callouts should recalculate IP checksums before they call packet injection functions that inject into the receive data path because the checksum in the original packet might not be correct when the packet is reassembled from IP packet fragments. There is no reliable mechanism that indicates whether a net buffer list is reassembled from fragments.

Inline Injection of TCP Packet from Transport Layers

Because of the TCP stack's locking behavior, a callout at the transport layer cannot inject a new or cloned TCP packet from the classifyFn callout function. If inline injection is desired, the callout must queue a DPC to perform the injection.

Outgoing IP Header Alignment

The MDL that describes the IP header in a net buffer list (**NET_BUFFER_CURRENT_MDL**(**NET_BUFFER_LIST_FIRST_NB**(*netBufferList*))) must be pointer-aligned when one of the packet injection functions is used to inject packet data into an outgoing path. Because an incoming packet's IP header MDL may be pointer-aligned, a callout must rebuild the IP header (if not already aligned) when injecting an incoming packet into an outgoing path.

Related topics

Windows Filtering Platform Callout Drivers

Porting Packet-Processing Drivers and Apps to WFP

Article • 12/15/2021

Windows Filtering Platform (WFP) enables TCP/IP packet filtering, inspection, and modification, connection monitoring or authorization, IPsec rules and processing, and RPC filtering. Generally, you must convert your TCP/IP filtering or connection monitoring component in Windows XP and Windows Server 2003 to use a WFP user-mode application or service, a WFP kernel-mode callout driver, or both for Windows Vista and Windows Server 2008 and later. The following table lists the existing methods for packet processing in Windows XP and Windows Server 2003 and how you must change them in Windows Vista and Windows Server 2008 and later to use WFP.

Note As of Windows 8, the Transport Driver Interface (TDI) feature and Layered Service Providers (LSPs) feature are deprecated.

Existing method in Windows XPand Windows Server 2003	New method in Windows Vista and Windows Server 2008 and later
Firewall hook or filter hook driver for simple packet filtering.	User-mode application or service that uses the WFP Win32 API.
Firewall hook or filter hook driver for deep packet inspection or modification.	IP layer, Transport layer, or Application Layer Enforcement (ALE) layer callout driver and optional user-mode application or service that uses the WFP Win32 API.
Transport Driver Interface (TDI) filter driver for simple packet filtering.	User-mode application or service that uses the WFP Win32 API.
TDI filter driver for deep packet or stream inspection or modification.	Transport layer, Stream layer, and/or ALE callout driver and optional user-mode application or service that uses the WFP Win32 API

Existing method in Windows XPand Windows Server 2003	New method in Windows Vista and Windows Server 2008 and later
TDI filter driver for TCP connection or User Datagram Protocol (UDP) traffic management.	For TCP connection management: ALE callout driver and optional user-mode application or service that uses the WFP Win32 API. For TCP proxying: • In Windows Vista: Packet modification callout driver. • In Windows 7 and later: ALE_REDIRECT layer callout driver. For MAC-level filtering: • In Windows 8 and later: MAC_FRAME layer callout driver. • In Windows Vista and Windows 7: NDIS lightweight filter driver. For UDP traffic management: Stream or Datagram Data layer callout driver and optional user-mode application or service that uses the WFP Win32 API.
Windows Sockets LSP for simple packet filtering.	User-mode application or service that uses the WFP Win32 API.
Windows Sockets LSP for deep packet inspection or modification.	IP layer, ALE, Transport (such as Datagram Data), or Stream layer callout driver and optional user-mode application or service that uses the WFP Win32 API.
Network Device Interface Specification (NDIS) intermediate driver for simple packet filtering.	For IP-based filtering: User-mode application or service that uses the WFP Win32 API. For MAC-based filtering: • In Windows 8 and later: MAC_FRAME layer callout driver. • In Windows Vista and Windows 7: NDIS lightweight filter driver.

Existing method in Windows XPand Windows Server 2003	New method in Windows Vista and Windows Server 2008 and later
NDIS intermediate driver for TCP connection or UDP traffic management.	TCP connection management: ALE callout driver and optional user-mode application or service that uses the WFP Win32 API. UDP traffic management: ALE or Transport layer callout driver and optional user-mode application or service that uses the WFP Win32 API.
NDIS lightweight filter driver to perform media access control (MAC)-level filtering.	In Windows 8 and later: MAC_FRAME layer callout driver. In Windows Vista and Windows 7: NDIS lightweight filter driver.

Developing IPsec-Compatible Callout Drivers

Article • 12/15/2021

Layers That Are Compatible With IPsec

To be fully compatible with the Windows implementation of IPsec that begins with Windows Vista and Windows Server 2008, a callout driver should be registered at one of the following run-time filtering layers:

TCP Packet Filtering
Stream Layers:

- FWPS_LAYER_STREAM_V4
- FWPS_LAYER_STREAM_V6

Non-TCP and Non-Error ICMP Packet Filtering
Datagram-Data Layers:

- FWPS_LAYER_DATAGRAM_DATA_V4
- FWPS_LAYER_DATAGRAM_DATA_V6
- FWPS_LAYER_DATAGRAM_DATA_V4_DISCARD
- FWPS_LAYER_DATAGRAM_DATA_V6_DISCARD

Except for the case when incoming packets must be rebuilt before they are receive-injected from a datagram-data layer, callout drivers that are registered at these data layers are compatible with IPsec.

Layers That Are Incompatible With IPsec

Network and forwarding layers are incompatible with IPsec because at these layers IPsec traffic has not yet been decrypted or verified. IPsec policies are enforced at the transport layer, which occurs after a network layer classify operation.

The following run-time filtering layers are incompatible with IPsec because IPsec processing in Windows occurs below the following layers:

FWPS_LAYER_INBOUND_IPPACKET_V4

FWPS_LAYER_INBOUND_IPPACKET_V6

FWPS_LAYER_INBOUND_IPPACKET_V4_DISCARD

FWPS_LAYER_INBOUND_IPPACKET_V6_DISCARD

FWPS_LAYER_OUTBOUND_IPPACKET_V4

FWPS_LAYER_OUTBOUND_IPPACKET_V6

FWPS_LAYER_OUTBOUND_IPPACKET_V4_DISCARD

FWPS_LAYER_OUTBOUND_IPPACKET_V6_DISCARD

Special Considerations for Transport Layers

To make a callout driver that is registered with a transport layer (FWPS_LAYER_*XXX*_TRANSPORT_V4 or _V6) compatible with IPsec, follow these guidelines:

1. Register the callout at ALE authorize receive/accept layers (**FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4** or **FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6**) in addition to transport layers (FWPS_LAYER_*XXX*_TRANSPORT_V4 or _V6).

2. To prevent interference with internal Windows IPsec processing, register the callout at a sublayer that has a lower weight than **FWPM_SUBLAYER_UNIVERSAL**. Use the **FwpmSubLayerEnum0** function to find the sublayer's weight. For information about this function, see the Windows Filtering Platform documentation in the Microsoft Windows SDK.

3. An incoming transport packet that requires ALE classification must be inspected at the ALE authorize receive/accept layers (**FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4** or **FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6**). Such a packet must be permitted from incoming transport layers. Beginning with Windows Vista with Service Pack 1 (SP1) and Windows Server 2008, use the **FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED** metadata flag to determine whether the incoming packet will be indicated to the **FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4** and **FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6** filtering layers. This metadata flag replaces the **FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY** condition flag that was used in Windows Vista.

4. To prevent interference with internal Windows IPsec processing, do not intercept IPsec tunnel-mode traffic at transport layers if the IPsec traffic is not yet detunneled. The following code example shows how to bypass such packets.

```C++
FWPS_PACKET_LIST_INFORMATION0 packetInfo = {0};
FwpsGetPacketListSecurityInformation0(
 layerData,
    FWPS_PACKET_LIST_INFORMATION_QUERY_IPSEC |
    FWPS_PACKET_LIST_INFORMATION_QUERY_INBOUND,
    &packetInfo
    );

if (packetInfo.ipsecInformation.inbound.isTunnelMode &&
    !packetInfo.ipsecInformation.inbound.isDeTunneled)
{
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
}
```

5. After an IPsec-protected packet is decrypted and verified at the transport layer, the AH/ESP header remains in the IP header. If such a packet has to be reinjected back into the TCP/IP stack, the IP header must be rebuilt to remove the AH/ESP header. Beginning with Windows Vista with SP1 and Windows Server 2008, you can do this by cloning the packet and calling the FwpsConstructIpHeaderForTransportPacket0 function that has the *headerIncludeHeaderSize* parameter set to the IP header size of the cloned packet.

6. At the ALE receive/accept layer, a callout can detect IPsec-protected traffic by checking whether the **FWP_CONDITION_FLAG_IS_IPSEC_SECURED** flag is set. At transport layers, a callout can detect IPsec-protected traffic by calling the FwpsGetPacketListSecurityInformation0 function and checking whether the **FWPS_PACKET_LIST_INFORMATION0** flag is set in the *queryFlags* parameter.

Working With IPsec ESP Packets

When the engine indicates decrypted encapsulating security payload (ESP) packets, it truncates them to exclude trailing ESP data. Because of the way the engine handles such packets, the MDL data in the NET_BUFFER structure does not reflect the correct packet length. The correct length can be obtained by using the NET_BUFFER_DATA_LENGTH macro to retrieve the data length of the **NET_BUFFER** structure.

Calling Other Windows Filtering Platform Functions

Article • 06/30/2022

Many of the other Windows Filtering Platform functions that are available to user-mode management applications are also available to callout drivers. This enables a callout driver to perform management tasks, such as adding filters to the filter engine. The only difference between the user-mode and kernel-mode versions of these functions is the data type that is returned. The user-mode functions return Win32 error codes, whereas the kernel-mode functions return the equivalent NTSTATUS codes.

Most of the Windows Filtering Platform management functions require a handle to an open session to the filter engine as a parameter. The following topics discuss how a callout driver can open and close a session to the filter engine.

Opening a Session to the Filter Engine

Closing a Session to the Filter Engine

For a list of the other Windows Filtering Platform functions that can be called from a callout driver, see Other Windows Filtering Platform Functions. For more information about how to use these functions, see the Windows Filtering Platform documentation in the Microsoft Windows SDK.

Opening a Session to the Filter Engine

Article • 12/15/2021

A callout driver must open a session to the filter engine to perform management tasks such as adding filters to the filter engine. A callout driver opens a session to the filter engine by calling the **FwpmEngineOpen0** function. For example:

C++

```
HANDLE engineHandle;
NTSTATUS status;

// Open a session to the filter engine
status =
 FwpmEngineOpen0(
    NULL,              // The filter engine on the local system
    RPC_C_AUTHN_WINNT, // Use the Windows authentication service
    NULL,              // Use the calling thread's credentials
    NULL,              // There are no session-specific parameters
    &engineHandle      // Pointer to a variable to receive the handle
    );
```

After a callout driver has successfully opened a session to the filter engine, it can use the returned handle to call the other Windows Filtering Platform management functions.

Closing a Session to the Filter Engine

Article • 12/15/2021

After a callout driver has performed the desired management tasks, it should close the session to the filter engine. A callout driver does this by calling the **FwpmEngineClose0** function. For example:

C++

```cpp
status =
 FwpmEngineClose0(
 engineHandle  // An handle to the open session
    );
```

Windows Filtering Platform constants

Article • 12/15/2021

This section describes constants used in Windows Filtering Platform callout drivers.

In this section

Built-in callout identifiers

Filtering layer identifiers

Filtering conditions

Metadata fields

Data field identifiers

Data offset positions

Discard reason identifiers

Related topics

Windows Filtering Platform Callout Drivers reference

Built-in callout identifiers

Article • 12/15/2021

The identifiers for the callouts that are built into the Windows Filtering Platform are each represented by a GUID. These identifiers are defined as follows.

> ⓘ **Note**
>
> The V4 and V6 suffixes at the end of the callout identifiers indicate whether the callout is for the IPv4 network stack or the IPv6 network stack.

Built-in callout identifier	Callout description
FWPM_CALLOUT_IPSEC_INBOUND_TRANSPORT_V4 FWPM_CALLOUT_IPSEC_INBOUND_TRANSPORT_V6	Verifies that each received packet that is supposed to arrive over a transport mode security association arrives securely. This callout is applicable at the transport layer.
FWPM_CALLOUT_IPSEC_OUTBOUND_TRANSPORT_V4 FWPM_CALLOUT_IPSEC_OUTBOUND_TRANSPORT_V6	Indicates to IPsec the outbound traffic that must be secured over transport mode security associations. This callout is applicable at the transport layer.
FWPM_CALLOUT_IPSEC_INBOUND_TUNNEL_V4 FWPM_CALLOUT_IPSEC_INBOUND_TUNNEL_V6	Verifies that each received packet that is supposed to arrive over a tunnel mode security association arrives securely. This callout is applicable at the transport layer.
FWPM_CALLOUT_IPSEC_OUTBOUND_TUNNEL_V4 FWPM_CALLOUT_IPSEC_OUTBOUND_TUNNEL_V6	Indicates to IPsec the outbound traffic that must be secured over tunnel mode security associations. This callout is applicable at the transport layer.
FWPM_CALLOUT_IPSEC_FORWARD_INBOUND_TUNNEL_V4 FWPM_CALLOUT_IPSEC_FORWARD_INBOUND_TUNNEL_V6	Verifies that each received packet that is supposed to arrive over a tunnel mode security association arrives securely. This callout is applicable at the forward layer.

Built-in callout identifier	Callout description
FWPM_CALLOUT_IPSEC_FORWARD_OUTBOUND_TUNNEL_V4 FWPM_CALLOUT_IPSEC_FORWARD_OUTBOUND_TUNNEL_V6	Indicates to IPsec the outbound traffic that must be secured over a tunnel mode security association. This callout is applicable at the forward layer.
FWPM_CALLOUT_IPSEC_INBOUND_INITIATE_SECURE_V4 FWPM_CALLOUT_IPSEC_INBOUND_INITIATE_SECURE_V6	Verifies that each incoming connection that is supposed to arrive securely does so. This callout is applicable at the ALE accept layer.
FWPM_CALLOUT_IPSEC_ALE_CONNECT_V4 FWPM_CALLOUT_IPSEC_ALE_CONNECT_V6	Applies IPsec policy modifiers to client applications.
FWPM_CALLOUT_WFP_TRANSPORT_LAYER_V4_SILENT_DROP FWPM_CALLOUT_WFP_TRANSPORT_LAYER_V6_SILENT_DROP	Silently drops all incoming packets for which TCP does not have a listening endpoint. This callout is applicable at the inbound transport layer.
FWPM_CALLOUT_TCP_CHIMNEY_CONNECT_LAYER_V4 FWPM_CALLOUT_TCP_CHIMNEY_CONNECT_LAYER_V6	Enables or disables TCP chimney offload for each outgoing connection.
FWPM_CALLOUT_TCP_CHIMNEY_ACCEPT_LAYER_V4 FWPM_CALLOUT_TCP_CHIMNEY_ACCEPT_LAYER_V6	Enables or disables TCP chimney offload for each incoming connection.

Management filtering layer identifiers

Article • 12/15/2021

The management filtering layer identifiers are generally used by user-mode applications and are each represented by a GUID, which is 128 bits in size. These identifiers are defined as follows.

> ⓘ Note
>
> The V4 and V6 suffixes at the end of the layer identifiers indicate whether the layer is located in the IPv4 network stack or in the IPv6 network stack.

Management filtering layer identifier	Filtering layer description
FWPM_LAYER_INBOUND_IPPACKET_V4 FWPM_LAYER_INBOUND_IPPACKET_V6	This filtering layer is located in the receive path just after the IP header of a received packet has been parsed but before any IP header processing takes place. No IPsec decryption or reassembly has occurred.
FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_INBOUND_IPPACKET_V6_DISCARD	This filtering layer is located in the receive path for processing any received packets that have been discarded at the network layer.
FWPM_LAYER_OUTBOUND_IPPACKET_V4 FWPM_LAYER_OUTBOUND_IPPACKET_V6	This filtering layer is located in the send path just before the sent packet is evaluated for fragmentation. All IP header processing is complete and all extension headers are in place. Any IPsec authentication and encryption has already occurred.
FWPM_LAYER_OUTBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_OUTBOUND_IPPACKET_V6_DISCARD	This filtering layer is located in the send path for processing any sent packets that have been discarded at the network layer.
FWPM_LAYER_IPFORWARD_V4 FWPM_LAYER_IPFORWARD_V6	This filtering layer is located in the forwarding path at the point where a received packet is forwarded.

Management filtering layer identifier	Filtering layer description
FWPM_LAYER_IPFORWARD_V4_DISCARD FWPM_LAYER_IPFORWARD_V6_DISCARD	This filtering layer is located in the forwarding path for processing any forwarded packets that have been discarded at the forward layer.
FWPM_LAYER_INBOUND_TRANSPORT_V4 FWPM_LAYER_INBOUND_TRANSPORT_V6	This filtering layer is located in the receive path just after a received packet's header has been parsed by the network stack at the transport layer, but before any transport layer processing takes place.
FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD	This filtering layer is located in the receive path for processing any received packets that have been discarded at the transport layer.
FWPM_LAYER_OUTBOUND_TRANSPORT_V4 FWPM_LAYER_OUTBOUND_TRANSPORT_V6	This filtering layer is located in the send path just after a sent packet has been passed to the network layer for processing but before any network layer processing takes place. This filtering layer is located at the top of the network layer instead of at the bottom of the transport layer so that any packets that are sent by third-party transports or as raw packets are filtered at this layer.
FWPM_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD	This filtering layer is located in the send path for processing any sent packets that have been discarded at the transport layer.
FWPM_LAYER_STREAM_V4 FWPM_LAYER_STREAM_V6	This filtering layer is located in the stream data path. This layer allows for processing network data on a per stream basis. At the stream layer, the network data is bidirectional.
FWPM_LAYER_STREAM_V4_DISCARD FWPM_LAYER_STREAM_V6_DISCARD	This filtering layer is reserved for future use.

Management filtering layer identifier	Filtering layer description
FWPM_LAYER_DATAGRAM_DATA_V4 FWPM_LAYER_DATAGRAM_DATA_V6	This filtering layer is located in the datagram data path. This layer allows for processing network data on a per datagram basis. At the datagram layer, the network data is bidirectional.
FWPM_LAYER_DATAGRAM_DATA_V4_DISCARD FWPM_LAYER_DATAGRAM_DATA_V6_DISCARD	This filtering layer is located in the datagram data path for processing any datagrams that have been discarded.
FWPM_LAYER_INBOUND_ICMP_ERROR_V4 FWPM_LAYER_INBOUND_ICMP_ERROR_V6	This filtering layer is located in the receive path for processing received ICMP messages for the transport protocol.
FWPM_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD FWPM_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD	This filtering layer is located in the receive path for processing received ICMP messages that have been discarded.
FWPM_LAYER_OUTBOUND_ICMP_ERROR_V4 FWPM_LAYER_OUTBOUND_ICMP_ERROR_V6	This filtering layer is located in the send path for processing sent ICMP messages for the transport protocol.
FWPM_LAYER_OUTBOUND_ICMP_ERROR_V4_DISCARD FWPM_LAYER_OUTBOUND_ICMP_ERROR_V6_DISCARD	This filtering layer is located in the send path for processing sent ICMP messages that have been discarded.
FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6	This filtering layer allows for authorizing connect requests for outgoing TCP connections, as well as authorizing outgoing non-TCP traffic based on the first packet sent.
FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD	This filtering layer allows for processing connect requests for outgoing TCP connections that have been discarded, as well as processing authorizations for outgoing non-TCP traffic that have been discarded.

Management filtering layer identifier	Filtering layer description
FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6	This filtering layer allows for notification of when a TCP connection has been established, or when non-TCP traffic has been authorized.
FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD	This filtering layer allows for processing when an established TCP connection has been discarded at the flow established layer, as well as when authorized non-TCP traffic has been discarded at the flow established layer.
FWPM_LAYER_ALE_AUTH_LISTEN_V4 FWPM_LAYER_ALE_AUTH_LISTEN_V6	This filtering layer allows for authorizing TCP listen requests.
FWPM_LAYER_ALE_AUTH_LISTEN_V4_DISCARD FWPM_LAYER_ALE_AUTH_LISTEN_V6_DISCARD	This filtering layer allows for processing TCP listen requests that have been discarded.
FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6	This filtering layer allows for authorizing accept requests for incoming TCP connections, as well as authorizing incoming non-TCP traffic based on the first packet received.
FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD	This filtering layer allows for processing accept requests for incoming TCP connections that have been discarded, as well as processing authorizations for incoming non-TCP traffic that have been discarded.
FWPM_LAYER_ALE_AUTH_ROUTE_V4 FWPM_LAYER_ALE_AUTH_ROUTE_V6	This filtering layer allows for inspecting and filtering the route and path parameters of bind and connect requests.
FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V4 FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V6	This filtering layer is used as an opportunity to reclaim resources allocated by the callout driver in any of the ALE_AUTH_CONNECT or ALE_AUTH_RECV_ACCEPT layers.

Management filtering layer identifier	Filtering layer description
FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6	This filtering layer allows for authorizing transport port assignments, bind requests, promiscuous mode requests, and raw mode requests.
FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD	This filtering layer allows for processing the following discarded items: transport port assignments, bind requests, promiscuous mode requests, and raw mode requests.
FWPM_LAYER_ALE_RESOURCE_RELEASE_V4 FWPM_LAYER_ALE_RESOURCE_RELEASE_V6	This filtering layer is used as an opportunity to reclaim resources allocated by the callout driver in any of the ALE_RESOURCE_ASSIGNMENT layers.
FWPM_LAYER_IPSEC_KM_DEMUX_V4 FWPM_LAYER_IPSEC_KM_DEMUX_V6	This filtering layer is used to determine which keying modules are invoked when the local system is the initiator. This is a user-mode filtering layer.
FWPM_LAYER_IPSEC_V4 FWPM_LAYER_IPSEC_V6	This filtering layer allows the keying module to look up quick-mode policy information when negotiating quick-mode security associations. This is a user-mode filtering layer.
FWPM_LAYER_IKEEXT_V4 FWPM_LAYER_IKEEXT_V6	This filtering layer allows the IKE and authenticated IP modules to look up main-mode policy information when negotiating main-mode security associations. This is a user-mode filtering layer.
FWPM_LAYER_RPC_UM	This filtering layer allows for inspecting the RPC data fields that are available in user mode. This is a user-mode filtering layer.
FWPM_LAYER_RPC_EPMAP	This filtering layer allows for inspecting the RPC data fields that are available in user mode during endpoint resolution. This is a user-mode filtering layer.

Management filtering layer identifier	Filtering layer description
FWPM_LAYER_RPC_EP_ADD	This filtering layer allows for inspecting the RPC data fields that are available in user mode when a new endpoint is added. This is a user-mode filtering layer.
FWPM_LAYER_RPC_PROXY_CONN	This filtering layer allows for inspecting RpcProxy connection requests. This is a user-mode filtering layer.
FWPM_LAYER_RPC_PROXY_IF	This filtering layer allows for inspecting the interface used for RpcProxy connections. This is a user-mode filtering layer.
FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET	This filtering layer allows for inspecting the MAC frame data at the inbound lower (to the NDIS protocol driver) layer. **Note**: Available only on Windows 8 and later.
FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET	This filtering layer allows for inspecting the MAC frame data at the outbound upper (to the NDIS protocol driver) layer. **Note**: Available only on Windows 8 and later.
FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE	This filtering layer allows for inspecting the MAC frame data at the inbound lower (to the NDIS miniport driver) layer. **Note**: Available only on Windows 8 and later.
FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE	This filtering layer allows for inspecting the MAC frame data at the outbound lower (to the NDIS miniport driver) layer. **Note**: Available only on Windows 8 and later.

Management filtering layer identifier	Filtering layer description
FWPM_LAYER_INGRESS_VSWITCH_ETHERNET	This filtering layer allows for inspecting the ingress 802.3 data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.
FWPM_LAYER_EGRESS_VSWITCH_ETHERNET	This filtering layer allows for inspecting the egress 802.3 data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.
FWPM_LAYER_INGRESS_VSWITCH_TRANSPORT_V4 FWPM_LAYER_INGRESS_VSWITCH_TRANSPORT_V6	This filtering layer allows for inspecting the ingress transport data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.
FWPM_LAYER_EGRESS_VSWITCH_TRANSPORT_V4 FWPM_LAYER_EGRESS_VSWITCH_TRANSPORT_V6	This filtering layer allows for inspecting the egress transport data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.

Run-time filtering layer identifiers

Article • 12/15/2021

The run-time filtering layer identifiers are used by kernel-mode callout drivers and are each represented by a locally unique identifier (LUID), which is 64 bits in size. These identifiers are constant values in the FWPS_BUILTIN_LAYERS enumeration that is defined in Fwpsk.h. These identifiers are defined as follows:

> ⓘ Note
>
> The V4 and V6 suffixes at the end of the run-time layer identifiers indicate whether the layer is located in the IPv4 network stack or in the IPv6 network stack.

Run-time filtering layer identifier	Filtering layer description
FWPS_LAYER_INBOUND_IPPACKET_V4 FWPS_LAYER_INBOUND_IPPACKET_V6	This filtering layer is located in the receive path just after the IP header of a received packet has been parsed but before any IP header processing takes place. No IPsec decryption or reassembly has occurred.
FWPS_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPS_LAYER_INBOUND_IPPACKET_V6_DISCARD	This filtering layer is located in the receive path for processing any received packets that have been discarded at the network layer.
FWPS_LAYER_OUTBOUND_IPPACKET_V4 FWPS_LAYER_OUTBOUND_IPPACKET_V6	This filtering layer is located in the send path just before the sent packet is evaluated for fragmentation. All IP header processing is complete and all extension headers are in place. Any IPsec authentication and encryption has already occurred.
FWPS_LAYER_OUTBOUND_IPPACKET_V4_DISCARD FWPS_LAYER_OUTBOUND_IPPACKET_V6_DISCARD	This filtering layer is located in the send path for processing any sent packets that have been discarded at the network layer.
FWPS_LAYER_IPFORWARD_V4 FWPS_LAYER_IPFORWARD_V6	This filtering layer is located in the forwarding path at the point where a received packet is forwarded.

Run-time filtering layer identifier	Filtering layer description
FWPS_LAYER_IPFORWARD_V4_DISCARD FWPS_LAYER_IPFORWARD_V6_DISCARD	This filtering layer is located in the forwarding path for processing any forwarded packets that have been discarded at the forward layer.
FWPS_LAYER_INBOUND_TRANSPORT_V4 FWPS_LAYER_INBOUND_TRANSPORT_V6	This filtering layer is located in the receive path just after a received packet's header has been parsed by the network stack at the transport layer, but before any transport layer processing takes place.
FWPS_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPS_LAYER_INBOUND_TRANSPORT_V6_DISCARD	This filtering layer is located in the receive path for processing any received packets that have been discarded at the transport layer.
FWPS_LAYER_OUTBOUND_TRANSPORT_V4 FWPS_LAYER_OUTBOUND_TRANSPORT_V6	This filtering layer is located in the send path just after a sent packet has been passed to the network layer for processing but before any network layer processing takes place. This filtering layer is located at the top of the network layer instead of at the bottom of the transport layer so that any packets that are sent by third-party transports or as raw packets are filtered at this layer.
FWPS_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPS_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD	This filtering layer is located in the send path for processing any sent packets that have been discarded at the transport layer.
FWPS_LAYER_STREAM_V4 FWPS_LAYER_STREAM_V6	This filtering layer is located in the stream data path. This layer allows for processing network data on a per stream basis. At the stream layer, the network data is bidirectional.
FWPS_LAYER_STREAM_V4_DISCARD FWPS_LAYER_STREAM_V6_DISCARD	This filtering layer is reserved for future use.

Run-time filtering layer identifier	Filtering layer description
FWPS_LAYER_DATAGRAM_DATA_V4 FWPS_LAYER_DATAGRAM_DATA_V6	This filtering layer is located in the datagram data path. This layer allows for processing network data on a per datagram basis. At the datagram layer, the network data is bidirectional.
FWPS_LAYER_DATAGRAM_DATA_V4_DISCARD FWPS_LAYER_DATAGRAM_DATA_V6_DISCARD	This filtering layer is located in the datagram data path for processing any datagrams that have been discarded.
FWPS_LAYER_INBOUND_ICMP_ERROR_V4 FWPS_LAYER_INBOUND_ICMP_ERROR_V6	This filtering layer is located in the receive path for processing received ICMP messages for the transport protocol.
FWPS_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD FWPS_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD	This filtering layer is located in the receive path for processing received ICMP messages that have been discarded.
FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4 FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6	This filtering layer is located in the send path for processing sent ICMP messages for the transport protocol.
FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4_DISCARD FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6_DISCARD	This filtering layer is located in the send path for processing sent ICMP messages that have been discarded.
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6	This filtering layer allows for authorizing transport port assignments, bind requests, promiscuous mode requests, and raw mode requests.
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD	This filtering layer allows for processing the following discarded items: transport port assignments, bind requests, promiscuous mode requests, and raw mode requests.
FWPS_LAYER_ALE_AUTH_LISTEN_V4 FWPS_LAYER_ALE_AUTH_LISTEN_V6	This filtering layer allows for authorizing TCP listen requests.
FWPS_LAYER_ALE_AUTH_LISTEN_V4_DISCARD FWPS_LAYER_ALE_AUTH_LISTEN_V6_DISCARD	This filtering layer allows for processing TCP listen requests that have been discarded.

Run-time filtering layer identifier	Filtering layer description
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6	This filtering layer allows for authorizing accept requests for incoming TCP connections, as well as authorizing incoming non-TCP traffic based on the first packet received.
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD	This filtering layer allows for processing accept requests for incoming TCP connections that have been discarded, as well as processing authorizations for incoming non-TCP traffic that have been discarded.
FWPS_LAYER_ALE_AUTH_CONNECT_V4 FWPS_LAYER_ALE_AUTH_CONNECT_V6	This filtering layer allows for authorizing connect requests for outgoing TCP connections, as well as authorizing outgoing non-TCP traffic based on the first packet sent.
FWPS_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPS_LAYER_ALE_AUTH_CONNECT_V6_DISCARD	This filtering layer allows for processing connect requests for outgoing TCP connections that have been discarded, as well as processing authorizations for outgoing non-TCP traffic that have been discarded.
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6	This filtering layer allows for notification of when a TCP connection has been established, or when non-TCP traffic has been authorized.
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD	This filtering layer allows for processing when an established TCP connection has been discarded at the flow established layer, as well as when authorized non-TCP traffic has been discarded at the flow established layer.
FWPS_LAYER_RESERVED1_V4 FWPS_LAYER_RESERVED1_V6	This filtering layer is not supported.
FWPS_LAYER_NAME_RESOLUTION_CACHE_V4 FWPS_LAYER_NAME_RESOLUTION_CACHE_V6	This filtering layer allows for querying the names recently resolved by the system.

Run-time filtering layer identifier	Filtering layer description
FWPS_LAYER_ALE_RESOURCE_RELEASE_V4 FWPS_LAYER_ALE_RESOURCE_RELEASE_V6	This filtering layer is used as an opportunity to reclaim resources allocated by the callout driver in any of the ALE_RESOURCE_ASSIGNMENT layers.
FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4 FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6	This filtering layer is used as an opportunity to reclaim resources allocated by the callout driver in any of the ALE_AUTH_CONNECT or ALE_AUTH_RECV_ACCEPT layers.
FWPS_LAYER_ALE_CONNECT_REDIRECT_V4 FWPS_LAYER_ALE_CONNECT_REDIRECT_V6	This filtering layer allows for the redirecting of connect requests to a different IPV4/ IPV6 address and TCP/UDP port.
FWPS_LAYER_ALE_BIND_REDIRECT_V4 FWPS_LAYER_ALE_BIND_REDIRECT_V6	This filtering layer allows for the redirecting of bind requests to a different local IPV4/ IPV6 address and/or local TCP/UDP port.
FWPS_LAYER_INBOUND_MAC_FRAME_ETHERNET	This filtering layer allows for inspecting the MAC frame data at the inbound lower (to the NDIS protocol driver) layer. **Note**: Available only on Windows 8 and later.
FWPS_LAYER_OUTBOUND_MAC_FRAME_ETHERNET	This filtering layer allows for inspecting the MAC frame data at the outbound upper (to the NDIS protocol driver) layer. **Note**: Available only on Windows 8 and later.
FWPS_LAYER_INBOUND_MAC_FRAME_NATIVE	This filtering layer allows for inspecting the MAC frame data at the inbound lower (to the NDIS miniport driver) layer. **Note**: Available only on Windows 8 and later.
FWPS_LAYER_OUTBOUND_MAC_FRAME_NATIVE	This filtering layer allows for inspecting the MAC frame data at the outbound lower (to the NDIS miniport driver) layer. **Note**: Available only on Windows 8 and later.

Run-time filtering layer identifier	Filtering layer description
FWPS_LAYER_INGRESS_VSWITCH_ETHERNET	This filtering layer allows for inspecting the ingress 802.3 data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.
FWPS_LAYER_EGRESS_VSWITCH_ETHERNET	This filtering layer allows for inspecting the egress 802.3 data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.
FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V4 FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V6	This filtering layer allows for inspecting the ingress transport data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.
FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V4 FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V6	This filtering layer allows for inspecting the egress transport data of the Hyper-V extensible switch. **Note**: Available only on Windows 8 and later.
FWPS_LAYER_STREAM_PACKET_V4 FWPS_LAYER_STREAM_PACKET_V6	This filtering layer allows for inspection of network data on a per-TCP packet basis, including handshake and flow control exchanges. At the stream packet layer, the network data is bidirectional.
FWPS_LAYER_IPSEC_KM_DEMUX_V4 FWPS_LAYER_IPSEC_KM_DEMUX_V6	This filtering layer is used to determine which keying modules are invoked when the local system is the initiator. This is a user-mode filtering layer.
FWPS_LAYER_IPSEC_V4 FWPS_LAYER_IPSEC_V6	This filtering layer allows the keying module to look up quick-mode policy information when negotiating quick-mode security associations. This is a user-mode filtering layer.

Run-time filtering layer identifier	Filtering layer description
FWPS_LAYER_IKEEXT_V4 FWPS_LAYER_IKEEXT_V6	This filtering layer allows the IKE and authenticated IP modules to look up main-mode policy information when negotiating main-mode security associations. This is a user-mode filtering layer.
FWPS_LAYER_RPC_UM	This filtering layer allows for inspecting the RPC data fields that are available in user mode. This is a user-mode filtering layer.
FWPS_LAYER_RPC_EPMAP	This filtering layer allows for inspecting the RPC data fields that are available in user mode during endpoint resolution. This is a user-mode filtering layer.
FWPS_LAYER_RPC_EP_ADD	This filtering layer allows for inspecting the RPC data fields that are available in user mode when a new endpoint is added. This is a user-mode filtering layer.
FWPS_LAYER_RPC_PROXY_CONN	This filtering layer allows for inspecting RpcProxy connection requests. This is a user-mode filtering layer.
FWPS_LAYER_RPC_PROXY_IF	This filtering layer allows for inspecting the interface used for RpcProxy connections. This is a user-mode filtering layer.
FWPS_LAYER_KM_AUTHORIZATION	This filtering layer allows for authorizing security association establishment.

Each run-time layer identifier has an associated run-time data field identifier that represents a set of constant values. These data field identifiers are declared as FWPS_FIELDS_XXX enumerations in Fwpsk.h. For more information, see Data Field Identifiers.

Filtering condition identifiers

Article • 12/15/2021

The filtering condition identifiers are each represented by a GUID. These identifiers are described in the following table.

Filtering condition identifier	Description
FWPM_CONDITION_ARRIVAL_INTERFACE_INDEX	The index of the arrival network interface, as enumerated by the network stack. WFP uses the Arrival interface to match this condition. The Arrival Interface is the first interface the packet sees before entering the IP stack inbound from the network, before weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an inbound condition. This means that WFP will use an empty value on this condition when reauthorizing an inbound connection on a response outbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of arrival interface conditions, the next hop class of interface conditions will have a valid interface on outbound packets. Note that this is available only in Windows Server 2008 R2, Windows 7, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_ARRIVAL_INTERFACE_TYPE	The type of the arrival network interface, as defined by the Internet Assigned Numbers Authority (IANA). For more information, see IANAifType-MIB Definitions ↗ . WFP uses the Arrival interface to match this condition. The Arrival Interface is the first interface the packet sees before entering the IP stack inbound from the network, before weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an inbound condition. This means that WFP will use an empty value on this condition when reauthorizing an inbound connection on a response outbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of arrival interface conditions, the next hop class of interface conditions will have a valid interface on outbound packets. Note that this is vailable only in Windows Server 2008 R2, Windows 7, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_ARRIVAL_TUNNEL_TYPE	The encapsulation method used by a tunnel if the IfType member of the IP_ADAPTER_ADDRESSES structure is IF_TYPE_TUNNEL. The tunnel type is defined by the IANA. For more information, see IANAifType-MIB Definitions and the Windows SDK IP Helper documentation. WFP uses the Arrival interface to match this condition. The Arrival Interface is the first interface the packet sees before entering the IP stack inbound from the network, before weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an inbound condition. This means that WFP will use an empty value on this condition when reauthorizing an inbound connection on a response outbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of arrival interface conditions, the next hop class of interface conditions will have a valid interface on outbound packets. Note that this is available only in Windows Server 2008 R2, Windows 7, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_IP_ARRIVAL_INTERFACE	The LUID for the network interface that is associated with the arrival IP address. WFP uses the Arrival interface to match this condition. The Arrival Interface is the first interface the packet sees before entering the IP stack inbound from the network, before weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an inbound condition. This means that WFP will use an empty value on this condition when reauthorizing an inbound connection on a response outbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of arrival interface conditions, the next hop class of interface conditions will have a valid interface on outbound packets. Note that this is available only in Windows Server 2008 R2, Windows 7, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_NEXTHOP_INTERFACE_INDEX	The index of the arrival network interface, as enumerated by the network stack. WFP uses the Next Hop interface to match this condition. The Next Hop Interface is the last interface the packet sees before leaving the IP stack outbound towards the network, after weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an outbound condition. This means that WFP will use an empty value on this condition when reauthorizing an outbound connection on a response inbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of next hop interface conditions, the arrival class of interface conditions will have a valid interface on inbound packets. Note that this is available only in Windows Server 2008 R2, Windows 7, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_NEXTHOP_INTERFACE_TYPE	The type of the arrival network interface, as defined by the Internet Assigned Numbers Authority (IANA). For more information, see IANAifType-MIB Definitions . WFP uses the Next Hop interface to match this condition. The Next Hop Interface is the last interface the packet sees before leaving the IP stack outbound towards the network, after weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an outbound condition. This means that WFP will use an empty value on this condition when reauthorizing an outbound connection on a response inbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of next hop interface conditions, the arrival class of interface conditions will have a valid interface on inbound packets. Note that this is available only in Windows Server 2008 R2, Windows 7, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_NEXTHOP_TUNNEL_TYPE	The encapsulation method used by a tunnel if the **IfType** member of the **IP_ADAPTER_ADDRESSES** structure is IF_TYPE_TUNNEL. The tunnel type is defined by the IANA. For more information, see IANAifType-MIB Definitions and the Windows SDK IP Helper documentation. WFP uses the Next Hop interface to match this condition. The Next Hop Interface is the last interface the packet sees before leaving the IP stack outbound towards the network, after weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an outbound condition. This means that WFP will use an empty value on this condition when reauthorizing an outbound connection on a response inbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of next hop interface conditions, the arrival class of interface conditions will have a valid interface on inbound packets. Note that this is available only in Windows Server 2008 R2, Windows 7, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_IP_NEXTHOP_INTERFACE	The LUID for the network interface that is associated with the arrival IP addres WFP uses the Next Hop interface to match this condition. The Next Hop Interface is the last interface the packet sees before leaving the IP stack outbound towards the network, after weak-host or forwarding are performed. This condition is asymmetric for reauthorization purposes, as it is intrinsically an outbound condition. This means that WFP will use an empty value on this condition when reauthorizing an outbound connection on a response inbound packet. To handle reauthorization a second filter must be used. This second filter can either permit or block the empty values, or use a different condition that will have a valid value for such circumstance. In the case of next hop interface conditions, the arrival class of interface conditions will have a valid interface on inbound packets. Note that this is available only in Windows Server 2008 R2, Windows 7, and later versions of Windows.
FWPM_CONDITION_IP_LOCAL_ADDRESS	The local IP address.
FWPM_CONDITION_IP_REMOTE_ADDRESS	The remote IP address.
FWPM_CONDITION_IP_SOURCE_ADDRESS	The source IP address for forwarded packets.
FWPM_CONDITION_IP_DESTINATION_ADDRESS	The destination IP address for forwarded packets.

Filtering condition identifier	Description
FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE	The local IP address type. The possible condition values are: - NlatUnspecified - NlatUnicast - NlatAnycast - NlatMulticast - NlatBroadcast
FWPM_CONDITION_IP_DESTINATION_ADDRESS_TYPE	The destination IP address type. The possible condition values are: - NlatUnspecified - NlatUnicast - NlatAnycast - NlatMulticast - NlatBroadcast
FWPM_CONDITION_IP_LOCAL_INTERFACE	The LUID for the network interface associated with the local IP address.
FWPM_CONDITION_IP_FORWARD_INTERFACE	The LUID for the network interface on which the packet being forwarded is to be sent out.
FWPM_CONDITION_IP_PROTOCOL	The IP protocol number, as specified in RFC 1700 .
FWPM_CONDITION_IP_LOCAL_PORT	The local transport protocol port number.
FWPM_CONDITION_IP_REMOTE_PORT	The remote transport protocol port number.
FWPM_CONDITION_ICMP_TYPE	The ICMP type field, as specified in RFC 792 .
FWPM_CONDITION_ICMP_CODE	The ICMP code field, as specified in RFC 792 .

Filtering condition identifier	Description
FWPM_CONDITION_EMBEDDED_LOCAL_ADDRESS_TYPE	The local IP address type that is embedded in the ICMP packet. The possible condition values are: - NlatUnspecified - NlatUnicast - NlatAnycast - NlatMulticast - NlatBroadcast
FWPM_CONDITION_EMBEDDED_REMOTE_ADDRESS	The remote IP address that is embedded in the ICMP packet.
FWPM_CONDITION_EMBEDDED_PROTOCOL	The IP protocol number that is embedded in the ICMP packet, as specified in RFC 1700 .
FWPM_CONDITION_EMBEDDED_LOCAL_PORT	The local transport protocol port number that is embedded in the ICMP packet.
FWPM_CONDITION_EMBEDDED_REMOTE_PORT	The remote transport protocol port number that is embedded in the ICMP packet.
FWPM_CONDITION_FLAGS	A bitwise OR of a combination of filtering condition flags. For information about the possible flags, see Filtering Condition Flags.

Filtering condition identifier	Description
FWPM_CONDITION_DIRECTION	The direction of the datagram traffic or data flow. The possible condition values are: - FWP_DIRECTION_INBOUND - FWP_DIRECTION_OUTBOUND In datagram data layers and stream packet layers, this condition specifies the direction of the packet. In stream layers and ALE flow established layers, this condition specifies the direction of the connection (for example, when a local application initiates the connection, an inbound packet has FWPM_CONDITION_DIRECTION set to FWP_DIRECTION_OUTBOUND).
FWPM_CONDITION_INTERFACE_INDEX	The index of the network interface, as enumerated by the network stack.
FWPM_CONDITION_INTERFACE_TYPE	The bus type of the network interface.
FWPM_CONDITION_SUB_INTERFACE_INDEX	The index of the logical network interface, as enumerated by the network stack.
FWPM_CONDITION_SOURCE_INTERFACE_INDEX	The index of the source network interface for forwarded packets, as enumerated by the network stack.
FWPM_CONDITION_SOURCE_SUB_INTERFACE_INDEX	The index of the source logical network interface for forwarded packets, as enumerated by the network stack.
FWPM_CONDITION_DESTINATION_INTERFACE_INDEX	The index of the destination network interface for forwarded packets, as enumerated by the network stack.

Filtering condition identifier	Description
FWPM_CONDITION_DESTINATION_SUB_INTERFACE_INDEX	The index of the destination logical network interface for forwarded packets, as enumerated by the network stack.
FWPM_CONDITION_ALE_APP_ID	The full path of the application.
FWPM_CONDITION_ALE_USER_ID	The identification of the local user.
FWPM_CONDITION_ALE_REMOTE_USER_ID	The identification of the remote user.
FWPM_CONDITION_ALE_REMOTE_MACHINE_ID	The identification of the remote machine.
FWPM_CONDITION_ALE_PROMISCUOUS_MODE	The raw socket mode that is allowed or denied. The possible condition values are: - SIO_RCVALL - SIO_RCVALL_IGMPMCAST - SIO_RCVALL_MCAST For a description of these raw socket modes, see WSAIoctl in the Microsoft Windows SDK documentation.
FWPM_CONDITION_ALE_SIO_FIREWALL_SYSTEM_PORT	Reserved for internal use.
FWPM_CONDITION_ALE_NAP_CONTEXT	Reserved for internal use.
FWPM_CONDITION_REMOTE_USER_TOKEN	The identification of the remote user.
FWPM_CONDITION_RPC_IF_UUID	The UUID of the RPC interface.
FWPM_CONDITION_RPC_IF_VERSION	The version of the RPC interface.
FWPM_CONDITION_RCP_IF_FLAG	Reserved for internal use.
FWPM_CONDITION_DCOM_APP_ID	The identification of the COM application.
FWPM_CONDITION_IMAGE_NAME	The name of the application.

Filtering condition identifier	Description
FWPM_CONDITION_RPC_PROTOCOL	The RPC protocol. The possible condition values are: - RPC_PROTSEQ_TCP - RPC_PROTSEQ_HTTP - RPC_PROTSEQ_NMP
FWPM_CONDITION_RPC_AUTH_TYPE	The authentication service type. For more information about authentication service types, see Authentication-Service Constants in the RPC section of the Windows SDK documentation.
FWPM_CONDITION_RPC_AUTH_LEVEL	The authentication service level. For more information about authentication service levels, see Authentication-Level Constants in the RPC section of the Windows SDK documentation.
FWPM_CONDITION_SEC_ENCRYPT_ALGORITHM	The certificate based security service provider interface (SSPI) encryption algorithm.
FWPM_CONDITION_SEC_KEY_SIZE	The certificate based security service provider interface (SSPI) encryption key size.
FWPM_CONDITION_IP_LOCAL_ADDRESS_V4	The local IPv4 address.
FWPM_CONDITION_IP_LOCAL_ADDRESS_V6	The local IPv6 address.
FWPM_CONDITION_PIPE	The name of the remote named pipe.
FWPM_CONDITION_IP_REMOTE_ADDRESS_V4	The remote IPv4 address.
FWPM_CONDITION_IP_REMOTE_ADDRESS_V6	The remote IPv6 address.
FWPM_CONDITION_PROCESS_WITH_RPC_IF_UUID	The UUID of the process with the RPC interface.
FWPM_CONDITION_RPC_EP_VALUE	Reserved for internal use.
FWPM_CONDITION_RPC_EP_FLAGS	Reserved for internal use.
FWPM_CONDITION_CLIENT_TOKEN	The identification of the client when using RpcProxy.

Filtering condition identifier	Description
FWPM_CONDITION_RPC_SERVER_NAME	The name of the RPC server when using RpcProxy.
FWPM_CONDITION_RPC_SERVER_PORT	The port on the RPC server when using RpcProxy.
FWPM_CONDITION_RPC_PROXY_AUTH_TYPE	The RPC proxy authentication service type. For more information about authentication service types, see Authentication-Service Constants in the RPC section of the Windows SDK documentation.
FWPM_CONDITION_TUNNEL_TYPE	The encapsulation method used by a tunnel.
FWPM_CONDITION_CLIENT_CERT_KEY_LENGTH	The secure socket layer (SSL) key length in the client certificate.
FWPM_CONDITION_CLIENT_CERT_OID	The object identifier (OID) in the client certificate.
FWPM_CONDITION_INTERFACE_MAC_ADDRESS	The physical address of the sending or receiving network interface. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_MAC_LOCAL_ADDRESS	The physical address of the local network interface. For inbound traffic this is the destination MAC address in the frame. For outbound traffic this is the source MAC address of the frame. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_MAC_REMOTE_ADDRESS	The physical address of the remote network interface. For inbound traffic this is the source MAC address in the frame. For outbound traffic this is the destination MAC address of the frame. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_ETHER_TYPE	The type indicated in the MAC frame. This value is 0x800 for IPv4 traffic, 0x86DD for IPv6 traffic or, 0x806 for ARP traffic. All of the possible values are defined as NDIS_ETH_TYPE_Xxx in ntddndis.h.
FWPM_CONDITION_VLAN_ID	The identifier of the VLAN in the ETHERNET SNAP header. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_NDIS_PORT	The port number identifying a miniport adapter port. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_NDIS_MEDIA_TYPE	The type of the NDIS medium specified as one of the NDIS_MEDIUM enumeration values. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_NDIS_PHYSICAL_MEDIA_TYPE	The type of the physical medium for the communicating interface specified as one of the NDIS_PHYSICAL_MEDIUM enumeration values. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_L2_FLAGS	A bitwise OR of a combination of filtering condition flags for the MAC layers. For information about the possible flags, see Filtering Condition L2 Flags. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_MAC_LOCAL_ADDRESS_TYPE	The Datalink type of the local MAC address. This is one of the values that are defined in the DL_ADDRESS_TYPE enumeration in FwpmTypes.h. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_MAC_REMOTE_ADDRESS_TYPE	The Datalink type of the remote MAC address. This is one of the values that are defined in the DL_ADDRESS_TYPE enumeration in FwpmTypes.h. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_INTERFACE	The LUID for the network interface that is associated with the local MAC address. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_ALE_PACKAGE_ID	The security identifier (SID) of the AppContainer restricted package. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_MAC_SOURCE_ADDRESS	The physical address of the network interface that created the MAC frame. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_MAC_DESTINATION_ADDRESS	The physical address of the network interface to which the frame is destined. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_MAC_SOURCE_ADDRESS_TYPE	The Datalink type of the MAC Address for the interface that created the frame. This is one of the values that are defined in the DL_ADDRESS_TYPE enumeration in FwpmTypes.h. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_MAC_DESTINATION_ADDRESS_TYPE	The Datalink type of the MAC Address for the interface to which the frame is destined. This is one of the values that are defined in the DL_ADDRESS_TYPE enumeration in FwpmTypes.h. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_IP_SOURCE_PORT	The transport protocol source port number. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_IP_DESTINATION_PORT	The transport protocol destination port number. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_VSWITCH_ID	The GUID of the virtual switch. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_VSWITCH_NETWORK_TYPE	The type of network that is associated with the virtual switch. This is one of the values that are defined in the FWP_VSWITCH_NETWORK_TYPE enumeration in FwpTypes.h. **Note** Supported in Windows 8 and later versions of Windows.
FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_ID	The GUID of the interface of the virtual switch that created the frame. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_ID	The GUID of the interface of the virtual switch to which the frame is destined. **Note** Supported in Windows 8 and later versions of Windows.
FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_TYPE	The type of the virtual switch interface that created the frame. This is one of the values that are defined in the NDIS_NIC_SWITCH_TYPE enumeration in Ntddndis.h. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_TYPE	The type of the virtual switch interface to which the frame is destined. This is one of the values that are defined in the NDIS_NIC_SWITCH_TYPE enumeration in Ntddndis.h. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_VSWITCH_SOURCE_VM_ID	Unique identifier of the vSwitch source virtual machine. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.

Filtering condition identifier	Description
FWPM_CONDITION_VSWITCH_DESTINATION_VM_ID	Unique identifier of the vSwitch destination virtual machine. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_VSWITCH_TENANT_NETWORK_ID	Unique identifier for the vSwitch network. Cannot be used in conjunction with VLAN_IDs. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_ALE_PACKAGE_ID	The security identifier (SID) of an app container. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_ALE_ORIGINAL_APP_ID	The original full path of the application before alteration from proxying. Note that if proxying is not involved, then this will be the same as the FWPM_CONDITION_ALE_APP_ID. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.
FWPM_CONDITION_QM_MODE	The quick mode (QM) mode. **Note** Supported in Windows 8, Windows Server 2012, and later versions of Windows.

Filtering condition flags

Article • 12/06/2022

The filtering condition flags are each represented by a bit field. These flags are defined as follows:

> ⓘ **Note**
>
> This topic contains filtering condition flags for kernel mode WFP callout drivers. For information about filtering condition flags that are shared between user mode and kernel mode, or if you are looking for information about a flag that isn't listed here, see **Filtering Condition Flags**.

Filtering condition flag	Description
FWP_CONDITION_FLAG_IS_LOOPBACK 0x00000001	Indicates that the network traffic is loopback traffic. This flag is applicable at the following filtering layers: FWPM_LAYER_INBOUND_IPPACKET_V4 FWPM_LAYER_INBOUND_IPPACKET_V6 FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_INBOUND_IPPACKET_V6_DISCARD FWPM_LAYER_OUTBOUND_IPPACKET_V4 FWPM_LAYER_OUTBOUND_IPPACKET_V6 FWPM_LAYER_OUTBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_OUTBOUND_IPPACKET_V6_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V4 FWPM_LAYER_INBOUND_TRANSPORT_V6 FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V4 FWPM_LAYER_OUTBOUND_TRANSPORT_V6 FWPM_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD FWPM_LAYER_DATAGRAM_DATA_V4 FWPM_LAYER_DATAGRAM_DATA_V6 FWPM_LAYER_DATAGRAM_DATA_V4_DISCARD FWPM_LAYER_DATAGRAM_DATA_V6_DISCARD FWPM_LAYER_INBOUND_ICMP_ERROR_V4 FWPM_LAYER_INBOUND_ICMP_ERROR_V6 FWPM_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD FWPM_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD FWPM_LAYER_OUTBOUND_ICMP_ERROR_V4 FWPM_LAYER_OUTBOUND_ICMP_ERROR_V6 FWPM_LAYER_OUTBOUND_ICMP_ERROR_V4_DISCARD FWPM_LAYER_OUTBOUND_ICMP_ERROR_V6_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD
FWP_CONDITION_FLAG_IS_IPSEC_SECURED	Indicates that the network traffic is protected by IPsec.

0x00000002	This flag is applicable at the following filtering layers: FWPM_LAYER_INBOUND_TRANSPORT_V4 FWPM_LAYER_INBOUND_TRANSPORT_V6 FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD
FWP_CONDITION_FLAG_IS_REAUTHORIZE 0x00000004	Indicates a policy change (as opposed to a new connection). This flag is applicable at the following filtering layers: FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD This flag is also applicable at the following filtering layers in Windows Server 2008 R2, Windows 7, and later versions of Windows: FWPM_LAYER_ALE_BIND_REDIRECT_V4 FWPM_LAYER_ALE_BIND_REDIRECT_V6
FWP_CONDITION_FLAG_IS_WILDCARD_BIND 0x00000008	Indicates that the application specified a wildcard address when binding to a local network address. This flag is applicable at the following filtering layers: FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD
FWP_CONDITION_FLAG_IS_RAW_ENDPOINT 0x00000010	Indicates that the local endpoint that is sending and receiving traffic is a raw endpoint. This flag is applicable at the following filtering layers: FWPM_LAYER_INBOUND_TRANSPORT_V4 FWPM_LAYER_INBOUND_TRANSPORT_V6 FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V4 FWPM_LAYER_OUTBOUND_TRANSPORT_V6 FWPM_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD FWPM_LAYER_DATAGRAM_DATA_V4 FWPM_LAYER_DATAGRAM_DATA_V6 FWPM_LAYER_DATAGRAM_DATA_V4_DISCARD FWPM_LAYER_DATAGRAM_DATA_V6_DISCARD FWPM_LAYER_INBOUND_ICMP_ERROR_V4 FWPM_LAYER_INBOUND_ICMP_ERROR_V6 FWPM_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD FWPM_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD

	FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD This flag is applicable at the following filtering layers in Windows Server 2008 R2, Windows 7, and later versions of Windows: FWPM_LAYER_ALE_CONNECT_REDIRECT_V4 FWPM_LAYER_ALE_CONNECT_REDIRECT_V6 FWPM_LAYER_ALE_BIND_REDIRECT_V4 FWPM_LAYER_ALE_BIND_REDIRECT_V6
FWP_CONDITION_FLAG_IS_FRAGMENT 0x00000020	Indicates that the **NET_BUFFER_LIST** structure passed to a callout driver is an IP packet fragment. This flag is applicable at the following filtering layers: FWPM_LAYER_INBOUND_IPPACKET_V4 FWPM_LAYER_INBOUND_IPPACKET_V6 FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_INBOUND_IPPACKET_V6_DISCARD
FWP_CONDITION_FLAG_IS_FRAGMENT_GROUP 0x00000040	Indicates that the **NET_BUFFER_LIST** structure passed to a callout driver describes a linked list of packet fragments. This flag is applicable at the following filtering layers: FWPM_LAYER_INBOUND_IPPACKET_V4 FWPM_LAYER_INBOUND_IPPACKET_V6 FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_INBOUND_IPPACKET_V6_DISCARD
FWP_CONDITION_FLAG_IS_IPSEC_NATT_RECLASSIFY 0x00000080	This flag is set when an NAT Traversal (UDP port 4500) packet is indicated. Once the decapsulation occurs, the flag is set for the reclassify using the information from the encapsulated packet. This flag is applicable at the following filtering layers: FWPM_LAYER_INBOUND_TRANSPORT_V4 FWPM_LAYER_INBOUND_TRANSPORT_V6 FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V4 FWPM_LAYER_OUTBOUND_TRANSPORT_V6 FWPM_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD
FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY 0x00000100	Indicates that the packet has not yet reached the ALE receive/accept layer (FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 or FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6), where its connection state will be tracked. This flag is applicable at the following filtering layers:

	FWPM_LAYER_INBOUND_TRANSPORT_V4 FWPM_LAYER_INBOUND_TRANSPORT_V6 FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD
FWP_CONDITION_FLAG_IS_IMPLICIT_BIND 0x00000200	Indicates that the socket was not explicitly bound. If the sender calls send without first calling bind, Windows Sockets performs an implicit bind. **Note** This flag is supported only in Windows Server 2008 and Windows Vista. It is deprecated in later Windows versions. This flag is applicable at the following filtering layers: FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD
FWP_CONDITION_FLAG_IS_REASSEMBLED 0x00000400	Indicates that the packet has been reassembled from a group of fragments. This flag is applicable at the following filtering layers in Windows Server 2008, Windows Vista with Service Pack 1 (SP1), and later versions of Windows: FWPM_LAYER_INBOUND_IPPACKET_V4 FWPM_LAYER_INBOUND_IPPACKET_V6 FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_INBOUND_IPPACKET_V6_DISCARD
FWP_CONDITION_FLAG_IS_NAME_APP_SPECIFIED 0x00004000	Indicates that the name of the peer machine that the application is expecting to connect to has been obtained by calling a function such as **WSASetSocketPeerTargetName** and not by using the caching heuristics. This flag is applicable at the following filtering layers in Windows Server 2008 R2, Windows 7, and later versions of Windows: FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD
FWP_CONDITION_FLAG_IS_PROMISCUOUS 0x00008000	Reserved for future use.
FWP_CONDITION_FLAG_IS_AUTH_FW 0x00010000	Indicates that a packet matches authenticated firewall policies. Only connections matching the "Allow the connection if it is secure" firewall rule option will have this flag set. For more information, see How to Enable Authenticated Firewall Bypass.

	This flag is also applicable at the following filtering layers in Windows Server 2008, Windows Vista with SP1, and later versions of Windows: FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD This flag is also applicable at the following filtering layers in Windows Server 2008 R2, Windows 7, and later versions of Windows: FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD
FWP_CONDITION_FLAG_IS_RECLASSIFY 0x00020000	This flag is set when the IPV6_PROTECTION_LEVEL socket option is set on a previously authorized socket. This flag is applicable at the following filtering layers: FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD FWPM_LAYER_ALE_AUTH_LISTEN_V6 FWPM_LAYER_ALE_AUTH_LISTEN_V6_DISCARD
FWP_CONDITION_FLAG_IS_OUTBOUND_PASS_THRU 0x00040000	Indicates that the packet is weak-host sent, which means that it isn't leaving this network interface and therefore must be forwarded to another interface. This flag is applicable at the following filtering layers in Windows Server 2008 R2, Windows 7, and later versions of Windows: FWPM_LAYER_IPFORWARD_V4 FWPM_LAYER_IPFORWARD_V6 FWPM_LAYER_IPFORWARD_V4_DISCARD FWPM_LAYER_IPFORWARD_V6_DISCARD
FWP_CONDITION_FLAG_IS_INBOUND_PASS_THRU 0x00080000	Indicates that the packet is weak-host received, which means that it isn't destined for the receiving network interface and therefore must be forwarded to another interface. This flag is applicable at the following filtering layers in Windows Server 2008 R2, Windows 7, and later versions of Windows: FWPM_LAYER_IPFORWARD_V4 FWPM_LAYER_IPFORWARD_V6 FWPM_LAYER_IPFORWARD_V4_DISCARD FWPM_LAYER_IPFORWARD_V6_DISCARD
FWP_CONDITION_FLAG_IS_CONNECTION_REDIRECTED 0x00100000	Indicates that the connection was redirected by an ALE_CONNECT_REDIRECT callout function.

	This flag is applicable at the following filtering layers in Windows Server 2008 R2, Windows 7, and later versions of Windows: FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD
FWP_CONDITION_FLAG_IS_PROXY_CONNECTION 0x00200000	Indicates that the connection has been proxied, and therefore previous redirect records exist. This flag is applicable at the following filtering layers in Windows Server 2012, Windows 8, and later versions of Windows: FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD FWPM_LAYER_ALE_CONNECT_REDIRECT_V4 FWPM_LAYER_ALE_CONNECT_REDIRECT_V6
FWP_CONDITION_FLAG_IS_APPCONTAINER_LOOPBACK 0x00400000	Indicates that the traffic is going to and from an AppContainer that is using loopback. This flag is applicable at the following filtering layers in Windows Server 2012, Windows 8, and later versions of Windows: FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD
FWP_CONDITION_FLAG_IS_NON_APPCONTAINER_LOOPBACK 0x00800000	Indicates that the traffic is going to and from a standard app (not an AppContainer) that is using loopback. This flag is applicable at the following filtering layers in Windows Server 2012, Windows 8, and later versions of Windows: FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD
FWP_CONDITION_FLAG_IS_RESERVED	Reserved for future use.

0x01000000	
FWP_CONDITION_FLAG_IS_HONORING_POLICY_AUTHORIZE 0x02000000	Indicates that the current classification is being performed to honor the intention of a redirected Universal Windows app to connect to a specified host. Such a classification will contain the same classifiable field values as if the app were never redirected. The flag also indicates that a future classification will be invoked to match the effective redirected destination. If the app is redirected to a proxy service for inspection, it also means a future classification will be invoked on the proxy connection. Callouts should use FWPS_FIELD_ALE_AUTH_CONNECT_V4_ALE_ORIGINAL_APP_ID to find the appid of the (original) redirected connection. This flag is applicable at the following filtering layers in Windows Server 2012, Windows 8, and later versions of Windows: FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD

Filtering condition L2 flags

Article • 12/15/2021

The filtering condition L2 flags are each represented by a bit field. These flags are defined as follows:

Filtering condition flag	Description
FWP_CONDITION_L2_IS_NATIVE_ETHERNET 0x00000001	Tests if the data passed to a callout driver describes a native Ethernet packet. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE Tests if the data passed to a callout driver describes a WIFI packet. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE
FWP_CONDITION_L2_IS_WIFI 0x00000002	Tests if the data passed to a callout driver describes a native Wi-Fi packet. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE
FWP_CONDITION_L2_IS_MOBILE_BROADBAND 0x00000004	Tests if the data passed to a callout driver describes a mobile broadband packet. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET

	FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE
FWP_CONDITION_L2_IS_WIFI_DIRECT_DATA 0x00000008	Tests if the data passed to a callout driver describes a WIFI direct data packet. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE
FWP_CONDITION_L2_IS_VM2VM 0x00000010	Tests if the data passed to a callout driver describes a native virtual machine to virtual machine packet. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE
FWP_CONDITION_L2_IS_MALFORMED_PACKET 0x00000020	Tests if the data passed to a callout driver describes a malformed Ethernet packet. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE
FWP_CONDITION_L2_IS_IP_FRAGMENT_GROUP 0x00000040	Tests if the data passed to a callout driver describes a part of an IP fragment group. This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE
FWP_CONDITION_L2_IF_CONNECTOR_PRESENT 0x00000080	Tests if a network interface connector is present on the underlying network adapter. This flag should be set for a physical adapter.

	This flag is applicable at the following filtering layers in Windows 8, Windows Server 2012, and later versions of Windows: FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE

Filtering condition data types

Article • 12/15/2021

The data type for the condition value for each filtering condition is specified as an FWP_DATA_TYPE value as follows.

Filtering condition identifier	Condition value data type
FWPM_CONDITION_IP_LOCAL_ADDRESS	For an IPv4 address: FWP_V4_ADDR_MASK or FWP_UINT32 For an IPv6 address: FWP_V6_ADDR_MASK or FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_IP_REMOTE_ADDRESS	For an IPv4 address: FWP_V4_ADDR_MASK or FWP_UINT32 For an IPv6 address: FWP_V6_ADDR_MASK or FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_IP_SOURCE_ADDRESS	For an IPv4 address: FWP_V4_ADDR_MASK or FWP_UINT32 For an IPv6 address: FWP_V6_ADDR_MASK or FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_IP_DESTINATION_ADDRESS	For an IPv4 address: FWP_V4_ADDR_MASK or FWP_UINT32 For an IPv6 address: FWP_V6_ADDR_MASK or FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE	FWP_UINT8
FWPM_CONDITION_IP_DESTINATION_ADDRESS_TYPE	FWP_UINT8
FWPM_CONDITION_IP_LOCAL_INTERFACE	FWP_UINT64
FWPM_CONDITION_IP_FORWARD_INTERFACE	FWP_UINT64
FWPM_CONDITION_IP_PROTOCOL	FWP_UINT8

FWPM_CONDITION_IP_LOCAL_PORT	FWP_UINT16
FWPM_CONDITION_IP_REMOTE_PORT	FWP_UINT16
FWPM_CONDITION_ICMP_TYPE	FWP_UINT16
FWPM_CONDITION_ICMP_CODE	FWP_UINT16
FWPM_CONDITION_EMBEDDED_LOCAL_ADDRESS_TYPE	FWP_UINT8
FWPM_CONDITION_EMBEDDED_REMOTE_ADDRESS	For an IPv4 address: FWP_V4_ADDR_MASK or FWP_UINT32 For an IPv6 address: FWP_V6_ADDR_MASK or FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_EMBEDDED_PROTOCOL	FWP_UINT8
FWPM_CONDITION_EMBEDDED_LOCAL_PORT	FWP_UINT16
FWPM_CONDITION_EMBEDDED_REMOTE_PORT	FWP_UINT16
FWPM_CONDITION_FLAGS	FWP_UINT32
FWPM_CONDITION_DIRECTION	FWP_UINT32
FWPM_CONDITION_INTERFACE_INDEX	FWP_UINT32
FWPM_CONDITION_SUB_INTERFACE_INDEX	FWP_UINT32
FWPM_CONDITION_SOURCE_INTERFACE_INDEX	FWP_UINT32
FWPM_CONDITION_SOURCE_SUB_INTERFACE_INDEX	FWP_UINT32
FWPM_CONDITION_DESTINATION_INTERFACE_INDEX	FWP_UINT32
FWPM_CONDITION_DESTINATION_SUB_INTERFACE_INDEX	FWP_UINT32
FWPM_CONDITION_ALE_APP_ID	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_ALE_USER_ID	FWP_SECURITY_DESCRIPTOR_TYPE
FWPM_CONDITION_ALE_REMOTE_USER_ID	FWP_SECURITY_DESCRIPTOR_TYPE
FWPM_CONDITION_ALE_REMOTE_MACHINE_ID	FWP_SECURITY_DESCRIPTOR_TYPE
FWPM_CONDITION_ALE_PROMISCUOUS_MODE	FWP_UNIT32
FWPM_CONDITION_ALE_SIO_FIREWALL_SYSTEM_PORT	FWP_UINT32
FWPM_CONDITION_ALE_NAP_CONTEXT	FWP_UINT32

FWPM_CONDITION_REMOTE_USER_TOKEN	FWP_SECURITY_DESCRIPTOR_TYPE
FWPM_CONDITION_RPC_IF_UUID	FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_RPC_IF_VERSION	FWP_UINT16
FWPM_CONDITION_RPC_IF_FLAG	FWP_UINT32
FWPM_CONDITION_DCOM_APP_ID	FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_IMAGE_NAME	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_RPC_PROTOCOL	FWP_UINT8
FWPM_CONDITION_RPC_AUTH_TYPE	FWP_UINT8
FWPM_CONDITION_RPC_AUTH_LEVEL	FWP_UINT8
FWPM_CONDITION_SEC_ENCRYPT_ALGORITHM	FWP_UINT32
FWPM_CONDITION_SEC_KEY_SIZE	FWP_UINT32
FWPM_CONDITION_IP_LOCAL_ADDRESS_V4	FWP_V4_ADDR_MASK or FWP_UINT32
FWPM_CONDITION_IP_LOCAL_ADDRESS_V6	FWP_V6_ADDR_MASK or FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_PIPE	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_IP_REMOTE_ADDRESS_V4	FWP_V4_ADDR_MASK or FWP_UINT32
FWPM_CONDITION_IP_REMOTE_ADDRESS_V6	FWP_V6_ADDR_MASK or FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_PROCESS_WITH_RPC_IF_UUID	FWP_BYTE_ARRAY16_TYPE
FWPM_CONDITION_RPC_EP_VALUE	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_RPC_EP_FLAGS	FWP_UINT32
FWPM_CONDITION_CLIENT_TOKEN	FWP_SECURITY_DESCRIPTOR_TYPE
FWPM_CONDITION_RPC_SERVER_NAME	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_RPC_SERVER_PORT	FWP_UINT16
FWPM_CONDITION_RPC_PROXY_AUTH_TYPE	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_CLIENT_CERT_KEY_LENGTH	FWP_UINT32
FWPM_CONDITION_CLIENT_CERT_OID	FWP_BYTE_BLOB_TYPE

FWPM_CONDITION_INTERFACE_MAC_ADDRESS	FWP_BYTE_ARRAY6_TYPE
FWPM_CONDITION_LOCAL_MAC_ADDRESS	FWP_BYTE_ARRAY6_TYPE
FWPM_CONDITION_REMOTE_MAC_ADDRESS	FWP_BYTE_ARRAY6_TYPE
FWPM_CONDITION_ETHER_TYPE	FWP_UINT16
FWPM_CONDITION_VLAN_ID	FWP_UINT16
FWPM_CONDITION_NDIS_PORT	FWP_UINT16
FWPM_CONDITION_NDIS_MEDIA_TYPE	FWP_UINT16
FWPM_CONDITION_NDIS_PHYSICAL_MEDIA_TYPE	FWP_UINT16
FWPM_CONDITION_L2_FLAGS	FWP_UINT16
FWPM_CONDITION_LOCAL_MAC_ADDRESS_TYPE	FWP_UINT8
FWPM_CONDITION_REMOTE_MAC_ADDRESS_TYPE	FWP_UINT8
FWPM_CONDITION_INTERFACE	FWP_UINT64
FWPM_CONDITION_PACKAGE_ID	FWP_SID
FWPM_CONDITION_MAC_SOURCE_ADDRESS	FWP_BYTE_ARRAY6_TYPE
FWPM_CONDITION_MAC_DESTINATION_ADDRESS	FWP_BYTE_ARRAY6_TYPE
FWPM_CONDITION_MAC_SOURCE_ADDRESS_TYPE	FWP_UINT8
FWPM_CONDITION_MAC_DESTINATION_ADDRESS_TYPE	FWP_UINT8
FWPM_CONDITION_IP_SOURCE_PORT	FWP_UINT16
FWPM_CONDITION_IP_DESTINATION_PORT	FWP_UINT16
FWPM_CONDITION_VSWITCH_ID	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_VSWITCH_NETWORK_TYPE	FWP_UINT8
FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_ID	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_ID	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_TYPE	FWP_UINT8
FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_TYPE	FWP_UINT8
FWPM_CONDITION_VSWITCH_SOURCE_VM_ID	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_VSWITCH_DESTINATION_VM_ID	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_VSWITCH_TENANT_NETWORK_ID	FWP_UINT16

FWPM_CONDITION_ALE_PACKAGE_ID	FWP_SID
FWPM_CONDITION_ALE_ORIGINAL_APP_ID	FWP_BYTE_BLOB_TYPE
FWPM_CONDITION_QM_MODE	FWP_UINT32

Filtering conditions available at each filtering layer

Article • 12/15/2021

The filtering conditions that are available at each filtering layer are as follows.

> **Note**
>
> The V4 and V6 suffixes at the end of the layer identifiers indicate whether the layer is located in the IPv4 network stack or in the IPv6 network stack.

Management filtering layer identifier	Available filtering conditions
FWPM_LAYER_INBOUND_IPPACKET_V4 FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_INBOUND_IPPACKET_V6 FWPM_LAYER_INBOUND_IPPACKET_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX FWPM_CONDITION_TUNNEL_TYPE
FWPM_LAYER_OUTBOUND_IPPACKET_V4 FWPM_LAYER_OUTBOUND_IPPACKET_V4_DISCARD FWPM_LAYER_OUTBOUND_IPPACKET_V6 FWPM_LAYER_OUTBOUND_IPPACKET_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX FWPM_CONDITION_TUNNEL_TYPE
FWPM_LAYER_IPFORWARD_V4 FWPM_LAYER_IPFORWARD_V4_DISCARD FWPM_LAYER_IPFORWARD_V6 FWPM_LAYER_IPFORWARD_V6_DISCARD	FWPM_CONDITION_IP_SOURCE_ADDRESS FWPM_CONDITION_IP_DESTINATION_ADDRESS FWPM_CONDITION_IP_DESTINATION_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE

	FWPM_CONDITION_IP_FORWARD_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_SOURCE_INTERFACE_INDEX FWPM_CONDITION_SOURCE_SUB_INTERFACE_INDEX FWPM_CONDITION_DESTINATION_INTERFACE_INDEX FWPM_CONDITION_DESTINATION_SUB_INTERFACE_INDEX
FWPM_LAYER_INBOUND_TRANSPORT_V4 FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_INBOUND_TRANSPORT_V6 FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX FWPM_CONDITION_TUNNEL_TYPE
FWPM_LAYER_OUTBOUND_TRANSPORT_V4 FWPM_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPM_LAYER_OUTBOUND_TRANSPORT_V6 FWPM_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_DESTINATION_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX FWPM_CONDITION_TUNNEL_TYPE
FWPM_LAYER_STREAM_V4 FWPM_LAYER_STREAM_V4_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS

FWPM_LAYER_STREAM_V6 FWPM_LAYER_STREAM_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_DIRECTION
FWPM_LAYER_DATAGRAM_DATA_V4 FWPM_LAYER_DATAGRAM_DATA_V4_DISCARD FWPM_LAYER_DATAGRAM_DATA_V6 FWPM_LAYER_DATAGRAM_DATA_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_DIRECTION FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX FWPM_CONDITION_TUNNEL_TYPE
FWPM_LAYER_INBOUND_ICMP_ERROR_V4 FWPM_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD FWPM_LAYER_INBOUND_ICMP_ERROR_V6 FWPM_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_ARRIVAL_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_ICMP_TYPE FWPM_CONDITION_ICMP_CODE FWPM_CONDITION_EMBEDDED_LOCAL_ADDRESS_TYPE FWPM_CONDITION_EMBEDDED_REMOTE_ADDRESS FWPM_CONDITION_EMBEDDED_PROTOCOL FWPM_CONDITION_EMBEDDED_LOCAL_PORT FWPM_CONDITION_EMBEDDED_REMOTE_PORT FWPM_CONDITION_ARRIVAL_INTERFACE_INDEX FWPM_CONDITION_ARRIVAL_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX

	Note In Windows Vista, this flag was called FWPM_CONDITION_ARRIVAL_SUB_INTERFACE_INDEX. In Windows Vista with Service Pack 1 (SP1) and later, both names are valid. FWPM_CONDITION_ARRIVAL_TUNNEL_TYPE FWPM_CONDITION_INTERFACE_INDEX **Note** In Windows Vista, this flag was called FWPM_CONDITION_LOCAL_INTERFACE_INDEX. In Windows Vista with SP1 and later, both names are valid. FWPM_CONDITION_INTERFACE_TYPE **Note** In Windows Vista, this flag was called FWPM_CONDITION_LOCAL_INTERFACE_TYPE. In Windows Vista with SP1 and later, both names are valid. FWPM_CONDITION_TUNNEL_TYPE **Note** In Windows Vista, this flag was called FWPM_CONDITION_LOCAL_TUNNEL_TYPE. In Windows Vista with SP1 and later, both names are valid.
FWPM_LAYER_OUTBOUND_ICMP_ERROR_V4 FWPM_LAYER_OUTBOUND_ICMP_ERROR_V4_DISCARD FWPM_LAYER_OUTBOUND_ICMP_ERROR_V6 FWPM_LAYER_OUTBOUND_ICMP_ERROR_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_ICMP_TYPE FWPM_CONDITION_ICMP_CODE FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX FWPM_CONDITION_TUNNEL_TYPE
FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6 FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID

	FWPM_CONDITION_ALE_PROMISCUOUS_MODE FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_TUNNEL_TYPE FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_ALE_AUTH_LISTEN_V4 FWPM_LAYER_ALE_AUTH_LISTEN_V4_DISCARD FWPM_LAYER_ALE_AUTH_LISTEN_V6 FWPM_LAYER_ALE_AUTH_LISTEN_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_TUNNEL_TYPE FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_IP_ARRIVAL_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_ALE_REMOTE_USER_ID FWPM_CONDITION_ALE_REMOTE_MACHINE_ID FWPM_CONDITION_ALE_SIO_FIREWALL_SYSTEM_PORT FWPM_CONDITION_ALE_NAP_CONTEXT FWPM_CONDITION_ARRIVAL_INTERFACE_INDEX FWPM_CONDITION_ARRIVAL_INTERFACE_TYPE FWPM_CONDITION_SUB_INTERFACE_INDEX

	Note In Windows Vista, this flag was called FWPM_CONDITION_ARRIVAL_SUB_INTERFACE_INDEX. In Windows Vista with SP1 and later, both names are valid. FWPM_CONDITION_ARRIVAL_TUNNEL_TYPE FWPM_CONDITION_INTERFACE_INDEX **Note** In Windows Vista, this flag was called FWPM_CONDITION_LOCAL_INTERFACE_INDEX. In Windows Vista with SP1 and later, both names are valid. FWPM_CONDITION_INTERFACE_TYPE **Note** In Windows Vista, this flag was called FWPM_CONDITION_LOCAL_INTERFACE_TYPE. In Windows Vista with SP1 and later, both names are valid. FWPM_CONDITION_TUNNEL_TYPE **Note** In Windows Vista, this flag was called FWPM_CONDITION_LOCAL_TUNNEL_TYPE. In Windows Vista with SP1 and later, both names are valid. FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_ALE_AUTH_CONNECT_V4 FWPM_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPM_LAYER_ALE_AUTH_CONNECT_V6 FWPM_LAYER_ALE_AUTH_CONNECT_V6_DISCARD	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_DESTINATION_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_ALE_REMOTE_USER_ID FWPM_CONDITION_ALE_REMOTE_MACHINE_ID FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_TUNNEL_TYPE FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE

FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD	FWPM_CONDITION_IP_DESTINATION_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_DIRECTION FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_ALE_REMOTE_USER_ID FWPM_CONDITION_ALE_REMOTE_MACHINE_ID FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_TUNNEL_TYPE FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_NAME_RESOLUTION_CACHE_V4 FWPM_LAYER_NAME_RESOLUTION_CACHE_V6	FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_PEER_NAME
FWPM_LAYER_ALE_RESOURCE_RELEASE_V4 FWPM_LAYER_ALE_RESOURCE_RELEASE_V6	FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V4 FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V6	FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_PROTOCOL

	FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_ALE_CONNECT_REDIRECT_V4 FWPM_LAYER_ALE_CONNECT_REDIRECT_V6	FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_DESTINATION_ADDRESS_TYPE FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_ORIGINAL_APP_ID FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_ALE_BIND_REDIRECT_V4 FWPM_LAYER_ALE_BIND_REDIRECT_V6	FWPM_CONDITION_ALE_APP_ID FWPM_CONDITION_ALE_USER_ID FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_LOCAL_ADDRESS_TYPE FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_FLAGS FWPM_CONDITION_ALE_PACKAGE_ID
FWPM_LAYER_STREAM_PACKET_V4 FWPM_LAYER_STREAM_PACKET_V6	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT FWPM_CONDITION_IP_LOCAL_INTERFACE FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_SUB_INTERFACE_INDEX FWPM_CONDITION_DIRECTION

	FWPM_CONDITION_FLAGS FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_TUNNEL_TYPE
FWPM_LAYER_INBOUND_MAC_FRAME_ETHERNET FWPM_LAYER_OUTBOUND_MAC_FRAME_ETHERNET	FWPM_CONDITION_INTERFACE_MAC_ADDRESS FWPM_CONDITION_MAC_LOCAL_ADDRESS FWPM_CONDITION_MAC_REMOTE_ADDRESS FWPM_CONDITION_MAC_LOCAL_ADDRESS_TYPE FWPM_CONDITION_MAC_REMOTE_ADDRESS_TYPE FWPM_CONDITION_ETHER_TYPE FWPM_CONDITION_VLAN_ID FWPM_CONDITION_INTERFACE FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_NDIS_PORT FWPM_CONDITION_L2_FLAGS
FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE	FWPM_CONDITION_NDIS_MEDIA_TYPE FWPM_CONDITION_NDIS_PHYSICAL_MEDIA_TYPE FWPM_CONDITION_INTERFACE FWPM_CONDITION_INTERFACE_TYPE FWPM_CONDITION_INTERFACE_INDEX FWPM_CONDITION_NDIS_PORT FWPM_CONDITION_L2_FLAGS
FWPM_LAYER_INGRESS_VSWITCH_ETHERNET	FWPM_CONDITION_MAC_SOURCE_ADDRESS FWPM_CONDITION_MAC_SOURCE_ADDRESS_TYPE FWPM_CONDITION_MAC_DESTINATION_ADDRESS FWPM_CONDITION_MAC_DESTINATION_ADDRESS_TYPE FWPM_CONDITION_ETHER_TYPE FWPM_CONDITION_VLAN_ID FWPM_CONDITION_VSWITCH_TENANT_NETWORK_ID FWPM_CONDITION_VSWITCH_ID FWPM_CONDITION_VSWITCH_NETWORK_TYPE FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_ID FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_TYPE FWPM_CONDITION_VSWITCH_SOURCE_VM_ID

	FWPM_CONDITION_L2_FLAGS
FWPM_LAYER_EGRESS_VSWITCH_ETHERNET	FWPM_CONDITION_MAC_SOURCE_ADDRESS FWPM_CONDITION_MAC_SOURCE_ADDRESS_TYPE FWPM_CONDITION_MAC_DESTINATION_ADDRESS FWPM_CONDITION_MAC_DESTINATION_ADDRESS_TYPE FWPM_CONDITION_ETHER_TYPE FWPM_CONDITION_VLAN_ID FWPM_CONDITION_VSWITCH_TENANT_NETWORK_ID FWPM_CONDITION_VSWITCH_ID FWPM_CONDITION_VSWITCH_NETWORK_TYPE FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_ID FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_TYPE FWPM_CONDITION_VSWITCH_SOURCE_VM_ID FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_ID FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_TYPE FWPM_CONDITION_VSWITCH_DESTINATION_VM_ID FWPM_CONDITION_L2_FLAGS
FWPM_LAYER_INGRESS_VSWITCH_TRANSPORT_V4 FWPM_LAYER_INGRESS_VSWITCH_TRANSPORT_V6	FWPM_CONDITION_IP_SOURCE_ADDRESS FWPM_CONDITION_IP_DESTINATION_ADDRESS FWPM_CONDITION_P_PROTOCOL FWPM_CONDITION_IP_SOURCE_PORT FWPM_CONDITION_IP_DESTINATION_PORT FWPM_CONDITION_VLAN_ID FWPM_CONDITION_VSWITCH_TENANT_NETWORK_ID FWPM_CONDITION_VSWITCH_ID FWPM_CONDITION_VSWITCH_NETWORK_TYPE FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_ID FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_TYPE FWPM_CONDITION_VSWITCH_SOURCE_VM_ID FWPM_CONDITION_L2_FLAGS
FWPM_LAYER_EGRESS_VSWITCH_TRANSPORT_V4 FWPM_LAYER_EGRESS_VSWITCH_TRANSPORT_V6	FWPM_CONDITION_IP_SOURCE_ADDRESS FWPM_CONDITION_IP_DESTINATION_ADDRESS FWPM_CONDITION_IP_PROTOCOL

	FWPM_CONDITION_IP_SOURCE_PORT FWPM_CONDITION_IP_DESTINATION_PORT FWPM_CONDITION_VLAN_ID FWPM_CONDITION_VSWITCH_TENANT_NETWORK_ID FWPM_CONDITION_VSWITCH_ID FWPM_CONDITION_VSWITCH_NETWORK_TYPE FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_ID FWPM_CONDITION_VSWITCH_SOURCE_INTERFACE_TYPE FWPM_CONDITION_VSWITCH_SOURCE_VM_ID FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_ID FWPM_CONDITION_VSWITCH_DESTINATION_INTERFACE_TYPE FWPM_CONDITION_VSWITCH_DESTINATION_VM_ID FWPM_CONDITION_L2_FLAGS
FWPM_LAYER_IPSEC_KM_DEMUX_V4 FWPM_LAYER_IPSEC_KM_DEMUX_V6	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS
FWPM_LAYER_IPSEC_V4 FWPM_LAYER_IPSEC_V6	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS FWPM_CONDITION_IP_PROTOCOL FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_IP_REMOTE_PORT
FWPM_LAYER_IKEEXT_V4 FWPM_LAYER_IKEEXT_V6	FWPM_CONDITION_IP_LOCAL_ADDRESS FWPM_CONDITION_IP_REMOTE_ADDRESS
FWPM_LAYER_RPC_UM	FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_REMOTE_USER_TOKEN FWPM_CONDITION_RPC_IF_UUID FWPM_CONDITION_RPC_IF_VERSION FWPM_CONDITION_RPC_PROTOCOL FWPM_CONDITION_RPC_IF_FLAG FWPM_CONDITION_DCOM_APP_ID FWPM_CONDITION_IMAGE_NAME FWPM_CONDITION_RPC_AUTH_TYPE FWPM_CONDITION_RPC_AUTH_LEVEL FWPM_CONDITION_SEC_ENCRYPT_ALGORITHM

	FWPM_CONDITION_SEC_KEY_SIZE FWPM_CONDITION_IP_LOCAL_ADDRESS_V4 FWPM_CONDITION_IP_LOCAL_ADDRESS_V6 FWPM_CONDITION_PIPE FWPM_CONDITION_IP_REMOTE_ADDRESS_V4 FWPM_CONDITION_IP_REMOTE_ADDRESS_V6
FWPM_LAYER_RPC_EPMAP	FWPM_CONDITION_IP_LOCAL_PORT FWPM_CONDITION_REMOTE_USER_TOKEN FWPM_CONDITION_RPC_IF_UUID FWPM_CONDITION_RPC_IF_VERSION FWPM_CONDITION_RPC_PROTOCOL FWPM_CONDITION_RPC_AUTH_TYPE FWPM_CONDITION_RPC_AUTH_LEVEL FWPM_CONDITION_SEC_ENCRYPT_ALGORITHM FWPM_CONDITION_SEC_KEY_SIZE FWPM_CONDITION_IP_LOCAL_ADDRESS_V4 FWPM_CONDITION_IP_LOCAL_ADDRESS_V6 FWPM_CONDITION_PIPE FWPM_CONDITION_IP_REMOTE_ADDRESS_V4 FWPM_CONDITION_IP_REMOTE_ADDRESS_V6
FWPM_LAYER_RPC_EP_ADD	FWPM_CONDITION_RPC_PROTOCOL FWPM_CONDITION_PROCESS_WITH_RPC_IF_UUID FWPM_CONDITION_RPC_EP_VALUE FWPM_CONDITION_RPC_EP_FLAGS
FWPM_LAYER_RPC_PROXY_CONN	FWPM_CONDITION_CLIENT_TOKEN FWPM_CONDITION_RPC_SERVER_NAME FWPM_CONDITION_RPC_SERVER_PORT FWPM_CONDITION_RPC_PROXY_AUTH_TYPE FWPM_CONDITION_CLIENT_CERT_KEY_LENGTH FWPM_CONDITION_CLIENT_CERT_OID
FWPM_LAYER_RPC_PROXY_IF	FWPM_CONDITION_RPC_IF_UUID FWPM_CONDITION_RPC_IF_VERSION FWPM_CONDITION_CLIENT_TOKEN

	FWPM_CONDITION_RPC_SERVER_NAME FWPM_CONDITION_RPC_SERVER_PORT FWPM_CONDITION_RPC_PROXY_AUTH_TYPE FWPM_CONDITION_CLIENT_CERT_KEY_LENGTH FWPM_CONDITION_CLIENT_CERT_OID
FWPM_LAYER_KM_AUTHORIZATION	FWPM_CONDITION_REMOTE_ID FWPM_CONDITION_AUTHENTICATION_TYPE FWPM_CONDITION_KM_TYPE FWPM_CONDITION_DIRECTION FWPM_CONDITION_KM_MODE FWPM_CONDITION_IPSEC_POLICY_KEY FWPM_CONDITION_KM_AUTH_NAP_CONTEXT

Metadata field identifiers

Article • 12/15/2021

The metadata field identifiers are each represented by a bit-field. These identifiers are defined as follows:

Metadata field identifier	Description
FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED	The inbound packet will also be indicated to the FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 and FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6 filtering layers. **Note:** Supported in Windows Server 2008, Windows Vista with Service Pack 1 (SP1), and later.
FWPS_METADATA_FIELD_COMPARTMENT_ID	The identifier of the routing compartment in which the packet was received or is being sent.
FWPS_METADATA_FIELD_COMPLETION_HANDLE	The completion handle used to pend the current filtering operation.
FWPS_METADATA_FIELD_DESTINATION_INTERFACE_INDEX	The index of the network interface where the outgoing packet is to be sent.
FWPS_METADATA_FIELD_DESTINATION_PREFIX	The destination IPV4 or IPV6 address and subnet mask for the outgoing packets. **Note:** Supported starting with Windows 7.
FWPS_METADATA_FIELD_DISCARD_REASON	The reason that the data was discarded.
FWPS_METADATA_FIELD_ETHER_FRAME_LENGTH	This metadata field identifier is not currently supported.
FWPS_METADATA_FIELD_FLOW_HANDLE	The handle for the data flow.
FWPS_METADATA_FIELD_FORWARD_LAYER_INBOUND_PASS_THRU	The packet that traverses the FWPM_LAYER_IPFORWARD_V4 or FWPM_LAYER_IPFORWARD_V6 forward layer is locally destined (its destination matches an address that is assigned to an interface of the host). **Note:** Supported in Windows Server 2008, Windows Vista with SP1, and later.

Metadata field identifier	Description
FWPS_METADATA_FIELD_FORWARD_LAYER_OUTBOUND_PASS_THRU	The packet that traverses the FWPM_LAYER_IPFORWARD_V4 or FWPM_LAYER_IPFORWARD_V6 forward layer originated locally. **Note:** Supported in Windows Server 2008, Windows Vista with SP1, and later.
FWPS_METADATA_FIELD_FRAGMENT_DATA	The fragment data for a received packet fragment.
FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE	The Identifier and Sequence Number fields of an ICMP Echo Request or Echo Reply packet. **Note:** Supported starting with Windows 7.
FWPS_METADATA_FIELD_IP_HEADER_SIZE	The size of the IP header.
FWPS_METADATA_FIELD_LOCAL_REDIRECT_TARGET_PID	The Process ID that a connection was redirected to. **Note:** Supported starting with Windows 7.
FWPS_METADATA_FIELD_ORIGINAL_DESTINATION	A SOCKADDR_STORAGE structure that indicate the packet's original destination. **Note:** Supported starting with Windows 7.
FWPS_METADATA_FIELD_PACKET_DIRECTION	The direction of network traffic (inbound or outbound).
FWPS_METADATA_FIELD_PACKET_SYSTEM_CRITICAL	Reserved for system use. Do not use. **Note:** Supported in Windows Server 2008, Windows Vista with SP1, and later.
FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE	The handle of the endpoint's parent socket. **Note:** Supported starting with Windows 7.
FWPS_METADATA_FIELD_PATH_MTU	The path maximum transmission unit (path MTU) for an outgoing packet.
FWPS_METADATA_FIELD_PROCESS_ID	The process ID for the process that owns the endpoint.
FWPS_METADATA_FIELD_PROCESS_PATH	The full path to the process that owns the endpoint.
FWPS_METADATA_FIELD_REDIRECT_RECORD_HANDLE	The redirect records handle indicated to ALE_CONNECT_REDIRECT callout by the classify metadata. **Note:** Supported starting with Windows 8.
FWPS_METADATA_FIELD_REMOTE_SCOPE_ID	The remote scope identifier to be used in outbound transport layer injection.
FWPS_METADATA_FIELD_RESERVED	Reserved for system use. Do not use.

Metadata field identifier	Description
FWPS_METADATA_FIELD_SOURCE_INTERFACE_INDEX	The index of the network interface where the incoming packet was received.
FWPS_METADATA_FIELD_SUB_PROCESS_TAG	Reserved for system use.
FWPS_METADATA_FIELD_SYSTEM_FLAGS	System flags that are used internally by the filter engine.
FWPS_METADATA_FIELD_TOKEN	The token used to validate the permissions for the user.
FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA	An optional socket control data object.
FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE	The handle for the end of the packet to be injected into the outbound transport layer.
FWPS_METADATA_FIELD_TRANSPORT_HEADER_INCLUDE_HEADER	The IP header if the packet is sent from a raw socket. **Note:** Supported in Windows Server 2008, Windows Vista with SP1, and later.
FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE	The size of the transport header.

Metadata field L2 identifiers

Article • 12/15/2021

Windows 8 and Windows Server 2012 introduce metadata field L2 identifiers.

The metadata field L2 identifiers are each represented by a bit field. These identifiers are defined as follows:

Metadata field identifier	Description
FWPS_L2_METADATA_FIELD_ETHERNET_MAC_HEADER_SIZE	The size, in bytes, of the MAC header.
FWPS_L2_METADATA_FIELD_VSWITCH_DESTINATION_PORT_ID	The identifier for the destination port on the virtual switch.
FWPS_L2_METADATA_FIELD_VSWITCH_PACKET_CONTEXT	A **HANDLE** to the virtual switch packet context.
FWPS_L2_METADATA_FIELD_VSWITCH_SOURCE_NIC_INDEX	The index for the source NIC on the virtual switch.
FWPS_L2_METADATA_FIELD_VSWITCH_SOURCE_PORT_ID	The identifier for the source port on the virtual switch.
FWPS_L2_METADATA_FIELD_WIFI_OPERATION_MODE	The current Native 802.11 operation mode.

Related topics

Metadata field identifiers

Metadata fields at each filtering layer

FWPS_INCOMING_METADATA_VALUES0

Metadata fields at each filtering layer

Article • 12/15/2021

The following table lists the possible metadata fields that are available by layer. Some fields are available only under specific circumstances. For example, the metadata field FWPS_METADATA_FIELD_FRAGMENT_DATA is available for inbound IP packet layers only if the packet is fragmented. Layers that are not listed in the table do not have any available metadata fields.

Run-time filtering layer identifier	Metadata fields
FWPS_LAYER_INBOUND_IPPACKET_V4 FWPS_LAYER_INBOUND_IPPACKET_V6	FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_FRAGMENT_DATA
FWPS_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPS_LAYER_INBOUND_IPPACKET_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_FRAGMENT_DATA
FWPS_LAYER_OUTBOUND_IPPACKET_V4 FWPS_LAYER_OUTBOUND_IPPACKET_V6	FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_FRAGMENT_DATA FWPS_METADATA_FIELD_PATH_MTU
FWPS_LAYER_OUTBOUND_IPPACKET_V4_DISCARD FWPS_LAYER_OUTBOUND_IPPACKET_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_FRAGMENT_DATA FWPS_METADATA_FIELD_PATH_MTU
FWPS_LAYER_IPFORWARD_V4 FWPS_LAYER_IPFORWARD_V6	FWPS_METADATA_FIELD_COMPARTMENT_ID
FWPS_LAYER_IPFORWARD_V4_DISCARD FWPS_LAYER_IPFORWARD_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_COMPARTMENT_ID
FWPS_LAYER_INBOUND_TRANSPORT_V4 FWPS_LAYER_INBOUND_TRANSPORT_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_SYSTEM_FLAGS

	FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE
FWPS_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPS_LAYER_INBOUND_TRANSPORT_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE
FWPS_LAYER_OUTBOUND_TRANSPORT_V4 FWPS_LAYER_OUTBOUND_TRANSPORT_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA FWPS_METADATA_FIELD_REMOTE_SCOPE_ID FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE
FWPS_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPS_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE

	FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA FWPS_METADATA_FIELD_REMOTE_SCOPE_ID FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE
FWPS_LAYER_STREAM_V4 FWPS_LAYER_STREAM_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_SYSTEM_FLAGS
FWPS_LAYER_STREAM_V4_DISCARD FWPS_LAYER_STREAM_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_SYSTEM_FLAGS
FWPS_LAYER_DATAGRAM_DATA_V4 FWPS_LAYER_DATAGRAM_DATA_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA FWPS_METADATA_FIELD_REMOTE_SCOPE_ID FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE
FWPS_LAYER_DATAGRAM_DATA_V4_DISCARD FWPS_LAYER_DATAGRAM_DATA_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA FWPS_METADATA_FIELD_REMOTE_SCOPE_ID FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE
FWPS_LAYER_INBOUND_ICMP_ERROR_V4 FWPS_LAYER_INBOUND_ICMP_ERROR_V6	FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE
FWPS_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON

FWPS_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD	FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE
FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4 FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6	FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA FWPS_METADATA_FIELD_REMOTE_SCOPE_ID
FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4_DISCARD FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA FWPS_METADATA_FIELD_REMOTE_SCOPE_ID
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6	FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_SUB_PROCESS_TAG

FWPS_LAYER_ALE_AUTH_LISTEN_V4 FWPS_LAYER_ALE_AUTH_LISTEN_V6	FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_AUTH_LISTEN_V4_DISCARD FWPS_LAYER_ALE_AUTH_LISTEN_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_PACKET_DIRECTION FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_IP_HEADER_SIZE

	FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_PACKET_DIRECTION FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_AUTH_CONNECT_V4 FWPS_LAYER_ALE_AUTH_CONNECT_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_REMOTE_SCOPE_ID FWPS_METADATA_FIELD_PACKET_DIRECTION FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPS_LAYER_ALE_AUTH_CONNECT_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED

	FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_COMPLETION_HANDLE FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_REMOTE_SCOPE_ID FWPS_METADATA_FIELD_PACKET_DIRECTION FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD	FWPS_METADATA_FIELD_DISCARD_REASON FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_PROCESS_PATH FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_RESERVED FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_INBOUND_MAC_FRAME_802_3	FWPS_METADATA_FIELD_ETHER_FRAME_LENGTH
FWPS_LAYER_ALE_RESOURCE_RELEASE_V4 FWPS_LAYER_ALE_RESOURCE_RELEASE_V6	FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_COMPARTMENT_ID

	FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4 FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6	FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_CONNECT_REDIRECT_V4 FWPS_LAYER_ALE_CONNECT_REDIRECT_V6	FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_ICMP_ID_AND_SEQUENCE FWPS_METADATA_FIELD_LOCAL_REDIRECT_TARGET_PID FWPS_METADATA_FIELD_REDIRECT_RECORD_HANDLE FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_ALE_BIND_REDIRECT_V4 FWPS_LAYER_ALE_BIND_REDIRECT_V6	FWPS_METADATA_FIELD_TOKEN FWPS_METADATA_FIELD_PROCESS_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_LOCAL_REDIRECT_TARGET_PID FWPS_METADATA_FIELD_SUB_PROCESS_TAG
FWPS_LAYER_STREAM_PACKET_V4 FWPS_LAYER_STREAM_PACKET_V6	FWPS_METADATA_FIELD_FLOW_HANDLE FWPS_METADATA_FIELD_IP_HEADER_SIZE FWPS_METADATA_FIELD_SYSTEM_FLAGS FWPS_METADATA_FIELD_TRANSPORT_HEADER_SIZE FWPS_METADATA_FIELD_COMPARTMENT_ID FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE FWPS_METADATA_FIELD_TRANSPORT_CONTROL_DATA FWPS_METADATA_FIELD_REMOTE_SCOPE_ID
FWPS_LAYER_INGRESS_VSWITCH_ETHERNET FWPS_LAYER_EGRESS_VSWITCH_ETHERNET	FWPS_L2_METADATA_FIELD_ETHERNET_MAC_HEADER_SIZE FWPS_L2_METADATA_FIELD_WIFI_OPERATION_MODE FWPS_L2_METADATA_FIELD_VSWITCH_SOURCE_PORT_ID

	FWPS_L2_METADATA_FIELD_VSWITCH_SOURCE_NIC_INDEX FWPS_L2_METADATA_FIELD_VSWITCH_PACKET_CONTEXT FWPS_L2_METADATA_FIELD_VSWITCH_DESTINATION_PORT_ID
FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V4 FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V6 FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V4 FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V6	FWPS_L2_METADATA_FIELD_WIFI_OPERATION_MODE FWPS_L2_METADATA_FIELD_VSWITCH_SOURCE_PORT_ID FWPS_L2_METADATA_FIELD_VSWITCH_SOURCE_NIC_INDEX FWPS_L2_METADATA_FIELD_VSWITCH_PACKET_CONTEXT FWPS_L2_METADATA_FIELD_VSWITCH_DESTINATION_PORT_ID

Data field identifiers

Article • 12/15/2021

The run-time filtering layers are associated with data field identifiers. These identifiers represent a set of constant values that are declared as FWPS_FIELDS_XXX enumerations in Fwpsk.h.

The following table lists the run-time filtering layers and the associated data field enumerations.

Run-time filtering layer	Data field enumeration
FWPS_LAYER_INBOUND_IPPACKET_V4 FWPS_LAYER_INBOUND_IPPACKET_V4_DISCARD	FWPS_FIELDS_INBOUND_IPPACKET_V4
FWPS_LAYER_INBOUND_IPPACKET_V6 FWPS_LAYER_INBOUND_IPPACKET_V6_DISCARD	FWPS_FIELDS_INBOUND_IPPACKET_V6
FWPS_LAYER_OUTBOUND_IPPACKET_V4 FWPS_LAYER_OUTBOUND_IPPACKET_V4_DISCARD	FWPS_FIELDS_OUTBOUND_IPPACKET_V4
FWPS_LAYER_OUTBOUND_IPPACKET_V6 FWPS_LAYER_OUTBOUND_IPPACKET_V6_DISCARD	FWPS_FIELDS_OUTBOUND_IPPACKET_V6
FWPS_LAYER_IPFORWARD_V4 FWPS_LAYER_IPFORWARD_V4_DISCARD	FWPS_FIELDS_IPFORWARD_V4
FWPS_LAYER_IPFORWARD_V6 FWPS_LAYER_IPFORWARD_V6_DISCARD	FWPS_FIELDS_IPFORWARD_V6
FWPS_LAYER_INBOUND_TRANSPORT_V4 FWPS_LAYER_INBOUND_TRANSPORT_V4_DISCARD	FWPS_FIELDS_INBOUND_TRANSPORT_V4
FWPS_LAYER_INBOUND_TRANSPORT_V6 FWPS_LAYER_INBOUND_TRANSPORT_V6_DISCARD	FWPS_FIELDS_INBOUND_TRANSPORT_V6
FWPS_LAYER_OUTBOUND_TRANSPORT_V4 FWPS_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD	FWPS_FIELDS_OUTBOUND_TRANSPORT_V4
FWPS_LAYER_OUTBOUND_TRANSPORT_V6 FWPS_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD	FWPS_FIELDS_OUTBOUND_TRANSPORT_V6
FWPS_LAYER_STREAM_V4 FWPS_LAYER_STREAM_V4_DISCARD	FWPS_FIELDS_STREAM_V4
FWPS_LAYER_STREAM_V6 FWPS_LAYER_STREAM_V6_DISCARD	FWPS_FIELDS_STREAM_V6
FWPS_LAYER_DATAGRAM_DATA_V4 FWPS_LAYER_DATAGRAM_DATA_V4_DISCARD	FWPS_FIELDS_DATAGRAM_DATA_V4
FWPS_LAYER_DATAGRAM_DATA_V6 FWPS_LAYER_DATAGRAM_DATA_V6_DISCARD	FWPS_FIELDS_DATAGRAM_DATA_V6

FWPS_LAYER_STREAM_PACKET_V4	FWPS_FIELDS_STREAM_PACKET_V4 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_STREAM_PACKET_V6	FWPS_FIELDS_STREAM_PACKET_V6 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_INBOUND_ICMP_ERROR_V4 FWPS_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD	FWPS_FIELDS_INBOUND_ICMP_ERROR_V4
FWPS_LAYER_INBOUND_ICMP_ERROR_V6 FWPS_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD	FWPS_FIELDS_INBOUND_ICMP_ERROR_V6
FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4 FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4_DISCARD	FWPS_FIELDS_OUTBOUND_ICMP_ERROR_V4
FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6 FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6_DISCARD	FWPS_FIELDS_OUTBOUND_ICMP_ERROR_V6
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD	FWPS_FIELDS_ALE_RESOURCE_ASSIGNMENT_V4
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6 FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD	FWPS_FIELDS_ALE_RESOURCE_ASSIGNMENT_V6
FWPS_LAYER_ALE_RESOURCE_RELEASE_V4	FWPS_FIELDS_ALE_RESOURCE_RELEASE_V4 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_ALE_RESOURCE_RELEASE_V6	FWPS_FIELDS_ALE_RESOURCE_RELEASE_V6 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4	FWPS_FIELDS_ALE_ENDPOINT_CLOSURE_V4 **Note** Supported in Windows 7 and later versions of Windows.

FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6	FWPS_FIELDS_ALE_ENDPOINT_CLOSURE_V6 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_ALE_AUTH_LISTEN_V4 FWPS_LAYER_ALE_AUTH_LISTEN_V4_DISCARD	FWPS_FIELDS_ALE_AUTH_LISTEN_V4
FWPS_LAYER_ALE_AUTH_LISTEN_V6 FWPS_LAYER_ALE_AUTH_LISTEN_V6_DISCARD	FWPS_FIELDS_ALE_AUTH_LISTEN_V6
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD	FWPS_FIELDS_ALE_AUTH_RECV_ACCEPT_V4
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6 FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD	FWPS_FIELDS_ALE_AUTH_RECV_ACCEPT_V6
FWPS_LAYER_ALE_BIND_REDIRECT_V4	FWPS_FIELDS_ALE_BIND_REDIRECT_V4 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_ALE_BIND_REDIRECT_V6	FWPS_FIELDS_ALE_BIND_REDIRECT_V6 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_ALE_CONNECT_REDIRECT_V4	FWPS_FIELDS_ALE_CONNECT_REDIRECT_V4 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_ALE_CONNECT_REDIRECT_V6	FWPS_FIELDS_ALE_CONNECT_REDIRECT_V6 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_ALE_AUTH_CONNECT_V4 FWPS_LAYER_ALE_AUTH_CONNECT_V4_DISCARD	FWPS_FIELDS_ALE_AUTH_CONNECT_V4

FWPS_LAYER_ALE_AUTH_CONNECT_V6 FWPS_LAYER_ALE_AUTH_CONNECT_V6_DISCARD	FWPS_FIELDS_ALE_AUTH_CONNECT_V6
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD	FWPS_FIELDS_ALE_FLOW_ESTABLISHED_V4
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6 FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD	FWPS_FIELDS_ALE_FLOW_ESTABLISHED_V6
FWPS_LAYER_NAME_RESOLUTION_CACHE_V4	FWPS_FIELDS_NAME_RESOLUTION_CACHE_V4 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_NAME_RESOLUTION_CACHE_V6	FWPS_FIELDS_NAME_RESOLUTION_CACHE_V6 **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_INBOUND_MAC_FRAME_ETHERNET	FWPS_FIELDS_INBOUND_MAC_FRAME_ETHERNET **Note** Supported in *Windows 8* and later versions of Windows.
FWPS_LAYER_OUTBOUND_MAC_FRAME_ETHERNET	FWPS_FIELDS_OUTBOUND_MAC_FRAME_ETHERNET **Note** Supported in *Windows 8* and later versions of Windows.
FWPM_LAYER_INBOUND_MAC_FRAME_NATIVE	FWPS_FIELDS_INBOUND_MAC_FRAME_NATIVE **Note** Supported in *Windows 8* and later versions of Windows.
FWPM_LAYER_OUTBOUND_MAC_FRAME_NATIVE	FWPS_FIELDS_OUTBOUND_MAC_FRAME_NATIVE **Note** Supported in *Windows 8* and later versions of Windows.

FWPS_LAYER_IPSEC_KM_DEMUX_V4	FWPS_FIELDS_IPSEC_KM_DEMUX_V4
FWPS_LAYER_IPSEC_KM_DEMUX_V6	FWPS_FIELDS_IPSEC_KM_DEMUX_V6
FWPS_LAYER_IPSEC_V4	FWPS_FIELDS_IPSEC_V4
FWPS_LAYER_IPSEC_V6	FWPS_FIELDS_IPSEC_V6
FWPS_LAYER_IKEEXT_V4	FWPS_FIELDS_IKEEXT_V4
FWPS_LAYER_IKEEXT_V6	FWPS_FIELDS_IKEEXT_V6
FWPS_LAYER_RPC_UM	FWPS_FIELDS_RPC_UM
FWPS_LAYER_RPC_EPMAP	FWPS_FIELDS_RPC_EPMAP
FWPS_LAYER_RPC_EP_ADD	FWPS_FIELDS_RPC_EP_ADD
FWPS_LAYER_RPC_PROXY_CONN	FWPS_FIELDS_RPC_PROXY_CONN
FWPS_LAYER_RPC_PROXY_IF	FWPS_FIELDS_RPC_PROXY_IF_IF
FWPS_LAYER_KM_AUTHORIZATION	FWPS_FIELDS_KM_AUTHORIZATION **Note** Supported in Windows 7 and later versions of Windows.
FWPS_LAYER_INGRESS_VSWITCH_ETHERNET	FWPS_FIELDS_INGRESS_VSWITCH_ETHERNET **Note** Supported in *Windows 8* and later versions of Windows.
FWPS_LAYER_EGRESS_VSWITCH_ETHERNET	FWPS_FIELDS_EGRESS_VSWITCH_ETHERNET **Note** Supported in *Windows 8* and later versions of Windows.
FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V4	FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V4 **Note** Supported in *Windows 8* and later versions of Windows.
WPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V6	FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V6

	Note Supported in *Windows 8* and later versions of Windows.
FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V4	FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V4 **Note** Supported in *Windows 8* and later versions of Windows.
FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V6	FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V6 **Note** Supported in *Windows 8* and later versions of Windows.

Data offset positions

Article • 12/15/2021

When the filter engine calls a callout driver's classifyFn callout function, it passes a pointer to a structure in the *layerData* parameter. For the layers that filter packet data, the pointer references a NET_BUFFER_LIST structure. Depending on the filtering layer at which the *classifyFn* callout function is called, the filter engine passes a pointer in the layerData* parameter to one of the following structures:

- For the stream layer, the *layerData* parameter contains a pointer to an FWPS_STREAM_CALLOUT_IO_PACKET0 structure. The streamData member of this structure contains a pointer to an FWPS_STREAM_DATA0 structure.

  The **netBufferListChain** member of the FWPS_STREAM_DATA0 structure contains a pointer to a NET_BUFFER_LIST structure.

- For all the other layers, the *layerData* parameter contains a pointer to a NET_BUFFER_LIST structure.

> ⓘ Note
>
> The *layerData* parameter might be NULL, depending on the layer being filtered and the conditions under which the driver's **classifyFn** callout function is called.

The NET_BUFFER_LIST structure contains a linked list of NET_BUFFER structures. Within the NET_BUFFER_DATA structure of each **NET_BUFFER** structure, the **DataOffset** member points to a specific position in the packet data. The position that the **DataOffset** member points to depends on the filtering layer at which the filter engine calls the callout driver's *classifyFn* callout function.

For each filtering layer, the position in the packet data as specified by the **DataOffset** member is defined as follows:

Run-time filtering layer identifier (starting with Windows Vista)	Position in the packet data
FWPS_LAYER_INBOUND_IPPACKET_V4 FWPS_LAYER_INBOUND_IPPACKET_V6	The beginning of the transport header.
FWPS_LAYER_INBOUND_IPPACKET_V4_DISCARD FWPS_LAYER_INBOUND_IPPACKET_V6_DISCARD	The offset where the TCP/IP stack stopped processing.

FWPS_LAYER_OUTBOUND_IPPACKET_V4 FWPS_LAYER_OUTBOUND_IPPACKET_V6	The beginning of the IP header.
FWPS_LAYER_OUTBOUND_IPPACKET_V4_DISCARD FWPS_LAYER_OUTBOUND_IPPACKET_V6_DISCARD	The offset where the TCP/IP stack stopped processing.
FWPS_LAYER_IPFORWARD_V4 FWPS_LAYER_IPFORWARD_V6	The beginning of the IP header.
FWPS_LAYER_IPFORWARD_V4_DISCARD FWPS_LAYER_IPFORWARD_V6_DISCARD	The beginning of the IP header.
FWPS_LAYER_INBOUND_TRANSPORT_V4 FWPS_LAYER_INBOUND_TRANSPORT_V6	The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header.
FWPS_LAYER_INBOUND_TRANSPORT_V4_DISCARD FWPS_LAYER_INBOUND_TRANSPORT_V6_DISCARD	The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header.
FWPS_LAYER_OUTBOUND_TRANSPORT_V4 FWPS_LAYER_OUTBOUND_TRANSPORT_V6	The beginning of the transport header.
FWPS_LAYER_OUTBOUND_TRANSPORT_V4_DISCARD FWPS_LAYER_OUTBOUND_TRANSPORT_V6_DISCARD	The beginning of the transport header.
FWPS_LAYER_STREAM_V4 FWPS_LAYER_STREAM_V6	The beginning of the data. **Note** The position in the packet data contains no IP, IPv6, and transport headers.

FWPS_LAYER_STREAM_V4_DISCARD FWPS_LAYER_STREAM_V6_DISCARD	The beginning of the data. **Note** The position in the packet data contains no IP, IPv6, or transport headers.
FWPS_LAYER_DATAGRAM_DATA_V4 FWPS_LAYER_DATAGRAM_DATA_V6	For inbound datagrams: The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header. For outbound datagrams: The beginning of the transport header.
FWPS_LAYER_DATAGRAM_DATA_V4_DISCARD FWPS_LAYER_DATAGRAM_DATA_V6_DISCARD	For inbound datagrams: The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header. For outbound datagrams: The beginning of the transport header.
FWPS_LAYER_INBOUND_ICMP_ERROR_V4 FWPS_LAYER_INBOUND_ICMP_ERROR_V6	The beginning of the inner IP header.
FWPS_LAYER_INBOUND_ICMP_ERROR_V4_DISCARD FWPS_LAYER_INBOUND_ICMP_ERROR_V6_DISCARD	The beginning of the inner IP header.
FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4 FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6	The beginning of the ICMP header.

FWPS_LAYER_OUTBOUND_ICMP_ERROR_V4_DISCARD FWPS_LAYER_OUTBOUND_ICMP_ERROR_V6_DISCARD	The beginning of the ICMP header.
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4 FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6	Not applicable.
FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V4_DISCARD FWPS_LAYER_ALE_RESOURCE_ASSIGNMENT_V6_DISCARD	Not applicable.
FWPS_LAYER_ALE_AUTH_LISTEN_V4 FWPS_LAYER_ALE_AUTH_LISTEN_V6	Not applicable.
FWPS_LAYER_ALE_AUTH_LISTEN_V4_DISCARD FWPS_LAYER_ALE_AUTH_LISTEN_V6_DISCARD	Not applicable.
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4 FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6	For inbound packet direction: The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header. For outbound packet direction: The beginning of the transport header.
FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4_DISCARD FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6_DISCARD	For inbound packet direction: The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header. For outbound packet direction: The beginning of the transport header.
FWPS_LAYER_ALE_AUTH_CONNECT_V4	For non-TCP traffic: The beginning of the transport header.

FWPS_LAYER_ALE_AUTH_CONNECT_V6	For TCP traffic: Not applicable.
FWPS_LAYER_ALE_AUTH_CONNECT_V4_DISCARD FWPS_LAYER_ALE_AUTH_CONNECT_V6_DISCARD	For non-TCP traffic: The beginning of the transport header. For TCP traffic: Not applicable.
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4 FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6	For inbound packet direction: The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header. For outbound packet direction: The beginning of the transport header.
FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4_DISCARD FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6_DISCARD	For inbound packet direction: The beginning of the data. **Note** For inbound packets received on the ICMP socket of the TCP/IP stack, the offset is the beginning of the ICMP header. For outbound packet direction: The beginning of the transport header.
FWPS_LAYER_IPSEC_KM_DEMUX_V4 FWPS_LAYER_IPSEC_KM_DEMUX_V6	Not applicable.
FWPS_LAYER_IPSEC_V4 FWPS_LAYER_IPSEC_V6	Not applicable.
FWPS_LAYER_IKEEXT_V4 FWPS_LAYER_IKEEXT_V6	Not applicable.
FWPS_LAYER_RPC_UM	Not applicable.
FWPS_LAYER_RPC_EPMAP	Not applicable.

FWPS_LAYER_RPC_EP_ADD	Not applicable.
FWPS_LAYER_RPC_PROXY_CONN	Not applicable.
FWPS_LAYER_RPC_PROXY_IF	Not applicable.

Run-time filtering layer identifier (starting with Windows 7)	Position in the packet data
FWPS_LAYER_NAME_RESOLUTION_CACHE_V4 FWPS_LAYER_NAME_RESOLUTION_CACHE_V6	Not applicable.
FWPS_LAYER_ALE_RESOURCE_RELEASE_V4 FWPS_LAYER_ALE_RESOURCE_RELEASE_V6	Not applicable.
FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4 FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6	Not applicable.
FWPS_LAYER_ALE_CONNECT_REDIRECT_V4 FWPS_LAYER_ALE_CONNECT_REDIRECT_V6	Not applicable. **Note** For these filtering layers, the *layerData* parameter contains a pointer to an **FWPS_CONNECT_REQUEST0** structure. This structure does not reference a **NET_BUFFER_LIST** structure that describes packet data.
FWPS_LAYER_ALE_BIND_REDIRECT_V4 FWPS_LAYER_ALE_BIND_REDIRECT_V6	Not applicable. **Note** For these filtering layers, the *layerData* parameter contains a pointer to an **FWPS_BIND_REQUEST0** structure. This structure does not reference a **NET_BUFFER_LIST** structure that describes packet data.

FWPS_LAYER_STREAM_PACKET_V4 FWPS_LAYER_STREAM_PACKET_V6	For inbound packet direction: The beginning of the data. For outbound packet direction: The beginning of the transport header.
FWPS_LAYER_KM_AUTHORIZATION	Not applicable.
Run-time filtering layer identifier (starting with Windows 8)	**Position in the packet data**
FWPS_LAYER_INBOUND_MAC_FRAME_ETHERNET	The beginning of the IP header.
FWPS_LAYER_OUTBOUND_MAC_FRAME_ETHERNET	The beginning of the MAC header.
FWPS_LAYER_INBOUND_MAC_FRAME_NATIVE	The beginning of the MAC header.
FWPS_LAYER_OUTBOUND_MAC_FRAME_NATIVE	The beginning of the MAC header.
FWPS_LAYER_INGRESS_VSWITCH_ETHERNET	The beginning of the ethernet header.
FWPS_LAYER_EGRESS_VSWITCH_ETHERNET	The beginning of the ethernet header.
FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V4	The beginning of the IP header.
FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V6	The beginning of the IP header.
FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V4	The beginning of the IP header.
FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V6	The beginning of the IP header.

General discard reasons

Article • 12/15/2021

The identifiers for the possible reasons that the data is discarded by the filter engine are as follows. These identifiers are constant values in the FWPS_GENERAL_DISCARD_REASON enumeration that is defined in Fwpstypes.h.

Discard reason identifier	Discard reason description
FWPS_DISCARD_FIREWALL_POLICY	An FWP_ACTION_BLOCK action was returned from a filtering decision.
FWPS_DISCARD_IPSEC	Reserved.

Network layer discard reasons

Article • 12/15/2021

The identifiers for the possible reasons that data is discarded by one of the network layers are as follows. These identifiers are constant values in the IP_DISCARD_REASON enumeration that is defined in Fwpsk.h.

Discard reason identifier	Discard reason description
IpDiscardBadSourceAddress	The outgoing packet's source address is a multicast address, a broadcast address, or an IPv6 address that contains an embedded IPv4 loopback or unspecified address.
IpDiscardNotLocallyDestined	The received packet's destination address does not exist on the system, and no appropriate forwarding interface exists.
IpDiscardProtocolUnreachable	There is either no transport protocol handler for the received packet or the transport protocol handler refused to process the packet.
IpDiscardPortUnreachable	There is no application that is receiving packets on the received packet's destination port.
IpDiscardBadLength	A length field specified within the received packet is inconsistent with the packet's length.
IpDiscardMalformedHeader	The received packet contains a recognized extension header or option whose contents are invalid.
IpDiscardNoRoute	The received packet cannot be forwarded to its destination address because the system's routing table does not contain a route to that destination.
IpDiscardBeyondScope	The received packet cannot be forwarded because the packet's incoming and outgoing network interfaces have different zone indexes for the packet's zone level.
IpDiscardInspectionDrop	Reserved for internal use by the network stack.
IpDiscardTooManyDecapsulations	The received packet cannot be forwarded to its destination address because there are too many decapsulations.
IpDiscardAdministrativelyProhibited	Reserved for internal use by the network stack.
IpDiscardHopLimitExceeded	The received packet's hop limit or time-to-live limit has been exceeded.

Discard reason identifier	Discard reason description
IpDiscardAddressUnreachable	The outgoing packet cannot be sent to the packet's destination address either because the destination does not exist or packets are not allowed to be sent to that destination.
IpDiscardRscPacket	The outgoing packet cannot be sent because it is a receive-side coalesced (RSC) packet.
IpDiscardArbitrationUnhandled	Reserved for internal use by the network stack.
IpDiscardInspectionAbsorb	The outgoing packet cannot be sent because WFP took ownership of the packet.
IpDiscardDontFragmentMtuExceeded	Reserved for internal use by the network stack.
IpDiscardBufferLengthExceeded	Reserved for internal use by the network stack.
IpDiscardAddressResolutionTimeout	Reserved for internal use by the network stack.
IpDiscardAddressResolutionFailure	Reserved for internal use by the network stack.
IpDiscardIpsecFailure	Reserved for internal use by the network stack.
IpDiscardExtensionHeadersFailure	Reserved for internal use by the network stack.
IpDiscardAllocationFailure	Reserved for internal use by the network stack.

Transport layer discard reasons

Article • 12/15/2021

The identifiers for the possible reasons that data is discarded by one of the transport layers are as follows. These identifiers are constant values in the INET_DISCARD_REASON enumeration that is defined in Fwpsk.h.

Discard reason identifier	Discard reason description
InetDiscardSourceUnspecified	The outgoing packet's source address is unspecified.
InetDiscardDestinationMulticast	The outgoing packet's destination address is an unspecified address, and the transport does not support multicast addresses.
InetDiscardHeaderInvalid	The packet's transport protocol header is invalid.
InetDiscardChecksumInvalid	The checksum in the packet's transport protocol header is invalid.
InetDiscardEndpointNotFound	The endpoint specified in the packet's header could not be found.
InetDiscardConnectedPath	The packet remote address does not match the remote address of a connected endpoint.
InetDiscardSessionState	The packet cannot be delivered based on network layer information.
InetDiscardReceiveInspection	The connection was closed due to a receive inspection failure.
InetDiscardAckInvalid	The packet is an invalid ACK segment.
InetDiscardExpectedSyn	A SYN segment was expected.
InetDiscardRst	The packet is an invalid RST segment.
InetDiscardSynRcvdSyn	A TCP connection in SYN_RCVD state received another SYN segment.
InetDiscardSimultaneousConnect	A TCP connection has encountered the simultaneous-connect condition.
InetDiscardPawsFailed	A TCP PAWS check failed.

Discard reason identifier	Discard reason description
InetDiscardLandAttack	The packet was detected as part of a Land Attack.
InetDiscardMissedReset	An SYN segment outside the receive window was received on a SYN_RCVD connection. An RST may have been missed.
InetDiscardOutsideWindow	A TCP segment was outside the receive window.
InetDiscardDuplicateSegment	A duplicate TCP segment was received.
InetDiscardClosedWindow	The TCP receive window was closed.
InetDiscardTcbRemoved	The TCP connection was closed.
InetDiscardFinWait2	The TCP connection is closing.
InetDiscardReassemblyConflict	A TCP data reassembly conflict was encountered on reception of a FIN segment.
InetDiscardFinReceived	A FIN was already received on a TCP connection; no more data can be received.
InetDiscardListenerInvalidFlags	A segment with invalid flags was received by a listening TCP socket.
InetDiscardUrgentDeliveryAllocationFailure	There is insufficient memory for URG delivery on a TCP connection.
InetDiscardTcbNotInTcbTable	A TCP connection was closed due to urgent delivery.
InetDiscardTimeWaitTcbReceivedRstOutsideWindow	A TIME_WAIT state TCP connection received an RSP segment outside the window.
InetDiscardTimeWaitTcbSynAndOtherFlags	A TIME_WAIT state TCP connection received a segment with SYN and one or more incompatible flags.
InetDiscardTimeWaitTcb	A TIME_WAIT state TCP connection received an invalid segment.

WFP user mode management functions

Article • 12/15/2021

The semantics of the Windows Filtering Platform user-mode management functions are exactly the same when called from a callout driver as when called from a user-mode application, except that the return type is an **NTSTATUS** code instead of a Win32 error code.

These functions are documented in the Management Functions section of the user-mode WFP Functions documentation.

ⓘ Note

The kernel-mode version of each function is defined in fwpmk.h. The user-mode version of each function is defined in fwpmu.h.

Callers of all of these functions except FwpmFreeMemory0 must be running at IRQL = PASSIVE_LEVEL. Callers of **FwpmFreeMemory0** must be running at IRQL <= DISPATCH_LEVEL.

Callout Management

- FwpmCalloutAdd0
- FwpmCalloutCreateEnumHandle0
- FwpmCalloutDeleteById0
- FwpmCalloutDeleteByKey0
- FwpmCalloutDestroyEnumHandle0
- FwpmCalloutEnum0
- FwpmCalloutGetById0
- FwpmCalloutGetByKey0
- FwpmCalloutGetSecurityInfoByKey0
- FwpmCalloutSetSecurityInfoByKey0

Connection Object Management

- FwpmConnectionCreateEnumHandle0
- FwpmConnectionDestroyEnumHandle0
- FwpmConnectionEnum0
- FwpmConnectionGetById0

- FwpmConnectionGetSecurityInfo0
- FwpmConnectionSetSecurityInfo0

Event Management

- FwpmNetEventCreateEnumHandle0
- FwpmNetEventDestroyEnumHandle0
- FwpmNetEventEnum:
  - FwpmNetEventEnum0 (Windows Vista)
  - FwpmNetEventEnum1 (Windows 7)
  - FwpmNetEventEnum2 (Windows 8)
- FwpmNetEventsGetSecurityInfo0
- FwpmNetEventsSetSecurityInfo0

Filter Management

- FwpmFilterAdd0
- FwpmFilterCreateEnumHandle0
- FwpmFilterDeleteById0
- FwpmFilterDeleteByKey0
- FwpmFilterDestroyEnumHandle0
- FwpmFilterEnum0
- FwpmFilterGetById0
- FwpmFilterGetByKey0
- FwpmFilterGetSecurityInfoByKey0
- FwpmFilterSetSecurityInfoByKey0

Layer Management

- FwpmLayerCreateEnumHandle0
- FwpmLayerDestroyEnumHandle0
- FwpmLayerEnum0
- FwpmLayerGetById0
- FwpmLayerGetByKey0
- FwpmLayerGetSecurityInfoByKey0
- FwpmLayerSetSecurityInfoByKey0

Provider Context Management

- [FwpmProviderContextAdd:
  - FwpmProviderContextAdd0 (Windows Vista)
  - FwpmProviderContextAdd1 (Windows 7)
  - FwpmProviderContextAdd2 (Windows 8)
- FwpmProviderContextCreateEnumHandle0
- FwpmProviderContextDeleteById0
- FwpmProviderContextDeleteByKey0
- FwpmProviderContextDestroyEnumHandle0
- FwpmProviderContextEnum:
  - FwpmProviderContextEnum0 (Windows Vista)
  - FwpmProviderContextEnum1 (Windows 7)
  - FwpmProviderContextEnum2 (Windows 8)
- FwpmProviderContextGetById:
  - FwpmProviderContextGetById0 (Windows Vista)
  - FwpmProviderContextGetById1 (Windows 7)
  - FwpmProviderContextGetById2 (Windows 8)
- FwpmProviderContextGetByKey:
  - FwpmProviderContextGetByKey0 (Windows Vista)
  - FwpmProviderContextGetByKey1 (Windows 7)
  - FwpmProviderContextGetByKey2 (Windows 8)
- FwpmProviderContextGetSecurityInfoByKey0
- FwpmProviderContextSetSecurityInfoByKey0

Provider Management

- FwpmProviderAdd0
- FwpmProviderCreateEnumHandle0
- FwpmProviderDeleteByKey0
- FwpmProviderDestroyEnumHandle0
- FwpmProviderEnum0
- FwpmProviderGetByKey0
- FwpmProviderGetSecurityInfoByKey0
- FwpmProviderSetSecurityInfoByKey0

Session Management

- FwpmEngineClose0
- FwpmEngineGetOption0
- FwpmEngineGetSecurityInfo0
- FwpmEngineOpen0

- FwpmEngineSetOption0
- FwpmEngineSetSecurityInfo0
- FwpmSessionCreateEnumHandle0
- FwpmSessionDestroyEnumHandle0
- FwpmSessionEnum0

Sublayer Management

- FwpmSubLayerAdd0
- FwpmSubLayerCreateEnumHandle0
- FwpmSubLayerDeleteByKey0
- FwpmSubLayerDestroyEnumHandle0
- FwpmSubLayerEnum0
- FwpmSubLayerGetByKey0
- FwpmSubLayerGetSecurityInfoByKey0
- FwpmSubLayerSetSecurityInfoByKey0

Transaction Management

- FwpmTransactionAbort0
- FwpmTransactionBegin0
- FwpmTransactionCommit0

vSwitch Management

- FwpmvSwitchEventsGetSecurityInfo0
- FwpmvSwitchEventsSetSecurityInfo0

Internet Key Exchange functions

Article • 12/06/2022

The semantics of the following functions are exactly the same when called from a callout driver as when called from a user-mode application except that the return type is an NTSTATUS code instead of a Win32 error code. For a description of each of these functions, see the Windows Filtering Platform.

Callers of these functions must be running at IRQL = PASSIVE_LEVEL.

- IkeextGetStatistics0
- IkeextSaCreateEnumHandle0
- IkeextSaDbGetSecurityInfo0
- IkeextSaDbSetSecurityInfo0
- IkeextSaDeleteById0
- IkeextSaDestroyEnumHandle0
- IkeextSaEnum0
- IkeextSaGetById0

IPsec functions

Article • 12/06/2022

The semantics of the following functions are exactly the same when called from a callout driver as when called from a user-mode application except that the return type is an NTSTATUS code instead of a Win32 error code. For a description of each of these functions, see the Windows Filtering Platform.

Callers of these functions must be running at IRQL = PASSIVE_LEVEL.

- FwpmIPsecTunnelAdd0
- FwpmIPsecTunnelDeleteByKey0
- IPsecGetStatistics0
- IPsecSaContextAddInbound0
- IPsecSaContextAddOutbound0
- IPsecSaContextCreate0
- IPsecSaContextCreateEnumHandle0
- IPsecSaContextDeleteById0
- IPsecSaContextDestroyEnumHandle0
- IPsecSaContextEnum0
- IPsecSaContextExpire0
- IPsecSaContextGetById0
- IPsecSaContextGetSpi0
- IPsecSaCreateEnumHandle0
- IPsecSaDbGetSecurityInfo0
- IPsecSaDbSetSecurityInfo0
- IPsecSaDestroyEnumHandle0
- IPsecSaEnum0

System Area Networks Overview

Article • 12/15/2021

A *system area network* (SAN) is a group of devices that are linked by a high-speed, high-performance connection. A SAN connection uses Internet Protocol (IP) addresses, which are assigned by TCP/IP to each SAN network interface controller (NIC), to determine data routing. It also uses a reliable transport, which is built into the SAN, to perform data delivery. A SAN can include, for example, clusters of client and server computers.

This section includes:

Supporting System Area Networks

Windows Sockets Direct

Supporting System Area Networks Topics

Article • 12/15/2021

This section describes the performance benefits that client and server computers can achieve by interfacing with a SAN, as well as the software components that you must implement and supply. These matters are discussed in the following topics:

Introduction to System Area Networks

Using a SAN with Windows Sockets Applications

Creating Components for Using a SAN

Virtual Interface Architecture and Support for SAN

Introduction to System Area Networks

Article • 01/29/2022

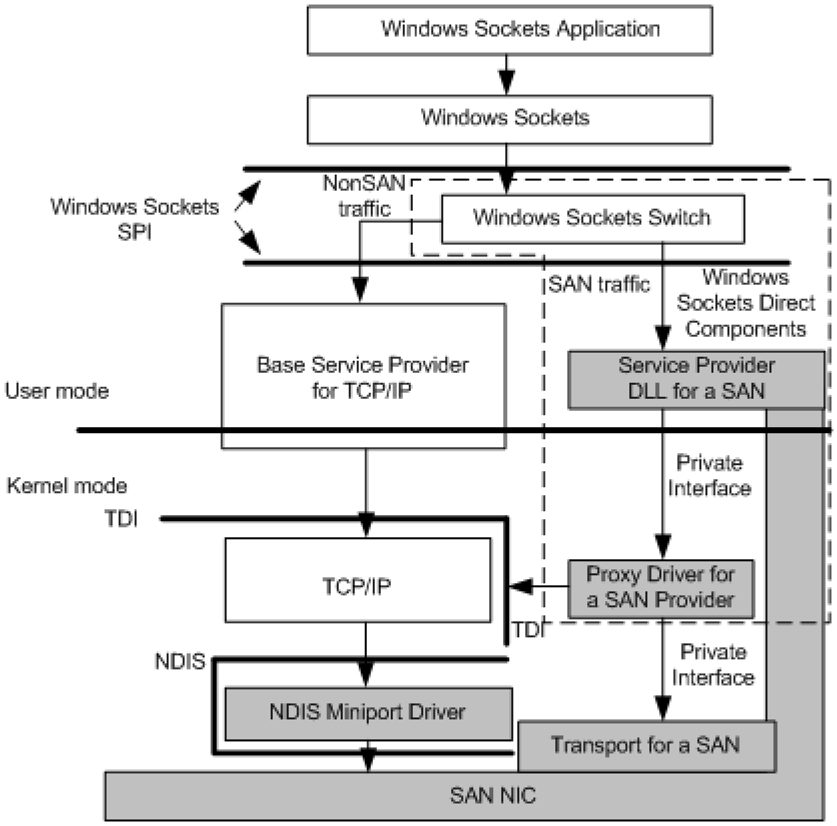
A *system area network (SAN)* is a high-performance, connection-oriented network that can link a cluster of computers. A SAN delivers high bandwidth (1 Gbps or greater) with low latency. A SAN is typically switched by hubs that support eight or more nodes. The cable lengths between nodes on a SAN range from a few meters to a few kilometers.

Unlike existing network technologies such as Ethernet and ATM, a SAN offers a reliable transport service; that is, a SAN guarantees to deliver uncorrupted data in the same order in which it was sent. Connection endpoints in a SAN are not required to use the TCP/IP protocol stack to transfer data unless traffic must be routed between subnets. SAN-local communication can use a native SAN transport, bypassing the TCP/IP protocol stack.

A SAN network interface controller (NIC), a transport driver for the SAN NIC, or a combination of both exposes a private transport interface. However, because most networking applications are written to use TCP/IP through Windows Sockets, they cannot use a SAN directly. The Windows Sockets Direct components shown in the following figure let these applications benefit from using a SAN transparently without requiring modification. Windows Sockets Direct is part of:

- Microsoft Windows 2000 Datacenter Server
- Microsoft Windows 2000 Advanced Server SP2
- Microsoft Windows 2000 Server Appliance Kit SP2
- Microsoft Windows Server 2003

The following figure shows the architecture required to support a SAN. The shaded areas represent components that a SAN NIC vendor supplies to enable using a SAN.



The following is a description of the components shown in this figure.

Windows Sockets application

Application that interfaces with Windows Sockets for network services.

Windows Sockets

The Windows Sockets interface (Ws2_32.dll).

Windows Sockets SPI

The Windows Sockets service provider interface (SPI).

Windows Sockets switch

The Windows Sockets switches between use of the standard TCP/IP service provider and particular SAN service providers.

TCP/IP service provider

A user-mode DLL and associated kernel-mode proxy driver that comprise the standard base Windows Sockets service provider for TCP/IP. The proxy driver exposes a TDI interface.

TCP/IP

The standard TCP/IP protocol driver.

SAN service provider

The user-mode DLL portion of the SAN service provider.

Proxy driver for a SAN service provider
The kernel-mode proxy driver of the SAN service provider.

NDIS miniport driver
The NDIS miniport driver that supports communication to the SAN NIC using the standard TCP/IP protocol driver.

SAN transport
A reliable transport service, which can be fully implemented in the NIC, fully implemented in software, or implemented in a combination of both hardware and software.

SAN NIC
The physical SAN network interface controller (NIC).

A kernel-mode provider for a particular SAN. (Reserved for future use.)

Supporting System Area Networks

Using a SAN with Windows Sockets Applications

Creating Components for Using a SAN

Virtual Interface Architecture and Support for SAN

Using a SAN with Windows Sockets Applications

Article • 12/15/2021

Windows Sockets applications can benefit from using a system area network (SAN). These applications can use a SAN to transfer data in bulk form and to drop data directly onto the SAN network, without copying across the user-kernel boundary, using a technology called Windows Sockets Direct. Windows Sockets Direct lets these applications use a SAN transparently.

For each Windows Sockets application, Windows Sockets Direct can either:

- Route data traffic that flows over a SAN directly to the SAN.

  The system-supplied Windows Sockets switch component of Windows Sockets Direct routes data traffic for a SAN that originates from a Windows Sockets application directly to the SAN NIC to be transferred over the SAN network. The switch uses that SAN's particular Windows Sockets service provider to transfer data.

- Route data traffic that flows over other networks through TCP/IP.

  To route data traffic that is not for a specific SAN from a Windows Sockets application, the switch must use the TCP/IP service provider. Non-SAN-specific data traffic includes, for example, datagrams, multicast, and connections that must be routed. Non-SAN-specific data traffic is then routed through TCP/IP and the NDIS miniport driver to the SAN NIC.

Creating Components for Using a SAN

Article • 12/15/2021

Windows Sockets applications can benefit from using a system area network (SAN). To use a SAN, these applications must have a SAN service provider DLL and a proxy driver for that DLL.

To use a specific SAN to transfer data, you also need a reliable transport for that SAN. If a reliable transport is not fully implemented in the SAN NIC hardware, you need a transport driver for the SAN. If required, a SAN transport driver is specified by the SAN NIC vendor and communicates with its overlying SAN proxy driver and underlying SAN NIC through private interfaces.

For information about implementing a SAN service provider DLL and its proxy driver, see Windows Sockets Direct. Note, however, that this section does not specify how to write a SAN transport driver.

You need an NDIS miniport driver to transfer data that must flow over networks other than your specific SAN such as Ethernet, ATM, or another SAN. TCP/IP uses the NDIS miniport driver to send data both to the SAN NIC and over such networks.

For information about implementing miniport and transport drivers, see *Miniport Drivers* and TDI Transports and Their Clients.

Virtual Interface Architecture and Support for SAN

Article • 12/15/2021

The Virtual Interface (VI) architecture, proposed by Compaq, Intel, and Microsoft, is a design for an interface between a SAN NIC and a host computer system. This architecture represents only one aspect of design with regard to system area networks (SAN). There are alternate designs that share the same fundamental characteristics.

The VI architecture defines a set of capabilities and characteristics for SAN interconnects. For example, the VI architecture includes support for remote direct memory access (RDMA) operations. The VI architecture also describes specific mechanisms for interacting with a SAN NIC to manage endpoints and connections, and to process data transfer requests.

The Windows Sockets switch works with a broader class of SAN interconnects beyond those that use the VI architecture. SAN extensions to the Windows Sockets service provider interface (SPI) shield the switch from the hardware interface for a particular SAN NIC. That hardware interface is encapsulated within a SAN service provider DLL and its kernel-mode proxy driver. These components are supplied by a SAN vendor. For more information about SAN extensions, see Windows Sockets SPI Extensions for SANs.

Windows Sockets Direct Overview

Article • 12/15/2021

Microsoft Windows Sockets Direct is a technology that, when possible, routes data at high speed and with high performance to and from Windows Sockets applications through a system area network (SAN) instead of the TCP/IP transport and an NDIS miniport driver.

Windows Sockets Direct is included in:

- Microsoft Windows 2000 Datacenter Server
- Microsoft Windows 2000 Advanced Server SP2
- Microsoft Windows 2000 Server Appliance Kit SP2
- Microsoft Windows Server 2003

The following topics describe Windows Sockets Direct features and operations, the software components that enable Windows Sockets Direct, and the requirements for writing a SAN service provider and its associated proxy driver:

Windows Sockets Direct Architecture

Windows Sockets Direct Component Operation

Creating a Service Provider for a SAN

Creating a Proxy Driver for a SAN Service Provider

Windows Sockets Direct Architecture

Article • 12/15/2021

Windows Sockets Direct provides a high-speed, high-performance connection between two network nodes on the same system area network (SAN) by mapping a SAN transport interface directly into an application process. This SAN connection enables user-mode processes to perform direct input and output (I/O) without copying across the user-kernel boundary.

The SAN architecture figure in Introduction to System Area Networks shows how Windows Sockets Direct provides a SAN connection. The shaded areas in the figure represent components that a SAN NIC vendor must supply to enable use of a SAN.

The following paragraphs describe the components that appear in the figure.

Supplied Components for SAN Network Interface Controllers

Each SAN network interface controller (NIC) uses the following software components to provide support for NDIS and for Windows Sockets Direct.

- An NDIS miniport driver for a SAN NIC provides support for NDIS so that it can communicate with Windows Sockets applications using a standard TCP/IP protocol driver. This NDIS miniport driver supports standard media types such as Ethernet or ATM.

- The SAN service provider DLL and its associated proxy driver provide support for Windows Sockets Direct. These Windows Sockets Direct components export the native transport semantics of an interconnect for the SAN to Windows Sockets applications. These semantics can include, for example, address family and message orientation.

The SAN NIC vendor supplies the NDIS miniport driver and Windows Sockets Direct components. The SAN NIC vendor might also supply a SAN transport driver if transport service is not implemented fully in the NIC. The proxy driver for a SAN service provider DLL and possibly a SAN transport driver are contained either in the NDIS miniport driver or in separate drivers, at the discretion of the SAN NIC vendor.

Windows Sockets Switch Components

The Windows Sockets switch is an operating system-supplied component of Windows Sockets Direct. The switch is a Windows Sockets service provider that is layered on top of TCP/IP and SAN service providers. The Windows operating system inserts the switch between the Windows Sockets interface and the other service providers. For clarity, the switch appears in the figure as a separate entity. However, the switch and the base TCP/IP service provider are actually implemented in the same DLL. The switch performs the following actions:

- Makes the installed collection of SAN service providers and the standard TCP/IP provider look like a single provider to Windows Sockets applications.
- Chooses, on a per-connection basis, whether to use a native SAN service provider or the standard TCP/IP provider to service an application socket.
- Emulates TCP/IP semantics when using a native SAN service provider.

The top and bottom interfaces of the switch conform to the Windows Sockets Service Provider Interface (SPI). The switch's bottom interface uses extensions to the Windows Sockets SPI to take advantage of a SAN's capabilities. Those extensions are described in Windows Sockets SPI Extensions for SANs and fully documented in the Windows Sockets Direct Reference.

The switch manages application access to all networks. A computer can contain multiple SAN NICs from multiple vendors, as well as one or more LAN and WAN NICs, such as a LAN NIC that supports an Ethernet network. The switch manages application access to all networks associated with these NICs transparently.

TCP/IP Functions

As with any NIC exposed through NDIS, the TCP/IP protocol driver assigns one or more IP addresses to each SAN NIC. The Windows Sockets switch and SAN service providers determine these assignments, as described in Receiving and Translating NIC Addresses. The switch uses this IP address information to determine which SAN service provider to use for a given socket connection. SAN service providers use this IP address information to translate IP addresses into native SAN addresses.

The switch works closely with the standard base TCP/IP service provider to obtain functionality that SAN service providers do not support. The TCP/IP service provider supports listening for connections on multiple providers and synchronization across multiple providers.

The TCP/IP service provider also handles all communication over standard LAN and WAN interconnects, raw IP sockets, all UDP sockets, and connections between subnets.

Windows Sockets Direct Component Operation Topics

Article • 12/15/2021

This sections describes Windows Sockets Direct operations, including the operation of the Windows Sockets switch, a SAN service provider, and the SAN service provider's associated proxy driver.

This section includes the following topics:

Installing Windows Sockets Direct Components

Article • 12/15/2021

The following topics describe how Windows Sockets Direct components (that is, the Windows Sockets switch and individual SAN service providers) are installed:

Installing the Windows Sockets Switch

Installing a SAN Service Provider

Installing the Windows Sockets Switch

Article • 12/15/2021

Microsoft Windows installs the Windows Sockets switch as a *layered service provider*, with the Windows Sockets service provider interface (SPI) as both the top and bottom interface. The switch exports the protocol characteristics of TCP/IP just like the TCP/IP service provider. The switch is the first visible TCP/IP provider, which makes the switch the default choice for applications that open sockets for the WSK address families.

Installing a SAN Service Provider

Article • 12/06/2022

A SAN service provider is typically installed as a base Windows Sockets service provider that interfaces with the Windows Sockets switch. Although a SAN service provider can be installed for direct use by an application instead, the Windows Sockets Direct technology does not support using a SAN service provider in this manner. A SAN service provider that is installed for direct use by an application exports its native address family and protocol characteristics rather than those of TCP/IP protocol.

A SAN service provider that is indirectly exposed to applications through the Windows Sockets switch must set the PFL_HIDDEN flag in the **dwProviderFlags** member of the SAN service provider's WSAPROTOCOL_INFOW structure. To install the SAN service provider on the operating system, the SAN service provider's installation mechanism passes this structure in a call to the WSCInstallProvider function. The SAN service provider's installation mechanism can be for example, a setup program or a function exported by the SAN service provider and called by a INF file directive.

The SAN service provider's installation mechanism must add a value of type REG_BINARY to the following key in the registry before the SAN service provider can be detected by the Windows Sockets switch as a base Windows Sockets service provider:

Console

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Winsock\
Parameters\TCP on SAN
```

This value contains the binary representation of the value in the **ProviderId** member from the WSAPROTOCOL_INFOW structure. This value registers a SAN service provider with the Windows Sockets switch. This member contains the globally unique identifier (GUID) that the vendor assigned to the SAN service provider.

The vendor can also assign a unique name that represents this GUID, for example:

- Trademarked name of the product
- Unique numeric value
- Textual representation of the GUID

To register a SAN service provider

1. The switch calls the **WSAProviderConfigChange** function to detect Windows Sockets provider installation and removal events.

2. After a new Windows Sockets service provider is installed, the switch calls the **WSCEnumProtocols** function to query the Windows Sockets catalog and the list of SAN service providers in the registry to determine whether the new service provider controls a SAN. For more information about **WSCEnumProtocols**, see the Windows SDK.

3. If the switch detects a new SAN service provider, the switch initializes that service provider as described in Initializing a SAN Service Provider.

4. The switch also calls the following functions of the newly installed SAN service provider after the SAN service provider is initialized to service any existing listening sockets bound to the wildcard IP address (0.0.0.0) (the wildcard IP address implies that the SAN service provider should accept incoming connection requests from all NICs it controls):

   WSPSocket
   Creates a socket

   WSPBind
   Binds the socket to the wildcard IP address

   WSPListen
   Sets the socket to acknowledge and queue incoming connection requests until accepted by the switch

   Note Beginning with Windows Vista, the wildcard IP address 0.0.0.0 is not available. Also beginning with Windows Vista, if the **IPAutoconfigurationEnabled** registry key is set to a value of 0, automatic IP address assignment is disabled, and no IP address is assigned. In this case, the **ipconfig** command line tool will not display an IP address. If the key is set to a nonzero value, an IP address is automatically assigned. This key can be located at the following paths in the registry:

   HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Tcpip\Parameters\IPAutoconfigurationEnabled

   HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Tcpip\Parameters\Interfaces\*GUID*\IPAutoconfigurationEnabled

Initializing the Use of a SAN

Article • 12/15/2021

The following topics describe how to initialize the use of a SAN:

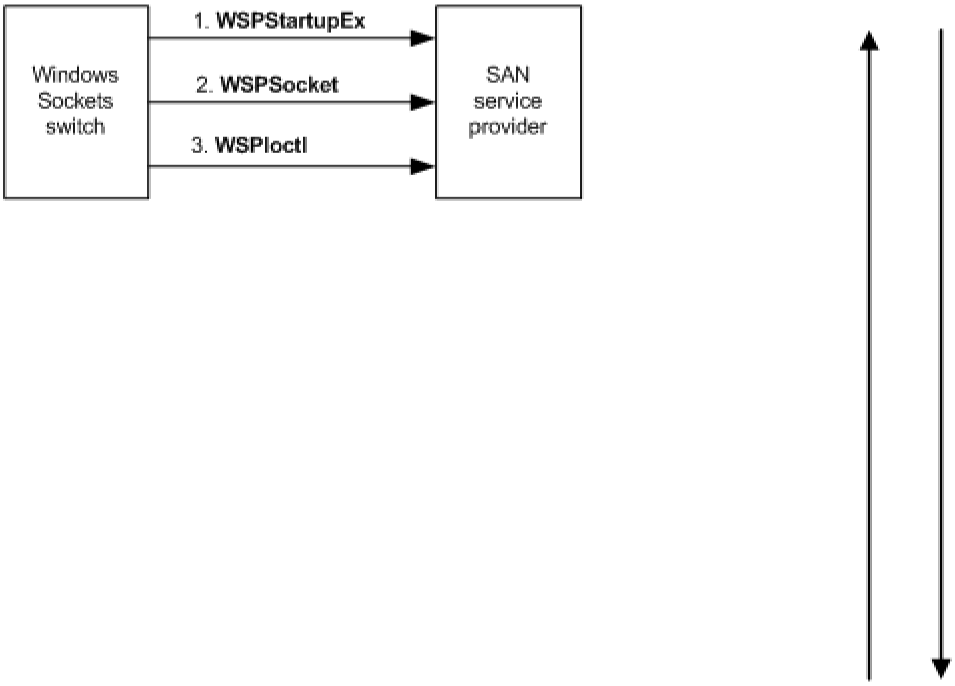
Initializing a SAN Service Provider

Receiving and Translating NIC Addresses

Initializing a SAN Service Provider

Article • 12/06/2022

The Windows Sockets switch initializes a SAN service provider as described in the following figure.



After Windows loads the Windows Sockets switch DLL into an application's process, the following sequence of events occur.

To initialize a SAN service provider

1. The switch detects and loads the TCP/IP provider and then queries the list of SAN service providers in the registry to detect all of those providers, as described in Installing a SAN Service Provider. The switch calls each detected provider's **WSPStartupEx** function to initiate use of that provider.

2. In the **WSPStartupEx** call, the switch passes a pointer to a **WSAPROTOCOL_INFOW** structure that contains the TCP/IP provider's protocol information. The TCP/IP provider's protocol indicates to the SAN service provider that it has been initialized by the switch rather than by other layered service providers or the Windows Sockets interface. The switch passes the TCP/IP provider's protocol information instead of the SAN service provider's transport information, as suggested in the Windows Sockets service provider Interface (SPI) section of the Microsoft Windows SDK documentation.

   Because a SAN service provider can detect that it is initialized by the switch, it can expose the appropriate set of entry-point functions to the switch. If the SAN service provider is initialized directly by an application, it can expose another set of entry-point functions to that application. If a SAN service provider is layered under the switch, that provider must adhere to the extensions and behavior described in this section.

3. A SAN service provider's proxy driver obtains the list of IP addresses assigned to each NIC under its control as described in Registering for SAN NIC Notifications. The SAN service provider uses a private interface to retrieve this list from its proxy driver. The switch calls a SAN service provider's **WSPSocket** function to create a socket. The switch uses this socket to retrieve the complete list of IP addresses assigned to the NICs under control of the SAN service provider's proxy driver. The switch retrieves this list as described in Receiving and Translating NIC Addresses. Based on this list and the lists of other SAN service providers, the switch builds a table that maps local IP subnets to SAN service providers.

4. The Windows Sockets switch must retrieve pointers to the SAN service provider's entry-point functions that extend Windows Sockets service provider Interface (SPI) for use with SANs. To retrieve each of these extended functions, the Windows Sockets switch calls a SAN service provider's **WSPIoctl** function and passes the SIO_GET_EXTENSION_FUNCTION_POINTER command code along with the GUID whose value identifies one of these extended functions.

   For a complete description of these functions, see Windows Sockets SPI Extensions for SANs.

5. The switch can create threads to support listening sockets as well as nonblocking connect requests, as described in Setting Up a SAN Connection.

Receiving and Translating NIC Addresses

Article • 12/15/2021

The Windows Sockets switch always uses the WSK address families, which contain IP addresses, when it interacts with SAN service providers and SAN NICs. The switch does not use a SAN's native address family. Therefore, a SAN service provider must use its associated proxy driver to retrieve the list of IP addresses assigned to its NICs. The SAN service provider uses these IP addresses when interacting with its proxy driver. The proxy driver must translate between IP addresses and native addresses.

During initialization, a proxy driver typically registers with Transport Driver Interface (TDI) for address change notifications. All Plug and Play (PnP) aware transports, including TCP/IP, supply address change notifications through TDI to clients that have registered for such notifications.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

Registering for Address Change Notification

During initialization, a proxy driver calls the **TdiRegisterPnPHandlers** function to register for address change notifications. In this call, the proxy driver passes pointers to callback functions for address additions and deletions in the **AddAddressHandlerV2** and **DelAddressHandlerV2** members of the TDI_CLIENT_INTERFACE_INFO structure. After the proxy driver registers to receive these notifications, TDI promptly indicates all currently active network addresses using the add-address callback.

TDI passes the following parameters to a proxy driver's add-address or delete-address callback functions:

Address
Pointer to a TA_ADDRESS structure that describes the network address assigned to or removed from the NIC. In the case of TCP/IP, this pointer is actually a pointer to a TA_ADDRESS_IP structure.

DeviceName
Pointer to a Unicode string that identifies the transport-to-NIC binding with which the address is associated. In case of TCP/IP, the Unicode string has the following format:

\Device\Tcpip_{NIC-GUID}

where NIC-GUID is the globally unique identifier assigned by the network configuration subsystem to the NIC.

The preceding structure definitions are defined in the tdi.h header file. The preceding registration and callback functions are defined in the tdikrnl.h header file. These header files are available in the Microsoft Windows Driver Development Kit (DDK) and the Windows Driver Kit (WDK). Detailed information about TDI PnP notifications is included in TDI Client Callbacks and TDI Client Event and PnP Notification Handlers.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

Maintaining a List of IP Addresses

A SAN service provider's proxy driver uses add-address and delete-address notifications to maintain the list of IP addresses assigned to each NIC under its control. The proxy driver uses this list to translate between one or more IP addresses assigned to a SAN NIC by the TCP/IP transport and native SAN addresses. The proxy driver must also supply a device-control routine that makes the list of IP addresses assigned to a NIC available to the Windows Sockets switch whenever the switch makes an SIO_ADDRESS_LIST_QUERY control-code query. The proxy driver's **DriverEntry** routine must specify an entry point for this device-control routine.

The Windows Sockets switch maintains a list of all IP addresses assigned to each SAN NIC. To retrieve IP addresses for this inclusive list, the switch calls each SAN service provider 's **WSPIoctl** function, passing the SIO_ADDRESS_LIST_QUERY control code. Each SAN service provider, in turn, queries its associated proxy driver for its individual list of IP addresses assigned to its SAN NICs. After the switch is notified of an address change, it again queries each SAN service provider for updates in each individual list.

Setting Up a SAN Connection

Article • 12/15/2021

During connection setup, the Windows Sockets switch determines which service provider will service the TCP socket. This provider will handle most subsequent operations on the socket. Regardless of whether the switch chooses a SAN service provider, the TCP/IP provider exclusively handles a few types of setup operations.

This section describes the connection setup operations that a SAN service provider performs and the connection setup operations that the TCP/IP provider handles. This information is provided in the following topics:

Creating and Binding SAN Sockets

Initiating a Connection

Listening for Connections on a SAN

Accepting Connection Requests

Registering Memory for Operations on a SAN

Caching Registered Memory

Creating and Binding SAN Sockets

Article • 12/15/2021

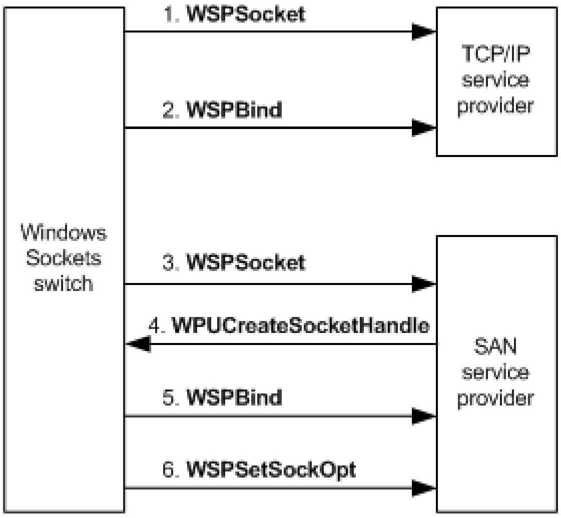
If the Windows Sockets switch determines that it can route data through a SAN connection rather than through the TCP/IP stack, it requests the appropriate SAN service provider to create, bind, and set options for a socket on which the data can be transferred.

The socket created by the SAN service provider is a *companion* to the socket that the TCP/IP service provider created at the request of the application, either from or to which the data is being transferred. The *companion socket* created by the SAN service provider has the same options as the socket created by the TCP/IP service provider, if the SAN service provider supports those options.

The companion socket also has the same IP address and TCP port as the socket that was created by the TCP/IP service provider. The SAN data is transferred through the companion socket created by the SAN service provider rather than the socket created by the TCP/IP service provider. The SAN socket is not visible to the application. From the application's perspective, the data is transferred on the socket that it requested to be created for the data transfer.

Note The switch always uses the TCP/IP service provider to transfer data over *raw sockets*. The switch therefore never requests a SAN service provider to create a raw socket.

The following figure shows an overview of how the Windows Sockets switch creates a companion socket. The sequence in the sections that follow describe creating a companion socket in more detail.



Initiating Creation of a TCP/IP Socket

1. After the Windows Sockets switch receives a **WSPSocket** call that was initiated by an application, the switch calls the TCP/IP provider's **WSPSocket** function to request the TCP/IP provider to create a socket.

2. The Windows Sockets switch returns the descriptor for the created socket to the application and stores this descriptor in a private data structure that is associated with the socket.

   From the application's perspective, the socket created by the TCP/IP provider is the socket used for data transfers, whether the switch uses the TCP/IP service provider or the SAN service provider to transfer the data.

Binding a TCP/IP Socket

1. The switch receives a **WSPBind** call if an application requests to bind the socket to a specific network interface controller (NIC) or to the wildcard IP address (0.0.0.0). A socket bound to the wildcard IP address can listen for incoming connection requests from all NICs.

   Note Beginning with Windows Vista, the wildcard IP address 0.0.0.0 is not available. Also beginning with Windows Vista, if the **IPAutoconfigurationEnabled** registry key is set to a value of 0, automatic IP address assignment is disabled, and no IP address is assigned. In this case, the **ipconfig** command line tool will not display an IP address. If the key is set to a nonzero value, an IP address is automatically assigned. This key can be located at the following paths in the registry:

   HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Tcpip\Parameters\IPAutoconfigurationEnabled

   HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Tcpip\Parameters\Interfaces\*GUID*\IPAutoconfigurationEnabled

2. The switch forwards this call to the TCP/IP service provider by calling the TCP/IP provider's **WSPBind** function.

Service Provider Determination

1. The switch determines whether to use a SAN service provider for data transfer on a socket after the application initiates a **WSPListen** or **WSPConnect** call to the switch, as described in Setting Up a SAN Connection.

2. If the switch determines that it cannot use a SAN service provider for a data transfer, the switch routes the data transfer through the TCP/IP service provider.

3. If the switch chooses a SAN service provider to service an application's socket, the switch calls the SAN service provider's WSPSocket function to create a companion socket.

Initiating Creation of a Companion Socket

1. The SAN service provider's **WSPSocket** function initializes an internal data structure in which it stores information about the companion socket.

2. The SAN service provider's **WSPSocket** function must next call the **WPUCreateSocketHandle** function to acquire a socket descriptor from the switch.

3. The SAN service provider must store the switch's socket descriptor in its internal data structure for the companion socket and must return its own descriptor for the companion socket to complete the **WSPSocket** call. The socket descriptor returned by the SAN service provider can be any meaningful value, such as a pointer to a private data structure.

4. To perform an operation on the socket, the switch supplies the socket descriptor that was returned by the SAN service provider to the appropriate function of the SAN service provider. Similarly, the SAN service provider must supply the socket descriptor that was acquired from the switch in the **WPUCreateSocketHandle** call if the SAN service provider makes any of the following up calls:

   WPUQuerySocketHandleContext

   WPUCloseSocketHandle

   WPUCompleteOverlappedRequest

Binding a Companion Socket

1. If a SAN service provider's **WSPSocket** function completes successfully, the switch immediately calls the SAN service provider's WSPBind function to assign a local IP address and TCP port to the socket.

2. The switch assigns the same IP address and TCP port to the SAN socket as was assigned to the socket that was created by the TCP/IP provider. The SAN service provider must translate this TCP/IP address into its native format.

3. The switch supplies a fully qualified IP address and TCP port (that is, nonzero values) to the SAN service provider's **WSPBind** function unless an application requested to listen for incoming connections from all NICs. In the later case, the switch supplies the wildcard IP address to the SAN service provider's **WSPBind** function.

Setting Options for a Companion Socket

- If the application specified any socket options, the switch stores those options. After creating the SAN socket, the switch calls the SAN service provider's WSPSetSockOpt function for each supported option that was specified by the application to immediately set these options for the SAN socket.

Failing a Companion Socket Call

- If a SAN service provider fails any of the preceding calls to its **WSPSocket**, **WSPBind**, or **WSPSetSockOpt** functions, the switch calls the SAN service provider's WSPCloseSocket function to destroy the SAN socket. The switch then uses the TCP/IP provider to continue servicing the application socket. Note that, after the switch establishes a connection using a SAN service provider, the switch cannot use the TCP/IP provider to service the application's socket. In this case, the switch returns the appropriate error to the application.

Connecting the Companion Socket

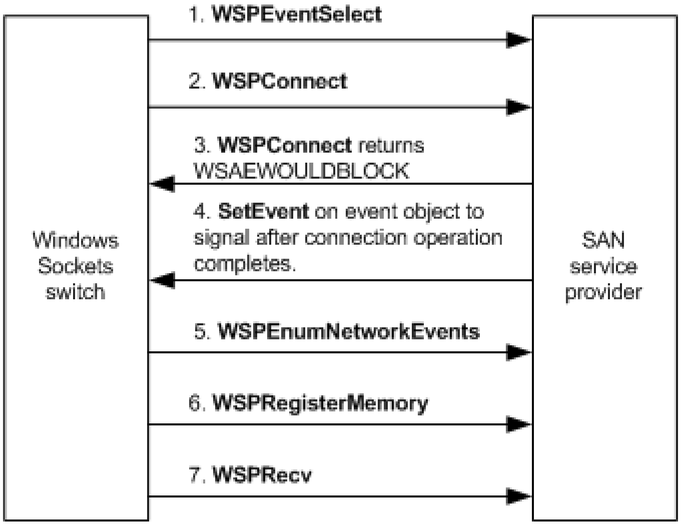
- After the switch sets up the companion socket, the switch calls either the **WSPListen** or **WSPConnect** function for the SAN service provider to perform the operation that caused the SAN service provider to originally set up the socket. For example, if an application originally requested to listen for incoming connections, the switch calls the SAN service provider's WSPListen function.

Initiating a Connection

Article • 12/15/2021

After the Windows Sockets switch receives a **WSPConnect** call that was initiated by an application, the switch compares the destination address of the connect request with addresses in the switch's table of IP subnets that SAN service providers serve. If one of those subnets includes this destination address, the switch calls the **WSPSocket** and **WSPBind** functions of the corresponding SAN service provider to create and bind a socket, as described in Creating and Binding SAN Sockets. The switch processes the application's connect request using the SAN socket. If the destination address of the connect request is not on a SAN subnet, or if the SAN service provider fails to create and bind a socket, the switch uses the TCP/IP provider to establish the connection.

The following figure shows an overview of how the Windows Sockets switch requests a connection with a remote peer. The sequence and sections that follow describe the connection request in more detail.



After creating and binding the SAN socket, the switch executes a connect request, using the SAN socket in *nonblocking mode*, as described in the following procedure.

To execute a connect request

1. The switch calls the SAN service provider's **WSPEventSelect** function. In this call, the switch passes the FD_CONNECT code and the event object to be associated with that code. The call to **WSPEventSelect** requests notification of connection events and informs the SAN service provider that any subsequent **WSPConnect** call executes in nonblocking mode.

2. After the **WSPEventSelect** function returns, the switch calls the SAN service provider's **WSPConnect** function. In this call, the switch passes the destination

address in the format of one of the WSK address families. The SAN service provider's proxy driver maps this destination address to a native address and attempts to establish the connection.

3. If the SAN service provider's **WSPConnect** function can complete or fail the connection operation immediately, it returns the appropriate success or failure code. If the SAN service provider's **WSPConnect** function cannot complete a connection request immediately, the SAN service provider's connection operation proceeds asynchronously in another thread. The SAN service provider's **WSPConnect** function returns with the error WSAEWOULDBLOCK to indicate that the socket is marked as nonblocking and that the connection operation cannot be completed immediately.

4. After the connection operation completes, the SAN service provider calls the Win32 **SetEvent** function to signal the event object that was previously registered in the **WSPEventSelect** call.

5. After the event object is signaled, the switch calls the SAN service provider's **WSPEnumNetworkEvents** function to obtain the result of the connection operation.

Note After the switch establishes a connection through a SAN service provider, the switch can no longer use the TCP/IP provider for that connection. SAN service providers must fully implement all functionality required to service an established connection.

Destroying the SAN Socket

If the SAN service provider's **WSPConnect** function fails, the switch calls the SAN service provider's **WSPCloseSocket** function to destroy the SAN socket. The switch then calls the TCP/IP service provider's **WSPConnect** function to forward the connection operation to the TCP/IP service provider unless the SAN service provider returned one of the following error codes as the result of its connection operation:

WSAECONNRESET
Indicates that no application is listening on the specified port at the destination address

WSAECONNREFUSED
Indicates that the remote application actively refused the connection request

WSAEHOSTUNREACH
Indicates that the destination address does not exist

These preceding error codes guarantee that an attempt to establish the connection through TCP/IP will also fail. A SAN service provider must not return one of these error codes if it cannot make that guarantee. For example, if a target computer that does not support Windows Sockets Direct exists on the SAN but can only communicate through NDIS, the SAN service provider cannot return WSAEHOSTUNREACH as the result of a failed SAN connection request to this target because a connection request through the TCP/IP provider might succeed. In this case, the SAN service provider should return WSAETIMEDOUT.

Session Negotiation

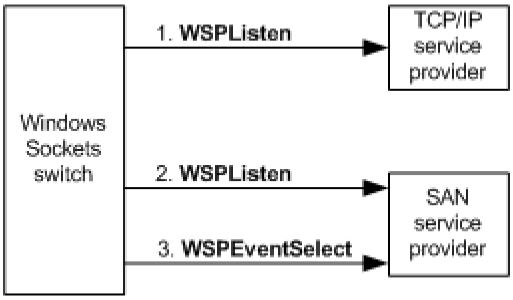
After the switch establishes a connection through a SAN service provider, the switch calls the SAN service provider's **WSPRegisterMemory** extension function to preregister the memory for the buffer array that is to receive incoming messages. The switch next calls the SAN service provider's **WSPRecv** function to post one or more buffers to receive incoming message data from the remote peer. The switch then negotiates a session with its remote peer by exchanging a pair of messages that contain initial flow control information. After the switch negotiates a session, it completes the **WSPConnect** call that the application initiated. The application can then begin sending and receiving data on the connection. For more information, see Accepting Connection Requests.

After a connection is established over a SAN socket, the switch does not call the SAN service provider's **WSPConnect** function. The switch internally handles applications that initiate a call to the switch's **WSPConnect** function to poll for connection requests.

Listening for Connections on a SAN

Article • 12/15/2021

The following figure shows an overview of how the Windows Sockets switch sets a SAN socket to acknowledge and queue--that is, listen for--incoming connection requests from a remote peer. The topics that follow describe the listening process in more detail.



When the Windows Sockets switch receives a **WSPListen** call that was initiated by an application, the switch always calls the TCP/IP provider's **WSPListen** function first to set the TCP/IP provider's socket to acknowledge and queue incoming connection requests. If the application's socket is bound to the IP address of a SAN NIC or to the wildcard IP address, the switch also uses the appropriate SAN service provider to create and bind an additional socket. For more information, see Creating and Binding SAN Sockets.

Listening for Incoming Connection Requests

After requesting a SAN service provider to create and bind the SAN socket, the switch calls the **WSPListen** function of the SAN service provider to cause the SAN socket to listen for incoming connections and to specify a limit on the number of incoming connection requests that the SAN service provider can queue.

Setting Up to Accept Incoming Connections

The switch accepts incoming connections only in nonblocking mode. The switch calls the SAN service provider's **WSPEventSelect** function to put a socket in nonblocking mode and to request notification of incoming connection events. In this call, the switch passes the FD_ACCEPT code and the event object to be associated with that code. After the SAN service provider receives a connection request on its socket that was previously established for listening, the SAN service provider calls the Win32 **SetEvent** function to signal the associated event object. The switch listens for incoming connection events in a dedicated thread and accepts or rejects the connection after the event object is signaled. For more information, see Accepting Connection Requests.

Indicating Refusal of a Connection Request to a Remote Peer

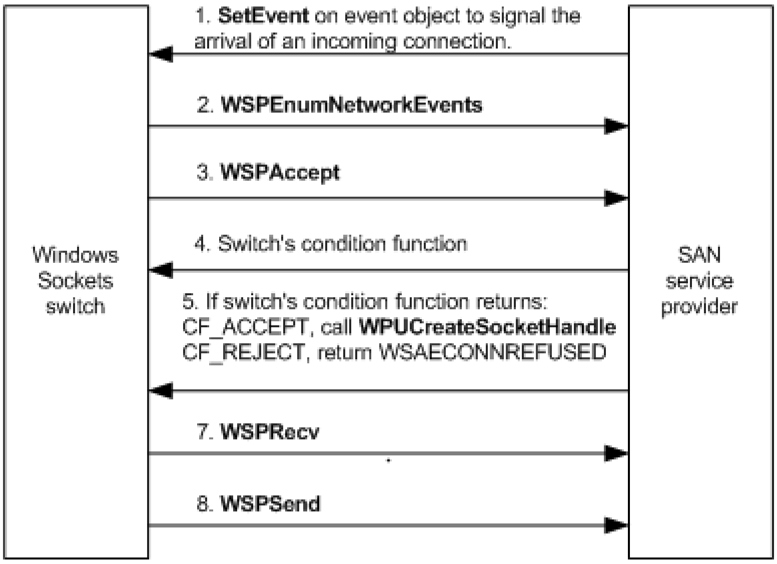
If a connection request arrives and the SAN service provider's backlog of connection requests is full, the SAN service provider should immediately indicate to the remote peer that it refuses the connection request. In this case, the SAN service provider does not signal the event object to inform the switch to accept or reject the connection request. The SAN service provider on the remote peer must then fail its connection operation that was initiated by a **WSPConnect** call with the WSAECONNREFUSED error code.

Accepting Connection Requests

Article • 12/15/2021

If an application calls the **WSAAccept**, **accept**, or **AcceptEx** function to accept an incoming connection request on a socket, the Windows Sockets switch always forwards this call to the TCP/IP service provider. If an incoming connection request arrives from a non-SAN network, it flows through the NDIS path and the TCP/IP service provider handles it. If a connection request arrives from a remote peer on a SAN, the switch acts as an intermediary between the TCP/IP service provider and the SAN service provider in determining whether to accept the connection request and in completing the application's **WSAAccept**, **accept**, or **AcceptEx** function.

The following figure shows an overview of the interaction between the Windows Sockets switch and the SAN service provider in determining whether to accept or reject an incoming connection request. The sequences and sections that follow describe acceptance determination in more detail.



1. **SetEvent** on event object to signal the arrival of an incoming connection.
2. **WSPEnumNetworkEvents**
3. **WSPAccept**
Windows Sockets switch
4. Switch's condition function
SAN service provider
5. If switch's condition function returns:
CF_ACCEPT, call **WPUCreateSocketHandle**
CF_REJECT, return WSAECONNREFUSED
7. **WSPRecv**
8. **WSPSend**

To accept or reject a connect request

1. On receiving an incoming connection request from a remote peer, the SAN service provider signals an event object as described in Listening for Connections on a SAN.

2. The Windows Sockets switch calls the SAN service provider's **WSPEnumNetworkEvents** function to receive the FD_ACCEPT event code.

3. On receiving the FD_ACCEPT event code, the switch calls the SAN service provider's **WSPAccept** function to accept or reject the incoming connection request.

4. In the switch's call to the SAN service provider's WSPAccept function, the switch specifies a condition function. The SAN service provider must call this condition function in the same thread in which the **WSPAccept** function was called before returning from the **WSPAccept** call.

5. The switch returns the CF_ACCEPT or CF_REJECT code from this condition function to indicate that it accepts or rejects the connection request, respectively.

Accepting a Connection Request and Creating an Accepting Socket

If an application accepts an incoming connection request, the switch returns the CF_ACCEPT code to the SAN service provider to complete the switch's condition function. On receiving CF_ACCEPT, the SAN service provider initializes an internal data structure in which it stores information about the accepting socket. The SAN service provider's WSPAccept function must next call the **WPUCreateSocketHandle** function to acquire a descriptor for the accepting socket from the switch. The SAN service provider must store the switch's descriptor in its internal data structure for the accepting socket and must return its own descriptor for the accepting socket to complete the **WSPAccept** call. The switch must supply the SAN service provider's internal descriptor for the accepting socket when calling the SAN service provider's functions, while the SAN service provider must supply the switch's socket descriptor in up calls to the switch.

Before successfully completing WSPAccept, the SAN service provider should call the Win32 **ResetEvent** function to reset the event object. Doing so enables the SAN service provider to later call the Win32 **SetEvent** function to signal the switch to accept the next incoming connection request.

Rejecting a Connection Request

If an application rejects an incoming connection request, the switch returns the CF_REJECT code to the SAN service provider to complete the switch's condition function. On receiving CF_REJECT, the SAN service provider should return the WSAECONNREFUSED error code to the switch to complete the WSPAccept call.

Indicating Acceptance or Refusal of a Connection Request to a Remote Peer

Before a SAN service provider can indicate to a remote peer that it accepts or refuses the remote peer's connection request, the SAN service provider must call the switch's condition function. Depending on the value that the switch's condition function returns, the SAN service provider should make one of the following indications to the remote peer:

If the switch's condition function returns CF_ACCEPT, the SAN service provider should indicate that it accepts the remote peer's connection request. The SAN service provider on the remote peer can then successfully complete its connection operation that was initiated by a **WSPConnect** call.

If the switch's condition function returns CF_REJECT, the SAN service provider should indicate that it refuses the remote peer's connection request. The SAN service provider on the remote peer must fail its connection operation that was initiated by a **WSPConnect** call with the WSAECONNREFUSED error code.

Session Negotiation

After the switch has successfully used a SAN service provider to accept a connection request from a remote peer, the switch negotiates a session with that peer.

To negotiate a session

1. The switch at the remote peer calls the SAN service provider's **WSPRecv** function to post a set of receive buffers.

2. The switch at the remote peer calls the SAN service provider's **WSPSend** function to send a session negotiation message to the switch at the local accepting endpoint. This message includes the number of receive buffers that the switch at the remote peer posted.

3. The switch at the local accepting endpoint calls the local SAN service provider's **WSPRecv** function to post its own receive buffers, but it might not be able to do so in time to receive the session negotiation message. If the local switch does not post a receive buffer in time and if the underlying NIC does not support flow control, the SAN service provider at the local accepting endpoint must buffer the remote switch's session negotiation message in its own private receive buffers. When the switch posts receive buffers, the SAN service provider copies data from its private receive buffers to the switch buffers on a one-to-one basis until all of the data has been copied from the private buffers to the switch buffers.

   The SAN service provider performs normal receive processing on subsequent switch buffers--that is, it posts all such switch buffers to the receive queue on the NIC.

Note that a SAN service provider must not drop a connection simply because the switch did not post a receive buffer before the session negotiation message arrived. The maximum length of a session negotiation message is 256 bytes.

4. The switch at the local accepting endpoint posts its receive buffers before responding to the session negotiation message. The local switch calls the local SAN service provider's WSPSend function to respond to the session negotiation message. The local switch's response includes the number of receive buffers that the local switch posted. From this point forward, the local switch guarantees that the posted set of receive buffers is of sufficient size to receive any message that arrives on the connection.

5. If an application specifies an initial receive buffer in its **AcceptEx** call, the switch waits until it receives the first data message from its remote peer before completing the application's **AcceptEx** call.

6. If the application cancels its own **accept** call, the switch calls the appropriate SAN service provider's WSPCloseSocket function to close the accepting SAN socket.

Registering Memory for Operations on a SAN

Article • 12/15/2021

The Windows Sockets switch calls a SAN service provider's extension functions to register all data buffers for sending and receiving messages and for RDMA operations on a system area network. These extension functions register a buffer to a region of physical memory for use on a particular SAN socket that is connected to a remote peer. For a description of these extension functions, see the Windows Sockets SPI Extensions for SANs.

Registering Data Buffers

The switch calls a SAN service provider's **WSPRegisterMemory** extension function on behalf of an application that runs in a local process to register data buffers that can be accessed only by that process. The buffer handles that **WSPRegisterMemory** returns are valid only in the context of the local process that performed the registration. The switch calls **WSPRegisterMemory** to register buffers that serve as the message receiving buffer in a call to the **WSPRecv** function or the message sending buffer in a call to the **WSPSend** function. The switch also calls **WSPRegisterMemory** to register buffers that serve as the local receiving RDMA buffer in a call to the **WSPRdmaRead** extension function or the local RDMA source in a call to the **WSPRdmaWrite** extension function. After the local process finishes using buffers that were registered with **WSPRegisterMemory**, the switch calls the **WSPDeregisterMemory** extension function to release those buffers.

The switch calls the SAN service provider's **WSPRegisterRdmaMemory** extension function on behalf of an application that runs in a local process to register RDMA buffers that both local and remote processes can access. The buffer descriptors that **WSPRegisterRdmaMemory** returns are valid only for RDMA data transfer operations that a remote peer initiates in the context of the peer's connection to the SAN socket on which the registration was performed. The switch at the remote peer connection uses these RDMA buffers as either the target in a call to the **WSPRdmaWrite** extension function or the source in a call to the **WSPRdmaRead** extension function. After the local and remote processes finish using buffers that were registered with **WSPRegisterRdmaMemory**, the switch calls the **WSPDeregisterRdmaMemory** extension function to release those buffers.

Managing Memory Access

A SAN service provider must prevent unauthorized access to registered memory.

Memory must be registered and accessible as follows:

Memory registered for local access should be available only to the process in which the switch called **WSPRegisterMemory**.

Memory registered for both local and remote access can be accessed by either the process in which the switch called **WSPRegisterRdmaMemory** to register memory, or by the remote peer that is connected to the SAN socket to which the memory is registered.

Memory must be accessible only while registered and while the connection is established. A SAN service provider must ensure that it does not inadvertently make such memory accessible to other processes running on the same computer or on other computers on the SAN.

Memory registered only for read access must not be available for write access. Memory registered only for write access must not be available for read access.

Using Registered Memory

The switch registers two virtually contiguous regions of memory for each connected TCP socket to use for negotiating a data-transfer session. The switch uses one region of memory to provide message buffers containing send data when calling a SAN service provider's **WSPSend** function. The switch uses the other region of memory to post message buffers to receive data when calling a SAN service provider's **WSPRecv** function.

The switch typically registers RDMA buffers only if it transfers application data in RDMA operations.

Before the switch closes a socket, the switch calls either **WSPDeregisterMemory** or **WSPDeregisterRdmaMemory** functions of a SAN service provider to release any memory that a pending data transfer operation is not currently using. The SAN service provider must also release memory associated with outstanding data transfer operations.

Caching Registered Memory

Article • 12/06/2022

SAN service providers can cache RDMA buffers that are exposed for either local or remote access to improve performance.

Caching RDMA Buffers Exposed for Local Access

The Windows Sockets switch calls a SAN service provider's WSPRegisterMemory extension function on behalf of an application to register all data buffers that serve as either the local receiving RDMA buffer in a call to the WSPRdmaRead extension function or the local RDMA source in a call to the WSPRdmaWrite extension function. As part of this registration process, the SAN service provider must lock down these buffers to regions of physical memory and register them with the SAN NIC. Both of these operations are resource intensive. Therefore, the SAN service provider should use caching to reduce the overhead of these registrations. If the SAN service provider uses caching, the performance of applications that reuse buffers for data transfers improves.

SAN service providers should cache and release RDMA buffers that are exposed for local access as described in the following list:

1. When the switch calls the WSPDeregisterMemory extension function to release a buffer, the SAN service provider should leave the buffer registered with the SAN NIC and locked down to a region of physical memory. The SAN service provider should also add the buffer to a cache of registered buffers, in case the buffer is used again in a subsequent RDMA operation, and secure possession of the buffer as described in the next list item.

2. A SAN service provider caches memory registrations based on virtual addresses. When the SAN service provider caches a buffer's registration, the SAN service provider's proxy driver must call the MmSecureVirtualMemory function to secure possession of that registered buffer so that the operating system notifies the switch if the buffer is released (for example, if an application calls the VirtualFree function to release a virtual address range back to the operating system).

3. When the switch subsequently calls **WSPRegisterMemory** to register a buffer, the SAN service provider should check its cache to determine if the buffer is already registered. If the SAN service provider finds the buffer in its cache, the SAN service provider should not perform any further registration action.

4. Before the virtual-to-physical mappings of the registered buffer subsequently change, the switch calls each SAN service provider's **WSPMemoryRegistrationCacheCallback** extension function. Each SAN service provider's proxy driver, in turn, must call the **MmUnsecureVirtualMemory** function to release ownership of the buffer. In addition, each SAN service provider must remove the buffer from its cache and must remove the buffer registration from the SAN NIC.

5. Before the connection between a local SAN socket and a remote peer is closed, the SAN service provider should release any cached buffers.

Note The proxy driver must use the **try/except** mechanism around code that accesses a user-mode buffer that was secured through a call to **MmSecureVirtualMemory** to prevent operating system crashes. For more information about how a proxy driver secures and releases buffers, see Securing and Releasing Ownership of Virtual Addresses.

Caching RDMA Buffers Exposed for Remote Access

The Windows Sockets switch calls a SAN service provider's **WSPRegisterRdmaMemory** extension function to register all data buffers that serve as either the remote RDMA target of a remote **WSPRdmaWrite** call or the remote RDMA source of a remote **WSPRdmaRead** call. That is, the switch exposes these buffers for access by a remote peer. After data transfers from these buffers are completed, the switch calls the SAN service provider's **WSPDeregisterRdmaMemory** extension function to release these buffers so that they are no longer accessible from the remote peer.

SAN service providers should cache RDMA buffers that are exposed for remote access as described in the following list:

1. When the Switch calls **WSPDeregisterRdmaMemory** to release a buffer, the SAN service provider should leave the buffer locked in physical memory and registered with the SAN NIC. The SAN service provider should also add the buffer to a cache of registered buffers, in case the buffer is used again in a subsequent RDMA operation. However, the SAN service provider should take appropriate action to ensure that the remote peer can no longer access the buffer. **Note** If the buffer can only be made inaccessible by the SAN service provider removing the buffer registration from the SAN NIC, the SAN service provider must do so. However, the SAN service provider should leave the buffer locked down to a region of physical memory. This scenario does not provide the best possible performance but is better than no caching.

2. To cache RDMA buffers exposed for remote access, the SAN service provider and its proxy driver should use the caching techniques as described in the preceding list for RDMA buffers that are exposed for local access.

3. When the switch subsequently calls **WSPRegisterRdmaMemory** to register a buffer, the SAN service provider should check its cache to determine if the buffer is already registered. If the SAN service provider finds the buffer in its cache, the SAN service provider should simply expose the buffer for remote access, no further registration action is required. However, if the buffer registration was previously removed from the SAN NIC, the SAN service provider should register the buffer again.

4. To release RDMA buffers exposed for remote access, the SAN service provider and its proxy driver should use the techniques as described in the preceding list.

Transferring Data on a SAN

Article • 12/15/2021

Many system area networks (SANs) lack flow control; therefore, the Windows Sockets switch uses a lightweight session protocol to transfer data on a SAN. The following topics describe features of the switch's session protocol that enable data transfer operations for a SAN service provider:

Using Session Protocol

Sending Urgent Data on a SAN

Completing Data Transfer Requests

Using Session Protocol

Article • 12/15/2021

The Windows Sockets switch uses its session protocol to transfer data over a SAN connection. If the switch transfers a small amount of data, it transfers that data within a control message. Each control message consists of a header and an optional payload of application data. If the switch transfers a large amount of data, it transfers that data using RDMA operations.

This section describes how to set up and perform a data transfer.

Note Depending on the behavior of applications that load the switch, the switch optimizes its session protocol to reduce the overhead involved in transferring application data.

This section also provides examples of how the switch's session protocol performs data transfers. However, these examples do not include definitive descriptions of these operations.

Setting Up a Data Transfer

The switch allocates a pool of control message buffers for each connected socket. The switch then makes calls to the SAN service provider's **WSPRegisterMemory** function to register those message buffers to regions of physical memory. The switch uses part of the buffer pool to send flow control information to a remote peer when calling a SAN service provider's **WSPSend** function. The switch uses the other part of the pool to post message buffers to receive flow control information from a remote peer when calling a SAN service provider's **WSPRecv** function. After the switch receives control messages, it immediately consumes them. After consuming control messages, the switch calls the SAN service provider's **WSPRecv** function and passes receive buffers to post them again so they can receive additional control messages from a remote peer.

Transferring Application Data

The size of the data transfer affects how the switch will handle the data transfer operation.

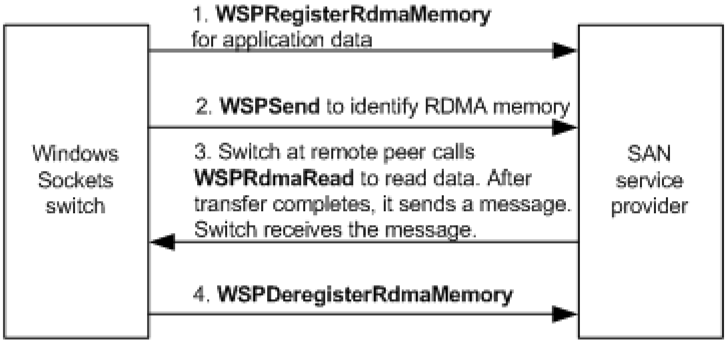
If an application requests to send a small amount of data, the switch transfers that data as described in Sending Urgent Data on a SAN.

If an application requests to send a large amount of data, the switch copies the initial portion of the data to a control message buffer that is used for sending. The header for this control message contains information that specifies the amount of application data. The switch then calls the SAN service provider's WSPSend function to send this control message to the SAN socket's remote peer.

How the switch completes the transfer of application data depends on whether the service provider supports the WSPRdmaRead function.

Data Transfer to a Provider That Supports the WSPRdmaRead Function

The following figure shows an overview of how the switch completes the transfer of application data if the SAN service provider at the remote peer supports a WSPRdmaRead function. The sequence that follows describes transferring application data in more detail.


Windows Sockets switch
1. WSPRegisterRdmaMemory for application data
2. WSPSend to identify RDMA memory
3. Switch at remote peer calls WSPRdmaRead to read data. After transfer completes, it sends a message. Switch receives the message.
4. WSPDeregisterRdmaMemory
SAN service provider

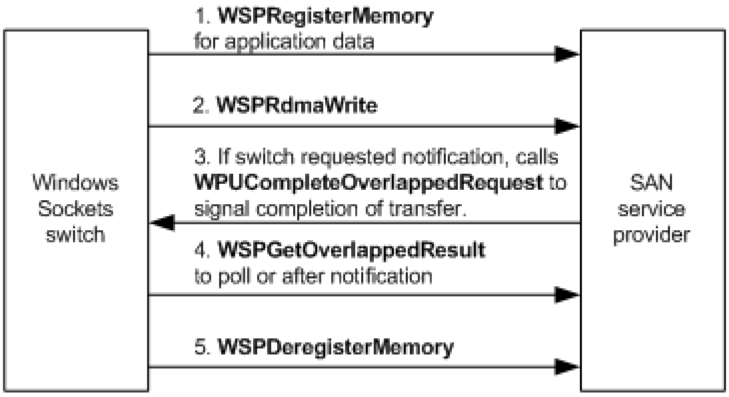
To transfer data when the remote peer supports WSPRdmaRead

- The local switch must call the SAN service provider's WSPRegisterRdmaMemory function to register RDMA memory for read access. In this case, the control header for the message buffer also identifies the descriptor for the RDMA memory that holds the application's remaining data.
- The switch at the remote peer then calls WSPRdmaRead to transfer application data from RDMA memory to receiving buffers that the switch at the remote peer previously registered with WSPRegisterMemory calls. The SAN service provider transmits the buffered data in the background. Doing so allows applications that do not post more than one send at a time to post another send request while the SAN service provider sends buffered data.
- The switch at the remote peer then calls WSPSend to send a control message to the local switch to indicate that the transfer is complete.

- The local switch calls the WSPDeregisterRdmaMemory function to release RDMA memory.
- The local switch completes the application's send request. If the switch cannot register memory for the application's data buffers or if temporary memory cannot be fully allocated, it completes an application's send request with the **WSAENOBUFS** error code.

Data Transfer to a Provider That Does Not Support the WSPRdmaRead Function

The following figure shows an overview of how the switch completes the transfer of application data if the SAN service provider at the remote peer does not support a WSPRdmaRead function. The sequence that follows describes transferring application data in more detail.


1. WSPRegisterMemory
for application data
2. WSPRdmaWrite
3. If switch requested notification, calls
WPUCompleteOverlappedRequest to
signal completion of transfer.
Windows
Sockets
switch
4. WSPGetOverlappedResult
to poll or after notification
SAN
service
provider
5. WSPDeregisterMemory

To transfer data when the remote peer does not support WSPRdmaRead

- The switch at the remote peer calls **WSPRegisterRdmaMemory** to register RDMA memory for write access.
- The switch at the remote peer then calls **WSPSend** to send a control message to the local switch that indicates the location of RDMA memory to which the local switch can write.
- The local switch calls the **WSPRdmaWrite** function to transfer application data to the RDMA memory. The SAN service provider transmits the buffered data in the background. Doing so allows applications that do not post more than one send at a time to post another send request while the SAN service provider sends buffered data.
- The local switch calls the **WSPGetOverlappedResult** function to obtain the results of the transfer. For more information, see Completing Data Transfer Requests.

- The local switch calls WSPSend to send a control message to the remote peer to indicate that the transfer is complete.
- The switch at the remote peer calls WSPDeregisterRdmaMemory to release RDMA memory.
- The local switch completes the application's send request. If the switch cannot register memory for the application's data buffers or if temporary memory cannot be allocated, it completes an application's send request with the **WSAENOBUFS** error code.

Related topics

Transferring Data on a SAN

Sending Urgent Data on a SAN

Article • 12/15/2021

If an application sends urgent data on a SAN, the Windows Sockets switch transfers that data as described in the following sequence:

1. For requests to send urgent data, the switch receives a **WSPSend** call in which the MSG_OOB flag is set.

2. The switch copies the urgent data to the payload portion of a control message buffer.

3. The switch calls the appropriate SAN service provider's **WSPSend** function to transmit the urgent data contained in the control message to the remote peer connection for a SAN socket. The SAN NIC in turn transmits the urgent data.

4. The switch at the remote peer receives the transmitted data into receive buffers that it posted with the **WSPRecv** function.

5. The switch at the remote peer copies the received data from the receive buffers to private storage.

6. The switch at the remote peer calls **WSPRecv** to repost the receive buffers.

7. The switch at the remote peer delivers the data to an application in accordance with standard Windows Sockets procedures.

Completing Data Transfer Requests

Article • 12/06/2022

The Windows Sockets switch transfers data on a SAN socket asynchronously. Whenever the switch calls the SAN service provider's **WSPSend**, **WSPRecv**, **WSPRdmaWrite**, or **WSPRdmaRead** data-transfer function, it specifies a pointer to an overlapped structure (WSAOVERLAPPED) and **NULL** for a completion routine. Even if the switch calls the SAN service provider's **WSPEventSelect** function to indicate that the socket is in nonblocking mode, the SAN service provider is not required to implement nonblocking semantics for these data-transfer functions.

As described in the Windows Sockets API and SPI documentation in the Microsoft Windows SDK documentation, both blocking and nonblocking sockets treat overlapped operations the same. That is, the SAN service provider starts the particular data transfer operation and then immediately returns control to the switch. These data-transfer functions return error code WSA_IO_PENDING to indicate that an asynchronous operation started and that completion of that operation occurs later. After the operation completes, the SAN service provider signals completion if the switch requires completion notification as described in following paragraphs.

Because the switch always specifies **NULL** for a completion routine for overlapped data transfer operations, a SAN service provider is not required to support completion through the use of asynchronous procedure calls (APCs).

Whenever possible, the switch attempts to call the SAN service provider's **WSPGetOverlappedResult** function to poll for completion of data transfer requests. In this way, the switch can avoid the overhead associated with active overlapped completion mechanisms. To indicate to a SAN service provider that the switch does not require completion notification for a particular overlapped data transfer operation, the switch sets the low-order bit of the **hEvent** member in the **WSAOVERLAPPED** structure to one. The SAN service provider must not notify the switch of the completion of requests submitted in this manner.

If the switch requires notification of the completion of an overlapped data transfer operation, it sets the low-order bit of the **hEvent** member in the WSAOVERLAPPED structure to zero. The SAN service provider must complete data transfer operations that are initiated in this way by calling the **WPUCompleteOverlappedRequest** function to signal completion. In this call, the SAN service provider passes a pointer to the WSAOVERLAPPED structure that corresponds to a completed data transfer operation. In this **WPUCompleteOverlappedRequest** call, the SAN service provider also passes the socket descriptor that was acquired from the switch in a call to the

WPUCreateSocketHandle function. The switch receives completion notifications, matches them to an application's I/O requests, and completes those I/O requests, as appropriate, for the application.

Synchronizing Operations on a SAN

Article • 12/15/2021

The Windows Sockets switch uses its session protocol to handle almost all synchronization between a SAN service provider and applications. That is, the switch, in conjunction with the TCP/IP provider, handles most **WSPAsyncSelect**, **WSPEventSelect**, and **WSPSelect** calls from applications. The switch does not forward these calls to a SAN service provider except when specifying the FD_ACCEPT and FD_CONNECT network event codes in calls to a SAN service provider's **WSPEventSelect** function, as described in Setting Up a SAN Connection.

Shutting Down the Use of a SAN

Article • 12/15/2021

The following topics describe how to terminate the use of a SAN:

Shutting Down a SAN Connection

Closing a SAN Socket

Cleaning up a Process for a SAN

Shutting Down a SAN Connection

Article • 12/15/2021

The Windows Sockets switch uses its session protocol to shut down a connection to a SAN socket. That is, the switch handles **WSPRecvDisconnect**, **WSPSendDisconnect**, and **WSPShutdown** calls from applications. The switch does not forward these calls to a SAN service provider. The switch uses its session protocol to disable the reception and transmission of data on a SAN socket.

Closing a SAN Socket

Article • 12/06/2022

After the Windows Sockets switch on either side of a connection calls a SAN service provider's WSPCloseSocket function, the SAN service provider performs the following procedure to close a SAN socket:

1. Each SAN service provider on either side of the connection tears down the connection and completes receive requests-- WSPRecv function calls--by returning the appropriate error code at the *lpErrno* parameter. For example, a SAN service provider returns WSAECONNRESET to indicate that the remote peer reset the connection.

   Each SAN service provider also signals completion of pending overlapped operations for the SAN socket to be closed. The SAN service provider calls the WPUCompleteOverlappedRequest function to signal completion of an overlapped operation. In this call, the SAN service provider passes a pointer to the WSAOVERLAPPED structure that is associated with the overlapped operation. The SAN service provider also passes the WSA_OPERATION_ABORTED error code to specify that the overlapped operation was canceled because the SAN socket was closed. Before signaling completion of an overlapped operation, the SAN service provider should release any memory that was required for the operation.

2. After the SAN service provider is done making up-calls--calls to functions that are prefixed with **WPU**--to the switch using the handle to the SAN socket that was obtained through a WPUCreateSocketHandle up-call, the SAN service provider must make a final up-call to the switch by calling the WPUCloseSocketHandle function to close the socket handle. The SAN service provider then cleans up everything related to the SAN socket. Up-calls are function calls from the switch's up-call dispatch table. The switch provides a pointer to this up-call dispatch table when it calls the SAN service provider's WSPStartupEx function to start using the provider.

As long as a SAN service provider performs the preceding procedure to close a SAN socket, the switch takes care of everything else.

To prevent race conditions between a SAN service provider and the switch initiating socket closures, the SAN service provider should never release data structures related to a SAN socket until the switch calls **WSPCloseSocket**.

Cleaning up a Process for a SAN

Article • 12/15/2021

When an application is ready to clean up the process in which it is running, it initiates a call to the Windows Sockets switch's **WSPCleanup** function. The switch, in turn, calls the **WSPCleanup** function of the TCP/IP provider and all SAN service providers. All providers are expected to release any resources that they were using. Resources can include, for example, objects used to synchronize events and memory used to perform data transfers.

Blocking Calls for a SAN

Article • 12/15/2021

The Windows Sockets switch handles blocking calls and the cancellation of such calls internally or forwards them to the TCP/IP service provider. The switch never calls a **WSPCancelBlockingCall** function for a SAN service provider to cancel a blocking request that is in progress. Therefore, a SAN service provider is not required to implement a **WSPCancelBlockingCall** function.

The switch handles the following blocking requests and corresponding cancellations in the following ways:

- When an application requests to connect a SAN socket to a specific destination address in blocking mode, the switch receives a blocking **WSPConnect** call. The switch forwards the connect request in nonblocking mode to the appropriate SAN service provider's WSPConnect function. If the switch must cancel this connection request for some reason, it calls the SAN service provider's WSPCloseSocket function. The SAN service provider must promptly abort the connection request and release resources for the socket.

- When the switch receives a blocking request that was initiated by an application to perform a data transfer operation on a SAN socket, it forwards the data transfer request in an overlapped (nonblocking) manner to the appropriate SAN service provider. For example, if the switch receives a synchronous (blocking) **WSPSend** call, it calls the appropriate SAN service provider's WSPSend function in an overlapped (nonblocking) manner. If the application later cancels the data transfer operation and the switch has control of the application's buffer, the switch completes the application's request with a failure status. If the application's buffer is involved in an outstanding RDMA operation, the switch waits for the operation to complete. If an RDMA transfer takes too long to complete, the switch calls the appropriate SAN service provider's **WSPCloseSocket** function to close the connection in an abortive manner, thereby forcing completion.

Note If an application cancels a blocking call, it cannot rely on a connection being preserved. Only the **WSPCloseSocket** call is guaranteed to succeed on the socket after the cancellation of a blocking request. For more information, see the Windows Sockets SPI documentation in the Microsoft Windows SDK.

Duplicating Socket Handles for a SAN

Article • 12/15/2021

Multiple applications that run in different processes can use the Windows Sockets switch to perform operations on a shared underlying socket. However, only one application at a time can perform operations on that shared underlying socket.

To use a shared underlying socket, an application must retrieve a duplicate handle to that underlying socket in one of the following ways:

- Directly, by calling the Windows Sockets **WSADuplicateSocket** function

  The call is made in the context of the controlling process (the process in which the socket was created).

- Indirectly, by calling the Win32 DuplicateHandle function

  The call is made in the context of a noncontrolling process (other than the process in which the socket was created).

- Using the handle inheritance mechanism

  A child process (the noncontrolling process) inherits all or some of the handles created in its parent process (the controlling process).

- During graceful connection closure

  If an application in the controlling process closes a socket and exits while some data still remains to be sent, this remaining data is buffered in the Windows Sockets DLL. Another application in the context of the system service process (the noncontrolling process) subsequently sends this data.

The Windows Sockets switch, in conjunction with the TCP/IP provider, detects and handles each of the preceding conditions. The switch allows only one process at a time to execute operations that either transfer data or change state for an underlying shared socket. Processes dynamically swap control of the underlying socket, as required, to execute requested operations. The switch serializes operations that different processes request to perform on a shared socket and executes those operations in first-in-first-out (FIFO) order. The switch waits for all in-progress operations to complete before swapping control of an underlying socket to another process. Logically, the switch takes control of the underlying socket away from the controlling process as soon as a noncontrolling process requests a qualifying operation. After control is taken away, the switch treats the original controlling process like a noncontrolling process if the original controlling process requests qualifying operations. Note that the switch takes no action on a duplicate socket handle until the noncontrolling process actually uses the duplicate socket handle for a data transfer or state-change operation.

Both the switch and the appropriate SAN service provider are loaded into all processes that share access to a particular underlying socket. The switch maintains its own socket context and connection state information in all processes that share the socket. The SAN service provider is required to maintain its socket context and connection state information only in the process that has control of the underlying socket at any given point in time. The SAN service provider must swap control of its context and connection state information from the current controlling process to the next controlling process whenever the switch requires the swap as described in the following sequence. To minimize the amount of resources that are required for swapping, a SAN service provider can maintain its context and connection state information in all processes that share an underlying socket.

Because the switch does not create the SAN socket that corresponds to an application socket until an application calls either the **connect** or **listen** function, the switch cannot request that the SAN service provider perform a swap operation before the application socket is connected or listening. Even after the application socket is connected or listening, one of the following conditions must be met before the switch requests that the SAN service provider swap control of the socket:

- A process that does not control the socket initiates a data transfer. The SAN service provider does not swap control of the socket until all data transfer operations that were initiated by the controlling process have completed.

- A process that does not control the socket calls the **WSAAccept**, **WSPAccept**, or **AcceptEx** function to start a connection-acceptance operation on a listening socket. The SAN service provider does not swap control of the socket until all accept requests that were initiated by the controlling process have completed.

The switch performs the following steps to swap control of a connected SAN socket from the controlling process to the next-controlling process (For an overview of the swapping process, see the table in the Remarks section of the documentation for the WSPDuplicateSocket function.):

1. The switch suspends processing of new requests from the application in the controlling process. When all send and RDMA operations in progress on the SAN socket have completed, the switch calls the SAN service provider's WSPSend function to send a message to a connected peer to request a suspension of the session and calls the SAN service provider's WSPDeregisterMemory function to release all local buffers used for send operations. As a result, the switch at the peer

connection suspends processing of new application requests, waits for all send and RDMA operations in progress on the SAN socket to complete, and releases all RDMA memory. The peer connection next sends a reply message indicating that the session is suspended. On receiving this confirmation message, the switch at the local endpoint calls the SAN service provider's **WSPDeregisterRdmaMemory** function to release all RDMA memory. At this point, SAN sockets at both endpoints of the connection can only have receive requests pending. These receive requests remain pending on the remote peer's SAN socket to permit reactivation of the session. The receive requests on the local SAN socket in the controlling process are completed in the next step. While the connection is suspended, the switch at the remote peer connection queues new blocking or overlapped requests, buffers new nonblocking sends up to the SO_SNDBUF setting, fails new nonblocking sends after the buffer limit is reached, and fails all new nonblocking receives with WSAEWOULDBLOCK. The local switch in the controlling process handles new requests on the application socket as if the process did not have control of the socket.

2. After the session is suspended, the switch calls the SAN service provider's **WSPDuplicateSocket** function in the controlling process to direct the SAN service provider to transfer the socket context into the address space of the next-controlling process. The switch specifies the next-controlling process in the *dwProcessId* parameter of **WSPDuplicateSocket**. The **WSPDuplicateSocket** function must call the **WPUCompleteOverlappedRequest** function to complete all outstanding receive requests on the socket with a success status and zero bytes. The SAN service provider must also automatically release all buffers associated with these requests. The SAN service provider releases all buffers since the switch does not request any more operations on the SAN socket after **WSPDuplicateSocket** returns. The only possible exception is a **WSPCloseSocket** function call, as described in the next step. After **WSPDuplicateSocket** returns, the switch saves the value in the **dwProviderReserved** member of the WSAPROTOCOL_INFOW structure to which the *lpProtocolInfo* output parameter points. The switch uses this value to identify the underlying socket in the context of the next-controlling process. Therefore, the value in **dwProviderReserved** must uniquely identify the underlying socket and the connection for that socket across all processes on the system. In addition, this value must be valid only in the context of the process that the switch specified in the *dwProcessId* parameter of **WSPDuplicateSocket**.

3. After the socket context is transferred into the address space of the next-controlling process, the switch calls the SAN service provider's WSPSocket function in the context of the next-controlling process. In this call, the switch passes the

value for the underlying socket that was returned in the **WSPDuplicateSocket** call to the **dwProviderReserved** member of the WSAPROTOCOL_INFOW structure to which the *lpProtocolInfo* input parameter points. If the next-controlling process did not request the creation of the SAN socket, the SAN service provider must create a new socket and call the **WPUCreateSocketHandle** function to obtain a handle, as required for any new socket. If the SAN socket was created in the context of the next-controlling process, the SAN service provider can reactivate the former socket and return the same descriptor for the socket that was used previously. In this case, the SAN service provider should not call **WPUCreateSocketHandle**, but should continue to use the original socket handle that the switch provided. Alternatively, the SAN service provider can create a new socket, regardless of whether a socket previously existed in the process. In this case, the switch must call the SAN service provider's WSPCloseSocket function in the context of the next-controlling process to dispose of the former socket descriptor.

4. The switch restarts processing of new requests from the application in the next-controlling process.

The switch duplicates a listening socket in a similar manner, except that the switch is not required to suspend a session. The switch waits until it completes all **WSPAccept** calls that were initiated by an application's **accept** and **AcceptEx** calls before calling the SAN service provider's **WSPDuplicateSocket** function in the controlling process.

Because the switch suspends processing of new requests on a SAN socket prior to calling the SAN service provider's **WSPDuplicateSocket** function, the SAN service provider can release all resources associated with a local endpoint in the controlling process. The SAN service provider can even terminate an underlying connection. If the SAN service provider closes an underlying connection in the controlling process, the SAN service provider must reestablish the connection after the switch calls the SAN service provider's **WSPSocket** function within the next-controlling process. After the **WSPSocket** call returns, the SAN socket within the next-controlling process must be in the same state, from the switch's perspective, as the SAN socket in the controlling process was prior to the switch calling the SAN service provider's **WSPDuplicateSocket** function.

If a SAN NIC supports sharing resources between endpoints that run in different processes, the SAN service provider does not have to release resources for a local endpoint in the controlling process prior to receiving a **WSPDuplicateSocket** call. In such a case, the SAN socket associated with a local endpoint remains inactive in the former-controlling process until the switch either swaps the socket context back from the next-controlling process or calls the SAN service provider's **WSPCloseSocket** function to explicitly close the socket. Because most applications perform their final access to the socket in the process that originally created it--generally to close the connection--the SAN service provider can improve performance if the SAN service provider preserves the socket context in the controlling process after the switch swaps control of the socket to the next-controlling process.

Note that, in all cases, a SAN socket descriptor must remain valid until the switch calls the SAN service provider's **WSPCloseSocket** function to explicitly close the socket. Even if the SAN service provider releases all resources for the socket in a particular process prior to receiving a **WSPDuplicateSocket** call, the SAN service provider must not reuse the descriptor for the socket until the switch calls **WSPCloseSocket** on that descriptor.

An unexpected process exit or some other error condition can interrupt a SAN service provider's socket-duplication operation. For example, a shortage of resources can cause such an interruption. The switch treats such error conditions as it does any other error situation. If necessary, the switch closes all descriptors that are associated with the underlying socket in all processes to forcefully terminate the socket's connection. If at all possible, the SAN service provider at the remote peer should complete **WSPRecv** calls that receive incoming data with an appropriate error code, such as WSAECONNRESET. This error code informs the remote peer of the connection termination. If the switch at the remote peer does not receive this connection-termination indication, the switch at the remote peer times out a suspended connection if the system that requested the suspension fails.

Handling Socket Options and Control Codes for a SAN

Article • 12/15/2021

The Windows Sockets switch, in conjunction with the TCP/IP provider, handles most **WSPGetSockOpt**, **WSPSetSockOpt**, and **WSPIoctl** calls initiated by applications. These requests are generally to set and retrieve options and operating parameters associated with an application's socket. The switch does not generally forward these calls to a SAN service provider except as described in the following sections.

Retrieving SAN socket options

The Windows Sockets switch calls a SAN service provider's WSPGetSockOpt function and passes one of the following socket options to retrieve the current value of that option, if the SAN service provider supports that option:

SO_DEBUG
SAN service providers are not required to support this option. They are encouraged, but not required, to supply output debug information if applications set the SO_DEBUG option.

SO_MAX_MSG_SIZE
A SAN service provider must support this option if the underlying SAN transport is message-oriented and the transport limits the amount of data that the switch can send in a call to the SAN service provider's WSPSend function. The switch does not subsequently pass send requests to the SAN service provider that exceed the size that the SAN service provider returns for the value of this option.

SO_MAX_RDMA_SIZE
A SAN service provider must support this option if the underlying SAN transport limits the amount of data that the switch can transfer in calls to either the SAN service provider's WSPRdmaRead or WSPRdmaWrite function. The switch does not subsequently pass RDMA transfer requests to the SAN service provider that exceed the size that the SAN service provider returns for the value of this option.

SO_RDMA_THRESHOLD_SIZE
A SAN service provider supports this option to indicate its preference for the minimum amount of data that the switch can transfer in calls to either the SAN service provider's **WSPRdmaRead** or **WSPRdmaWrite** function. However, the switch can set the actual threshold to a value different from the value returned by the SAN service provider. The

switch subsequently calls the **WSPRdmaRead** or **WSPRdmaWrite** function to transfer data blocks (RDMA transfers) that exceed the size of this threshold and the **WSPSend** or **WSPRecv** function to transfer data blocks (message-oriented transfers) that are less than or equal to the size of this threshold.

SO_GROUP_ID, SO_GROUP_PRIORITY
A SAN service provider must support these options if it supports quality of service (QoS). Otherwise, the switch forwards these options to the TCP/IP provider, which maintains default values. A SAN service provider indicates that it supports QoS by setting the XP1_QOS_SUPPORTED bit in the **dwServiceFlags** member of the WSAPROTOCOL_INFO structure.

Setting SAN socket options

The Windows Sockets switch calls a SAN service provider's **WSPSetSockOpt** function and passes one of the following socket options to set a value for that option, if the SAN service provider supports that option:

SO_DEBUG
For a description of this socket option, see the preceding list.

SO_GROUP_PRIORITY
For a description of this socket option, see the preceding list.

Accessing SAN socket information

The Windows Sockets switch calls a SAN service provider's **WSPIoctl** function and passes one of the following control codes to set or retrieve information for that SAN service provider, if the SAN service provider supports that control code:

SIO_GET_EXTENSION_FUNCTION_POINTER
Retrieves a pointer to an extension function that a SAN service provider must support. For more information about extension functions, see Windows Sockets SPI Extensions for SANs. The input buffer of the **WSPIoctl** call contains the GUID whose value identifies the specified extension function. The SAN service provider returns the pointer to the requested function in **WSPIoctl**'s output buffer. The following table contains GUIDs for extension functions that a SAN service provider can support:

Extension function	GUID
WSPRegisterMemory	{C0B422F5-F58C-11d1-AD6C-00C04FA34A2D}
WSPDeregisterMemory	{C0B422F6-F58C-11d1-AD6C-00C04FA34A2D}

Extension function	GUID
WSPRegisterRdmaMemory	{C0B422F7-F58C-11d1-AD6C-00C04FA34A2D}
WSPDeregisterRdmaMemory	{C0B422F8-F58C-11d1-AD6C-00C04FA34A2D}
WSPRdmaWrite	{C0B422F9-F58C-11d1-AD6C-00C04FA34A2D}
WSPRdmaRead	{C0B422FA-F58C-11d1-AD6C-00C04FA34A2D}
WSPMemoryRegistrationCacheCallback	{E5DA4AF8-D824-48CD-A799-6337A98ED2AF}

SIO_GET_QOS, SIO_GET_GROUP_QOS, SIO_SET_QOS, SIO_SET_GROUP_QOS
A SAN service provider must support these control codes if it supports QoS. Otherwise, the switch forwards these options to the TCP/IP provider, which maintains default values. A provider indicates that it supports QoS by setting the XP1_QOS_SUPPORTED bit in the **dwServiceFlags** member of the WSAPROTOCOL_INFO structure.

SIO_ADDRESS_LIST_QUERY
Retrieves the list of local IP addresses that are assigned to the network interface cards (NICs) that the SAN service provider controls. The SAN service provider uses a SOCKET_ADDRESS_LIST structure, defined as follows, to return the list in **WSPIoctl**'s output buffer:

C++

```
typedef struct _SOCKET_ADDRESS_LIST {
    INT             iAddressCount;
    SOCKET_ADDRESS  Address[1];
} SOCKET_ADDRESS_LIST, FAR * LPSOCKET_ADDRESS_LIST;
```

The members of this structure contain the following information:

iAddressCount
Specifies the number of address structures in the list.

Address
Array of IP address structures.

The switch uses this IOCTL code internally to decide whether to use a given SAN service provider to execute an application's requests to make connections or to listen for incoming connections. The switch forwards actual application requests for the list of local IP addresses to the TCP/IP provider. The switch also uses the TCP/IP provider to detect changes in address lists that all SAN service providers service. After TCP/IP reports a change, the switch queries all SAN service providers to refresh their lists.

Handling Microsoft Extensions to Windows Sockets

Article • 12/15/2021

The Windows Sockets switch handles all Microsoft-specific Windows Sockets extension functions internally. The Windows Sockets documentation in the Microsoft Windows SDK defines an extension as a mechanism that exposes advanced transport functionality to application programs. These extension functions are: **TransmitFile**, **AcceptEx**, and **GetAcceptExSockAddrs**. The switch converts these calls, as necessary, and forwards them to the appropriate SAN service provider function: **WSPSend**, **WSPAccept**, **WSPRdmaWrite**, or **WSPRdmaRead**.

Creating a Service Provider for a SAN

Article • 12/15/2021

This section provides a brief description of the functions that make up the interface between a SAN service provider DLL and the Windows Sockets switch. The SAN service provider DLL exports a single entry point for its initialization function **WSPStartupEx**. The SAN service provider's **WSPStartupEx** function in turn makes most of the other interface functions accessible to the Windows Sockets switch through a dispatch table. The remaining interface functions are supplied to the switch through calls to the SAN service provider's **WSPIoctl** function. The interface functions include Windows Sockets SPI functions and SAN-specific extensions to the Windows Sockets SPI interface.

The Windows Sockets Direct reference provides detailed information about these functions as implemented in a SAN service provider. Do not use the Microsoft Windows SDK descriptions of the Windows Sockets SPI functions. The Windows SDK descriptions do not contain SAN-specific requirements.

This section also lists the Windows Sockets SPI functions that a SAN service provider is not required to supply.

Windows Sockets SPI Functions Required for SANs

Article • 12/15/2021

This section describes the functions of the Windows Sockets SPI that a SAN service provider DLL must supply. These functions are defined in Ws2spi.h and are fully documented in the Windows Sockets Direct Reference section:

WSPAccept
Conditionally accepts a connection for a socket that is listening for connections, based on the return value of a supplied condition function.

WSPBind
Associates the local IP address, or name, of a network interface with a socket. This network interface is serviced by the SAN service provider.

WSPCleanup
Terminates use of the SAN service provider DLL.

WSPCloseSocket
Closes a socket.

WSPConnect
Establishes the connection of a socket to a peer, exchanges connect data, and specifies required quality of service (QoS) based on the supplied flow specification.

WSPDuplicateSocket
Retrieves a WSAPROTOCOL_INFOW structure that can be used to create a new socket descriptor for a shared socket in the context of another process.

WSPEnumNetworkEvents
Reports occurrences of network events for a socket.

WSPEventSelect
Specifies an event object for a socket. This event object is subsequently set by the occurrence of the supplied set of network events.

WSPGetOverlappedResult
Returns the results of an asynchronous (overlapped) operation on a socket. This operation previously indicated that it was pending completion.

WSPGetQOSByName
Initializes a QoS structure based on a named template, or retrieves an enumeration of

the available template names.

A SAN service provider DLL that supports QoS must fully implement **WSPGetQOSByName**. If the SAN service provides does not support QoS, its **WSPGetQOSByName** function must at least return the error WSAEOPNOTSUPP.

WSPGetSockOpt
Retrieves the current value of an option for a socket.

WSPIoctl
Sets or retrieves operating parameters associated with a socket.

WSPListen
Establishes a socket to listen for incoming connections.

WSPRecv
Receives data on a connected socket.

WSPSend
Sends data on a connected socket.

WSPSetSockOpt
Sets the value of an option for a socket.

WSPSocket
Creates a socket that uses the TCP/IP protocol and asynchronous (overlapped) data transfer.

Windows Sockets SPI Functions not Required for SANs

Article • 12/15/2021

This section describes the functions of the Windows Sockets SPI that a SAN service provider is not required to implement. These functions are defined in Ws2spi.h.

WSPAddressToString
The Windows Sockets switch uses the TCP/IP provider to convert all components of a SOCKADDR structure into a human-readable numeric string that represents the IP address of a socket.

WSPAsyncSelect
The Windows Sockets switch uses its session protocol internally to handle notification of network events for a socket, if necessary.

WSPCancelBlockingCall
The Windows Sockets switch internally handles the cancellation of blocking requests that are in progress. Therefore, it never issues cancel blocking calls to a SAN service provider DLL. The Windows Sockets switch can either:

Cancel an outstanding connect request by closing the SAN socket. The SAN service provider DLL should abort the connect request.

Cancel outstanding send and receive requests by discarding data for those requests if the switch buffers that data internally, or by waiting for those requests to complete if they are RDMA transfers to or from application buffers. For lengthy RDMA transfers, the switch can close the connection altogether.

The Windows Sockets SPI documentation in the Microsoft Windows SDK warns that if a blocking call is canceled, an application cannot rely on a connection being preserved. In this case, the only call that is guaranteed to succeed on the socket after the cancellation of a blocking request is **WSPCloseSocket**.

WSPGetPeerName The Windows Sockets switch caches the IP address of a peer when the switch establishes a connection to the peer in a **WSPConnect** call or accepts a connection to the peer in a **WSPAccept** call. The switch provides this cached value to applications, if necessary.

WSPGetSockName The Windows Sockets switch caches the local IP address for a socket when the switch associates the address with the socket in a **WSPBind** call or accepts a connection to a peer in a **WSPAccept** call. The switch provides this cached value to applications, if necessary.

WSPJoinLeaf The Windows Sockets switch exclusively uses the TCP/IP provider to handle multipoint sessions.

WSPRecvDisconnect The Windows Sockets switch internally handles termination of data reception on a socket and retrieves any incoming disconnect data from the remote party.

WSPRecvFrom The current version of Windows Sockets Direct does not support SAN service providers handling sockets that receive datagrams with User Datagram Protocol (UDP) semantics. Therefore, the Windows Sockets switch calls a SAN service provider's **WSPRecv** function on a connected socket to receive stream data with Transmission Control Protocol (TCP) semantics.

WSPSelect The Windows Sockets switch uses its session protocol internally in cooperation with the TCP/IP provider to determine the status of sockets, if necessary.

WSPSendDisconnect The Windows Sockets switch internally handles termination of the connection for a socket and sends disconnect data to the remote party.

WSPSendTo The current version of Windows Sockets Direct does not support SAN service providers handling sockets that send datagrams with User Datagram Protocol (UDP) semantics. Therefore, the Windows Sockets switch calls a SAN service provider's **WSPSend** function on a connected socket to send stream data with Transmission Control Protocol (TCP) semantics.

WSPShutdown The Windows Sockets switch internally disables the reception and transmission of data on a socket.

WSPStartup The Windows Sockets switch does not call **WSPStartup** to start the operation of a SAN service provider. The switch instead uses the SAN service provider's **WSPStatupEx** function.

WSPStringToAddress The Windows Sockets switch uses the TCP/IP provider to convert a human-readable numeric string that represents the IP address of a socket into a socket address structure (SOCKADDR) that is suitable to pass to Windows Sockets routines that take such a structure.

Windows Sockets SPI Extensions for SANs

Article • 12/15/2021

This section provides a brief description of the SAN extension functions that a SAN service provider DLL must supply. These functions extend the Windows Sockets SPI for use with a SAN. The extended functions are defined in Ws2san.h and are fully documented in the Windows Sockets Direct Reference section.

Except for the **WSPStartupEx** function, the extended functions listed in this section are retrieved by the Windows Sockets switch. To retrieve the entry point to each of these extended functions, the Windows Sockets switch calls a SAN service provider's **WSPIoctl** function and passes the SIO_GET_EXTENSION_FUNCTION_POINTER command code along with the GUID whose value identifies one of these extended functions.

A SAN service provider must implement all of the following extension functions with the exception of the **WSPRdmaRead** and **WSPMemoryRegistrationCacheCallback** functions. If a SAN service provider does not support either the **WSPRdmaRead** or **WSPMemoryRegistrationCacheCallback** extension function, its **WSPIoctl** function must return the error WSAEOPNOTSUPP when the Windows Sockets switch requests the entry point to either **WSPRdmaRead** or **WSPMemoryRegistrationCacheCallback**.

WSPStartupEx
Initiates the Windows Sockets switch's use of a SAN service provider.

WSPRegisterMemory
Registers a buffer array that a socket uses as either the local source or the local target of a data transfer operation. Such a socket can use this buffer array as the source buffer in **WSPRdmaWrite** and **WSPSend** calls and the receiving buffer in **WSPRdmaRead** and **WSPRecv** calls.

WSPDeregisterMemory
Releases a buffer array that was registered by a previous call to the **WSPRegisterMemory** function.

WSPRegisterRdmaMemory
Registers an RDMA buffer array that is exposed to a remote peer connection for transferring data to or from that peer connection. A socket at the remote peer can use this RDMA buffer array as the target buffer in a **WSPRdmaWrite** call and the source buffer in a **WSPRdmaRead** call.

WSPDeregisterRdmaMemory

Releases a buffer array that was registered by a previous call to the **WSPRegisterRdmaMemory** function.

WSPMemoryRegistrationCacheCallback

Releases ownership of an application's buffer and the lock between the buffer and physical memory and removes the buffer from the SAN service provider's cache and the buffer registration from the SAN NIC.

WSPRdmaRead

Transfers data from an RDMA buffer in the address space that a socket's remote peer can access to a buffer in the address space that the local socket can access.

WSPRdmaWrite

Transfers data from a source buffer in the address space that a local socket can access to a target RDMA buffer in the address space that the socket's remote peer can access.

Creating a Proxy Driver for a SAN Service Provider

Article • 12/15/2021

A proxy driver for a SAN service provider is a kernel-mode driver that performs tasks required by the Windows Sockets switch and the SAN service provider. Such tasks include managing memory and determining the IP addresses of network interface controllers (NICs) that are under the proxy driver's control. The proxy driver is not required to be a Windows Driver Model (WDM) driver. That is, it is not required to support Plug and Play or power management. For more information about developing a kernel-mode driver, see Kernel-Mode Driver Components.

Different vendors might use different underlying technologies to implement their SAN network interface controllers (NICs), therefore Windows Sockets Direct does not specify an interface between a SAN service provider and its proxy driver or between the proxy driver and a SAN transport.

A SAN NIC vendor must implement a transport interface that is suitable for its underlying technologies. A vendor can implement this interface in the SAN NIC, in a kernel-mode driver for the SAN NIC, or both. A SAN service provider maps this interface directly into a user-mode process's address space. A vendor must ensure that all buffers passed across this interface are locked down and registered with the SAN NIC.

The following sections describe how to create a proxy driver for a SAN service provider DLL:

Initializing and Unloading a SAN Proxy Driver

Allocating and Releasing Memory for a SAN Proxy Driver

Securing and Releasing Ownership of Virtual Addresses

Registering for SAN NIC Notifications

Translating to a SAN Native Address

Implementing IOCTLs for a SAN Service Provider

Initializing and Unloading a SAN Proxy Driver

Article • 12/15/2021

In addition to creating and initializing a device object for the driver object, the proxy driver's **DriverEntry** routine can register to be notified when NICs under the driver's control are either added or removed. For more information, see Registering for SAN NIC Notifications.

If the proxy driver's SAN service provider sends I/O control requests down to the proxy driver, then **DriverEntry** must specify an *entry point* that enables device control. The provider might request, for example, to retrieve the list of IP addresses assigned to the driver's NICs. An entry point for this request is an **IRP_MJ_DEVICE_CONTROL** dispatch routine that returns the list of IP addresses assigned to the driver's NICs. For more information, see Implementing IOCTLs for a SAN Service Provider.

The **DriverEntry** routine must specify an entry point for a routine that unloads the proxy driver. This unload routine removes the device that was created in **DriverEntry**.

Allocating and Releasing Memory for a SAN Proxy Driver

Article • 12/15/2021

The proxy driver must set up access to user buffers so that the Windows Sockets switch can transfer control messages and perform RDMA operations. To request this type of buffer access, the proxy driver sets a bit in the **Flags** member of its device object to DO_DIRECT_IO. The proxy driver must also allocate or release memory that is used for message transfer and RDMA whenever requested to do so. When the Windows Sockets switch requests a SAN service provider to register or release memory, the SAN service provider requests its proxy driver to respectively allocate or release physical memory. For more information about setting up buffer access and allocating and releasing memory, see Memory Management and Buffer Management.

Allocating Low Memory for RDMA

A proxy driver must allocate memory that can be accessed for RDMA operations. The proxy driver can allocate low memory for RDMA operations even on an system that is configured so that no physical memory below 4 GB can be allocated. (This is called a NOLOWMEM configuration.) The proxy driver calls either the **MmAllocateContiguousMemorySpecifyCache** function or its own DMA **AllocateCommonBuffer** function to retrieve low memory.

To retrieve a pointer to its DMA **AllocateCommonBuffer** function, the proxy driver performs the following steps:

1. Zero-initializes a **DEVICE_DESCRIPTION** structure and then writes relevant information for its SAN NIC to this structure.

2. Calls **IoGetDmaAdapter** to retrieve a pointer to the DMA adapter structure for its SAN NIC. In this call, the driver passes a pointer to the filled-in DEVICE_DESCRIPTION structure. **IoGetDmaAdapter** returns a pointer to a DMA adapter structure that contains a pointer to a **DMA_OPERATIONS** structure. DMA_OPERATIONS contains pointers to a system-defined set of DMA functions. One of these functions is **AllocateCommonBuffer**, which allocates a physically contiguous DMA buffer.

Securing and Releasing Ownership of Virtual Addresses

Article • 12/15/2021

The proxy driver must secure ownership of the virtual addresses of user-mode buffers whenever the SAN service provider for the proxy driver caches those buffers. For more information about caching buffers, see Caching Registered Memory. The proxy driver secures ownership of a user-mode buffer, so that the operating system notifies the Windows Sockets switch if the buffer is released back to the operating system by an application. To secure ownership of a buffer, the proxy driver must call the **MmSecureVirtualMemory** function. In this call, the proxy driver passes a pointer to the starting address of the buffer and the size, in bytes, of the buffer.

If the virtual-to-physical mappings for the cached buffer are scheduled to change, the switch is notified and calls the SAN service provider's **WSPMemoryRegistrationCacheCallback** function to remove the buffer registration from the SAN NIC and the buffer from the SAN service provider's cache. The SAN service provider's proxy driver, in turn, must call the **MmUnsecureVirtualMemory** function to release ownership of the buffer. In this call, the proxy driver passes the handle to the buffer that was previously returned from the **MmSecureVirtualMemory** call.

Note A driver that tries to access a user-mode buffer that was secured through a call to **MmSecureVirtualMemory** can potentially bring down the operating system. Therefore, when the proxy driver accesses such a user-mode buffer, it must also use the **try/except** mechanism around the code that accesses the buffer. For more information about **try/except**, see the Visual C++ documentation.

A SAN service provider can send I/O control (IOCTL) requests to the proxy driver to secure and release ownership of a buffer. For more information, see Implementing IOCTLs for a SAN Service Provider.

Registering for SAN NIC Notifications

Article • 12/15/2021

When a proxy driver receives a request from its associated SAN service provider to supply the list of IP addresses assigned to NICs under the driver's control, the driver determines and passes this list to the provider.

In order to obtain these IP addresses, the proxy driver must register with the Transport Driver Interface (TDI) to receive address change notifications. The proxy driver calls the TdiRegisterPnPHandlers function. In this call, this proxy driver passes pointers to callback functions in the **AddAddressHandlerV2** and **DelAddressHandlerV2** members of the TDI_CLIENT_INTERFACE_INFO structure to specify callback functions for address additions and deletions. After the **TdiRegisterPnPHandlers** function has returned successfully, TDI immediately indicates all currently active network addresses to the proxy driver, using the address-addition callback. The indication contains both network addresses and identifiers for the devices to which those addresses are bound.

Whenever TDI calls either of these callback functions to indicate address additions or deletions, the proxy driver requires the following parameters:

Address
Pointer to a TA_ADDRESS structure that describes the network address either assigned to or removed from the NIC. In the case of TCP/IP, this pointer is actually be a pointer to a TA_ADDRESS_IP structure.

DeviceName
Pointer to a Unicode string that identifies the transport-to-NIC binding with which the address is associated. In case of TCP/IP, the Unicode string has the following format: \Device\Tcpip_{NIC-GUID}, where NIC-GUID is the globally unique identifier assigned by the network configuration subsystem to the NIC.

The preceding structure definitions are defined in the tdi.h header file. The preceding registration and callback functions are defined in the tdikrnl.h header file. These header files are available in the Microsoft Windows Driver Development Kit (DDK) and the Windows Driver Kit (WDK). For detailed information about TDI Plug and Play (PnP) notifications, see TDI Client Callbacks and TDI Client Event and PnP Notification Handlers.

At system startup, TDI calls the proxy driver's address-addition callback to indicate all currently active IP addresses. TDI also calls this callback whenever the TCP/IP transport protocol registers a new IP address with TDI. The proxy driver includes in its list of IP addresses only those addresses assigned to the proxy driver's NICs. The driver's address-addition callback should return control promptly if the driver does not recognize the NIC at *DeviceName*.

TDI calls the proxy driver's address-removal callback whenever the TCP/IP transport protocol indicates to TDI that a NIC has been removed. If the IP address of the NIC belongs to one of the proxy driver's NICs, the proxy driver removes the IP address from the list.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

Translating to a SAN Native Address

Article • 12/15/2021

The Windows Sockets switch always uses the WSK address families to interact with a SAN service provider, not the SAN's native address family. Therefore, a proxy driver for a SAN service provider must translate between WSK address families and native addresses accordingly.

A proxy driver uses TDI Plug and Play (PnP) notifications to maintain the list of IP addresses assigned to each NIC under its control, as described in Registering for SAN NIC Notifications. The proxy driver uses this list to translate between native SAN addresses and IP addresses.

The proxy driver receives requests from its SAN service provider that contain IP addresses. These requests include, for example, the request to bind to a specific NIC and the request to connect to a remote peer. The proxy driver must translate to native SAN addresses to complete these requests. The proxy driver also receives incoming connection requests from remote peers that contain SAN native addresses of those remote peers. The proxy driver must translate to the IP addresses of those remote peers to complete these requests.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

Implementing IOCTLs for a SAN Service Provider

Article • 12/15/2021

If a SAN service provider sends I/O control (IOCTL) requests to the proxy driver, the driver should implement an IRP_MJ_DEVICE_CONTROL dispatch routine to process these requests. An IOCTL request can be a request to retrieve the list of IP addresses assigned to the driver's NICs, for example, or a request to allocate or release memory. The **DriverEntry** routine must specify an entry point for the dispatch routine.

The proxy driver's device control routine calls the IoGetCurrentIrpStackLocation function, in which the device control routine passes a pointer to the IRP that was passed to the routine. The device control routine then determines which IOCTL request was received and processes the request accordingly.

After the current IOCTL request completes, the device control routine calls the IoCompleteRequest function and passes the status of the operation. This status is returned to the SAN service provider.

Introduction to Remote NDIS (RNDIS)

Article • 09/27/2024

Remote NDIS (RNDIS) is a bus-independent class specification for Ethernet (802.3) network devices on dynamic Plug and Play (PnP) buses such as USB, 1394, Bluetooth, and InfiniBand. Remote NDIS defines a bus-independent message protocol between a host computer and a Remote NDIS device over abstract control and data channels. Remote NDIS is precise enough to allow vendor-independent class driver support for Remote NDIS devices on the host computer.

Microsoft Windows versions beginning with Windows XP include a Remote NDIS driver for USB devices. This NDIS miniport driver, Rndismp.sys, is implemented and maintained by Microsoft and is distributed as part of all supported Windows versions. You can find it in the %SystemRoot%\System32\drivers directory.

To use this driver with a USB device, an IHV must provide an INF file that follows the template in Remote NDIS INF Template.

Remote NDIS messages are sent to a Remote NDIS device from the host, and a Remote NDIS device responds with an appropriate completion message. Messages are also sent in an unsolicited fashion from a Remote NDIS device to the host.

This section includes:

Overview of Remote NDIS (RNDIS)

Remote NDIS Communication

Remote NDIS To USB Mapping

Feedback

Was this page helpful? Yes No

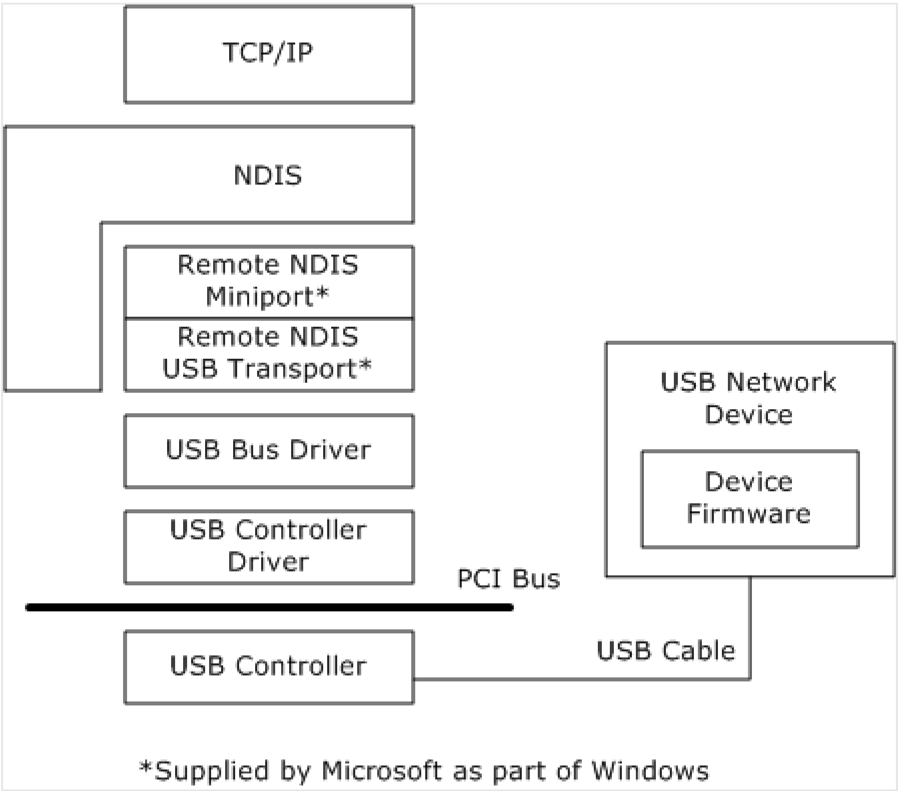
Provide product feedback | Get help at Microsoft Q&A

Overview of Remote NDIS (RNDIS)

Article • 09/27/2024

Remote NDIS (RNDIS) eliminates the need for hardware vendors to write an NDIS miniport device driver for a network device attached to the USB bus. Remote NDIS accomplishes this by defining a bus-independent message set and a description of how this message set operates over the USB bus. Because this Remote NDIS interface is standardized, one set of host drivers can support any number of networking devices attached to the USB bus. This significantly reduces the development burden on device manufacturers, improves the overall stability of the system because no new drivers are required, and improves the end-user experience because there are no drivers to install to support a new USB bus-connected network device. Currently Microsoft Windows provides support for Remote NDIS over USB.

The following figure shows the replacement of the device manufacturer's NDIS miniport with the combination of a Remote NDIS miniport driver and a USB transport driver. The device manufacturer can therefore concentrate on device implementation and not have to develop a Windows NDIS device driver.



Microsoft provides an NDIS miniport driver, Rndismp.sys, which implements the Remote NDIS message set and communicates with generic bus transport drivers, which in turn communicate with the appropriate bus driver. This NDIS miniport driver is implemented and maintained by Microsoft and is distributed as part of Windows.

The following Remote NDIS message set mirrors the semantics of the NDIS miniport driver interface:

- Initializing, resetting, and halting device operation
- Transmitting and receiving networking data packets
- Setting and querying device operational parameters
- Indicating media link status and monitoring device status

Microsoft also provides a USB bus transport driver that implements a mechanism for carrying the Remote NDIS messages across the USB bus. This driver transports standardized Remote NDIS messages between the Remote NDIS miniport driver and the bus-specific driver, such as USB. The bus-specific drivers are also required to map any bus-specific requirements, such as power management, into standardized Remote NDIS messages. The transport driver for USB 1.1 and 2.0 is implemented and maintained by Microsoft and distributed as part of Windows.

This structure allows a single device driver to be used for any Remote NDIS device for which there's a bus-specific transport layer. In addition, only one bus transport layer is required for all network devices on a specific bus.

This section includes the following articles:

Benefits of Remote NDIS

Remote NDIS Concepts and Definitions

Remote NDIS File Naming Conventions

Remote NDIS Messaging

Remote NDIS Device Control

Remote NDIS INF Template

Types of Remote NDIS Devices

Related article

USB class drivers included in Windows

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Benefits of Remote NDIS

Article • 03/14/2023

Remote NDIS is an extension of the well-understood and time-tested NDIS architecture. NDIS defines a function-call interface for device-specific NDIS miniport drivers. This interface defines primitives to send and receive network data, and to query and set configuration parameters and statistics. Remote NDIS leverages NDIS by defining a message wrapping for the NDIS miniport driver interface, thus moving the NDIS-handling code from a miniport driver into the device itself. In this and other ways, Remote NDIS allows for a wide range of device functionality and performance levels. The Remote NDIS model has many advantages:

- Extensibility without change to the bus-specific message transport mechanisms
- Ability to support more protocols over more buses in a short time
- Driver architecture that has been proven for both networking and external bus device models

Value-added mechanisms that already exist in the NDIS network stack are supported for Remote NDIS devices.

Remote NDIS Concepts and Definitions

Article • 03/14/2023

This section presents an overview of the Remote NDIS requirements on the communication channel and lower-layer drivers that are used to communicate between the host and the Remote NDIS device. Device state transitions and major operations such as initialization, halt and reset are also described in this section.

- **Control Channel**

  The control channel must be reliable and ensure sequenced delivery. It is used for all communication except for the transmission of network data packets. All required control messages, except REMOTE_NDIS_HALT_MSG and REMOTE_NDIS_INDICATE_STATUS_MSG, are request and response exchanges initiated by the host. The device must respond within the time-out period as specified for each bus.

- **Data Channel**

  The data channel is used exclusively for the transmission of network data packets. It may consist of multiple subchannels (for example, for varying quality of service) as defined for the appropriate bus.

- **Initialization and Teardown**

  The control and data channels are initialized and set up as specified for the appropriate bus. The host sends a REMOTE_NDIS_INITIALIZE_MSG message to the Remote NDIS device. The Remote NDIS device provides information about its type (connectionless or connection-oriented), supported medium, and version in the response message REMOTE_NDIS_INITIALIZE_CMPLT.

  Either the host or the Remote NDIS device can tear down the communication channel through the REMOTE_NDIS_HALT_MSG message. All outstanding requests and packets are discarded on receipt of this message.

- **Device State Definitions**

  Following bus-level initialization, the device is said to be in the RNDIS-uninitialized state. Upon receiving a REMOTE_NDIS_INITIALIZE_MSG and responding with a REMOTE_NDIS_INITIALIZE_CMPLT with a status of RNDIS_STATUS_SUCCESS, the device enters the RNDIS-initialized state.

Upon receiving REMOTE_NDIS_SET_MSG specifying a nonzero filter value for OID_GEN_CURRENT_PACKET_FILTER, the device enters the RNDIS-data-initialized state.

When in the state RNDIS-data-initialized, reception of a REMOTE_NDIS_SET_MSG specifying a zero filter value for OID_GEN_CURRENT_PACKET_FILTER forces the device back to the RNDIS-initialized state.

Reception of REMOTE_NDIS_HALT_MSG or a bus-level disconnect or hard-reset at any time forces the device to the RNDIS-uninitialized state.

- **Halt**

  At any time that the device is in the RNDIS-initialized or RNDIS-data-initialized state, the host computer may terminate the Remote NDIS functionality of the device by sending REMOTE_NDIS_HALT_MSG to the device.

- **Resetting the Communication Channel**

  The communication channel is reset when an error, such as message time-out, occurs. The host may initiate a reset at any time when the device is in the RNDIS-initialized state by sending the message REMOTE_NDIS_RESET_MSG to the device and the device must send a response message when it has completed the reset. For example, the host may initiate a reset when an error, such as a message time-out, has occurred.

  Note that this is a soft reset in the sense that any handles (for example, VCs for connection-oriented devices) continue to be valid after the reset. The Remote NDIS device discards all outstanding requests and packets as part of the reset process. The remote device might reset some of its hardware components, but keeps the communication channel intact.

  If the Remote NDIS device performs a reboot, this event is equivalent to "Remove" followed by "Add" Plug and Play events. The host NDIS miniport driver will be halted and removed, and a new instance will be added and started. All bus-level and Remote NDIS initialization will be re-executed. A Remote NDIS device may reboot itself in the event of a critical device failure.

- **Flow Control**

  The Remote NDIS device may need to exercise flow control to prevent the host from overflowing its data buffers with packets. Any flow control provisions or requirements are bus specific.

- **Numeric Byte Ordering**

All numeric values in Remote NDIS messages must be coded in little-endian format (least significant byte first).

- **NDIS Message Encapsulation**

  There is no Remote NDIS specification for the way NDIS messages are encapsulated in native bus messages or primitives.

Remote NDIS File Naming Conventions

Article • 03/14/2023

In order to support legacy Remote NDIS devices, multiple Remote NDIS drivers have been included with various versions of Windows. The following table lists the Remote NDIS driver names used in each version of Windows.

Remote NDIS file name	Windows version in which this driver is available
Rndismp.sys Usb8023.sys	These binaries are shipped only for legacy device support. No Remote NDIS device INF file should reference these drivers.
Rndismpy.sys Usb8023y.sys	Windows 2000. Were provided separately from the operating system. These are the only binaries for which Microsoft grants redistribution rights.
Rndismpx.sys Usb8023x.sys Netrndis.inf	• Windows XP SP2 and later • Windows XP x64 • Windows Server 2003 SP1 (x86, x64, ia64) and later • Windows Vista (x86, x64) and later The Rndismpx.sys and Usb8023x.sys binaries ship as part of the operating system. The Netrndis.inf file is an internal file that is part of the operating system. All these files must be referenced by the IHV-provided INF file as described in Remote NDIS INF Template.

Note Remote NDIS is not supported on Windows 98/Me/SE or prior versions.

Remote NDIS Messaging

Article • 03/14/2023

There are two types of Remote NDIS messages: *control messages* and *data messages*. Control messages allow the host and Remote NDIS device to communicate with each other over the communication channel. Data messages contain the message data information needed for the communication between the host and device and are communicated over the data channel.

- **Remote NDIS Control Messages**

  Remote NDIS control messages can be sent by the host to the Remote NDIS device and by the Remote NDIS device to the host. The following Remote NDIS control messages must be supported by an Ethernet 802.3 connectionless device:

  REMOTE_NDIS_INITIALIZE_MSG

  REMOTE_NDIS_INITIALIZE_CMPLT

  REMOTE_NDIS_HALT_MSG

  REMOTE_NDIS_QUERY_MSG

  REMOTE_NDIS_QUERY_CMPLT

  REMOTE_NDIS_SET_MSG

  REMOTE_NDIS_SET_CMPLT

  REMOTE_NDIS_RESET_MSG

  REMOTE_NDIS_RESET_CMPLT

  REMOTE_NDIS_INDICATE_STATUS_MSG

  REMOTE_NDIS_KEEPALIVE_MSG

  REMOTE_NDIS_KEEPALIVE_CMPLT

- **Remote NDIS Data Message**

  A Remote NDIS device must send and receive data through Remote NDIS data packets contained in the REMOTE_NDIS_PACKET_MSG message structure. Remote NDIS data packets may also contain out of band data as well as the data that goes across the network.

Both connectionless (for example, 802.3) and connection-oriented (for example, ATM) devices use the same **REMOTE_NDIS_PACKET_MSG** message structure, in order to facilitate common code for packet processing.

Remote NDIS Device Control

Article • 03/14/2023

The host uses REMOTE_NDIS_QUERY_MSG and REMOTE_NDIS_SET_MSG to control the operation of the Remote NDIS device. An NDIS object ID (OID) is used with each such message to identify a device operational parameter or statistics counter. The lists of Remote NDIS OIDs are broken down into two groups: *general OID* and *802.3-specific OID*. Additionally, each group includes a subsection of statistic OID queries. The general OIDs are required of any networking device.

Remote NDIS INF Template

Article • 09/27/2024

Microsoft provides an NDIS miniport driver, Rndismp.sys, which implements the Remote NDIS message set and communicates with generic bus transport drivers, which in turn communicate with the appropriate bus driver. This NDIS miniport driver is implemented and maintained by Microsoft and is distributed as part of all supported Windows versions. You can find it in the %SystemRoot%\System32\drivers directory.

To use the Remote NDIS driver with a USB device, an IHV must provide an INF file according to one of the following templates:

- RNDIS INF template for NDIS 5.1 (Windows XP and later)
- RNDIS INF template for NDIS 6.0 (Windows 7 and later)

RNDIS INF template for NDIS 5.1 (Windows XP and later)

INF

```
; Remote NDIS template device setup file
; Copyright (c) Microsoft Corporation
;
; This is the template for the INF installation script
; for the RNDIS-over-USB host driver.
; This INF works for Windows XP SP2, Windows XP x64,
; Windows Server 2003 SP1 x86, x64, and ia64, and
 ; Windows Vista x86 and x64.
; This INF will work with Windows XP, Windows XP SP1,
; and Windows 2003 after applying specific hotfixes.

[Version]
Signature           = "$Windows NT$"
Class               = Net
ClassGUID           = {4d36e972-e325-11ce-bfc1-08002be10318}
Provider            = %Microsoft%
DriverVer           = 06/21/2006,6.0.6000.16384
;CatalogFile        = device.cat
PnpLockdown         = 1

[Manufacturer]
%Microsoft%         = RndisDevices,NTx86,NTamd64,NTia64

; Decoration for x86 architecture
[RndisDevices.NTx86]
%RndisDevice%    = RNDIS.NT.5.1, USB\VID_xxxx&PID_yyyy

; Decoration for x64 architecture
[RndisDevices.NTamd64]
```

```
%RndisDevice%    = RNDIS.NT.5.1, USB\VID_xxxx&PID_yyyy

; Decoration for ia64 architecture
[RndisDevices.NTia64]
%RndisDevice%    = RNDIS.NT.5.1, USB\VID_xxxx&PID_yyyy

;@@@ This is the common setting for setup
[ControlFlags]
ExcludeFromSelect=*

; DDInstall section
; References the in-build Netrndis.inf
[RNDIS.NT.5.1]
Characteristics = 0x84   ; NCF_PHYSICAL + NCF_HAS_UI
BusType         = 15
; NEVER REMOVE THE FOLLOWING REFERENCE FOR NETRNDIS.INF
include         = netrndis.inf
needs           = Usb_Rndis.ndi
AddReg          = Rndis_AddReg_Vista

; DDInstal.Services section
[RNDIS.NT.5.1.Services]
include     = netrndis.inf
needs       = Usb_Rndis.ndi.Services

; Optional registry settings. You can modify as needed.
[RNDIS_AddReg_Vista]
HKR, NDI\params\VistaProperty, ParamDesc,  0, %Vista_Property%
HKR, NDI\params\VistaProperty, type,       0, "edit"
HKR, NDI\params\VistaProperty, LimitText,  0, "12"
HKR, NDI\params\VistaProperty, UpperCase,  0, "1"
HKR, NDI\params\VistaProperty, default,    0, " "
HKR, NDI\params\VistaProperty, optional,   0, "1"

; No sys copyfiles - the sys files are already in-build
; (part of the operating system).

; Modify these strings for your device as needed.
[Strings]
Microsoft             = "Microsoft Corporation"
RndisDevice           = "Remote NDIS based Device"
Vista_Property        = "Optional Vista Property"
```

RNDIS INF template for NDIS 6.0 (Windows 7 and later)

INF

```
; Remote NDIS template device setup file
; Copyright (c) Microsoft Corporation
;
; This is the template for the INF installation script  for the RNDIS-over-
USB
```

```
; host driver that leverages the newer NDIS 6.x miniport (rndismp6.sys) for
; improved performance. This INF works for Windows 7, Windows Server 2008
R2,
; and later operating systems on x86, amd64 and ia64 platforms.

[Version]
Signature           = "$Windows NT$"
Class               = Net
ClassGUID           = {4d36e972-e325-11ce-bfc1-08002be10318}
Provider            = %Microsoft%
DriverVer           = 07/21/2008,6.0.6000.16384
;CatalogFile        = device.cat
PnpLockdown         = 1

[Manufacturer]
%Microsoft%         = RndisDevices,NTx86,NTamd64,NTia64

; Decoration for x86 architecture
[RndisDevices.NTx86]
%RndisDevice%    = RNDIS.NT.6.0, USB\VID_xxxx&PID_yyyy

; Decoration for x64 architecture
[RndisDevices.NTamd64]
%RndisDevice%    = RNDIS.NT.6.0, USB\VID_xxxx&PID_yyyy

; Decoration for ia64 architecture
[RndisDevices.NTia64]
%RndisDevice%    = RNDIS.NT.6.0, USB\VID_xxxx&PID_yyyy

;@@@ This is the common setting for setup
[ControlFlags]
ExcludeFromSelect=*

; DDInstall section
; References the in-build Netrndis.inf
[RNDIS.NT.6.0]
Characteristics = 0x84   ; NCF_PHYSICAL + NCF_HAS_UI
BusType         = 15
; NEVER REMOVE THE FOLLOWING REFERENCE FOR NETRNDIS.INF
include         = netrndis.inf
needs           = usbrndis6.ndi
AddReg          = Rndis_AddReg
*IfType            = 6    ; IF_TYPE_ETHERNET_CSMACD.
*MediaType         = 16   ; NdisMediumNative802_11
*PhysicalMediaType = 14   ; NdisPhysicalMedium802_3

; DDInstal.Services section
[RNDIS.NT.6.0.Services]
include     = netrndis.inf
needs       = usbrndis6.ndi.Services

; Optional registry settings. You can modify as needed.
[RNDIS_AddReg]
HKR, NDI\params\RndisProperty, ParamDesc,  0, %Rndis_Property%
HKR, NDI\params\RndisProperty, type,       0, "edit"
```

```
HKR, NDI\params\RndisProperty, LimitText,  0, "12"
HKR, NDI\params\RndisProperty, UpperCase,  0, "1"
HKR, NDI\params\RndisProperty, default,    0, " "
HKR, NDI\params\RndisProperty, optional,   0, "1"

; No sys copyfiles - the sys files are already in-build
; (part of the operating system).

; Modify these strings for your device as needed.
[Strings]
Microsoft             = "Microsoft Corporation"
RndisDevice           = "Remote NDIS6 based Device"
Rndis_Property         = "Optional RNDIS Property"
```

Related topics

Overview of Remote NDIS (RNDIS)

USB class drivers included in Windows

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Types of Remote NDIS Devices

Article • 03/14/2023

Remote NDIS supports Ethernet 802.3 connectionless devices for Microsoft Windows Me and Windows XP.

Remote NDIS Communication

Article • 03/14/2023

This section includes the following topics:

Remote NDIS Control Messages

Remote NDIS Data Message

Setting Device-Specific Parameters

Example Connectionless (802.3) Initialization Sequence

Remote NDIS OIDs

Remote NDIS Version

Status Values

REMOTE_NDIS_INITIALIZE_MSG

Article • 03/03/2023

This message is sent by the host to a Remote NDIS device to initialize the network connection. It is sent through the control channel and only when the device is not in a state initialized by Remote NDIS.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x00000002.
4	4	MessageLength	Specifies in bytes the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is used to match messages sent by the host with device responses.
12	4	MajorVersion	Specifies the Remote NDIS protocol version implemented by the host. The current specification uses RNDIS_MAJOR_VERSION = 1.
16	4	MinorVersion	Specifies the Remote NDIS protocol version implemented by the host. The current specification uses RNDIS_MINOR_VERSION = 0.
20	4	MaxTransferSize	Specifies the maximum size in bytes of any single bus data transfer that the host expects to receive from the device. Typically, each bus data transfer accommodates a single Remote NDIS message. However, the device may bundle several Remote NDIS messages that contain data packets into a single transfer (see REMOTE_NDIS_PACKET_MSG).

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_INITIALIZE_CMPLT

Article • 12/15/2021

The REMOTE_NDIS_INITIALIZE_CMPLT message is sent by the Remote NDIS device to the host in response to a REMOTE_NDIS_INITIALIZE_MSG message. In the REMOTE_NDIS_INITIALIZE_CMPLT message, the device reports its medium type, Remote NDIS version numbers, and its type (connectionless or connection-oriented or both).

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x80000002.
4	4	MessageLength	Specifies in bytes the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is used to match messages sent by the host with device responses.
12	4	Status	Specifies RNDIS_STATUS_SUCCESS if the device initialized successfully; otherwise, it specifies an error code that indicates the failure.
16	4	MajorVersion	Specifies the highest Remote NDIS major protocol version supported by the device.
20	4	MinorVersion	Specifies the highest Remote NDIS minor protocol version supported by the device.
24	4	DeviceFlags	Specifies the miniport driver type as either connectionless or connection-oriented. This value can be one of the following: RNDIS_DF_CONNECTIONLESS 0x00000001 RNDIS_DF_CONNECTION_ORIENTED 0x00000002

Offset	Size	Field	Description
28	4	Medium	Specifies the medium supported by the device. Set to RNDIS_MEDIUM_802_3 (0x00000000)
32	4	MaxPacketsPerMessage	Specifies the maximum number of Remote NDIS data messages that the device can handle in a single transfer to it. This value should be at least one.
36	4	MaxTransferSize	Specifies the maximum size in bytes of any single bus data transfer that the device expects to receive from the host.
40	4	PacketAlignmentFactor	Specifies the byte alignment that the device expects for each Remote NDIS message that is part of a multimessage transfer to it. This value is specified in powers of 2. For example, this value is set to three to indicate 8-byte alignment. This value has a maximum setting of seven, which specifies 128-byte alignment.
44	4	AFListOffset	Reserved for connection-oriented devices. Set value to zero.
48	4	AFListSize	Reserved for connection-oriented devices. Set value to zero.

Remarks

The *Status* field should be set to RNDIS_STATUS_SUCCESS if the device initialized successfully; otherwise, it is set to an error code that indicates the failure. The device should return the highest Remote NDIS protocol version that it can support, in *MajorVersion* and *MinorVersion*--the combined version number should be less than or equal to the version number the host specified in the REMOTE_NDIS_INITIALIZE_MSG message.

The *AFListSize* and *AFListOffset* fields are relevant only for connection-oriented devices that include a call manager. Connectionless devices should set these fields to zero.

In this message, the Remote NDIS device indicates the following:

- Highest Remote NDIS protocol version number that the device can support. The combined version number should be less than or equal to the version number that the host specifies in the REMOTE_NDIS_INITIALIZE_MSG message. This allows the device to fall back to a compatibility mode when the host implements a Remote NDIS protocol version that is lower than that supported by the device.
- Maximum size in bytes of a single data transfer that the device expects to receive from the host. The device can specify the byte alignment it expects for each Remote NDIS message that is part of a multimessage transfer to it. This alignment value is specified in terms of powers of two. For example, this value is set to 3 to indicate 8-byte alignment.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_HALT_MSG

Article • 03/14/2023

This message is sent by the host to terminate the network connection. Unlike the other host-initiated control messages, the device does not respond to REMOTE_NDIS_HALT_MSG.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x00000003.
4	4	MessageLength	Specifies in bytes the total length of this message from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is used to match messages sent by the host with device responses.

Remarks

It is optional for the device to implement REMOTE_NDIS_HALT_MSG. If implemented, the device sends this message to the host through the control channel only when the device is in a state initialized by Remote NDIS. The device must terminate all communication immediately after sending this message. Sending this message causes the device to enter a state not initialized by Remote NDIS.

All outstanding requests and packets should be discarded on receipt of this message.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.

Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_QUERY_MSG

Article • 03/03/2023

This message is sent to a Remote NDIS device from a host when it needs to query the device for its characteristics, statistics information, or status. The parameter or statistics counter being queried for is identified by means of an NDIS Object Identifier (OID). The host may send REMOTE_NDIS_QUERY_MSG to the device through the control channel at any time that the device is in either a state initialized by Remote NDIS. The Remote NDIS device will respond to this message by sending a REMOTE_NDIS_QUERY_CMPLT to the host.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x00000004.
4	4	MessageLength	Specifies in bytes the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is used to match messages sent by the host with device responses.
12	4	Oid	Specifies the NDIS OID that identifies the parameter being queried.
16	4	InformationBufferLength	Specifies in bytes the length of the input data for the query. Set to zero when there is no OID input buffer.

Offset	Size	Field	Description
20	4	InformationBufferOffset	Specifies the byte offset, from the beginning of the *RequestId* field, at which input data for the query is located. Set to zero if there is no OID input buffer.
24	4	DeviceVcHandle	Reserved for connection-oriented devices. Set to zero.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_QUERY_CMPLT

Article • 03/03/2023

A Remote NDIS device will respond to a REMOTE_NDIS_QUERY_MSG message with a REMOTE_NDIS_QUERY_CMPLT message. This message is used to relay the result of a query for a device parameter or statistics counter to the host. The Remote NDIS device also returns the requested information to the host in this message.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x80000004.
4	4	MessageLength	Specifies, in bytes, the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is copied from the REMOTE_NDIS_QUERY_MSG being responded to.
12	4	Status	Specifies the status of processing the OID query request.
16	4	InformationBufferLength	Specifies, in bytes, the length of the response data for the query. Set to zero when there is no OID result buffer.
20	4	InformationBufferOffset	Specifies the byte offset, from the beginning of the *RequestId* field, at which response data for the query is located. Set to zero if there is no response data.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems.

	Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_SET_MSG

Article • 03/03/2023

This message is sent to a Remote NDIS device from a host, when it requires to set the value of some operational parameter on the device. The specific parameter being set is identified by means of an Object Identifier (OID), and the value it is to be set to is contained in an information buffer sent along with the message. The host may send REMOTE_NDIS_SET_MSG to the device through the control channel at any time that the device is in a state initialized by Remote NDIS. The Remote NDIS device will respond to this message by sending a REMOTE_NDIS_SET_CMPLT to the host.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x00000005.
4	4	MessageLength	Specifies, in bytes, the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is used to match messages sent by the host with device responses.
12	4	Oid	Specifies the NDIS OID that identifies the parameter being set.
16	4	InformationBufferLength	Specifies, in bytes, the length of the input data for the request.
20	4	InformationBufferOffset	Specifies the byte offset, from the beginning of the *RequestId* field, at which input data for the request is located.

Offset	Size	Field	Description
24	4	DeviceVcHandle	Reserved for connection-oriented devices. Set to zero.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_SET_CMPLT

Article • 03/03/2023

A Remote NDIS device will respond to a REMOTE_NDIS_SET_MSG message with a REMOTE_NDIS_SET_CMPLT message. This message is used to relay the result of setting the value of a device operational parameter to the host.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x80000005.
4	4	MessageLength	Specifies, in bytes, the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is copied from the REMOTE_NDIS_SET_MSG being responded to.
12	4	Status	Specifies the status of processing the OID set request.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_RESET_MSG

Article • 03/03/2023

This message is sent to a Remote NDIS device from a host to reset the device and return status. The host may send REMOTE_NDIS_RESET_MSG to the device through the control channel at any time that the device is in a state initialized by Remote NDIS. The Remote NDIS device will respond to this message by sending a REMOTE_NDIS_RESET_CMPLT to the host.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x00000006.
4	4	MessageLength	Specifies, in bytes, the total length of this message, from the beginning of the message.
8	4	Reserved	Reserved. Set to zero.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_RESET_CMPLT

Article • 03/03/2023

A Remote NDIS device will respond to a REMOTE_NDIS_RESET_MSG message from the host by resetting the device and returning the status of the request in the REMOTE_NDIS_RESET_CMPLT message.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x80000006.
4	4	MessageLength	Specifies, in bytes, the total length of this message, from the beginning of the message.
8	4	Status	Specifies the status of processing the Reset request.
12	4	AddressingReset	Indicates if addressing information (multicast address list, packet filter) has been lost during the concluded reset operation. If the device requires the host to resend addressing information, set this field to one; otherwise set it to zero.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_INDICATE_STATUS_MSG

Article • 03/03/2023

This message is sent from a Remote NDIS device to a host to indicate a change in the status of the device. A REMOTE_NDIS_INDICATE_STATUS_MSG message can also be used to indicate an error event, such as an unrecognized message. The Remote NDIS device may send this message at any time that it is in a state initialized by Remote NDIS. There is no response to this message.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x00000007.
4	4	MessageLength	Specifies, in bytes, the total length of this message, from the beginning of the message.
8	4	Status	Specifies the current status of the host request.
12	4	StatusBufferLength	Specifies the length of the status data, in bytes.
16	4	StatusBufferOffset	Specifies the byte offset, from the beginning of this message, at which Rndis_Diagnostic_Info status data for the device indication is located.

Remarks

The most common use of REMOTE_NDIS_INDICATE_STATUS_MSG is to indicate the state of the link for an 802.3 device. A status value of RNDIS_STATUS_MEDIA_CONNECT indicates a transition from disconnected (for example no 802.3 link pulse) to connected state (802.3 link pulse detected). A status value of RNDIS_STATUS_MEDIA_DISCONNECT indicates a transition from connected to disconnected state. The device must send

REMOTE_NDIS_INDICATE_STATUS_MSG with one of these values every time the 802.3 link state changes. No status buffer is required to return these two common indications.

In the specific case where this message is sent in response to a host message that the device could not handle, the *Status* field must be set to RNDIS_STATUS_INVALID_DATA, and the Rndis_Diagnostic_Info status buffer is formatted as follows.

Offset	Size	Field	Description
0	4	DiagStatus	Contains status information about the error itself (for example, RNDIS_STATUS_NOT_SUPPORTED)
4	4	ErrorOffset	Specifies the zero-based byte offset in the original message at which the error was detected.

If the error condition was caused by an Remote NDIS message (for example, the device can't recognize a particular RNDIS message), then the device should append the original message at the end of the status message defined above.

This message is used to report an error condition only in circumstances where the device is not able to generate a response message with appropriate status. Examples of appropriate usage are:

- On receiving a message with unsupported message type.
- On receiving a REMOTE_NDIS_PACKET_MSG with unacceptable contents.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_KEEPALIVE_MSG

Article • 03/03/2023

The host sends this message periodically when there has been no other control or data traffic from the device to the host for the bus-defined *KeepAliveTimeoutPeriod*. This message is sent by the host at least every RNDIS_KEEPALIVE_TIMEOUT seconds, in the absence of other message traffic, to detect the state of the remote device. The remote device may use the same message in the reverse direction, but it is not required.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x00000008.
4	4	MessageLength	Specifies in bytes the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is used to match messages sent by the host with device responses.

Remarks

The host will not send a REMOTE_NDIS_KEEPALIVE_MSG message until RNDIS_KEEPALIVE_TIMEOUT seconds have elapsed since the last message received from the remote device. This avoids unnecessary exchange of keep-alive messages when the communication channel is active.

The device can optionally send this message to the host as well. For example, the device may use this message to trigger a response from the host for computing round-trip delay time. If implemented, the device must send REMOTE_NDIS_KEEPALIVE_MSG through the control channel and only when the device is in a state initialized by Remote NDIS.

The host sends a **REMOTE_NDIS_KEEPALIVE_MSG** message to the device through the control channel to check the health of the device. When the device is in a state initialized by Remote NDIS, the host sends this message periodically when there has been no other control or data traffic from the device to the host for the *KeepAliveTimeoutPeriod*. *KeepAliveTimeoutPeriod* is bus-dependent and is defined in the appropriate bus-mapping specifications.

Upon receiving this message, the remote device must return a response whose *Status* field indicates whether the device solicits a **REMOTE_NDIS_RESET_MSG** message from the host.

The device does not have to perform any specific action if it stops seeing **REMOTE_NDIS_KEEPALIVE_MSG** messages from the host.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

REMOTE_NDIS_KEEPALIVE_CMPLT

Article • 03/03/2023

A Remote NDIS device will respond to a REMOTE_NDIS_KEEPALIVE_MSG message from the host by sending back a REMOTE_NDIS_KEEPALIVE_CMPLT response message. If the returned Status is not RNDIS_STATUS_SUCCESS, the host will send REMOTE_NDIS_RESET_MSG to reset the device.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x80000008.
4	4	MessageLength	Specifies, in bytes, the total length of this message, from the beginning of the message.
8	4	RequestId	Specifies the Remote NDIS message ID value. This value is used to match messages sent by the host with device responses.
12	4	Status	Specifies the current status of the device. If the returned *Status* is not RNDIS_STATUS_SUCCESS, the host will send an REMOTE_NDIS_RESET_MSG message to reset the device.

Remarks

If the device implements the option of sending REMOTE_NDIS_KEEPALIVE_MSG, the host will respond with REMOTE_NDIS_KEEPALIVE_CMPLT through the control channel.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems.

	Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

Remote NDIS Data Message

Article • 03/14/2023

A Remote NDIS device encapsulates NDIS packets to transfer them across the data channel. Data messages are used to do this because they can contain out-of-band (OOB) data or per-packet information.

The data message that is used to encapsulate data for transfer across the data channel is described in the following topic:

REMOTE_NDIS_PACKET_MSG

REMOTE_NDIS_PACKET_MSG

Article • 03/03/2023

REMOTE_NDIS_PACKET_MSG encapsulates NDIS data packets to form a single data message.

Concatenating multiple REMOTE_NDIS_PACKET_MSG elements forms a multipacket message. Each individual REMOTE_NDIS_PACKET_MSG component is constructed as described below. The difference from the single-packet message is that the *MessageLength* field in each REMOTE_NDIS_PACKET_MSG header includes some additional padding bytes. These padding bytes are appended to all but the last REMOTE_NDIS_PACKET_MSG so that the succeeding REMOTE_NDIS_PACKET_MSG starts at an appropriate byte boundary. For messages sent from the device to the host, this padding should result in each REMOTE_NDIS_PACKET_MSG starting at a byte offset that is a multiple of 8 bytes starting from the beginning of the multipacket message. When the host sends a multipacket message to the device, it will adhere to the *PacketAlignmentFactor* that the device specifies.

The REMOTE_NDIS_PACKET_MSG format is defined in the following table.

Offset	Size	Field	Description
0	4	MessageType	Specifies the type of message being sent. Set to 0x1.
4	4	MessageLength	Message length in bytes, including appended packet data, OOB data, per-packet information data, and both internal and external padding.
8	4	DataOffset	Specifies the offset in bytes from the start of the DataOffset field of this message to the start of the data. This is an integer multiple of 4.
12	4	DataLength	Specifies the number of bytes in the data content of this message.

Offset	Size	Field	Description
16	4	OOBDataOffset	Specifies the offset in bytes of the first OOB data record from the start of the *DataOffset* field of this message. Set to zero if there is no OOB data. Otherwise, this is an integer multiple of 4.
20	4	OOBDataLength	Specifies in bytes the total length of the OOB data.
24	4	NumOOBDataElements	Specifies the number of OOB records in this message.
28	4	PerPacketInfoOffset	Specifies in bytes the offset from the beginning of the *DataOffset* field in the REMOTE_NDIS_PACKET_MSG data message to the start of the first per-packet information data record. Set to zero if there is no per-packet data. Otherwise, this is an integer multiple of 4.
32	4	PerPacketInfoLength	Specifies in bytes the total length of the per-packet information contained in this message.
36	4	VcHandle	Reserved for connection-oriented devices. Set to zero.
40	4	Reserved	Reserved. Set to zero.

The format of a single OOB data record is indicated in the following table.

Offset	Size	Field	Description
0	4	Size	Length in bytes of this OOB header and appended OOB data and padding. This is an integer multiple of 4.

Offset	Size	Field	Description
4	4	Type	None defined for 802.3 devices.
8	4	ClassInformationOffset	The byte offset from the beginning of this OOB data record to the beginning of the OOB data.
(N)	...	OOB Data	OOB Data; consult Microsoft Windows Driver Development Kit (DDK) documentation for more information.

Note (N) is equal to the value of *ClassInformationOffset*.

The following table defines the format of a per-packet information data record.

Offset	Size	Field	Description
0	4	Size	Length in bytes of this per-packet header and appended per-packet data and padding. This value is an integer multiple of 4.
4	4	Type	Set to one of the legal values for NDIS_PER_PACKET_INFO_FROM_PACKET, as described in the Windows 2000 Driver Development Kit (DDK).
8	4	PerPacketInformationOffset	The byte offset from the beginning of this per-packet information data record to the beginning of the per-packet information data.
(N)	...	Per-Packet Data	Per-Packet Data; consult Windows 2000 DDK documentation for more information.

Note (N) is equal to the value of *PerPacketInformationOffset*.

Remarks

Each REMOTE_NDIS_PACKET_MSG may contain one or more OOB data records. *NumOOBDataElements* indicates the number of OOB data records in this message. The OOB data records must appear in sequence. The *OOBDataLength* field indicates the length in bytes of the entire OOB data block. The *OOBDataOffset* field indicates the byte offset from the beginning of the *DataOffset* field to the beginning of the OOB data block. For more information about OOB packet data, see the NDIS specification in the Windows 2000 DDK.

If multiple OOB data blocks are attached to a REMOTE_NDIS_PACKET_MSG message, each subsequent OOB data record must immediately follow the previous OOB record's data.

No OOB information is currently defined for 802.3 devices.

Each REMOTE_NDIS_PACKET_MSG may contain one or more per-packet-info data records. Per-packet-info is used to convey packet metadata, such as TCP checksum. The *PerPacketInfoOffset* field indicates the byte offset from the beginning of the *DataOffset* field to the beginning of the per-packet information data record. The *OOBDataLength* field indicates the byte length of the per-packet information data record. For more information about per-packet information data, see the Windows 2000 DDK.

If there are multiple per-packet information data blocks, each subsequent per-packet information data record must immediately follow the previous per-packet information record's data.

A Remote NDIS device must send and receive data through NDIS data packets. The bus that the device uses determines how these packets are passed from host to device and device to host. It could be shared memory or, in the case of USB, Isoch and Bulk pipes. NDIS packets may also contain out-of-band (OOB) data as well as the data that goes across the network.

A Remote NDIS device transfers NDIS packets, encapsulated as **REMOTE_NDIS_PACKET_MSG** across the data channel. Both connectionless (such as 802.3) and connection-oriented (such as ATM) devices use the same packet message structure to facilitate common code for packet processing. Each **REMOTE_NDIS_PACKET_MSG** message contains information about a single network data unit (such s an Ethernet 802.3 frame).

For more information about out-of-band packet data or per-packet-info data, see the Windows 2000 DDK NDIS sections.

Requirements

Version	Available in Microsoft Windows XP and later versions of the Windows operating systems. Also available in Windows 2000 as redistributable binaries.
Header	Rndis.h (include Rndis.h)

Multipacket Messages

Article • 12/15/2021

Multiple REMOTE_NDIS_PACKET_MSG messages may be sent in a single transfer, in either direction. A multipacket message is formed by concatenating multiple **REMOTE_NDIS_PACKET_MSG** elements. The maximum length of such a transfer is governed by the *MaxTransferSize* parameter passed in the REMOTE_NDIS_INITIALIZE_MSG and response messages. The host will also limit the number of messages it bundles into a single transfer to the *MaxPacketsPerMessage* parameter returned by the device in the REMOTE_NDIS_INITIALIZE_CMPLT response message.

The difference from the single-packet message case is that the *MessageLength* field in each REMOTE_NDIS_PACKET_MSG header includes some additional padding bytes. These padding bytes are added to all but the last **REMOTE_NDIS_PACKET_MSG** such that the succeeding REMOTE_NDIS_PACKET_MSG starts at an appropriate byte boundary. For messages sent from the device to the host, this padding should result in each REMOTE_NDIS_PACKET_MSG starting at a byte offset that is a multiple of 8 bytes starting from the beginning of the multipacket message. When the host sends a multipacket message to the device, it will adhere to the *PacketAlignmentFactor* specified by the device in the REMOTE_NDIS_INITIALIZE_CMPLT response message.

Note that neither the combined length of a multipacket message nor the number of REMOTE_NDIS_PACKET_MSG elements in a combined message is given explicitly in any Remote NDIS defined field. The combined length is implicit in the bus-specific transfer mechanism, and the host or device must walk the *MessageLength* fields of the combined message to determine the number of combined messages.

The following table is an example of a multipacket message that is made up of two REMOTE_NDIS_PACKET_MSGs, sent from the host to the device. During the REMOTE_NDIS_INITIALIZE_MSG exchange, the device requested a *PacketAlignmentFactor* of 3 (an alignment along an 8-byte boundary).

Offset	Size	Field	Value
0	4	MessageType	0x1
4	4	MessageLength	72 (includes 2 padding bytes; see below)
8	4	DataOffset	36

Offset	Size	Field	Value
12	4	DataLength	26
16	4	OOBDataOffset	0
20	4	OOBDataLength	0
24	4	NumOOBDataElements	0
28	4	PerPacketInfoOffset	0
32	4	PerPacketInfoLength	0
36	4	VcHandle	0
40	4	Reserved	0
44	26	Payload (data)	Some network data of 26 bytes in length
70	2	Padding	Doesn't matter - unused
72	4	MessageType (start of second REMOTE_NDIS_PACKET_MSG)	0x1
76	4	MessageLength	60
80	4	DataOffset	36
84	4	DataLength	16
88	4	OOBDataOffset	0
92	4	OOBDataLength	0
96	4	NumOOBDataElements	0
100	4	PerPacketInfoOffset	0
104	4	PerPacketInfoLength	0
108	4	VcHandle	0
112	4	Reserved	0
116	16	Payload (data)	Some network data of 16 bytes in length

Setting Device-Specific Parameters

Article • 12/15/2021

It is expected that most Remote NDIS devices will function well without the need to configure parameters on the host. However, there may be cases where proper network operation requires some configuration on the host. If the device supports configurable parameters, then it should include the following optional OID in the list of supported OIDs it reports in response to a query for OID_GEN_SUPPORTED_LIST:

C++

```
#define OID_GEN_RNDIS_CONFIG_PARAMETER 0x0001021B
```

If the device supports the OID_GEN_RNDIS_CONFIG_PARAMETER OID, the host uses it to set device-specific parameters, soon after the device enters a state initialized by Remote NDIS from the uninitialized state. The host will send zero or more REMOTE_NDIS_SET_MSGs to the device, with OID_GEN_RNDIS_CONFIG_PARAMETER as the OID value to set. Each such **REMOTE_NDIS_SET_MSG** corresponds to one device-specific parameter that is configured on the host.

The *InformationBuffer* associated with each such **REMOTE_NDIS_SET_MSG** has the following format. Note that the Offset values are relative to the beginning of the information buffer.

Offset	Size	Field	Description
0	4	ParameterNameOffset	Specifies the byte offset from the beginning of the ParameterNameOffset field at which a Unicode character string representing the parameter name is located. The string does not include a NULL terminator.
4	4	ParameterNameLength	Specifies the byte length of the parameter name string.

Offset	Size	Field	Description
8	4	ParameterType	Specifies the data type of the parameter value. This is one of the following: 0 - numeric value; 2 - string value.
12	4	ParameterValueOffset	Specifies the byte offset from the beginning of the ParameterNameOffset field at which the parameter value is located.
16	4	ParameterValueLength	Specifies the byte length of the parameter value.

The device sends a REMOTE_NDIS_SET_CMPLT in response to each REMOTE_NDIS_SET_MSG, after applying the parameter value. If the parameter setting is acceptable, it returns a status of RNDIS_STATUS_SUCCESS in the response. If the parameter setting is not acceptable, and the device cannot apply a useful default value for this parameter, then the device returns an appropriate error status value (see section on status values). If an error status is returned, then the host will initiate a halt process for the device.

Device-specific parameters are expected to be configured in the Windows registry. The keys that define parameter values are typically created in the registry during the process of device installation. The list of keys, type information, default values and optional range of valid values are specified in the INF file for the device. For more information about using an INF to set up configuration parameters in the registry for network devices, consult the Windows 2000 Driver Development Kit (DDK).

Example Connectionless (802.3) Initialization Sequence

Article • 12/15/2021

This section describes the general order of events that a device can expect upon startup as a Remote NDIS connectionless device. Because the basic operation of Remote NDIS is the same, regardless of the underlying bus, the require bus enumeration and start up process has been left out of the example.

Host	Device	Description
REMOTE_NDIS_INITIALIZE_MSG		Hosts sends Remote NDIS Initialization message to device.
	REMOTE_NDIS_INITIALIZE_CMPLT	Device response with Initialize Complete message.
Receiving. Successful Initialization		Host starts accepting data on incoming data channel. (Example: on USB starts doing reads on IN pipe).
REMOTE_NDIS_QUERY_MSG AND REMOTE_NDIS_SET_MSG	REMOTE_NDIS_QUERY_CMPLT OR REMOTE_NDIS_SET_CMPLT	Host initiates a series of sets and queries to determine state of device and to setup initial parameters. The device responses appropriately with the correct complete messages. The following NDIS OIDs may be queried: OID_802_3_CURRENT_ADDRESS, OID_802_3_MAXIMUM_LIST_SIZE, and so on.
REMOTE_NDIS_SET_MSG		Host sends an OID_GEN_CURRENT_PACKET_FILTER OID with a nonzero filter value to the device. At this point the device should start sending data packets on the incoming data channel. The host will also start sending data packets on the outgoing data channel.

Remote NDIS OIDs

Article • 03/14/2023

This section lists the required and optional NDIS OIDs for Remote NDIS Ethernet devices. The list takes into account the unique properties of a Remote NDIS device and the Remote NDIS miniport driver, so the list is not identical to the list that a normal NDIS connectionless miniport driver would support. Some OIDs are both *set* and *query* OIDs; if a mandatory OID is defined as both, then it must be supported by a Remote NDIS device for both REMOTE_NDIS_SET_MSG and REMOTE_NDIS_QUERY_MSG. For a detailed explanation of the OIDs, see the Microsoft Windows Driver Development Kit (DDK).

The following lists of Remote NDIS OIDs are broken down into two groups -- general OID and 802.3 specific OID. Additionally, each group includes a subsection of statistic OID queries. The general OIDs are required for any networking device.

- General OIDs
- General Statistic OIDs
- 802.3 OIDs
- 802.3 Statistic OIDs
- Optional Power Management OIDs
- Optional Network Wake Up OIDs

General OIDs

Article • 12/15/2021

The following table lists the general OIDs for Remote NDIS Ethernet devices.

Support	OID	Description
Required	OID_GEN_SUPPORTED_LIST	List of supported OIDs.
Required	OID_GEN_HARDWARE_STATUS	Hardware status.
Required	OID_GEN_MEDIA_SUPPORTED	Media types supported (encoded).
Required	OID_GEN_MEDIA_IN_USE	Media types in use (encoded).
Required	OID_GEN_MAXIMUM_FRAME_SIZE	Maximum frame size in bytes.
Required	OID_GEN_LINK_SPEED	Link speed in units of 100 bps.
Required	OID_GEN_TRANSMIT_BLOCK_SIZE	Minimum amount of storage, in bytes, that a single packet occupies in the transmit buffer space of the NIC.
Required	OID_GEN_RECEIVE_BLOCK_SIZE	Amount of storage, in bytes, that a single packet occupies in the receive buffer space of the NIC.
Required	OID_GEN_VENDOR_ID	Vendor NIC code.
Required	OID_GEN_VENDOR_DESCRIPTION	Vendor network card description.
Required	OID_GEN_VENDOR_DRIVER_VERSION	Vendor-assigned version number of driver.
Required	OID_GEN_CURRENT_PACKET_FILTER	Current packet filter (encoded).
Required	OID_GEN_MAXIMUM_TOTAL_SIZE	Maximum total packet length in bytes.

Support	OID	Description
Optional	OID_GEN_RNDIS_CONFIG_PARAMETER	Device-specific configuration parameter (set only).
Optional	OID_GEN_PHYSICAL_MEDIUM	Information about the underlying physical medium.
Required	OID_GEN_MEDIA_CONNECT_STATUS	Status of the NIC network connection.
Optional	OID_GEN_MAC_OPTIONS	A bitmask that specifies optional properties of the NIC. Must be supported only by NICs that support 802.1p packet priority.

General Statistic OIDs

Article • 12/15/2021

The following table lists the general statistic OIDs for Remote NDIS Ethernet devices.

Support	OID	Description
Required	OID_GEN_XMIT_OK	Frames transmitted without errors
Required	OID_GEN_RCV_OK	Frames received without errors
Required	OID_GEN_XMIT_ERROR	Frames transmitted with errors
Required	OID_GEN_RCV_ERROR	Frames received with errors
Required	OID_GEN_RCV_NO_BUFFER	Frame missed, no buffers
Optional	OID_GEN_DIRECTED_BYTES_XMIT	Directed bytes transmitted without errors
Optional	OID_GEN_DIRECTED_FRAMES_XMIT	Directed frames transmitted without errors
Optional	OID_GEN_MULTICAST_BYTES_XMIT	Multicast bytes transmitted without errors
Optional	OID_GEN_MULTICAST_FRAMES_XMIT	Multicast frames transmitted without errors
Optional	OID_GEN_BROADCAST_BYTES_XMIT	Broadcast bytes transmitted without errors
Optional	OID_GEN_BROADCAST_FRAMES_XMIT	Broadcast frames transmitted without errors
Optional	OID_GEN_DIRECTED_BYTES_RCV	Directed bytes received without errors
Optional	OID_GEN_DIRECTED_FRAMES_RCV	Directed frames received without errors
Optional	OID_GEN_MULTICAST_BYTES_RCV	Multicast bytes received without errors
Optional	OID_GEN_MULTICAST_FRAMES_RCV	Multicast frames received without errors

Support	OID	Description
Optional	OID_GEN_BROADCAST_BYTES_RCV	Broadcast bytes received without errors
Optional	OID_GEN_BROADCAST_FRAMES_RCV	Broadcast frames received without errors
Optional	OID_GEN_RCV_CRC_ERROR	Frames received with circular redundancy check (CRC) or frame check sequence (FCS) errors
Optional	OID_GEN_TRANSMIT_QUEUE_LENGTH	Length of transmit queue

802.3 OIDs

Article • 12/15/2021

The following table lists the specific 802.3 OIDs for Remote NDIS Ethernet devices.

Support	OID	Description
Required	OID_802_3_PERMANENT_ADDRESS	Permanent station address
Required	OID_802_3_CURRENT_ADDRESS	Current station address
Required	OID_802_3_MULTICAST_LIST	Current multicast address list
Optional	OID_802_3_MAC_OPTIONS	NIC flags (encoded)
Required	OID_802_3_MAXIMUM_LIST_SIZE	Maximum size of multicast address list

802.3 Statistic OIDs

Article • 12/15/2021

The following table lists the 802.3 statistic OIDs for Remote NDIS Ethernet devices.

Support	OID	Description
Optional	OID_802_3_XMIT_DEFERRED	Frames transmitted after deferral
Optional	OID_802_3_XMIT_MAX_COLLISIONS	Frames not transmitted due to collisions
Optional	OID_802_3_RCV_OVERRUN	Frames not received due to overrun
Optional	OID_802_3_XMIT_UNDERRUN	Frames not transmitted due to underrun
Optional	OID_802_3_XMIT_HEARTBEAT_FAILURE	Frames transmitted with heartbeat failure
Optional	OID_802_3_XMIT_TIMES_CRS_LOST	Times carrier sense signal lost during transmission
Optional	OID_802_3_XMIT_LATE_COLLISIONS	Late collisions detected

Optional Power Management OIDs

Article • 12/15/2021

For NDIS to consider a device power-management -- aware, it must respond to the three power management OIDs listed in the following table. If the device returns a failure status code in response to a query for OID_PNP_CAPABILITIES, then the host will consider the device as not being power manageable. NDIS decides whether to query this OID based on the underlying bus technology that the Remote NDIS device is connected to. Some buses are power-manageable, such as USB, so it is expected that these types of Remote NDIS devices will support the minimal OIDs to be considered power-manageable.

Support	OID	Description
Optional	OID_PNP_CAPABILITIES	The NIC's Power Management abilities
Optional	OID_PNP_QUERY_POWER	A query to determine whether the device can transition to a specific power state.
Optional	OID_PNP_SET_POWER	A command to set the device to specified power state

Optional Network Wake Up OIDs

Article • 12/15/2021

To support network wake up events, a Remote NDIS device must additionally support the OID_PNP_ENABLE_WAKE_UP OID that is used by both the network protocols (TCP/IP) and NDIS to enable the wake up capabilities. Additionally, the options listed in the following table are available to enable specific types of wake up patterns. For further details, consult the Microsoft Windows 2000 Driver Development Kit (DDK).

Support	OID	Description
Optional	OID_PNP_ENABLE_WAKE_UP	The Remote NDIS device's wake-up capabilities that can be enabled
Optional	OID_PNP_ADD_WAKE_UP_PATTERN	Wake-up patterns that the Remote NDIS miniport driver should load into the device
Optional	OID_PNP_REMOVE_WAKE_UP_PATTERN	Wake-up patterns that the Remote NDIS miniport driver should remove from the device

Remote NDIS Version

Article • 03/14/2023

The following table defines the Remote NDIS protocol version identifiers exchanged between host and device.

Version Identifier	Value	Description
RNDIS_MAJOR_VERSION	1	Remote NDIS Major Version
RNDIS_MINOR_VERSION	0	Remote NDIS Minor Version

Status Values

Article • 12/15/2021

The Remote NDIS status values are generally equivalent to the 32-bit status values that are defined in the Microsoft Windows 2000 Driver Development Kit (DDK). The specific Remote NDIS status values used in this specification are listed below, others can be inferred from the Windows 2000 DDK or online documentation. A device may return any semantically correct Remote NDIS status value in a *Status* field of a message that it generates.

Status Identifier	Value	Description
RNDIS_STATUS_SUCCESS	0x00000000	Success
RNDIS_STATUS_FAILURE	0xC00000001	Unspecified error (equivalent to STATUS_UNSUCCESSFUL)
RNDIS_STATUS_INVALID_DATA	0xC0010015	Invalid data error
RNDIS_STATUS_NOT_SUPPORTED	0xC00000BB	Unsupported request error
RNDIS_STATUS_MEDIA_CONNECT	0x4001000B	Device is connected to network medium (equivalent to NDIS_STATUS_MEDIA_CONNECT from Windows 2000 DDK)
RNDIS_STATUS_MEDIA_DISCONNECT	0x4001000C	Device is disconnected from network medium (equivalent to NDIS_STATUS_MEDIA_DISCONNECT from Windows 2000 DDK)
RNDIS_STATUS_Xxx	...	Equal to NDIS_STATUS_Xxx values defined in Windows 2000 DDK or online documentation

Remote NDIS To USB Mapping overview

Article • 09/27/2024

A USB Remote NDIS device is implemented as a USB Communication Device Class (CDC) device with two interfaces. A Communication Class interface, of type Abstract Control, and a Data Class interface combine to form a single functional unit representing the USB Remote NDIS device. The Communication Class interface includes a single endpoint for event notification and uses the shared bidirectional Control endpoint for control messages. The Data Class interface includes two bulk endpoints for data traffic.

> ⓘ **Note**
>
> An understanding of the Universal Serial Bus (USB) Specification versions 1.1 and 2.0 is required. The USB Communication Device Class (CDC) Specifications are suggested as references. These documents can be found at https://www.usb.org.

This section includes the following topics:

USB-Level Initialization

USB-Level Termination

Control Channel Characteristics

Data Channel Characteristics

Power Management

Timer Constants

USB 802.3 Device Sample

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

USB-Level Initialization

Article • 12/15/2021

The host issues standard USB requests to obtain a set of standard USB descriptors for the device. The relevant portions of those descriptors are given below. See the USB Specification for generic USB device initialization.

USB Device Descriptor

Article • 12/15/2021

The device returns a USB Device Descriptor as defined in the USB Specification. The following table defines the prominent fields of the USB Device Descriptor.

Offset (bytes)	Field	Size (bytes)	Value	Description
4	bDeviceClass	1	02h	Communication Device Class code.
5	bDeviceSubClass	1	00h	Communication Device Subclass code, unused at this time.
6	bDeviceProtocol	1	00h	Communication Device Protocol code, unused at this time.

USB Configuration Descriptor

Article • 12/15/2021

The device returns a Configuration Descriptor as defined in the USB Specification. See the USB Specification for details.

Communication Class Interface

Article • 12/15/2021

The Communication Class interface is described by a USB interface descriptor, three class-specific descriptors, and an endpoint descriptor for the notification endpoint. The notification endpoint descriptor is a standard USB Interrupt-type IN endpoint descriptor whose **wMaxPacketSize** field is 8 bytes. The following table defines the prominent fields of the Communication Class interface descriptor.

Offset (bytes)	Field	Size (bytes)	Value	Description
5	bInterfaceClass	1	02h	Communication Interface Class code.
6	bInterfaceSubClass	1	02h	Communication Interface Class SubClass code for Abstract Control Model.
7	bInterfaceProtocol	1	FFh	Communication Interface Class Protocol code for vendor specific protocol.

Data Class Interface

Article • 12/15/2021

The Data Class interface is described by a standard USB Interface Descriptor followed by two endpoint descriptors. The two endpoint descriptors in the Data Class interface define standard USB Bulk-type endpoints: one Bulk-IN and one Bulk-OUT. The following table defines the prominent fields of the Data Class Interface Descriptor.

Offset (bytes)	Field	Size (bytes)	Value	Description
5	bInterfaceClass	1	0Ah	Data Interface Class code.
6	bInterfaceSubClass	1	00h	Data Class SubClass code.
7	bInterfaceProtocol	1	00h	Data Class Protocol code.

USB-Level Termination

Article • 12/15/2021

See the USB Specification for a description of generic USB bus-level termination.

Control Channel Characteristics

Article • 12/15/2021

The Control channel for the device is its USB Control endpoint. A control message from the host to the device is sent as a SEND_ENCAPSULATED_COMMAND transfer. This transfer is defined in the following table.

BmRequestType	bRequest	wValue	wIndex	wLength	Data
0x21	0x00	0x0000	*bInterfaceNumber* field of Communication Class interface descriptor	Byte length of control message block	Control message block

The host does not continuously poll the USB Control endpoint for input control messages. Upon placing a control message on its Control endpoint, the device must return a notification on the Communication Class interface's Interrupt IN endpoint, which is polled by the host whenever the device can return control messages. The transfer from the device's interrupt IN endpoint to the host is a standard USB Interrupt IN transfer. The only defined device notification is the RESPONSE_AVAILABLE notification, defined in the following table.

Offset (bytes)	Length (bytes)	Field	Data
0	4	Notification	RESPONSE_AVAILABLE (0x00000001)
4	4	Reserved	0

Upon receiving the RESPONSE_AVAILABLE notification, the host reads the control message from the Control endpoint using a GET_ENCAPSULATED_RESPONSE transfer, defined in the following table.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0xA1	0x01	0x0000	*bInterfaceNumber* field of Communication Class interface descriptor	0x0400 (this is the minimum byte length of the buffer posted by host)	Control message block

If for some reason the device receives a GET_ENCAPSULATED_RESPONSE and is unable to respond with a valid data on the Control endpoint, then it should return a one-byte packet set to 0x00, rather than stalling the Control endpoint.

Data Channel Characteristics

Article • 12/15/2021

The data channel for the device consists of the *Bulk* IN and OUT endpoints in the *Data Class* interface.

A single USB data transfer in either direction may consist of a single REMOTE_NDIS_PACKET_MSG or a longer multipacket message.

The USB transfer to send a data message from the host to the device is a standard USB bulk transfer to the Bulk OUT endpoint of the Data Class interface.

The USB transfer to send a data message from the device to the host is a standard USB bulk transfer from the Bulk IN endpoint of the Data Class interface. The host will read up to the number of bytes indicated by the *MaxTransferSize* field of REMOTE_NDIS_INITIALIZE_MSG, which will be no greater than 0x4000 bytes for a USB 1.1 device.

USB Short Packets

Article • 12/15/2021

USB passes data over the wire in the form of USB packets, which should not be confused with NDIS or networking packets. The maximum length of a USB packet to or from a USB endpoint is limited to the value of the **wMaxPacketSize** field of the endpoint's descriptor. For bulk pipes the maximum packet size is 64 bytes. Due to constraints of certain USB host controllers, there is a penalty associated with using short USB packets (for example, those of less then 64 bytes, when streaming data).

To work around this limitation, Remote NDIS USB devices may append zero-byte padding to data messages so that a short packet will not occur (within the constraints of the **MaxTransferSize** field of REMOTE_NDIS_INITIALIZE_MSG). The **MessageLength** field of the final REMOTE_NDIS_PACKET_MSG does not include these appended padding bytes.

If the device has transmitted its last available REMOTE_NDIS_PACKET_MSG (so no more are left in the device's queue), then it is acceptable to send a short USB packet.

If the last REMOTE_NDIS_PACKET_MSG of a device-send Remote NDIS data message (without any zero-byte padding) ends with a USB packet whose length is exactly the **wMaxPacketSize** for that endpoint, then the device may send an additional one-byte zero packet as an appended part of the transfer. Some device implementations are simplified by this allowance.

Similarly, some device-side USB chipsets do not detect the end of a received USB transfer that ends with a USB packet whose length is the **wMaxPacketSize** for that endpoint. For this reason, the host must append a one-byte zero packet to a data transfer that otherwise would have a length that is a multiple of the **wMaxPacketSize** of the receiving endpoint. USB Remote NDIS devices must tolerate the appended byte. The **MessageLength** field of the final REMOTE_NDIS_PACKET_MSG does not include this appended byte.

Flow Control

Article • 12/15/2021

Flow control for a USB Remote NDIS device is defined by the USB Specification.

Since all communication on USB is based on a host to device transaction, all the host must do to slow the flow of data is stop issuing IN tokens to the device on the bulk pipe. If the device needs to assert flow control, then it should NAK data transfers from the host until it is able to process data again. For a detailed explanation of this process, review Section 8.4.4 in the USB Specification, version 1.1.

USB Remote NDIS power management

Article • 03/14/2023

A USB Remote NDIS device must support the Power Management OIDs as well as Network wake-up OIDs that are listed in the Microsoft Windows 2000 Driver Development Kit (DDK) under NDIS OIDs. See the USB specification for a description of USB bus-level power management.

Timer Constants

Article • 12/15/2021

The **ControlTimeoutPeriod** for a USB Remote NDIS device is 10 seconds.

The **KeepAliveTimeoutPeriod** for a USB Remote NDIS device is 5 seconds.

USB 802.3 Device Sample

Article • 03/22/2024

This section contains a sample set of descriptors for a USB Remote NDIS Ethernet Device. It includes a CDC Communication Class interface and a CDC Data Class interface. The Device Descriptor is returned independently. The Configuration descriptor and all following descriptors are returned as a single block in the order shown.

Control messages are sent on the Control endpoint. Notification messages are sent on the Interrupt In endpoint in the CDC Communication Class interface. Data messages are sent on the Bulk In and Bulk Out endpoints in the CDC Data Class interface. String descriptors are not shown.

The Remote NDIS implementation in Windows Millennium Edition assumes that the Communication Class interface precedes the Data Class interface. Vendors should choose this descriptor ordering so that devices initialize correctly on Windows Millennium Edition.

If any portion of this sample contradicts a controlling specification, follow the specification.

This sample in this section includes:

Device Descriptor

Configuration Descriptor

Interface Descriptor for Communication Class Interface

Notification Endpoint Descriptor

Interface Descriptor for Data Class Interface

Data In Endpoint Descriptor

Data Out Endpoint Descriptor

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Device Descriptor

Article • 12/15/2021

Offset	Field	Size	Value	Description
0	bLength	1	0x12	Size of this descriptor, in bytes
1	bDescriptorType	1	0x01	DEVICE descriptor
2	bcdUSB	2	0x0110	1.1 - current revision of USB spec
4	bDeviceClass	1	0x02	Communication Device Class
5	bDeviceSubClass	1	0x00	Unused
6	bDeviceProtocol	1	0x00	Unused
7	bMaxPacketSize0	1	0x08	Max packet size on control pipe
8	idVendor	2	0xXXXX	Vendor ID
10	idProduct	2	0xXXXX	Product ID
12	bcdDevice	2	0xXXXX	Device Release Code
14	iManufacturer	1	0x01	Index of manufacturer string
15	iProduct	1	0x02	Index of product string
16	iSerialNumber	1	0x03	Index of device serial number string
17	bNumConfigurations	1	0x01	One configuration

Configuration Descriptor

Article • 12/15/2021

Offset	Field	Size	Value	Description
0	bLength	1	0x09	Size of this descriptor, in bytes
1	bDescriptorType	1	0x02	CONFIGURATION descriptor
2	wTotalLength	2	0x003E	Length of the total configuration block, including this descriptor and all following descriptors, in bytes
4	bNumInterfaces	1	0x02	Two interfaces
5	bConfigurationValue	1	0x01	ID of this configuration
6	iConfiguration	1	0x00	Unused
7	bmAttributes	1	0x80	Bus Powered
8	MaxPower	1	0x64	200 mA

Interface Descriptor for Communication Class Interface

Article • 12/15/2021

Offset	Field	Size	Value	Description
1	bDescriptorType	1	0x04	INTERFACE descriptor
2	bInterfaceNumber	1	0x00	Index of this interface
3	bAlternateSetting	1	0x00	Index of this setting
4	bNumEndpoints	1	0x01	1 endpoint
5	bInterfaceClass	1	0x02	Communication Class
6	bInterfaceSubclass	1	0x02	Abstract Control Model
7	bInterfaceProtocol	1	0xFF	Vendor-specific protocol
8	iInterface	1	0x00	Unused

Notification Endpoint Descriptor

Article • 12/15/2021

Offset	Field	Size	Value	Description
0	bLength	1	0x07	Size of this descriptor, in bytes
1	bDescriptorType	1	0x05	ENDPOINT descriptor
2	bEndpointAddress	1	0x81	Endpoint 1 IN
3	bmAttributes	1	0x03	Interrupt Endpoint
4	wMaxPacketSize	2	0x0008	8 byte maximum packet size
6	bInterval	1	0x01	Polling interval

Interface Descriptor for Data Class Interface

Article • 12/15/2021

Offset	Field	Size	Value	Description
0	bLength	1	0x09	Size of this descriptor, in bytes
1	bDescriptorType	1	0x04	INTERFACE descriptor
2	bInterfaceNumber	1	0x01	Index of this interface
3	bAlternateSetting	1	0x00	Index of this setting
4	bNumEndpoints	1	0x02	2 endpoints
5	bInterfaceClass	1	0x0A	Data Class
6	bInterfaceSubclass	1	0x00	Unused
7	bInterfaceProtocol	1	0x00	Unused
8	iInterface	1	0x00	Unused

Data In Endpoint Descriptor

Article • 12/15/2021

Offset	Field	Size	Value	Description
0	bLength	1	0x07	Size of this descriptor, in bytes
1	bDescriptorType	1	0x05	ENDPOINT descriptor
2	bEndpointAddress	1	0x82	Endpoint 2 IN
3	bmAttributes	1	0x02	Bulk Endpoint
4	wMaxPacketSize	2	0x0040	64 byte maximum packet size
6	bInterval	1	0x00	Unused

Data Out Endpoint Descriptor

Article • 12/15/2021

Offset	Field	Size	Value	Description
0	bLength	1	0x07	Size of this descriptor, in bytes
1	bDescriptorType	1	0x05	ENDPOINT descriptor
2	bEndpointAddress	1	0x03	Endpoint 3 OUT
3	bmAttributes	1	0x02	Bulk Endpoint
4	wMaxPacketSize	2	0x0040	64 byte maximum packet size
6	bInterval	1	0x00	Unused

Kernel Mode SDK Topics for Network Drivers

Article • 12/15/2021

This section lists header files and topics for kernel mode Windows network drivers. The header files in this section are included in the Windows Software Development Kit (SDK) instead of the Windows Driver Kit (WDK) as they are also shared with user mode networking applications.

ⓘ **Important**

This section's header topics contains pages for definitions, OIDs, status indications, and other data structures that are not part of network driver reference. Reference topics include structures, enumerations, functions, and callbacks.

For more information about network driver reference for these headers, see **Network driver reference in SDK header files**.

This section contains:

- Mstcpip.h
- Ntddndis.h
- Ws2def.h

Mstcpip.h

Article • 12/15/2021

This section contains kernel mode network driver topics for the Mstcpip.h header. This header is included in the Windows SDK as it is also shared with user mode networking applications.

The Mstcpip.h header contains definitions for Microsoft-specific extensions to the core Winsock definitions.

In this section

- SIO_LOOPBACK_FAST_PATH control code
- SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT control code
- SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS control code
- SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS control code

SIO_LOOPBACK_FAST_PATH control code

Article • 03/03/2023

Important The **SIO_LOOPBACK_FAST_PATH** is deprecated and is not recommended to be used in your code.

The **SIO_LOOPBACK_FAST_PATH** socket I/O control code allows a WSK application to configure a TCP socket for faster operations on the loopback interface.

To use this IOCTL, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	**SIO_LOOPBACK_FAST_PATH**
Level	0
InputSize	The size, in bytes, of the input buffer.
InputBuffer	A pointer to the input buffer. This parameter contains a pointer to a **Boolean** value that indicates if the socket should be configured for fast loopback operations.
OutputSize	0
OutputBuffer	**NULL**
OutputSizeReturned	**NULL**
Irp	A pointer to an IRP.

An application can use the **SIO_LOOPBACK_FAST_PATH** IOCTL to improve the performance of loopback operations on a TCP socket. This IOCTL requests that the TCP/IP stack use a special fast path for loopback operations on this socket. The **SIO_LOOPBACK_FAST_PATH** IOCTL can be used only with TCP sockets. This IOCTL must be used on both sides of the loopback session. The TCP loopback fast path is supported using either the IPv4 or IPv6 loopback interface.

The socket that plans to initiate the connection request must apply this IOCTL before making the connection request. The socket that is listening for the connection request must apply this IOCTL before accepting the connection.

Once an application establishes the connection on a loopback interface using the fast path, all packets for the lifetime of the connection must use the fast path.

Applying **SIO_LOOPBACK_FAST_PATH** to a socket which will be connected to a non-loopback path will have no effect.

This TCP loopback optimization results in packets that flow through Transport Layer (TL) instead of the traditional loopback through Network Layer. This optimization improves the latency for loopback packets. Once an applications opts in for a connection level setting to use the loopback fast path, all packets will follow the loopback path. For loopback communications, congestion and packet drop are not expected. The notion of congestion control and reliable delivery in TCP will be unnecessary. This, however, is not true for flow control. Without flow control, the sender can overwhelm the receive buffer, leading to erroneous TCP loopback behavior. The flow control in the TCP optimized loopback path is maintained by placing send requests in a queue. When the receive buffer is full, the TCP/IP stack guarantees that the sends won't complete until the queue is serviced, maintaining flow control.

TCP fast path loopback connections in the presence of a Windows Filtering Platform (WFP) callout for connection data must take the unoptimized slow path for loopback. So WFP filters will prevent this new loopback fast path from being used. When a WFP filter is enabled, the system will use the slow path even if the **SIO_LOOPBACK_FAST_PATH** IOCTL was set. This ensues that user-mode applications have the full WFP security capability.

By default, **SIO_LOOPBACK_FAST_PATH** is disabled.

Only a subset of the TCP/IP socket options are supported when the **SIO_LOOPBACK_FAST_PATH** IOCTL is used to enable the loopback fast path on a socket. The list of supported options includes the following:

- IP_TTL
- IP_UNICAST_IF
- IPV6_UNICAST_HOPS
- IPV6_UNICAST_IF
- IPV6_V6ONLY
- **SO_CONDITIONAL_ACCEPT**
- SO_EXCLUSIVEADDRUSE
- **SO_PORT_SCALABILITY**
- SO_RCVBUF
- SO_REUSEADDR

- TCP_BSDURGENT

A WSK application must specify a pointer to an IRP and a completion routine when calling the **WskControlSocket** function for this type of request. The application must not release the buffer till the WSK subsystem has completed the IRP. When it completes the IRP, the subsystem invokes the completion routine. In the completion routine, the application must check the IRP status and release all resources that it had previously allocated for the request.

For more information about WSK IRP handling, see Using IRPs with Winsock Kernel Functions.

When completing the IRP, the subsystem will set *Irp->IoStatus.Status* to **STATUS_SUCCESS** if the request is successful. Otherwise, *Irp->IoStatus.Status* will be set to **STATUS_INVALID_BUFFER_SIZE** or **STATUS_NOT_SUPPORTED** if the call is not successful.

Return value

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Header	Mstcpip.h
IRQL	PASSIVE_LEVEL

See also

SIO_LOOPBACK_FAST_PATH (SDK)

Using IRPs with Winsock Kernel Functions

SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT control code

Article • 03/03/2023

The **SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT** socket I/O control operation allows a Winsock client to retrieve the redirect context for a redirect record for a redirected connection.

A WFP redirect record is a buffer of opaque data that WFP must set on an outbound proxy connection so that the redirected connection and the original connection are logically related.

Note The SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS query can only be used if the connection was redirected at the **FWPS_LAYER_ALE_CONNECT_REDIRECT_V4** or **FWPS_LAYER_ALE_CONNECT_REDIRECT_V6** layer by a WFP client.

For more information about redirection, see Using Bind or Connect Redirection.

To query the redirect context for a redirect record, a Winsock client calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	**SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT**
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	The size, in bytes, of the buffer that is pointed to by the *OutputBuffer* parameter.
OutputBuffer	A pointer to the buffer that receives the redirect context for the redirect record for the accepted TCP connection. The size of the buffer is specified in the *OutputSize* parameter.
OutputSizeReturned	A pointer to a **ULONG**-typed variable that receives the number of bytes of data that is copied into the buffer that is pointed to by the *OutputBuffer* parameter.

Parameter	Value
Irp	A pointer to an IRP.

The caller can perform this query in either of the following ways:

- It can set the *OutputBuffer* to a large buffer approximately 1 KB in size. If the output buffer size is not large enough, WskControlSocket will return **STATUS_BUFFER_TOO_SMALL** and *OutputSizeReturned* will contain the required size of the buffer. A larger buffer can then be allocated and **WskControlSocket** called again with the **SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT** request and *OutputBuffer* set to the larger buffer.
- Or it can set the *OutputSize* parameter to 0 and the *OutputBuffer* to NULL and then call WskControlSocket. Upon completion, the **WskControlSocket** function retrieves the output buffer size, in bytes, in the *OutputSizeReturned* parameter. An appropriately sized buffer can then be allocated and **WskControlSocket** called again with the **SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT** request and *OutputBuffer* set to the buffer.

Note It is also possible to perform this query in a user-mode application by using SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT (SDK).

For this type of request, the Winsock client must specify a pointer to an IRP and a pointer to its completion routine. The IRP can be passed to the client by a higher driver or the client can choose to allocate the IRP. To specify the completion routine, the client must call IoSetCompletionRoutine. For more details, see Using IRPs with Winsock Kernel Functions.

The Winsock client must not free the allocated buffer till the IRP is completed by WSK subsystem. When the WSK subsystem completes the IRP, it notifies the client by invoking the completion routine. A reference to that buffer is passed to the client by the WSK subsystem in the *Context* parameter of the completion routine. The size of the buffer is stored in *Irp->IoStatus.Information*.

The client can get the status of the IRP by checking *Irp->IoStatus.Status*. *Irp->IoStatus.Status* will be set to **STATUS_SUCCESS** if the request is successful. Otherwise, it will contain **STATUS_INTEGER_OVERFLOW**, **STATUS_NOT_FOUND**, **STATUS_BUFFER_TOO_SMALL**, or **STATUS_ACCESS_DENIED** if the call is not successful.

Requirements

Minimum supported client	Windows 8

Minimum supported server	Windows Server 2012
Header	Mstcpip.h
IRQL	PASSIVE_LEVEL

See also

Using Bind or Connect Redirection

Using IRPs with Winsock Kernel Functions

SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS

SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT (SDK)

SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS control code

Article • 03/03/2023

The **SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS** socket I/O control operation allows a Winsock client to retrieve the redirect record for a redirected connection.

A WFP redirect record is a buffer of opaque data that WFP must set on an outbound proxy connection so that the redirected connection and the original connection are logically related.

Note The **SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS** query can only be used if the connection was redirected at the **FWPS_LAYER_ALE_CONNECT_REDIRECT_V4** or **FWPS_LAYER_ALE_CONNECT_REDIRECT_V6** layer by a WFP client.

For more information about redirection, see Using Bind or Connect Redirection.

To query the redirect record for the redirected connection, a Winsock client calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	**SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS**
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	The size, in bytes, of the buffer that is pointed to by the *OutputBuffer* parameter.
OutputBuffer	A pointer to the buffer that receives the redirect record for the accepted TCP connection. The size of the buffer is specified in the *OutputSize* parameter.
OutputSizeReturned	A pointer to a **ULONG**-typed variable that receives the number of bytes of data that is copied into the buffer that is pointed to by the *OutputBuffer* parameter.
Irp	A pointer to an IRP.

The caller can perform this query in either of the following ways:

- It can set the *OutputBuffer* to a large buffer approximately 1 KB in size. If the output buffer size is not large enough, **WskControlSocket** will return a **STATUS_BUFFER_TOO_SMALL** and *OutputSizeReturned* will contain the required size of the buffer. A larger buffer can then be allocated and **WskControlSocket** called again with the **SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS** request and *OutputBuffer* set to the larger buffer.
- Or it can set the *OutputSize* parameter to 0 and the *OutputBuffer* to NULL and then call **WskControlSocket**. Upon completion, the **WskControlSocket** function retrieves the output buffer size, in bytes, in the *OutputSizeReturned* parameter. An appropriately sized buffer can then be allocated and **WskControlSocket** called again with the **SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS** request and *OutputBuffer* set to the buffer.

Note It is also possible to perform this query in a user-mode application by using SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS (SDK).

For this type of request, the Winsock client must specify a pointer to an IRP and a pointer to its completion routine. The IRP can be passed to the client by a higher driver or the client can choose to allocate the IRP. To specify the completion routine, the client must call **IoSetCompletionRoutine**. For more details, see Using IRPs with Winsock Kernel Functions.

The Winsock client must not free the allocated buffer till the IRP is completed by WSK subsystem. When the WSK subsystem completes the IRP, it notifies the client by invoking the completion routine. A reference to that buffer is passed to the client by the WSK subsystem in the *Context* parameter of the completion routine. The size of the buffer is stored in *Irp->IoStatus.Information*.

The client can get the status of the IRP by checking *Irp->IoStatus.Status*. *Irp->IoStatus.Status* will be set to **STATUS_SUCCESS** if the request is successful. Otherwise, it will contain **STATUS_INTEGER_OVERFLOW**, **STATUS_NOT_FOUND**, **STATUS_BUFFER_TOO_SMALL**, or **STATUS_ACCESS_DENIED** if the call is not successful.

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Header	Mstcpip.h

IRQL	PASSIVE_LEVEL

See also

Using Bind or Connect Redirection

Using IRPs with Winsock Kernel Functions

SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT

SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS (SDK)

SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS

SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS control code

Article • 03/03/2023

The **SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS** socket I/O control operation allows a Winsock client to specify the redirect record to the new TCP socket used for connecting to the final destination.

A WFP redirect record is a buffer of opaque data that WFP must set on an outbound proxy connection so that the redirected connection and the original connection are logically related.

For more information about redirection, see Using Bind or Connect Redirection.

To set the redirect record to the new TCP socket used for connecting to the final destination, a Winsock client calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	**SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS**
Level	0
InputSize	The size of the redirect record pointed to by the InputBuffer parameter.
InputBuffer	A pointer to the redirect record associated with the socket.
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL
Irp	A pointer to an IRP.

The Winsock client must allocate a buffer and specify a pointer to the buffer and its size in *InputBuffer* and *InputSize*.

A Winsock client must specify a pointer to an IRP and a completion routine when calling the **WskControlSocket** function for this type of request. The client must not release the buffer till the WSK subsystem has completed the IRP. When it completes the IRP, the subsystem invokes the completion routine. In the completion routine, the client must check the IRP status and release all resources that it had previously allocated for the request.

Note It is also possible to perform this query in a user-mode application by using SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS (SDK).

For more information about WSK IRP handling, see Using IRPs with Winsock Kernel Functions.

The client can get the status of the IRP by checking *Irp->IoStatus.Status*. *Irp->IoStatus.Status* will be set to **STATUS_SUCCESS** if the request is successful. Otherwise, it will contain **STATUS_INTEGER_OVERFLOW**, or **STATUS_ACCESS_DENIED** if the call is not successful.

Requirements

Minimum supported client	Windows 8
Minimum supported server	Windows Server 2012
Header	Mstcpip.h
IRQL	PASSIVE_LEVEL

See also

Using Bind or Connect Redirection

Using IRPs with Winsock Kernel Functions

SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS

SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS (SDK)

Ntddndis.h

Article • 12/15/2021

ⓘ Important

Bias-free communication

Microsoft supports a diverse and inclusive environment. This article contains references to terminology that the Microsoft **style guide for bias-free communication** recognizes as exclusionary. The word or phrase is used in this article for consistency because it currently appears in the software. When the software is updated to remove the language, this article will be updated to be in alignment.

This section contains kernel mode network driver topics for the Ntddndis.h header. This header is included in the Windows SDK as it is also shared with user mode networking applications.

The Ntddndis.h header contains definitions for constants and types for interfacing with network drivers.

ⓘ Note

This section's topics contains pages for definitions and OIDs, which are not part of network driver DDI reference.

For DDI reference for this header, see **ntddndis.h header**.

In this section

- GUID_NDIS_GEN_PCI_DEVICE_CUSTOM_PROPERTIES
- OID_802_3_ADD_MULTICAST_ADDRESS
- OID_802_3_CURRENT_ADDRESS
- OID_802_3_DELETE_MULTICAST_ADDRESS
- OID_802_3_MAC_OPTIONS
- OID_802_3_MAXIMUM_LIST_SIZE
- OID_802_3_MULTICAST_LIST
- OID_802_3_PERMANENT_ADDRESS

- OID_802_3_RCV_OVERRUN
- OID_802_3_XMIT_DEFERRED
- OID_802_3_XMIT_HEARTBEAT_FAILURE
- OID_802_3_XMIT_LATE_COLLISIONS
- OID_802_3_XMIT_MAX_COLLISIONS
- OID_802_3_XMIT_TIMES_CRS_LOST
- OID_802_3_XMIT_UNDERRUN
- OID_CO_ADD_ADDRESS
- OID_CO_ADD_PVC
- OID_CO_ADDRESS_CHANGE
- OID_CO_AF_CLOSE
- OID_CO_DELETE_ADDRESS
- OID_CO_DELETE_PVC
- OID_CO_GET_ADDRESSES
- OID_CO_GET_CALL_INFORMATION
- OID_CO_SIGNALING_DISABLED
- OID_CO_SIGNALING_ENABLED
- OID_CO_TAPI_ADDRESS_CAPS
- OID_CO_TAPI_CM_CAPS
- OID_CO_TAPI_GET_CALL_DIAGNOSTICS
- OID_CO_TAPI_LINE_CAPS
- OID_CO_TAPI_TRANSLATE_NDIS_CALLPARAMS
- OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS
- OID_CO_TAPI_TRANSLATE_TAPI_SAP
- OID_GEN_ADMIN_STATUS
- OID_GEN_ALIAS
- OID_GEN_BROADCAST_BYTES_RCV
- OID_GEN_BROADCAST_BYTES_XMIT
- OID_GEN_BROADCAST_FRAMES_RCV
- OID_GEN_BROADCAST_FRAMES_XMIT
- OID_GEN_BYTES_RCV
- OID_GEN_BYTES_XMIT
- OID_GEN_CO_BYTES_RCV
- OID_GEN_CO_BYTES_XMIT
- OID_GEN_CO_BYTES_XMIT_OUTSTANDING
- OID_GEN_CO_DRIVER_VERSION
- OID_GEN_CO_GET_NETCARD_TIME
- OID_GEN_CO_GET_TIME_CAPS
- OID_GEN_CO_HARDWARE_STATUS
- OID_GEN_CO_LINK_SPEED

- OID_GEN_CO_MAC_OPTIONS
- OID_GEN_CO_MEDIA_CONNECT_STATUS
- OID_GEN_CO_MEDIA_IN_USE
- OID_GEN_CO_MEDIA_SUPPORTED
- OID_GEN_CO_MINIMUM_LINK_SPEED
- OID_GEN_CO_NETCARD_LOAD
- OID_GEN_CO_PROTOCOL_OPTIONS
- OID_GEN_CO_RCV_CRC_ERROR
- OID_GEN_CO_RCV_PDUS_ERROR
- OID_GEN_CO_RCV_PDUS_NO_BUFFER
- OID_GEN_CO_RCV_PDUS_OK
- OID_GEN_CO_SUPPORTED_GUIDS
- OID_GEN_CO_SUPPORTED_LIST
- OID_GEN_CO_TRANSMIT_QUEUE_LENGTH
- OID_GEN_CO_VENDOR_DESCRIPTION
- OID_GEN_CO_VENDOR_DRIVER_VERSION
- OID_GEN_CO_VENDOR_ID
- OID_GEN_CO_XMIT_PDUS_ERROR
- OID_GEN_CO_XMIT_PDUS_OK
- OID_GEN_CURRENT_LOOKAHEAD
- OID_GEN_CURRENT_PACKET_FILTER
- OID_GEN_DEVICE_PROFILE
- OID_GEN_DIRECTED_BYTES_RCV
- OID_GEN_DIRECTED_BYTES_XMIT
- OID_GEN_DIRECTED_FRAMES_RCV
- OID_GEN_DIRECTED_FRAMES_XMIT
- OID_GEN_DISCONTINUITY_TIME
- OID_GEN_DRIVER_VERSION
- OID_GEN_ENUMERATE_PORTS
- OID_GEN_FRIENDLY_NAME
- OID_GEN_HARDWARE_STATUS
- OID_GEN_HD_SPLIT_CURRENT_CONFIG
- OID_GEN_HD_SPLIT_PARAMETERS
- OID_GEN_INIT_TIME_MS
- OID_GEN_INTERFACE_INFO
- OID_GEN_INTERRUPT_MODERATION
- OID_GEN_ISOLATION_PARAMETERS
- OID_GEN_LAST_CHANGE
- OID_GEN_LINK_PARAMETERS
- OID_GEN_LINK_SPEED

- OID_GEN_LINK_SPEED_EX
- OID_GEN_LINK_STATE
- OID_GEN_MAC_OPTIONS
- OID_GEN_MACHINE_NAME
- OID_GEN_MAX_LINK_SPEED
- OID_GEN_MAXIMUM_FRAME_SIZE
- OID_GEN_MAXIMUM_LOOKAHEAD
- OID_GEN_MAXIMUM_SEND_PACKETS
- OID_GEN_MAXIMUM_TOTAL_SIZE
- OID_GEN_MEDIA_CAPABILITIES
- OID_GEN_MEDIA_CONNECT_STATUS
- OID_GEN_MEDIA_CONNECT_STATUS_EX
- OID_GEN_MEDIA_DUPLEX_STATE
- OID_GEN_MEDIA_IN_USE
- OID_GEN_MEDIA_SENSE_COUNTS
- OID_GEN_MEDIA_SUPPORTED
- OID_GEN_MINIPORT_RESTART_ATTRIBUTES
- OID_GEN_MULTICAST_BYTES_RCV
- OID_GEN_MULTICAST_BYTES_XMIT
- OID_GEN_MULTICAST_FRAMES_RCV
- OID_GEN_MULTICAST_FRAMES_XMIT
- OID_GEN_NDIS_RESERVED_1
- OID_GEN_NDIS_RESERVED_2
- OID_GEN_NDIS_RESERVED_5
- OID_GEN_NETWORK_LAYER_ADDRESSES
- OID_GEN_OPERATIONAL_STATUS
- OID_GEN_PCI_DEVICE_CUSTOM_PROPERTIES
- OID_GEN_PHYSICAL_MEDIUM
- OID_GEN_PHYSICAL_MEDIUM_EX
- OID_GEN_PORT_AUTHENTICATION_PARAMETERS
- OID_GEN_PORT_STATE
- OID_GEN_PROMISCUOUS_MODE
- OID_GEN_PROTOCOL_OPTIONS
- OID_GEN_RCV_CRC_ERROR
- OID_GEN_RCV_DISCARDS
- OID_GEN_RCV_ERROR
- OID_GEN_RCV_LINK_SPEED
- OID_GEN_RCV_NO_BUFFER
- OID_GEN_RCV_OK
- OID_GEN_RECEIVE_BLOCK_SIZE

- OID_GEN_RECEIVE_BUFFER_SPACE
- OID_GEN_RECEIVE_HASH
- OID_GEN_RECEIVE_SCALE_CAPABILITIES
- OID_GEN_RECEIVE_SCALE_PARAMETERS
- OID_GEN_RECEIVE_SCALE_PARAMETERS_V2
- OID_GEN_RESET_COUNTS
- OID_GEN_RNDIS_CONFIG_PARAMETER
- OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES
- OID_GEN_STATISTICS
- OID_GEN_SUPPORTED_GUIDS
- OID_GEN_SUPPORTED_LIST
- OID_GEN_TRANSMIT_BLOCK_SIZE
- OID_GEN_TRANSMIT_BUFFER_SPACE
- OID_GEN_TRANSMIT_QUEUE_LENGTH
- OID_GEN_TRANSPORT_HEADER_OFFSET
- OID_GEN_UNKNOWN_PROTOS
- OID_GEN_VENDOR_DESCRIPTION
- OID_GEN_VENDOR_DRIVER_VERSION
- OID_GEN_VENDOR_ID
- OID_GEN_VLAN_ID
- OID_GEN_XMIT_DISCARDS
- OID_GEN_XMIT_ERROR
- OID_GEN_XMIT_LINK_SPEED
- OID_GEN_XMIT_OK
- OID_IP4_OFFLOAD_STATS
- OID_IP6_OFFLOAD_STATS
- OID_NDK_CONNECTIONS
- OID_NDK_LOCAL_ENDPOINTS
- OID_NDK_SET_STATE
- OID_NDK_STATISTICS
- OID_NIC_SWITCH_ALLOCATE_VF
- OID_NIC_SWITCH_CREATE_SWITCH
- OID_NIC_SWITCH_CREATE_VPORT
- OID_NIC_SWITCH_CURRENT_CAPABILITIES
- OID_NIC_SWITCH_DELETE_SWITCH
- OID_NIC_SWITCH_DELETE_VPORT
- OID_NIC_SWITCH_ENUM_SWITCHES
- OID_NIC_SWITCH_ENUM_VFS
- OID_NIC_SWITCH_ENUM_VPORTS
- OID_NIC_SWITCH_FREE_VF

- OID_NIC_SWITCH_HARDWARE_CAPABILITIES
- OID_NIC_SWITCH_PARAMETERS
- OID_NIC_SWITCH_VF_PARAMETERS
- OID_NIC_SWITCH_VPORT_PARAMETERS
- OID_OFFLOAD_ENCAPSULATION
- OID_PACKET_COALESCING_FILTER_MATCH_COUNT
- OID_PD_CLOSE_PROVIDER
- OID_PD_OPEN_PROVIDER
- OID_PD_QUERY_CURRENT_CONFIG
- OID_PM_ADD_PROTOCOL_OFFLOAD
- OID_PM_ADD_WOL_PATTERN
- OID_PM_CURRENT_CAPABILITIES
- OID_PM_GET_PROTOCOL_OFFLOAD
- OID_PM_HARDWARE_CAPABILITIES
- OID_PM_PARAMETERS
- OID_PM_PROTOCOL_OFFLOAD_LIST
- OID_PM_REMOVE_PROTOCOL_OFFLOAD
- OID_PM_REMOVE_WOL_PATTERN
- OID_PM_WOL_PATTERN_LIST
- OID_PNP_ADD_WAKE_UP_PATTERN
- OID_PNP_CAPABILITIES
- OID_PNP_ENABLE_WAKE_UP
- OID_PNP_QUERY_POWER
- OID_PNP_REMOVE_WAKE_UP_PATTERN
- OID_PNP_SET_POWER
- OID_PNP_WAKE_UP_ERROR
- OID_PNP_WAKE_UP_OK
- OID_PNP_WAKE_UP_PATTERN_LIST
- OID_QOS_CURRENT_CAPABILITIES
- OID_QOS_HARDWARE_CAPABILITIES
- OID_QOS_OFFLOAD_CREATE_SQ
- OID_QOS_OFFLOAD_CURRENT_CAPABILITIES
- OID_QOS_OFFLOAD_DELETE_SQ
- OID_QOS_OFFLOAD_ENUM_SQS
- OID_QOS_OFFLOAD_HARDWARE_CAPABILITIES
- OID_QOS_OFFLOAD_SQ_STATS
- OID_QOS_OFFLOAD_UPDATE_SQ
- OID_QOS_OPERATIONAL_PARAMETERS
- OID_QOS_PARAMETERS
- OID_QOS_REMOTE_PARAMETERS

- OID_RECEIVE_FILTER_ALLOCATE_QUEUE
- OID_RECEIVE_FILTER_CLEAR_FILTER
- OID_RECEIVE_FILTER_CURRENT_CAPABILITIES
- OID_RECEIVE_FILTER_ENUM_FILTERS
- OID_RECEIVE_FILTER_ENUM_QUEUES
- OID_RECEIVE_FILTER_FREE_QUEUE
- OID_RECEIVE_FILTER_GLOBAL_PARAMETERS
- OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES
- OID_RECEIVE_FILTER_MOVE_FILTER
- OID_RECEIVE_FILTER_PARAMETERS
- OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE
- OID_RECEIVE_FILTER_QUEUE_PARAMETERS
- OID_RECEIVE_FILTER_SET_FILTER
- OID_SRIOV_BAR_RESOURCES
- OID_SRIOV_CURRENT_CAPABILITIES
- OID_SRIOV_HARDWARE_CAPABILITIES
- OID_SRIOV_PF_LUID
- OID_SRIOV_PROBED_BARS
- OID_SRIOV_READ_VF_CONFIG_BLOCK
- OID_SRIOV_READ_VF_CONFIG_SPACE
- OID_SRIOV_RESET_VF
- OID_SRIOV_SET_VF_POWER_STATE
- OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK
- OID_SRIOV_VF_SERIAL_NUMBER
- OID_SRIOV_VF_VENDOR_DEVICE_ID
- OID_SRIOV_WRITE_VF_CONFIG_BLOCK
- OID_SRIOV_WRITE_VF_CONFIG_SPACE
- OID_SWITCH_FEATURE_STATUS_QUERY
- OID_SWITCH_NIC_ARRAY
- OID_SWITCH_NIC_CONNECT
- OID_SWITCH_NIC_CREATE
- OID_SWITCH_NIC_DELETE
- OID_SWITCH_NIC_DISCONNECT
- OID_SWITCH_NIC_REQUEST
- OID_SWITCH_NIC_RESTORE
- OID_SWITCH_NIC_RESTORE_COMPLETE
- OID_SWITCH_NIC_SAVE
- OID_SWITCH_NIC_SAVE_COMPLETE
- OID_SWITCH_NIC_UPDATED
- OID_SWITCH_PARAMETERS

- OID_SWITCH_PORT_ARRAY
- OID_SWITCH_PORT_CREATE
- OID_SWITCH_PORT_DELETE
- OID_SWITCH_PORT_FEATURE_STATUS_QUERY
- OID_SWITCH_PORT_PROPERTY_ADD
- OID_SWITCH_PORT_PROPERTY_DELETE
- OID_SWITCH_PORT_PROPERTY_ENUM
- OID_SWITCH_PORT_PROPERTY_UPDATE
- OID_SWITCH_PORT_TEARDOWN
- OID_SWITCH_PORT_UPDATED
- OID_SWITCH_PROPERTY_ADD
- OID_SWITCH_PROPERTY_DELETE
- OID_SWITCH_PROPERTY_ENUM
- OID_SWITCH_PROPERTY_UPDATE
- OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG
- OID_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES
- OID_TCP_CONNECTION_OFFLOAD_PARAMETERS
- OID_TCP_OFFLOAD_CURRENT_CONFIG
- OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES
- OID_TCP_OFFLOAD_PARAMETERS
- OID_TCP_RSC_STATISTICS
- OID_TCP_TASK_IPSEC_ADD_SA
- OID_TCP_TASK_IPSEC_ADD_UDPESP_SA
- OID_TCP_TASK_IPSEC_DELETE_SA
- OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA
- OID_TCP_TASK_IPSEC_OFFLOAD_V2_UPDATE_SA
- OID_TCP_TASK_OFFLOAD
- OID_TCP4_OFFLOAD_STATS
- OID_TCP6_OFFLOAD_STATS
- OID_TUNNEL_INTERFACE_RELEASE_OID
- OID_TUNNEL_INTERFACE_SET_OID
- OID_WAN_CO_GET_COMP_INFO
- OID_WAN_CO_GET_INFO
- OID_WAN_CO_GET_LINK_INFO
- OID_WAN_CO_GET_STATS_INFO
- OID_WAN_CO_SET_COMP_INFO
- OID_WAN_CO_SET_LINK_INFO

- OID_WWAN_AUTH_CHALLENGE
- OID_WWAN_BASE_STATIONS_INFO
- OID_WWAN_CONNECT
- OID_WWAN_CREATE_MAC
- OID_WWAN_DELETE_MAC
- OID_WWAN_DEVICE_CAPS
- OID_WWAN_DEVICE_CAPS_EX
- OID_WWAN_DEVICE_RESET
- OID_WWAN_DEVICE_SERVICE_COMMAND
- OID_WWAN_DEVICE_SERVICE_SESSION
- OID_WWAN_DEVICE_SERVICE_SESSION_WRITE
- OID_WWAN_DEVICE_SERVICES
- OID_WWAN_DEVICE_SLOT_MAPPING_INFO
- OID_WWAN_DRIVER_CAPS
- OID_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS
- OID_WWAN_ENUMERATE_DEVICE_SERVICES
- OID_WWAN_LTE_ATTACH_CONFIG
- OID_WWAN_LTE_ATTACH_STATUS
- OID_WWAN_HOME_PROVIDER
- OID_WWAN_MODEM_CONFIG_INFO
- OID_WWAN_MODEM_LOGGING_CONFIG
- OID_WWAN_MPDP
- OID_WWAN_NETWORK_BLACKLIST
- OID_WWAN_NETWORK_IDLE_HINT
- OID_WWAN_NITZ
- OID_WWAN_PACKET_SERVICE
- OID_WWAN_PCO
- OID_WWAN_PIN
- OID_WWAN_PIN_EX
- OID_WWAN_PIN_EX2
- OID_WWAN_PIN_LIST
- OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS
- OID_WWAN_PREFERRED_PROVIDERS
- OID_WWAN_PRESHUTDOWN
- OID_WWAN_PROVISIONED_CONTEXTS
- OID_WWAN_RADIO_STATE
- OID_WWAN_READY_INFO
- OID_WWAN_REGISTER_STATE
- OID_WWAN_SAR_CONFIG
- OID_WWAN_SAR_TRANSMISSION_STATUS

- OID_WWAN_SERVICE_ACTIVATION
- OID_WWAN_SIGNAL_STATE
- OID_WWAN_SLOT_INFO
- OID_WWAN_SMS_CONFIGURATION
- OID_WWAN_SMS_DELETE
- OID_WWAN_SMS_READ
- OID_WWAN_SMS_SEND
- OID_WWAN_SMS_STATUS
- OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS
- OID_WWAN_SYS_CAPS_INFO
- OID_WWAN_UICC_ACCESS_BINARY
- OID_WWAN_UICC_ACCESS_RECORD
- OID_WWAN_UICC_APP_LIST
- OID_WWAN_UICC_FILE_STATUS
- OID_WWAN_UICC_RESET
- OID_WWAN_USSD
- OID_WWAN_VENDOR_SPECIFIC
- OID_WWAN_VISIBLE_PROVIDERS

GUID_NDIS_GEN_PCI_DEVICE_CUSTOM_PROPERTIES

Article • 03/14/2023

WMI clients can use the GUID_NDIS_GEN_PCI_DEVICE_CUSTOM_PROPERTIES method GUID to determine the current link state.

Remarks

NDIS handles this GUID and miniport drivers do not receive an OID query.

When a WMI client issues a GUID_NDIS_GEN_PCI_DEVICE_CUSTOM_PROPERTIES WMI method request, NDIS returns the PCI custom properties of a PCI device for the miniport adapter. The WMI method identifier should be NDIS_WMI_DEFAULT_METHOD_ID, and the WMI input buffer should contain an NDIS_WMI_METHOD_HEADER structure.

The data buffer that NDIS returns with this GUID contains an NDIS_PCI_DEVICE_CUSTOM_PROPERTIES structure.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PCI_DEVICE_CUSTOM_PROPERTIES

NDIS_WMI_METHOD_HEADER

OID_802_3_ADD_MULTICAST_ADDRESS

Article • 02/18/2023

As a set request, NDIS and overlying protocol drivers use the OID_802_3_ADD_MULTICAST_ADDRESS OID request to add an 802.3 multicast address to the multicast address list of a miniport adapter. The multicast address is an array of 6 bytes. Adding an address enables that address to receive multicast packets.

Version Information

Windows Vista
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

Remarks

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains the 6-byte address to be added to the multicast address list.

The OID_802_3_ADD_MULTICAST_ADDRESS OID request can add only one address. To add more than one address, the overlying driver must issue multiple OID_802_3_ADD_MULTICAST_ADDRESS OID requests.

NDIS miniport drivers do not receive this OID request directly. Instead, NDIS consolidates each sequence of OID_802_3_ADD_MULTICAST_ADDRESS and OID_802_3_DELETE_MULTICAST_ADDRESS OID requests into a single OID_802_3_MULTICAST_LIST OID request, which it sends to the miniport driver.

To receive multicast packets, the overlying driver must use the OID_GEN_CURRENT_PACKET_FILTER OID to set the packet filter **NDIS_PACKET_TYPE_MULTICAST** flag.

The miniport driver can set a limit on the number of multicast addresses that the multicast address list can contain. To specify the maximum number of multicast addresses, the miniport driver sets the **MaxMulticastListSize** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure that it passes to the **NdisMSetMiniportAttributes** function. For miniport drivers that are based on NDIS versions before NDIS 6.0, NDIS queries the maximum number of multicast addresses by sending an OID_802_3_MAXIMUM_LIST_SIZE OID request. NDIS returns

NDIS_STATUS_MULTICAST_FULL if an OID_802_3_ADD_MULTICAST_ADDRESS request exceeds this limit.

To delete a previously added multicast address, make a set request with the OID_802_3_DELETE_MULTICAST_ADDRESS OID. The overlying driver can add a given multicast address multiple times. If NDIS succeeds the first add request for a given multicast address, NDIS will succeed all subsequent add requests for that address. To delete a multicast address that was added more than once, the overlying driver must delete the address the same number of times that it added the address.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES

NDIS_OID_REQUEST

NdisMSetMiniportAttributes

OID_802_3_DELETE_MULTICAST_ADDRESS

OID_802_3_MAXIMUM_LIST_SIZE

OID_802_3_MULTICAST_LIST

OID_GEN_CURRENT_PACKET_FILTER

OID_802_3_CURRENT_ADDRESS

Article • 02/18/2023

The address the NIC is currently using.

The network management software cannot set the current station address using the NDIS interface library. It must set this address as a configuration parameter.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_802_3_DELETE_MULTICAST_ADDRESS

Article • 02/18/2023

As a set request, NDIS and overlying protocol drivers use the OID_802_3_DELETE_MULTICAST_ADDRESS OID to delete a previously added multicast address from the multicast address list of a miniport adapter. The multicast address is an array of 6 bytes. Deleting an address disables that address so that it can no longer receive multicast packets.

Version Information

Windows Vista
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

Remarks

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains the 6-byte address to be deleted from the multicast address list.

The OID_802_3_DELETE_MULTICAST_ADDRESS OID request can delete only one address. To delete more than one address, the protocol driver must issue multiple OID_802_3_DELETE_MULTICAST_ADDRESS OID requests.

NDIS miniport drivers do not receive this OID request directly. Instead, NDIS consolidates each sequence of OID_802_3_ADD_MULTICAST_ADDRESS and OID_802_3_DELETE_MULTICAST_ADDRESS OID requests into a single OID_802_3_MULTICAST_LIST OID request.

To replace or delete the entire multicast list, the protocol driver can use the OID_802_3_MULTICAST_LIST OID request.

To add an address to the list, the protocol driver can use the OID_802_3_ADD_MULTICAST_ADDRESS OID request.

The overlying protocol driver can add a given multicast address multiple times by sending multiple OID_802_3_ADD_MULTICAST_ADDRESS OID requests. If NDIS succeeds the first add request for a given multicast address, NDIS will succeed all subsequent add requests for that address. To delete a multicast address that was added more than once, the overlying driver must delete the address the same number of times that it added the address.

Return status codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the **NdisMOidRequestComplete** function, passing NDIS_STATUS_SUCCESS for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport driver is resetting.
NDIS_STATUS_REQUEST_ABORTED	The miniport driver stopped processing the request. For example, NDIS called the *MiniportResetEx* function.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_802_3_ADD_MULTICAST_ADDRESS

OID_802_3_MAXIMUM_LIST_SIZE

OID_802_3_MULTICAST_LIST

OID_802_3_MAC_OPTIONS

Article • 02/18/2023

A protocol can use this OID to determine features supported by the underlying driver, which could be emulating Ethernet.

The underlying driver returns zero, indicating that it supports no options.

Note This OID is obsolete for NDIS 6 drivers.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_802_3_MAXIMUM_LIST_SIZE

Article • 02/18/2023

NDIS and overlying protocol drivers use the OID_802_3_MAXIMUM_LIST_SIZE OID request to query or set the maximum number of 6-byte multicast addresses that the miniport adapter's multicast address list can hold.

This multicast address list is shared by all protocol drivers that are bound to the miniport adapter. Because it is a shared resource, a protocol driver can receive **NDIS_STATUS_MULTICAST_FULL** from the miniport adapter in response to an OID_802_3_MULTICAST_LIST OID set request, even if the number of elements in the list is less than the number that NDIS previously returned for an OID_802_3_MAXIMUM_LIST_SIZE OID query request.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_802_3_ADD_MULTICAST_ADDRESS

OID_802_3_DELETE_MULTICAST_ADDRESS

OID_802_3_MULTICAST_LIST

OID_802_3_MULTICAST_LIST

Article • 02/18/2023

As a set request, NDIS and overlying protocol drivers use the OID_802_3_MULTICAST_LIST OID request to replace the current multicast address list on a miniport adapter. If an address is present in the list, that address is enabled to receive multicast packets.

As a query request, NDIS and protocol drivers use the OID_802_3_MULTICAST_LIST OID request to obtain the current multicast address list.

NDIS handles OID_802_3_MULTICAST_LIST query requests for miniport drivers, so miniport drivers never receive these query requests.

Miniport drivers that support multicast address lists must support OID_802_3_MULTICAST_LIST set requests.

For a set request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains the multicast address list as an array of addresses.

- Each address is an array of 6 bytes.
- The **InformationBufferLength** member contains the length, in bytes, of the **InformationBuffer** array.
- If there are duplicate addresses in the list in the **InformationBuffer** member, NDIS removes the duplicates before sending the OID_802_3_MULTICAST_LIST set request to the miniport driver.
- If the **InformationBufferLength** member is zero, the miniport driver must clear the multicast address list.
- If the **InformationBufferLength** member is greater than zero, the miniport driver must replace any existing multicast address list with the list in the **InformationBuffer** member.

The miniport adapter's multicast address list is shared by all protocol drivers that are bound to the miniport adapter. NDIS controls access to this list. If multiple protocol drivers try to modify the list at the same time, NDIS combines their requests into a single OID_802_3_MULTICAST_LIST set request, which it sends to the miniport driver.

When a miniport adapter is initialized, it resets the NIC so the multicast address list is zero. NDIS also initializes the packet filter so it does not allow the protocol driver to receive multicast packets.

To receive a multicast packet, the protocol driver must later do one of the following:

- Set the packet filter to include the **NDIS_PACKET_TYPE_MULTICAST** flag. At any time, it can disable multicast packet reception by canceling this flag. The order in which the protocol driver enables reception for multicast packets is not important. For more information, see the OID_GEN_CURRENT_PACKET_FILTER OID request.
- Set the packet filter to include the **NDIS_PACKET_TYPE_ALL_MULTICAST** flag, which enables all multicast packets, and do the filtering itself.

The miniport driver can set a limit on the number of multicast addresses that the multicast address list can contain. NDIS returns **NDIS_STATUS_MULTICAST_FULL** if a protocol driver exceeds this limit or if it specifies an invalid multicast address.

For a query request, NDIS returns a multicast address list that is the union of all multicast address lists for all protocol bindings.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_802_3_ADD_MULTICAST_ADDRESS

OID_802_3_DELETE_MULTICAST_ADDRESS

OID_802_3_MAXIMUM_LIST_SIZE

OID_GEN_CURRENT_PACKET_FILTER

OID_802_3_PERMANENT_ADDRESS

Article • 02/18/2023

The address of the NIC encoded in the hardware.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_802_3_RCV_OVERRUN

Article • 02/18/2023

The number of frames not received due to overrun errors on the NIC.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_802_3_XMIT_DEFERRED

Article • 02/18/2023

The number of frames successfully transmitted after the NIC defers transmission at least once.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_802_3_XMIT_HEARTBEAT_FAILURE

Article • 02/18/2023

The number of frames successfully transmitted without detection of the collision-detect heartbeat.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_802_3_XMIT_LATE_COLLISIONS

Article • 02/18/2023

The number of collisions detected after the normal window.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_802_3_XMIT_MAX_COLLISIONS

Article • 02/18/2023

The number of frames not transmitted due to excessive collisions.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_802_3_XMIT_TIMES_CRS_LOST

Article • 02/18/2023

The number of times the CRS signal has been lost during packet transmission.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_802_3_XMIT_UNDERRUN

Article • 02/18/2023

The number of frames not transmitted due to underrun errors on the NIC.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_ADD_ADDRESS

Article • 02/18/2023

The OID_CO_ADD_ADDRESS OID is sent by a client to a call manager to specify an alias address for a host. The alias address is formatted as a CO_ADDRESS structure, defined as follows:

C++

```
typedef struct _CO_ADDRESS{
    ULONG   AddressSize;
    UCHAR   Address[1];
} CO_ADDRESS, *PCO_ADDRESS;
```

The members of this structure contain the following information:

AddressSize
Specifies the size in bytes of the structure at **Address**.

Address
Specifies a variable-length array that contains the alias address. The address format is specific to the signaling protocol used by the call manager.

This OID is typically used to specify a well-known address at which a host offers a particular service. For example, a client could specify a well-known address for a LAN emulation server. The call manager's response to this OID is specific to the signaling protocol used by the call manager. An ATM call manager, for example, sends a message to the switch that notifies the switch of the alias address.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_ADD_PVC

Article • 02/18/2023

The OID_CO_ADD_PVC OID is sent by a client to a call manager to add a permanent virtual connection (PVC) to the call manager's list of configured PVCs. The PVC is formatted as a CO_PVC structure, defined as follows:

C++

```
typedef struct _CO_PVC {
    NDIS_HANDLE             NdisAfHandle;
    CO_SPECIFIC_PARAMETERS  PvcParameters;
} CO_PVC, *PCO_PVC;
```

The members of this structure contain the following information:

NdisAfHandle
Specifies the NDIS-supplied handle returned by NdisClOpenAddressFamilyEx.

PvcParameters
A formatted CO_SPECIFIC_PARAMETERS structure. This structure contains protocol-specific parameters that describe the PVC.

A PVC is configured manually by an administrator. A client that monitors such activity notifies a call manager of a newly configured PVC by sending this OID to the call manager. Other clients can then use the newly-configured PVC.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_ADDRESS_CHANGE

Article • 02/18/2023

The OID_CO_ADDRESS_CHANGE OID is sent by the call manager to each client that opened an address family with the call manager. This action is taken in response to a change in the switch address that the call manager uses. For example, the call manager sends this request if someone disconnects the NIC from one switch and plugs it into another switch. Each notified client must send an OID_CO_GET_ADDRESSES query to the call manager to retrieve a list of currently valid addresses.

The call manager also sends OID_CO_ADDRESS_CHANGE to a client immediately after the client opens an address family with the call manager. This ensures that a client that opens an address family after the switch address has changed is notified of the change. The client must then must then send an OID_CO_GET_ADDRESSES query to the call manager.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_AF_CLOSE

Article • 02/18/2023

The OID_CO_AF_CLOSE OID is sent by a call manager that must unbind itself from an underlying miniport driver. Before unbinding itself from the miniport driver, the call manager sends this OID to each client that has an address family open with the call manager. In response, the client should do the following:

1. If the client has any active multipoint connections, call NdisClDropParty as many times as necessary until only a single party remains active on each multipoint VC
2. Call NdisClCloseCall as many times as necessary to close all calls still open with the call manager
3. Call NdisClDeregisterSap as many times as necessary to deregister all SAPs that the client has registered with the call manager
4. Call NdisClCloseAddressFamily to close the address family referenced by NdisAfHandle in the request that contained OID_CO_AF_CLOSE

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_DELETE_ADDRESS

Article • 02/18/2023

The OID_CO_DELETE_ADDRESS OID is sent by a client to a call manager to delete an alias address for a host. The alias address is formatted as a CO_ADDRESS structure, defined as follows:

```c++
typedef struct _CO_ADDRESS {
    ULONG   AddressSize;
    UCHAR   Address[1];
} CO_ADDRESS, *PCO_ADDRESS;
```

The members of this structure contain the following information:

AddressSize
Specifies the size in bytes of the structure at Address .

Address
Specifies a variable-length array that contains the alias address. The address format is specific to the signaling protocol used by the call manager.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_DELETE_PVC

Article • 02/18/2023

The OID_CO_DELETE_PVC OID is sent by a client to a call manager to delete a permanent virtual connection (PVC) from the call manager's list of configured PVCs. The PVC is formatted as a CO_PVC structure, defined as follows:

```c++
typedef struct _CO_PVC {
    NDIS_HANDLE             NdisAfHandle;
    CO_SPECIFIC_PARAMETERS  PvcParameters;
} CO_PVC, *PCO_PVC;
```

The members of this structure contain the following information:

NdisAfHandle
Specifies the NDIS-supplied handle returned by NdisClOpenAddressFamilyEx.

PvcParameters
A formatted CO_SPECIFIC_PARAMETERS structure. This structure contains protocol-specific parameters that describe the PVC.

A PVC is removed manually by an administrator. A client that monitors such activity notifies a call manager of a PVC that has been removed by sending this OID to the call manager.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_GET_ADDRESSES

Article • 02/18/2023

The OID_CO_GET_ADDRESSES OID is used by the client to make a query to the call manager. This query is made in response to the call manager sending an OID_CO_ADDRESS_CHANGE to the client. In response to this query, the call manager sends the client an address list that is formatted as a CO_ADDRESS_LIST structure, defined as follows:

```c++
typedef struct _CO_ADDRESS_LIST {
    ULONG       NumberOfAddressesAvailable;
    ULONG       NumberOfAddresses;
    CO_ADDRESS  AddressList;
} CO_ADDRESS_LIST, *PCO_ADDRESS_LIST;
```

The members of this structure contain the following information:

NumberOfAddressesAvailable
Specifies the maximum number of addresses in the call manager's list of addresses. Regardless of the actual number of addresses that the call manager returns to the client at **AddressList**, the size of the buffer at **AddressList** is always **NumberOfAddressesAvailable** multiplied by the address size, which is a fixed size specific to the call manager.

NumberOfAddresses
Specifies the number of addresses that the call manager has written to **AddressList**.

AddressList
The alias address is formatted as a CO_ADDRESS structure, defined as follows:

```c++
typedef struct _CO_ADDRESS {
    ULONG   AddressSize;
    UCHAR   Address[1];
} CO_ADDRESS, *PCO_ADDRESS;
```

The members of this structure contain the following information:

AddressSize
Specifies the size in bytes of the structure at **Address** .

Address
Specifies a variable-length array that contains the list of addresses. The address format is specific to the signaling protocol used by the call manager.

The **AddressList** contains network addresses at which the local host can be reached. The **AddressList** returned to a particular client contains addresses that are common to all clients, as well as any addresses that the client itself has added to call manager's list of addresses with OID_CO_ADD_ADDRESS.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_GET_CALL_INFORMATION

Article • 02/18/2023

The OID_CO_GET_CALL_INFORMATION OID is reserved for future use.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_SIGNALING_DISABLED

Article • 02/18/2023

The OID_CO_SIGNALING_DISABLED OID is sent by a call manager to indicate that it cannot make calls or dispatch incoming calls. The call manager sends this OID to each client that has an address family open with the call manager.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_SIGNALING_ENABLED

Article • 02/18/2023

The OID_CO_SIGNALING_ENABLED OID is sent by a call manager to indicate that it is ready to make calls and dispatch incoming calls. The call manager sends this OID to each client that has an address family open with the call manager.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_TAPI_ADDRESS_CAPS

Article • 02/18/2023

The OID_CO_TAPI_ADDRESS_CAPS OID requests a call manager or an integrated miniport call manager (MCM) driver to return the telephony capabilities for a specified address on a specified line.

This request uses a CO_TAPI_ADDRESS_CAPS structure, which is defined as follows:

```c++
typedef struct _CO_TAPI_ADDRESS_CAPS {
    IN  ULONG                ulLineID;
    IN  ULONG                ulAddressID;
    OUT ULONG                ulFlags;
    OUT LINE_ADDRESS_CAPS    LineAddressCaps;
} CO_TAPI_ADDRESS_CAPS, *PCO_TAPI_ADDRESS_CAPS;
```

The members of this structure contain the following information:

ulLineID
Specifies the zero-based line identifier of the line on which the given address is located.

ulAddressID
Specifies the zero-based address identifier on the line for which capabilities should be returned.

ulFlags
These flags are reserved.

LineAddressCaps
Specifies the telephony capabilities of an address, formatted as a LINE_ADDRESS_CAPS structure. For more information about this structure, see the Microsoft Windows SDK and the ndistapi.h header file.

Remarks

After querying the line capabilities of a call manager's or MCM driver's device with OID_CO_TAPI_LINE_CAPS, a connection-oriented client queries the capabilities of the address(es) for each line as follows:

- If the previous query of OID_CO_TAPI_LINE_CAPS indicated that the line supports only one address or that all addresses on the line have the same address capabilities, the client queries OID_CO_TAPI_ADDRESS_CAPS once to determine the capabilities of all the addresses on the line. In this case, the address capabilities returned by the call manager or MCM driver apply to all addresses on the line.

- If a line supports multiple addresses that have dissimilar capabilities, the client queries OID_CO_TAPI_ADDRESS_CAPS once for each address on the line. In this case, the address capabilities returned by the call manager or MCM driver apply to a specified address on a specified line.

The call manager or MCM driver returns the address capabilities for a specified address in **LineAddressCaps**.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_TAPI_CM_CAPS

Article • 02/18/2023

The OID_CO_TAPI_CM_CAPS OID requests a call manager or an integrated miniport call manager (MCM) driver to return the number of lines supported by its device (the device for which it provides call management services). This OID also requests the call manager or MCM driver to indicate whether these lines have dissimilar line capabilities.

This request uses a CO_TAPI_CM_CAPS structure, which is defined as follows:

```C++
typedef struct _CO_TAPI_CM_CAPS {
    OUT ULONG   ulCoTapiVersion;
    OUT ULONG   ulNumLines;
    OUT ULONG   ulFlags;
} CO_TAPI_CM_CAPS, *PCO_TAPI_CM_CAPS;
```

The members of this structure contain the following information:

ulCoTapiVersion

Specifies the TAPI version supported by the call manager or MCM driver. The call manager or MCM driver should set this to CO_TAPI_VERSION.

ulNumLines

Specifies the number of lines supported by the device.

ulFlags

If the device supports multiple lines that have dissimilar line capabilities or if the addresses on any of these lines have dissimilar address capabilities, the call manager or MCM driver sets the CO_TAPI_FLAG_PER_LINE_CAPS bit in **ulFlags**; otherwise, the call manager or MCM driver clears this bit. All undefined bits are reserved for future use and must be set to 0.

Remarks

A connection-oriented client uses the information returned from this OID to determine how it will query the line capabilities of the call manager's or MCM driver's device with OID_CO_TAPI_LINE_CAPS.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_TAPI_GET_CALL_DIAGNOSTICS

Article • 02/18/2023

The OID_CO_TAPI_GET_CALL_DIAGNOSTICS OID requests a call manager or MCM driver to return diagnostic information about a failed call or a call torn down by the remote TAPI party.

This request uses a CO_TAPI_CALL_DIAGNOSTICS structure, which is defined as follows:

```C++
typedef struct _CO_TAPI_CALL_DIAGNOSTICS {
    OUT ULONG                   ulOrigin;
    OUT ULONG                   ulReason;
    OUT NDIS_VAR_DATA_DESC  DiagInfo;
} CO_TAPI_CALL_DIAGNOSTICS, *PCO_TAPI_CALL_DIAGNOSTICS;
```

ulOrigin
Specifies the origination of the call as one of the following LINECALLORIGIN_ constants:

- **LINECALLORIGIN_OUTBOUND**
  The call is an outgoing call.

- **LINECALLORIGIN_INTERNAL**
  The call is incoming and originated internally (on the same PBX, for example).

- **LINECALLORIGIN_EXTERNAL** The call is incoming and originated externally.

- **LINECALLORIGIN_UNKNOWN**
  The call is incoming. Its origin is currently unknown but may become known later.

- **LINECALLORIGIN_UNAVAIL**
  The call is incoming. Its origin is not available and will never be known.

- **LINECALLORIGIN_CONFERENCE**
  The call handle is for a conference call--that is, for the application's connection to the conference bridge in the switch.

ulReason
Specifies the reason for the call as one of the following LINECALLREASON_ constants:

- **LINECALLREASON_DIRECT**
  The call is direct.

- LINECALLREASON_FWDBUSY

  The call was forwarded from a busy extension.

- LINECALLREASON_FWDNOANSWER

  The call was forwarded after some number of rings from an unanswered extension.

- LINECALLREASON_FWDUNCOND

  The call was forwarded unconditionally from another number.

- LINECALLREASON_PICKUP

  The call was picked up from another extension.

- LINECALLREASON_UNPARK

  The call was retrieved as a parked call.

- LINECALLREASON_REDIRECT

  The call was redirected to this station.

- LINECALLREASON_CALLCOMPLETION

  The call was the result of a call completion request.

- LINECALLREASON_TRANSFER

  The call was transferred from another number. Party identifier information may indicate who the caller is and from where the call was transferred.

- LINECALLREASON_REMINDER

  The call is a reminder (or "recall") that the user has a call parked or on hold for a potentially long time.

- LINECALLREASON_UNKNOWN

  The reason for the call is currently unknown but may become known later.

- LINECALLREASON_UNAVAIL

  The reason for the call is unavailable and cannot become known later.

DiagInfo

Specifies an NDIS_VAR_DATA_DESC structure that contains an offset to, as well as the length of, optional diagnostic information supplied by the call manager or MCM driver. The content and format of the diagnostic information is driver-determined.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_TAPI_LINE_CAPS

Article • 02/18/2023

The OID_CO_TAPI_LINE_CAPS OID requests a call manager or an integrated miniport call manager (MCM) driver to return the telephony capabilities for a specified line. This OID also requests the call manager or MCM driver to indicate whether addresses on this line have dissimilar telephony capabilities.

This request uses a CO_TAPI_LINE_CAPS structure, defined as follows, to query the telephony capabilities of a specified line:

```c++
typedef struct _CO_TAPI_LINE_CAPS {
    IN  ULONG           ulLineID;
    OUT ULONG           ulFlags;
    OUT LINE_DEV_CAPS   LineDevCaps;
} CO_TAPI_LINE_CAPS, *PCO_TAPI_LINE_CAPS;
```

The members of this structure contain the following information:

ulLineID
Specifies the line for which telephony capabilities should be returned. **ulLineID** is a zero-based identifier.

ulFlags
If the line supports multiple addresses that have dissimilar telephony capabilities, the call manager or MCM driver sets the CO_TAPI_FLAG_PER_ADDRESS_CAPS bit in ulFlags; otherwise, the call manager or MCM driver clears this bit. All undefined bits are reserved and must be set to 0.

LineDevCaps
Specifies the telephony capabilities of a line, formatted as a LINE_DEV_CAPS structure. For more information about this structure, see the Microsoft Windows SDK and the ndistapi.h header file.

Remarks

After querying the telephony capabilities of a call manager's or MCM driver's device with OID_CO_TAPI_CM_CAPS, a connection-oriented client queries the telephony capabilities of the line(s) supported by the device.

- If all lines supported by the device have the same line capabilities and all the addresses on these lines have the same address capabilities, the client queries OID_CO_TAPI_LINE_CAPS once to obtain the line capabilities of the device. In this case, the line capabilities returned by the call manager or MCM driver apply to all the lines supported by the device.
- If the device supports multiple lines with dissimilar capabilities, however, and/or if addresses on these lines have dissimilar address capabilities, the client queries OID_CO_TAPI_LINE_CAPS once for each line supported by the device to obtain the capabilities of each line.

The **ulFlags** setting determines how many times the client subsequently queries the capabilities of the address(es) on the line:

- If the line supports only one address, or if the line supports multiple addresses that have the same address capabilities, the client queries OID_CO_TAPI_ADDRESS_CAPS once.
- If the line supports multiple addresses that have dissimilar capabilities, the client must query OID_CO_TAPI_ADDRESS_CAPS once for each address on the line.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_TAPI_TRANSLATE_NDIS_CALLPARAMS

Article • 03/14/2023

The OID_CO_TAPI_TRANSLATE_NDIS_CALLPARAMS OID requests a call manager or MCM driver to translate NDIS call parameters (passed in a CO_CALL_PARAMETERS structure to the client's ProtocolClIncomingCall function) to TAPI call parameters. The client uses the translated TAPI call parameters returned by the call manager or MCM driver to determine whether to accept or reject the incoming call.

This request uses a CO_TAPI_TRANSLATE_NDIS_CALLPARAMS structure, which is defined as follows:

```C++
typedef struct _CO_TAPI_TRANSLATE_NDIS_CALLPARAMS {
    IN  ULONG               ulFlags;
    IN  NDIS_VAR_DATA_DESC  NdisCallParams;
    OUT NDIS_VAR_DATA_DESC  LineCallInfo;
} CO_TAPI_TRANSLATE_NDIS_CALLPARAMS, *PCO_TAPI_TRANSLATE_NDIS_CALLPARAMS;
```

The members of this structure contain the following information:

ulFlags
The client must set the CO_TAPI_FLAG_INCOMING_CALL bit in **ulFlags**.

NdisCallParams
Specifies an NDIS_VAR_DATA_DESC structure that contains an offset from the beginning of the NDIS_VAR_DATA_DESC structure to a CO_CALL_PARAMETERS structure. The NDIS_VAR_DATA_DESC structure also contains the length of the CO_CALL_PARAMETERS structure. The client fills in the CO_CALL_PARAMETERS structure with the NDIS call parameters to be translated to TAPI call parameters.

LineCallInfo
Specifies an NDIS_VAR_DATA_DESC structure that contains an offset from the beginning of the NDIS_VAR_DATA_DESC structure to a LINE_CALL_INFO structure. The NDIS_VAR_DATA_DESC structure also contains the length of the CO_CALL_PARAMETERS structure. The LINE_CALL_INFO structure specifies the TAPI call parameters into which the given NDIS call parameters have been translated. For more information about the LINE_CALL_INFO structure, see the Windows SDK and the ndistapi.h header file.

Remarks

If the request is successful, the call manager or MCM driver fills in the LINE_CALL_PARAMS structure referred to by **LineCallInfo** with the translated TAPI call parameters. The call manager or MCM driver must allocate the LINE_CALL_INFO structure within the flat memory section referred to **LineCallInfo**. The client writes the total length of the LINE_CALL_INFO structure to **LineCallInfo.Length**.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS

Article • 02/18/2023

The OID_CO_TAPI_TRANSLATE_TAPI_CALLPARAMS OID requests a call manager or integrated call manager miniport (MCM) driver to translate TAPI call parameters to NDIS call parameters. The client that queries this OID uses the returned NDIS call parameters as an input (formatted as a CO_CALL_PARAMETERS structure) to NdisClMakeCall, with which the client places an outgoing call.

This OID uses a CO_TAPI_TRANSLATE_TAPI_CALLPARAMS structure, which is defined as follows:

C++

```
typedef struct _CO_TAPI_TRANSLATE_TAPI_CALLPARAMS {
    IN  ULONG                   ulLineID;
    IN  ULONG                   ulAddressID;
    IN  ULONG                   ulFlags;
    IN  NDIS_VAR_DATA_DESC   DestAddress;
    IN  NDIS_VAR_DATA_DESC   LineCallParams;
    OUT NDIS_VAR_DATA_DESC   NdisCallParams;
} CO_TAPI_TRANSLATE_TAPI_CALLPARAMS, *PCO_TAPI_TRANSLATE_TAPI_CALLPARAMS;
```

The members of this structure contain the following information:

ulLineID

Specifies a zero-based line identifier to which the outgoing call will be directed.

ulAddressID

Specifies a zero-based address identifier (on the line specified by **ulLineID**) to which the outgoing call will be directed.

ulFlags

The client must set the CO_TAPI_FLAG_OUTGOING_CALL bit in **ulFlags**. The client can optionally set the CO_TAPI_USE_DEFAULT_CALLPARAMS bit in **ulFlags** to require the call manager or MCM driver to ignore the **LineCallParams** and return the default NDIS call parameters for the device.

DestAddress

Specifies an NDIS_VAR_DATA_DESC structure that contains an offset from the beginning of the NDIS_VAR_DATA_DESC structure to a destination address formatted as a character array. The NDIS_VAR_DATA_DESC structure also contains the length of the destination address. The destination address is the address to which the outgoing call will be directed.

LineCallParams
Specifies an NDIS_VAR_DATA_DESC structure that contains an offset from the beginning of the NDIS_VAR_DATA_DESC structure to a LINE_CALL_PARAMS structure. The NDIS_VAR_DATA_DESC structure also contains the length of the LINE_CALL_PARAMS structure. The LINE_CALL_PARAMS structure specifies the TAPI call parameters to be translated into NDIS call parameters. For more information about the LINE_CALL_PARAMS structure, see the Microsoft Windows SDK and the ndistapi.h header file.

NdisCallParams
Specifies an NDIS_VAR_DATA_DESC structure that contains an offset from the beginning of the NDIS_VAR_DATA_DESC structure to a CO_CALL_PARAMETERS structure. The NDIS_VAR_DATA_DESC structure also contains the length of the CO_CALL_PARAMETERS structure. The CO_CALL_PARAMETERS structure specifies the NDIS call parameters into which the given TAPI call parameters have been translated.

Remarks

If the request is successful, the call manager or MCM driver fills in the CO_CALL_PARAMETERS structure referenced by **NdisCallParams** with the translated NDIS call parameters. The call manager or MCM driver must allocate the CO_CALL_PARAMETERS structure within the flat memory section referred to by **NdisCallParams**. The client writes the total length of the CO_CALL_PARAMETERS structure to **NdisCallParams.Length**.

If the client sets the CO_TAPI_USE_DEFAULT_CALLPARAMS bit in **ulFlags**, the client does not specify TAPI call parameters. In this case, the call manager or MCM driver should return the default NDIS call parameters for the device. If there are no default NDIS call parameters for the device, the call manager or MCM driver should return NDIS_STATUS_FAILURE.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_CO_TAPI_TRANSLATE_TAPI_SAP

Article • 02/18/2023

The OID_CO_TAPI_TRANSLATE_TAPI_SAP OID requests a call manager or integrated MCM driver to prepare one or more SAPs from TAPI call parameters. The client that queries this OID uses an NDIS SAP returned by the call manager or MCM driver as an input (formatted as a CO_SAP structure) to NdisClRegisterSap, which the client calls to register a SAP on which to receive incoming calls.

This request uses a CO_TAPI_TRANSLATE_SAP structure, which is defined as follows:

C++

```
typedef struct _CO_TAPI_TRANSLATE_SAP {
    IN  ULONG               ulLineID;
    IN  ULONG               ulAddressID;
    IN  ULONG               ulMediaModes;
    IN  ULONG               Reserved;
    OUT ULONG               NumberOfSaps;
    OUT NDIS_VAR_DATA_DESC  NdisSapParams[1];
} CO_AF_TAPI_SAP, *PCO_AF_TAPI_SAP;
```

The members of this structure contain the following information:

ulLineID

Specifies a zero-based line identifier.

ulAddressID

Specifies a zero-based address identifier on the line specified by **ulLineID**.

ulMediaModes

Specifies the media mode of the information stream of calls that the client is interested in, as one or more of the following LINEMEDIAMODE_constants:

- LINEMEDIAMODE_UNKNOWN
  A media stream exists but its mode is currently unknown and may become known later. This corresponds to a call with an unclassified media type. In typical analog telephony environments, the media mode of an incoming call may be unknown until after the call has been answered and the media stream has been filtered to make a determination.

  If the **LINEMEDIAMODE_UNKNOWN** flag is set, other media flags can also be set. This signifies that the media is unknown but that it is likely to be one of the other indicated media modes.

- LINEMEDIAMODE_INTERACTIVEVOICE
  The presence of voice energy on the call, and the call is treated as an interactive call with humans on both ends.

- LINEMEDIAMODE_AUTOMATEDVOICE
  The presence of voice energy on the call, and the voice is locally handled by an automated application.

- LINEMEDIAMODE_DATAMODEM
  A data modem session on the call.

- LINEMEDIAMODE_G3FAX
  A group 3 fax is being sent or received over the call.

- LINEMEDIAMODE_G4FAX
  A group 4 fax is being sent or received over the call.

- LINEMEDIAMODE_TDD
  A TDD (telecommunication device for the deaf) session on the call.

- LINEMEDIAMODE_DIGITALDATA
  Digital data is being sent or received over the call.

- LINEMEDIAMODE_TELETEX
  A teletex session on the call. (Teletex is one of the telematic services.)

- LINEMEDIAMODE_VIDEOTEX
  A videotex session on the call. (Videotex is one the telematic services.)

- LINEMEDIAMODE_TELEX
  A telex session on the call. (Telex is one of the telematic services.)

- LINEMEDIAMODE_MIXED
  A mixed session on the call. (Mixed is one of the ISDN telematic services.)

- LINEMEDIAMODE_ADSI
  An ADSI (Analog Display Service Interfaces) session on the call.

- LINEMEDIAMODE_VOICEVIEW
  The media mode of the call is VoiceView.

Reserved
This is reserved. The client must set this field to 0.

NumberOfSaps
Specifies the number of NDIS_VAR_DATA_DESC structures contained in the buffer at

NdisSapParams.

NdisSapParams
Specifies a variable-length array that contains one or more NDIS_VAR_DATA_DESC structures. Each NDIS_VAR_DATA_DESC structure contains an offset to, as well as the length of, a CO_SAP structure. Each CO_SAP structure specifies a service access point (SAP) on which a connection-oriented client can receive incoming calls.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_ADMIN_STATUS

Article • 02/18/2023

As a query, use the OID_GEN_ADMIN_STATUS OID to determine the administrative status for an interface (*ifAdminStatus* from RFC 2863).

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

The administrative status is the status that the system administrator requested.

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the query succeeds, the interface provider returns NDIS_STATUS_SUCCESS, and the result of the query can be one of the values in the **NET_IF_ADMIN_STATUS** enumeration.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NET_IF_ADMIN_STATUS

NDIS Network Interface OIDs

OID_GEN_ALIAS

Article • 02/18/2023

As a query, use the OID_GEN_ALIAS OID to obtain the alias string for an interface (*ifAlias* from RFC 2863).

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

An NDIS network interface provider can assign unique alias strings for its interfaces. If the name should remain associated with the same interface, the provider can make the strings persistent after the computer restarts and reinitializations.

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the interface provider returns NDIS_STATUS_SUCCESS, the result of the query is an alias string that is returned in an NDIS_IF_COUNTED_STRING structure.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS Network Interface OIDs

OID_GEN_BROADCAST_BYTES_RCV

Article • 02/18/2023

As a query, the OID_GEN_BROADCAST_BYTES_RCV OID specifies the number of bytes in broadcast packets that are received without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_BROADCAST_BYTES_XMIT

Article • 02/18/2023

As a query, the OID_GEN_BROADCAST_BYTES_XMIT OID specifies the number of bytes in broadcast packets that are transmitted without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_BROADCAST_FRAMES_RCV

Article • 02/18/2023

As a query, the OID_GEN_BROADCAST_FRAMES_RCV OID specifies the number of broadcast packets that are received without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

The count from this OID, combined with the count from OID_GEN_MULTICAST_FRAMES_RCV, is identical to the *ifInNUcastPkts* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_BROADCAST_FRAMES_XMIT

Article • 02/18/2023

As a query, the OID_GEN_BROADCAST_FRAMES_XMIT OID specifies the number of broadcast packets that are transmitted without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

The count from this OID, combined with the count from OID_GEN_MULTICAST_FRAMES_XMIT, is identical to the *ifOutNUcastPkts* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_BYTES_RCV

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_BYTES_RCV OID to determine the total number of bytes that a miniport adapter received.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

NDIS handles this OID for miniport drivers. See the OID_GEN_STATISTICS OID for more information about statistics.

The total byte count is the sum of the receive-directed byte count, receive-multicast byte count and receive-broadcast byte count. This value is the same as the sum of the values that are returned by the OID_GEN_DIRECTED_BYTES_RCV, OID_GEN_MULTICAST_BYTES_RCV, and OID_GEN_BROADCAST_BYTES_RCV OIDs.

The count is identical to the *ifInOctets* counter described in RFC 2863.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_BROADCAST_BYTES_RCV

OID_GEN_DIRECTED_BYTES_RCV

OID_GEN_MULTICAST_BYTES_RCV

OID_GEN_STATISTICS

OID_GEN_BYTES_XMIT

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_BYTES_XMIT OID to determine the total bytes that a miniport adapter transmitted.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

NDIS handles this OID for miniport drivers. See the OID_GEN_STATISTICS OID for more information about statistics.

The total byte count is the sum of the transmit-directed byte count, transmit-multicast byte count and transmit-broadcast byte count. This value is the same as the sum of the values that are returned by the OID_GEN_DIRECTED_BYTES_XMIT, OID_GEN_MULTICAST_BYTES_XMIT, and OID_GEN_BROADCAST_BYTES_XMIT OIDs.

The count is identical to the *ifOutOctets* counter described in RFC 2863.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_BROADCAST_BYTES_XMIT

OID_GEN_DIRECTED_BYTES_XMIT

OID_GEN_MULTICAST_BYTES_XMIT

OID_GEN_STATISTICS

OID_GEN_CO_BYTES_RCV

Article • 02/18/2023

The OID_GEN_CO_BYTES_RCV OID specifies the number of bytes in PDUs received without errors.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_BYTES_XMIT

Article • 02/18/2023

The OID_GEN_CO_BYTES_XMIT OID specifies the number of bytes in PDUs transmitted without errors.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_BYTES_XMIT_OUTSTANDING

Article • 02/18/2023

The OID_GEN_CO_BYTES_XMIT_OUTSTANDING OID specifies the number of bytes in PDUs that are queued for transmission.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_DRIVER_VERSION

Article • 02/18/2023

The NDIS version in use by the NIC driver. This OID is two bytes in length; the high byte is the major version number, and the low byte is the minor version number.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_GET_NETCARD_TIME

Article • 02/18/2023

> ⓘ **Note**
>
> OID_GEN_CO_GET_NETCARD_TIME is the same as OID_GEN_GET_NETCARD_TIME.

The OID_GEN_CO_GET_NETCARD_TIME OID requests the miniport driver to return a NIC's local time, as derived from a clock on the NIC or from the network. The time is formatted as a GEN_GET_NETCARD_TIME structure, defined as follows:

```c++
typedef struct _GEN_GET_NETCARD_TIME{
    ULONGLONG   ReadTime;
} GEN_GET_NETCARD_TIME, *PGEN_GET_NETCARD_TIME;
```

The member of this structure contains the following information:

ReadTime
The NIC's local time.

Remarks

The miniport driver specified the units for its local time in the **ClockPrecision** element of the GEN_GET_TIME_CAPS structure that the miniport driver returned in response to a previous OID_GEN_CO_GET_TIME_CAPS query.

If the miniport driver set the READABLE_LOCAL_CLOCK flag in its response to an OID_GEN_CO_GET_TIME_CAPS query, the NIC derives its local time from an onboard clock. If the miniport driver set the CLOCK_NETWORK_DERIVED flag in its response to an OID_GEN_CO_GET_TIME_CAPS query, the NIC derives its local time from the network.

If the local time is derived from an onboard clock, the miniport driver should be able to report the clock precision in parts per million. In general, a network-derived clock is preferable, because it is likely to be more precise and can be used to synchronize many machines attached to the same network or switch.

The miniport driver must return its local time synchronously in its response to the OID_GEN_CO_GET_NETCARD_TIME query since this query synchronizes protocol drivers with the NIC's local time. Protocol drivers should send the

OID_GEN_CO_GET_NETCARD_TIME query several times in succession to filter out response-time latencies.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_GET_TIME_CAPS

Article • 02/18/2023

> ⓘ **Note**
>
> OID_GEN_CO_GET_TIME_CAPS is the same as OID_GEN_GET_TIME_CAPS.

The OID_GEN_CO_GET_TIME_CAPS OID requests a miniport driver to return its capabilities for reporting a NIC's local time formatted as a GEN_GET_TIME_CAPS structure, which is defined as follows:

C++

```cpp
typedef struct _GEN_GET_TIME_CAPS{
    ULONG   Flags;
    ULONG   ClockPrecision;
} GEN_GET_TIME_CAPS, *PGEN_GET_TIME_CAPS;
```

The members of this structure contain the following information:

Flags
The following flags can be ORed together. All unspecified flags must be set to zero.

READABLE_LOCAL_CLOCK
When set, indicates the presence of a readable clock on the NIC. Even without such a hardware clock, a miniport driver can use the system clock by calling NdisGetCurrentSystemTime, so long as it reports the correct precision in the ClockPrecision member.

CLOCK_NETWORK_DERIVED
When set, indicates that the NIC's local time is derived from the network connection, as opposed to a free-running, onboard clock.

CLOCK_PRECISION
When set, indicates that the ClockPrecision member contains valid information.

RECEIVE_TIME_INDICATION_CAPABLE
When set, indicates that the NIC hardware can note the local time at which it receives the first cell of a received PDU and that the miniport driver propagates this receive time for each PDU when indicating the packet to a protocol.

TIMED_SEND_CAPABLE
When set, indicates that the NIC can schedule a packet for transmission according to its local time. Protocols can use NDIS_SET_PACKET_TIME_TO_SEND to set the TimeToSend timestamp in the out-of-band data block of a packet descriptor. Setting the timestamp does not affect when the packet is actually transmitted; instead, the timestamp is used for recordkeeping. A protocol driver can use the timestamp to determine how long it takes to complete the sending of a paket.

TIME_STAMP_CAPABLE
When set, indicates that the NIC can stamp (in the appropriate field of the outgoing packet) the time at which the first byte of the packet is transmitted and that the NIC can retrieve this time from the same field of an inbound packet.

ClockPrecision
Specifies the clock precision in parts per million. For this information to be considered valid, the CLOCK_PRECISION flag must be set.

Remarks

A miniport driver can provide support for certain timing parameters even in the absence of a local or network clock. In particular, a miniport driver can use the system clock for receive time indications, timed sends, and even time stamping. A NIC-based clock is better since it is likely to provide higher precision and to be accessible with lower latencies than the system clock. In all cases, the miniport driver must specify the precision of the clock that it uses. This allows protocols to determine how to best use the miniport driver's timing support.

If the miniport driver reports the presence of a readable clock, it must be prepared to immediately respond to an OID_GEN_GET_NETCARD_TIME query. The miniport driver's response to this call is time-critical and therefore must be synchronous.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_HARDWARE_STATUS

Article • 02/18/2023

The OID_GEN_CO_HARDWARE_STATUS OID specifies the current hardware status of the underlying NIC, as one of the following NDIS_HARDWARE_STATUS-type values:

NdisHardwareStatusReady
Available and capable of sending and receiving data over the wire.

NdisHardwareStatusInitializing
Initializing.

NdisHardwareStatusReset
Resetting.

NdisHardwareStatusClosing
Closing.

NdisHardwareStatusNotReady
Not ready.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_LINK_SPEED

Article • 02/18/2023

The OID_GEN_CO_LINK_SPEED OID requests the miniport driver to return its current transmit and receive speeds formatted as an NDIS_CO_LINK_SPEED structure, which is defined as follows:

```c++
typedef struct _NDIS_CO_LINK_SPEED{
    ULONG   Outbound;
    ULONG   Inbound;
} NDIS_CO_LINK_SPEED, *PNDIS_CO_LINK_SPEED;
```

The members of this structure contain the following information:

Outbound
The current transmit speed of the NIC. The unit of measurement is 100bps, so a value of 100,000 represents a hardware bit rate of 10 Mbps.

Inbound
The current receive speed of the NIC. The unit of measurement is 100bps, so a value of 100,000 represents a hardware bit rate of 10 Mbps.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_MAC_OPTIONS

Article • 02/18/2023

The OID_GEN_CO_MAC_OPTIONS OID is reserved. Do not use it in your code.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_MEDIA_CONNECT_STATUS

Article • 02/18/2023

The OID_GEN_CO_MEDIA_CONNECT_STATUS OID requests the miniport driver to return the connection status of the NIC on the network as one of the following system-defined values:

NdisMediaStateConnected

NdisMediaStateDisconnected

When a miniport driver senses that the network connection has been lost, it should call NdisMCoIndicateStatus with NDIS_STATUS_MEDIA_DISCONNECT. When the connection is restored, it should call NdisMCoIndicateStatus with NDIS_STATUS_MEDIA_CONNECT.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_MEDIA_IN_USE

Article • 02/18/2023

A complete list of the media types the NIC is currently supporting, defined as some, none (also called the NULL filter), or all of the following:

NdisMedium802_3
Ethernet (802.3).

NdisMedium802_5
Token Ring (802.5).

NdisMediumFddi
FDDI.

NdisMediumWan
WAN.

NdisMediumLocalTalk
LocalTalk.

NdisMediumDix
DIX.

NdisMediumArcnetRaw
ARCNET (raw).

NdisMediumArcnet878_2
ARCNET (878.2).

NdisMediumWirelessWan
Various types of NdisWirelessXxx media.

NdisMediumAtm
ATM.

If the underlying miniport driver returns **NULL** for this query or if an experimental media type is used, the driver must indicate receives with NdisMCoIndicateReceivePacket.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_MEDIA_SUPPORTED

Article • 02/18/2023

A complete list of the media types the NIC supports, as a proper subset of the following system-defined values:

NdisMedium802_3
Ethernet (802.3).

NdisMedium802_5
Token Ring (802.5).

NdisMediumFddi
FDDI.

NdisMediumWan
WAN.

NdisMediumLocalTalk
LocalTalk.

NdisMediumDix
DEC/Intel/Xerox (DIX) Ethernet.

NdisMediumArcnetRaw
ARCNET (raw).

NdisMediumArcnet878_2
ARCNET (878.2).

NdisMediumWirelessWan
Various types of NdisWirelessXxx media.

NdisMediumAtm
ATM.

NdisMediumIrda
Reserved for future use on Windows 2000 and later platforms.

Remarks

A LAN-emulation driver for ATM networks declares its medium as **NdisMedium802_3** , rather than **NdisMediumAtm**. Such a driver emulates Ethernet to higher-level NDIS drivers, complies with the ATM Forum's LANE, and provides UNI signaling support.

A wireless-WAN NIC driver must report its medium type as **NdisMediumWirelessWan**. However, such a miniport driver also must provide **NdisWWDIXEthernetFrames** header format to any bound protocol that selects this format, and the miniport driver can provide its NIC's native header format as well. To support existing LAN-based protocols, the driver writer can provide an NDIS intermediate driver to "translate" a wireless NIC's native header formats and medium-specific information into a form understood by existing protocols.

If the underlying miniport driver returns **NULL** for this query or if an experimental media type is used, the driver must indicate receives with NdisMCoIndicateReceivePacket.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_MINIMUM_LINK_SPEED

Article • 02/18/2023

The OID_GEN_CO_MINIMUM_LINK_SPEED OID requests the miniport driver to return its minimum transmit and receive speeds formatted as an NDIS_CO_LINK_SPEED structure, which is defined as follows:

```c++
typedef struct _NDIS_CO_LINK_SPEED{
    ULONG   Outbound;
    ULONG   Inbound;
} NDIS_CO_LINK_SPEED, *PNDIS_CO_LINK_SPEED;
```

The members of this structure contain the following information:

Outbound
The minimum transmit speed of the NIC. The unit of measurement is 100bps, so a value of 100,000 represents a hardware bit rate of 10 Mbps.

Inbound
The minimum receive speed of the NIC. The unit of measurement is 100bps, so a value of 100,000 represents a hardware bit rate of 10 Mbps.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_NETCARD_LOAD

Article • 02/18/2023

ⓘ Note

OID_GEN_CO_NETCARD_LOAD is the same as OID_GEN_NETCARD_LOAD.

The OID_GEN_CO_NETCARD_LOAD OID returns the relative load on the transmit system of a connection-oriented miniport driver. The miniport driver derives this number by calculating the difference between the amount of data delivered for transmission from protocols and the amount of data actually sent, as indicated by the packets returned to protocols with NdisMCoSendComplete. The result is the amount of outstanding transmit data in the miniport driver at any time.

Because this statistic changes at a very high frequency, the miniport driver port should filter it. The simplest filtering method is to maintain a running average of samples of the outstanding transmit data. For example, each time MiniportCoSendPackets is called, the miniport driver could add the submitted packet size to a miniport driver-defined variable called *OutstandingBytes*. Each time the miniport driver calls NdisMCoSendComplete, the miniport driver would then subtract the returned packet size from *OutstandingBytes*. The miniport driver must also maintain a running average, which is the value that the miniport driver should return in response to the OID_GEN_CO_NETCARD_LOAD query. This variable, which could be called *RunningAverage*, must be updated on each *MiniportCoSendPackets*, as follows:

```C++
RunningAverage = [(RunningAverage * C) + (OutstandingBytes * (128 - C))] /
128;
```

In this case, 1 < *C* < 128. Larger values of *C* produce smoother filtering.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_PROTOCOL_OPTIONS

Article • 02/18/2023

A bitmask that defines optional properties of the protocol driver. A protocol informs NDIS of its properties, which can optionally take advantage of them. If the protocol driver does not set its flags on a binding, NDIS assumes they are all clear.

The following flags are currently defined:

NDIS_PROT_OPTION_ESTIMATED_LENGTH
Indicates that packets can be indicated at the worst-case estimate of packet size, instead of an exact value, to this protocol.

NDIS_PROT_OPTION_NO_LOOPBACK
The protocol does not require loopback support on the binding.

NDIS_PROT_OPTION_NO_RSVD_ON_RCVPKT
The protocol does not use the **ProtocolReserved** section of indicated receive packets. This allows NDIS to indicate a receive packet to more than one protocol.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_RCV_CRC_ERROR

Article • 02/18/2023

The OID_GEN_CO_RCV_CRC_ERROR OID specifies the number of PDUs received with cyclic redundancy check (CRC) errors.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_RCV_PDUS_ERROR

Article • 02/18/2023

The OID_GEN_CO_RCV_PDUS_ERROR OID specifies the number of PDUs that a NIC received but did not indicate to bound protocols due to errors.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_RCV_PDUS_NO_BUFFER

Article • 02/18/2023

The OID_GEN_CO_RCV_PDUS_NO_BUFFER OID specifies the number of PDUs that the NIC could not receive because of a lack of NIC receive buffer space. Instead of providing the exact number, some NICs provide only the number of times that they have missed at least one PDU because of such a problem.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_RCV_PDUS_OK

Article • 02/18/2023

The OID_GEN_CO_RCV_PDUS_OK OID specifies the number of PDUs the NIC received without errors and indicated to bound protocols.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_SUPPORTED_GUIDS

Article • 02/18/2023

The OID_GEN_CO_SUPPORTED_GUIDS OID requests the miniport driver to return an array of structures of the type NDIS_GUID. Each structure in the array specifies the mapping of a custom GUID (globally unique identifier) to either a custom OID or to an NDIS_STATUS that the miniport driver sends through NdisMCoIndicateStatusEx.

The NDIS_GUID structure is defined as follows:

C++

```
typedef struct _NDIS_GUID {
    GUID    Guid;
    union {
        NDIS_OID    Oid;
        NDIS_STATUS Status;
    };
    ULONG   Size;
    ULONG   Flags;
} NDIS_GUID, *PNDIS_GUID;
```

The members of this structure contain the following information:

Guid
The custom GUID defined for the miniport driver.

Oid
The custom OID to which **Guid** maps.

Status
The NDIS_STATUS to which **Guid** maps.

Size
When the fNDIS_GUID_ARRAY flag is set, **Size** specifies the size in bytes of each data item in the array returned by the miniport driver. If the fNDIS_GUID_ANSI_STRING or fNDIS_GUID_NDIS_STRING flag is set, **Size** is set to -1. Otherwise, **Size** specifies the size in bytes of the data item that the GUID represents.

Flags
The following flags can be ORed together to indicate whether the GUID maps to an OID or to an NDIS_STATUS string and to indicate the type of data supplied for the GUID:

fNDIS_GUID_TO_OID
When set, indicates that the NDIS_GUID structure maps a GUID to an OID.

fNDIS_GUID_TO_STATUS
When set, indicates that the NDIS_GUID structure maps a GUID to an NDIS_STATUS string.

fNDIS_GUID_ANSI_STRING
When set, indicates that a null-terminated ANSI string is supplied for the GUID.

fNDIS_GUID_UNICODE_STRING
When set, indicates that a Unicode string is supplied for the GUID.

fNDIS_GUID_ARRAY
When set, indicates that an array of data items is supplied for the GUID. The specified Size indicates the length of each data item in the array.

fNDIS_GUID_ALLOW_READ
When set, indicates that all users are allowed to query this GUID.

fNDIS_GUID_ALLOW_WRITE
When set, indicates that all users are allowed to set this GUID.

Remarks

> **Note**
>
> By default, custom WMI GUIDs supplied by a miniport driver are only accessible to users with administrator privileges. A user with administrator privileges can always read or write to a custom GUID if the miniport driver supports the read or write operation for that GUID. Set the fNDIS_GUID_ALLOW_READ and fNDIS_GUID_ALLOW_WRITE flags to allow all users to access a custom GUID.

Note that all custom GUIDs registered by a miniport driver must set either fNDIS_GUID_TO_OID or fNDIS_GUID_TO_STATUS (never set both). All other flags may be combined by using the OR operator as applicable.

In the following example, an NDIS_GUID structure maps a GUID to OID_GEN_CO_RCV_PDUS_NO_BUFFER:

C++

```
NDIS_GUID NdisGuid =  {{0x0a214809, 0xe35f, 0x11d0, 0x96, 0x92, 0x00,
 0xc0, 0x4f, 0xc3, 0x35, 0x8c},
 GUID_NDIS_GEN_CO_RCV_PDUS_NO_BUFFER,
 OID_GEN_CO_RCV_PDUS_NO_BUFFER,
 4,
 fNDIS_GUID_TO_OID};
```

A GUID is an identifier used by Windows Management Instrumentation (WMI) to obtain or set information. NDIS intercepts a GUID sent by WMI to an NDIS driver, maps the GUID to an OID, and sends the OID to the driver. The driver returns the data item(s) to NDIS, which then returns the data to WMI.

NDIS also translates changes in NIC status into GUIDs recognized by WMI. When a miniport driver reports a change in NIC status with NdisMCoIndicateStatusEx, NDIS translates the NDIS_STATUS indicated by the miniport driver into a GUID that NDIS sends to WMI.

If a connection-oriented miniport driver supports customs GUIDs, it must support OID_GEN_CO_SUPPORTED_GUIDS, which returns to NDIS the mapping of custom GUIDs to custom OIDs or NDIS_STATUS strings. After querying the miniport driver with OID_GEN_CO_SUPPORTED_GUIDS, NDIS registers the miniport driver's custom GUIDs with WMI.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_SUPPORTED_LIST

Article • 02/18/2023

The OID_GEN_CO_SUPPORTED_LIST OID specifies an array of OIDs for objects that the underlying driver or its NIC supports. Objects include general, media-specific, and implementation-specific objects.

The underlying driver should order the OID list it returns in increasing numeric order. NDIS forwards a subset of the returned list to protocols that make this query. That is, NDIS filters any supported statistics OIDs out of the list since protocols never make statistics queries subsequently.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_TRANSMIT_QUEUE_LENGTH

Article • 02/18/2023

The OID_GEN_CO_TRANSMIT_QUEUE_LENGTH OID specifies the number of PDUs currently queued for transmission, whether on the NIC or in a driver-internal queue. The number returned is always the total number of PDUs currently queued, which can include unsubmitted send requests queued in the NDIS library.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_VENDOR_DESCRIPTION

Article • 02/18/2023

A pointer to a null-terminated, counted string describing the NIC.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_VENDOR_DRIVER_VERSION

Article • 02/18/2023

The vendor-assigned version number of the NIC driver.

This OID is two bytes in length; the low-order half of the return value specifies the minor version, while the high-order half specifies the major version.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_VENDOR_ID

Article • 02/18/2023

A 3-byte IEEE-registered vendor code, followed by a single byte that the vendor assigns to identify a particular NIC.

The IEEE code uniquely identifies the vendor and is the same as the three bytes appearing at the beginning of the NIC hardware address.

Vendors without an IEEE-registered code should use the value 0xFFFFFF.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_XMIT_PDUS_ERROR

Article • 02/18/2023

The OID_GEN_CO_XMIT_PDUS_ERROR OID specifies the number of PDUs a NIC failed to transmit.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CO_XMIT_PDUS_OK

Article • 02/18/2023

The OID_GEN_CO_XMIT_PDUS_OK OID specifies the number of PDUs transmitted without errors.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_CURRENT_LOOKAHEAD

Article • 02/18/2023

As a query, the OID_GEN_CURRENT_LOOKAHEAD OID returns the number of bytes of received packet data that will be indicated to the protocol driver. This specification does not include the header.

As a set, the OID_GEN_CURRENT_LOOKAHEAD OID specifies the number of bytes of received packet data that the miniport driver should indicate to the protocol driver. This specification does not include the header.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory. (see Remarks section)

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

NDIS handles query and unsuccessful set requests for NDIS 6.0 and later miniport drivers. NDIS obtains the information from the miniport driver during initialization and miniport adapter restart. However, NDIS sends valid set requests to the miniport driver.

For a query, NDIS returns the largest lookahead size from among all the bindings. A protocol driver can set a suggested value for the number of bytes to be used in its binding; however, the underlying miniport driver is never required to limit its indications to the value set.

If the underlying driver supports multipacket receive indications, bound protocol drivers are given full net packets on every indication. Consequently, this value is identical to that returned for OID_GEN_RECEIVE_BLOCK_SIZE.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_RECEIVE_BLOCK_SIZE

OID_GEN_CURRENT_PACKET_FILTER

Article • 02/18/2023

As a query, the OID_GEN_CURRENT_PACKET_FILTER OID reports the types of net packets that are in receive indications from a miniport driver.

As a set, the OID_GEN_CURRENT_PACKET_FILTER OID specifies the types of net packets for which a protocol receives indications from a miniport driver.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory. (see Remarks section)

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

For NDIS 6.0 and later miniport drivers, the query is not requested and the set is mandatory. NDIS handles the query for miniport drivers. The miniport driver reports the packet filter information during initialization.

The miniport driver reports its medium type as one for which the system provides a filter library. The packet filter uses the OR operation to inclusively combine the following types:

NDIS_PACKET_TYPE_DIRECTED
Directed packets. Directed packets contain a destination address equal to the station address of the NIC.

NDIS_PACKET_TYPE_MULTICAST
Multicast address packets sent to addresses in the multicast address list.

A protocol driver can receive Ethernet (802.3) multicast packets by specifying the multicast or functional address packet type. Setting the multicast address list or functional address determines which multicast address groups the NIC driver enables.

NDIS_PACKET_TYPE_ALL_MULTICAST
All multicast address packets, not just the ones enumerated in the multicast address list.

NDIS_PACKET_TYPE_BROADCAST
Broadcast packets.

NDIS_PACKET_TYPE_PROMISCUOUS
Specifies all packets regardless of whether VLAN filtering is enabled or not and whether the VLAN identifier matches or not.

NDIS_PACKET_TYPE_ALL_FUNCTIONAL
All functional address packets, not just the ones in the current functional address.

NDIS_PACKET_TYPE_ALL_LOCAL
All packets sent by installed protocols and all packets indicated by the NIC that is identified by a given *NdisBindingHandle* .

NDIS_PACKET_TYPE_FUNCTIONAL
Functional address packets sent to addresses included in the current functional address.

NDIS_PACKET_TYPE_GROUP
Packets sent to the current group address.

NDIS_PACKET_TYPE_MAC_FRAME
NIC driver frames that a Token Ring NIC receives.

NDIS_PACKET_TYPE_SMT
SMT packets that an FDDI NIC receives.

NDIS_PACKET_TYPE_SOURCE_ROUTING
All source routing packets. If the protocol driver sets this bit, the NDIS library attempts to act as a source routing bridge.

For miniport adapters whose media type is **NdisMedium802_3** or **NdisMedium802_5**, NDIS disables packet reception, along with multicast and functional addresses during a call to the NdisOpenAdapterEx function.

For miniport adapters with all other media types, the protocol driver can begin receiving packets at any time during the NdisOpenAdapterEx call. Note that the protocol can even receive packets before **NdisOpenAdapterEx** returns. In general, packet filtering is best effort, and protocol drivers must be prepared to handle receive indications even when the packet filter is zero.

For a query, NDIS returns the binding filters that are combined using the OR operator.

For a set, the specified packet filter replaces the previous packet filter for the binding. If the miniport driver previously enabled a packet type but the protocol driver does not specify that type in a new filter, the protocol driver will not receive packets of this type.

For miniport adapters whose media type is **NdisMedium802_3** or **NdisMedium802_5**, if the miniport driver does not set a bit for a particular packet type in response to this query, the protocol driver will not receive packets of that type. Consequently, a protocol driver can disable packet reception by calling the **NdisOidRequest** or **NdisCoOidRequest** function using a filter of zero.

For miniport adapters with all other media types, NDIS does not check the packet type. For these media types, a protocol driver cannot disable packet reception by specifying a filter of zero.

When a miniport driver's *MiniportInitializeEx* function is called, the miniport driver's packet filter should be set to zero. When the packet filter is zero, receive indications are disabled. After a miniport driver's *MiniportInitializeEx* function has returned, a protocol driver can set OID_GEN_CURRENT_PACKET_FILTER to a nonzero value, thereby enabling the miniport driver to indicate received packets to that protocol.

If promiscuous mode is enabled with the NDIS_PACKET_TYPE_PROMISCUOUS bit, the protocol driver continues to receive packets even if the sending network node does not direct them to it. NDIS then sends the protocol driver all packets the NIC receives.

Setting a specific packet filter does not alter the packet filter for other protocol drivers that are bound to (or above) the same NIC. For example, if one bound protocol enables promiscuous mode, other bound protocol drivers do not receive packets that they have not specifically requested with their own packet filters.

Native 802.11 Packet Filters

The Native 802.11 miniport driver must only support the following standard packet filter types:

- NDIS_PACKET_TYPE_DIRECTED
- NDIS_PACKET_TYPE_MULTICAST
- NDIS_PACKET_TYPE_BROADCAST

- NDIS_PACKET_TYPE_PROMISCUOUS

When enabled, these standard packet filters are only applicable to 802.11 data packets.

In addition, the Native 802.11 miniport driver must support the following packet filter types, which are specific to the Native 802.11 media:

NDIS_PACKET_TYPE_802_11_RAW_DATA
An 802.11 media access control (MAC) protocol data unit (MPDU) frame, which contains all of the data in the format received by the 802.11 station. When this filter is set, the driver must indicate every unmodified MPDU fragment before it indicates the MAC service data unit (MSDU) packet reassembled from the MPDU fragments.

If an MPDU fragment is encrypted, it must not decrypt the fragment before it is indicated. However, the miniport driver must decrypt each MPDU fragment before reassembling and indicating the MSDU packet.

If enabled, this filter type only affects other standard packet filters, such as NDIS_PACKET_TYPE_DIRECTED or NDIS_PACKET_TYPE_BROADCAST.

For more information about the method for indicating raw 802.11 data packets, see Indicating Raw 802.11 Packets.

NDIS_PACKET_TYPE_802_11_DIRECTED_MGMT
Directed 802.11 management packets. Directed packets contain a destination address equal to the station address of the NIC.

NDIS_PACKET_TYPE_802_11_MULTICAST_MGMT
Multicast 802.11 management packets sent to addresses in the multicast address list.

NDIS_PACKET_TYPE_802_11_ALL_MULTICAST_MGMT
All multicast 802.11 management packets received by the 802.11 station, regardless of whether the destination address in the 802.11 MAC header is in the multicast address list.

NDIS_PACKET_TYPE_802_11_BROADCAST_MGMT
Broadcast 802.11 management packets received by the 802.11 station.

NDIS_PACKET_TYPE_802_11_PROMISCUOUS_MGMT
All 802.11 management packets received by the 802.11 station.

NDIS_PACKET_TYPE_802_11_RAW_MGMT
An 802.11 MPDU management frame, which contains all of the data in the format received by the 802.11 station. When this filter is set, the driver must indicate every

unmodified MPDU fragment before it indicates the MAC management protocol data unit (MMPDU) packet reassembled from the MPDU fragments.

If enabled, this filter type only affects other 802.11 management packet filters, such as NDIS_PACKET_TYPE_802_11_DIRECTED_MGMT or NDIS_PACKET_TYPE_802_11_MULTICAST_MGMT.

For more information about the method for indicating raw 802.11 management packets, see Indicating Raw 802.11 Packets.

NDIS_PACKET_TYPE_802_11_DIRECTED_CTRL
Directed 802.11 control packets. Directed packets contain a destination address equal to the station address of the NIC.

NDIS_PACKET_TYPE_802_11_BROADCAST_CTRL
Broadcast 802.11 control packets received by the 802.11 station.

NDIS_PACKET_TYPE_802_11_PROMISCUOUS_CTRL
All 802.11 control packets received by the 802.11 station.

If a miniport driver is operating in Native 802.11 Network Monitor (NetMon) or Extensible Access Point (AP) modes, the driver must enable the following packet filters through a set request of OID_GEN_CURRENT_PACKET_FILTER.

- NDIS_PACKET_TYPE_PROMISCUOUS
- NDIS_PACKET_TYPE_802_11_RAW_DATA
- NDIS_PACKET_TYPE_802_11_PROMISCUOUS_MGMT
- NDIS_PACKET_TYPE_802_11_RAW_MGMT
- NDIS_PACKET_TYPE_802_11_PROMISCUOUS_CTRL

A miniport driver operating in other Native 802.11 modes besides NetMon must not enable these packet filter settings, with the exception of NDIS_PACKET_TYPE_802_11_PROMISCUOUS_CTRL. A miniport driver that is not operating in NetMon mode can optionally enable NDIS_PACKET_TYPE_802_11_PROMISCUOUS_CTRL through a set request of OID_GEN_CURRENT_PACKET_FILTER.

Note When the miniport driver is in Native 802.11 modes other than NetMon, and OID_GEN_CURRENT_PACKET_FILTER is set, the driver must not fail the set request if any promiscuous or raw filter settings are enabled in the OID data.

For more information about the NetMon and ExtAP operating modes, see the following topics:

Network Monitor Operation Mode

Extensible Access Point Operation Mode

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

NdisCoOidRequest

NdisOidRequest

NdisOpenAdapterEx

OID_GEN_DEVICE_PROFILE

Article • 02/18/2023

The OID_GEN_DEVICE_PROFILE OID is obsolete. NDIS and NDIS drivers do not use this OID.

Requirements

Version	Not supported.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_DIRECTED_BYTES_RCV

Article • 02/18/2023

As a query, the OID_GEN_DIRECTED_BYTES_RCV OID specifies the number of bytes in directed packets that are received without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

The count is identical to the *ifInUcastPkts* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_DIRECTED_BYTES_XMIT

Article • 02/18/2023

As a query, the OID_GEN_DIRECTED_BYTES_XMIT OID specifies the number of bytes in directed packets that are transmitted without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_DIRECTED_FRAMES_RCV

Article • 02/18/2023

As a query, the OID_GEN_DIRECTED_FRAMES_RCV OID specifies the number of directed packets that are received without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_DIRECTED_FRAMES_XMIT

Article • 02/18/2023

As a query, the OID_GEN_DIRECTED_FRAMES_XMIT OID specifies the number of directed packets that are transmitted without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

The count is identical to the *ifOutUcastPkts* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_DISCONTINUITY_TIME

Article • 02/18/2023

As a query, use the OID_GEN_DISCONTINUITY_TIME OID to determine the discontinuity time of a network interface (*ifCounterDiscontinuityTime* from RFC 2863).

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

This OID returns the time, starting from the last computer restart, when the interface had a discontinuity in maintaining the statistics counters. For example, there was a discontinuity because the interface was disabled or the associated adapter was removed from the computer. For more information about the statistics counters, see OID_GEN_STATISTICS. To get the current time, an interface provider can call the **NdisGetSystemUpTimeEx** function.

If no such discontinuity occurred since the last re-initialization of the interface this value should be zero. If the interface provider does not track discontinuity time, this value should be zero.

If the interface provider returns NDIS_STATUS_SUCCESS, the result of the query is a ULONG64 value that specifies the discontinuity time, in milliseconds, since the last computer restart.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS Network Interface OIDs

OID_GEN_DRIVER_VERSION

Article • 02/18/2023

As a query, the OID_GEN_DRIVER_VERSION OID specifies the NDIS version in use by the miniport driver.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

NDIS handles this OID for NDIS 6.0 and later miniport drivers.

The high byte is the major version number; the low byte is the minor version number.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_ENUMERATE_PORTS

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_ENUMERATE_PORTS OID to determine the characteristics of the active NDIS ports that are associated with an underlying miniport adapter.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

NDIS handles this OID and miniport drivers do not receive this OID query.

If the query succeeds, NDIS returns NDIS_STATUS_SUCCESS and provides the results of the query in an NDIS_PORT_ARRAY structure. The **NumberOfPorts** member of NDIS_PORT_ARRAY contains the number of active ports that are associated with the miniport adapter. The **Ports** member of NDIS_PORT_ARRAY contains a list of pointers to NDIS_PORT_CHARACTERISTICS structures. Each NDIS_PORT_CHARACTERISTICS structure defines the characteristics of a single NDIDS port.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PORT_ARRAY

NDIS_PORT_CHARACTERISTICS

OID_GEN_FRIENDLY_NAME

Article • 02/18/2023

As a query, the OID_GEN_FRIENDLY_NAME OID returns the friendly name of a miniport adapter.

Remarks

The OID_GEN_FRIENDLY_NAME OID returns the friendly name of a miniport adapter.

Friendly names are intended to help the user quickly and accurately identify a miniport adapter--for example, "PCI Ethernet Adapter" and "Virtual Private Networking Adapter" are considered friendly names.

Requirements

Version	Supported for NDIS 5.1 and later drivers in Windows Vista and later versions of Windows and later versions of Windows. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_HARDWARE_STATUS

Article • 02/18/2023

As a query, the OID_GEN_HARDWARE_STATUS OID specifies the current hardware status of the underlying NIC.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Obsolete.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

The OID_GEN_HARDWARE_STATUS OID specifies the current hardware status of the underlying NIC as one of the following NDIS_HARDWARE_STATUS-type values:

NdisHardwareStatusReady
Available and capable of sending and receiving data over the wire

NdisHardwareStatusInitializing
Initializing

NdisHardwareStatusReset
Resetting

NdisHardwareStatusClosing
Closing

NdisHardwareStatusNotReady
Not ready

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_HD_SPLIT_CURRENT_CONFIG

Article • 02/18/2023

As a query, overlying drivers or administrative utilities can use the OID_GEN_HD_SPLIT_CURRENT_CONFIG OID to determine the current header-data split configuration of a miniport adapter. A system administrator can use the GUID that is associated with this OID through the WMI interface.

Remarks

NDIS handles this OID on behalf of the miniport driver. NDIS maintains the current header-data split configuration information based on the miniport driver initialization attributes and the NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG status indication.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains an NDIS_HD_SPLIT_CURRENT_CONFIG structure.

Requirements

Version	Supported in NDIS 6.1 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_HD_SPLIT_CURRENT_CONFIG

NDIS_OID_REQUEST

NDIS_STATUS_HD_SPLIT_CURRENT_CONFIG

OID_GEN_HD_SPLIT_PARAMETERS

Article • 02/18/2023

As a set, NDIS and overlying drivers or user-mode applications use the OID_GEN_HD_SPLIT_PARAMETERS OID to set the current header-data split settings of a miniport adapter. NDIS 6.1 and later miniport drivers that provide header-data split services must support this OID. Otherwise, this OID is optional.

Remarks

The **InformationBuffer** member of NDIS_OID_REQUEST structure contains an NDIS_HD_SPLIT_PARAMETERS structure.

NDIS might set the OID_GEN_HD_SPLIT_PARAMETERS OID when an NDIS 5.*x* protocol driver binds to an NDIS 6.1 miniport. NDIS processes this OID before passing it to the miniport driver and updates the miniport adapter's ***HeaderDataSplit** standardized keyword, if required. If header-data split is disabled, NDIS does not send this OID to the miniport adapter.

NDIS will send this OID to the miniport driver only if header-data split was enabled with the NDIS_HD_SPLIT_ENABLE_HEADER_DATA_SPLIT flag in the NDIS_HD_SPLIT_ATTRIBUTES structure during miniport initialization.

Requirements

Version	Supported in NDIS 6.1 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_HD_SPLIT_ATTRIBUTES

NDIS_HD_SPLIT_PARAMETERS

NDIS_OID_REQUEST

OID_GEN_INIT_TIME_MS

Article • 02/18/2023

As a query, the OID_GEN_INIT_TIME_MS OID returns the time in milliseconds that a driver required to initialize.

Remarks

The OID_GEN_INIT_TIME_MS OID returns the time in milliseconds that a driver required to initialize.

Requirements

Version	Supported for NDIS 5.1 and later drivers in Windows Vista and later versions of Windows and later versions of Windows. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_INTERFACE_INFO

Article • 02/18/2023

As a query, use the OID_GEN_INTERFACE_INFO OID to obtain the current state and statistics information for a network interface.

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the query succeeds, the interface provider returns NDIS_STATUS_SUCCESS, and the result of the query is an **NDIS_INTERFACE_INFORMATION** structure. This structure contains information that changes during the lifetime of the interface.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_INTERFACE_INFORMATION

NDIS Network Interface OIDs

OID_GEN_INTERRUPT_MODERATION

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_INTERRUPT_MODERATION OID to determine if interrupt moderation is enabled on a miniport adapter. If the query succeeds, NDIS returns an NDIS_INTERRUPT_MODERATION_PARAMETERS structure that contains the current interrupt moderation settings.

As a set, NDIS and overlying drivers use the OID_GEN_INTERRUPT_MODERATION OID to enable or disable the interrupt moderation on a miniport adapter.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory. Set and query.

Remarks

For a query, if a miniport driver does not support interrupt moderation, the driver must specify **NdisInterruptModerationNotSupported** in the **InterruptModeration** member of the NDIS_INTERRUPT_MODERATION_PARAMETERS structure.

For a set, if the driver reported **NdisInterruptModerationNotSupported** in response to the OID_GEN_INTERRUPT_MODERATION query, the driver should return NDIS_STATUS_INVALID_DATA in response to the set request. The miniport driver receives an NDIS_INTERRUPT_MODERATION_PARAMETERS structure. If the **InterruptModeration** member of NDIS_INTERRUPT_MODERATION_PARAMETERS is set to **NdisInterruptModerationEnabled**, the miniport driver should enable interrupt moderation. Otherwise, it should disable interrupt moderation.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_INTERRUPT_MODERATION_PARAMETERS

OID_GEN_ISOLATION_PARAMETERS

Article • 02/18/2023

NDIS and overlying drivers issue an object identifier (OID) request of OID_GEN_ISOLATION_PARAMETERS to obtain the multi-tenancy configuration (isolation) parameters that are set on a VM network adapter's port.

Although each routing domain is configured separately on the port, this OID returns parameters for all of the routing domains in a single query.

An overlying driver should issue this OID in two steps:

1. Io query the required buffer size, issue the OID query with the **Size** member of the **Header** member of the NDIS_ISOLATION_PARAMETERS structure set to **NDIS_SIZEOF_NDIS_ISOLATION_PARAMETERS_REVISION_1**. (See **NDIS_STATUS_INVALID_LENGTH** below.)
2. Issue the OID with an **InformationBuffer** of the required size.

If the OID query request is completed successfully, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data, in order:

1. An NDIS_ISOLATION_PARAMETERS structure

2. One or more NDIS_ROUTING_DOMAIN_ENTRY structures, one for each routing domain

3. One or more NDIS_ROUTING_DOMAIN_ISOLATION_ENTRY structures, grouped by routing domain

In each NDIS_ROUTING_DOMAIN_ENTRY structure, the **FirstIsolationInfoEntryOffset** member contains the offset from the beginning of the OID information buffer (that is, the beginning of the buffer that the **InformationBuffer** member of the NDIS_OID_REQUEST structure points to) to the first NDIS_ROUTING_DOMAIN_ISOLATION_ENTRY for that routing domain. The offset in the **NextIsolationInfoEntryOffset** member of the last structure in the list is zero.

If no multi-tenancy configuration parameters are set on the VM network adapter, the network adapter miniport driver sets the **DATA.QUERY_INFORMATION.BytesWritten** member of the NDIS_OID_REQUEST structure to zero and returns **NDIS_STATUS_SUCCESS**. In this case, the data within the **DATA.QUERY_INFORMATION.InformationBuffer** member is not modified by the miniport driver.

Remarks

Return Status Codes

The VM network adapter miniport driver returns one of the following status codes for this OID request:

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to return the requested information. The VM network adapter miniport driver sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size, in bytes, that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.40 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_ISOLATION_PARAMETERS

NDIS_OID_REQUEST

NDIS_ROUTING_DOMAIN_ENTRY

NDIS_ROUTING_DOMAIN_ISOLATION_ENTRY

NDIS_STATUS_ISOLATION_PARAMETERS_CHANGE

OID_GEN_LAST_CHANGE

Article • 02/18/2023

As a query, use the OID_GEN_LAST_CHANGE OID to determine the time of the last operational state change of a network interface (*ifLastChange* from RFC 2863).

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

This OID returns the time, starting from the last computer restart, when the interface entered its current operational state. For more information about the operational state, see **NDIS_STATUS_OPER_STATUS** and OID_GEN_OPERATIONAL_STATUS. To get the current time, an interface provider can call the **NdisGetSystemUpTimeEx** function.

If the current operational state was entered before the last reinitialization of the interface, this value should be zero. . If the interface provider does not track operational state change time, the value should be zero.

If the interface provider returns NDIS_STATUS_SUCCESS, the result of the query is a ULONG64 value that specifies the state change time, in milliseconds, since the last computer restart.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_OPERATIONAL_STATUS

NDIS_STATUS_OPER_STATUS

NdisGetSystemUpTimeEx

NDIS Network Interface OIDs

OID_GEN_LINK_PARAMETERS

Article • 02/18/2023

As a set, NDIS and overlying drivers use the OID_GEN_LINK_PARAMETERS OID to set the current link state of a miniport adapter. The miniport driver receives the duplex state, link speeds, and pause functions in an NDIS_LINK_PARAMETERS structure.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Optional.

The NDIS_LINK_PARAMETERS structure is defined as follows:

ManagedCPlusPlus

```
    typedef struct _NDIS_LINK_PARAMETERS {
         NDIS_OBJECT_HEADER Header;
         NDIS_MEDIA_DUPLEX_STATE MediaDuplexState;
         ULONG64 XmitLinkSpeed;
         ULONG64 RcvLinkSpeed;
         NDIS_SUPPORTED_PAUSE_FUNCTIONS PauseFunctions;
         ULONG AutoNegotiationFlags;
    } NDIS_LINK_PARAMETERS, *PNDIS_LINK_PARAMETERS;
```

This structure contains the following members:

Header
The **NDIS_OBJECT_HEADER** structure for the NDIS_LINK_PARAMETERS structure. Set the **Type** member of the structure that **Header** specifies to NDIS_OBJECT_TYPE_DEFAULT, the **Revision** member to NDIS_LINK_PARAMETERS_REVISION_1, and the **Size** member to NDIS_SIZEOF_LINK_PARAMETERS_REVISION_1.

MediaDuplexState
The media duplex state. This value is the same as the value that is returned by the OID_GEN_MEDIA_DUPLEX_STATE OID.

XmitLinkSpeed
The transmit link speed in bits per second.

RcvLinkSpeed
The receive link speed in bits per second.

PauseFunctions
The type of support for the IEEE 802.3 pause frames. This member must be one of the following pause functions:

NdisPauseFunctionsUnsupported
The adapter or link partner does not support pause frames.

NdisPauseFunctionsSendOnly
The adapter and link partner support only sending pause frames from the adapter to the link partner.

NdisPauseFunctionsReceiveOnly
The adapter and link partner support only sending pause frames from the link partner to the adapter

NdisPauseFunctionsSendAndReceive
The adapter and link partner support sending and receiving pause frames in both transmit and receive directions.

AutoNegotiationFlags
The auto-negotiation settings for the miniport adapter. This member is created from a bitwise OR of the following flags:

NDIS_LINK_STATE_XMIT_LINK_SPEED_AUTO_NEGOTIATED
The adapter should auto-negotiate the transmit link speed with the link partner. If this flag is not set, the miniport driver should set the transmit link speed to the value that is specified in the **XmitLinkSpeed** member.

NDIS_LINK_STATE_RCV_LINK_SPEED_AUTO_NEGOTIATED
The adapter should auto-negotiate the receive link speed with the link partner. If this flag is not set, the miniport driver should set the receive link speed to the value that is specified in the **RcvLinkSpeed** member.

NDIS_LINK_STATE_DUPLEX_AUTO_NEGOTIATED
The adapter should auto-negotiate the duplex state with the link partner. If this flag is not set, the miniport driver should set the duplex state to the value that is specified in the **MediaDuplexState** member.

NDIS_LINK_STATE_PAUSE_FUNCTIONS_AUTO_NEGOTIATED
The miniport driver should auto-negotiate the support for pause frames with the other end. If this flag is not set, the miniport driver should use the pause frame support that is specified in the **PauseFunctions** member.

Remarks

Note Setting OID_GEN_LINK_PARAMETERS can cause a loss of connectivity. Miniport drivers must reconfigure the miniport adapter when this OID is set. For example, the miniport driver can reset the miniport adapter with the resulting loss of existing connections. The specific mechanism for reconfiguration is application dependent.

If the link state of the miniport adapter changes because of the OID_GEN_LINK_PARAMETERS set request, the miniport driver should generate an NDIS_STATUS_LINK_STATE status indication to notify NDIS and overlying drivers of the new link state.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OBJECT_HEADER

NDIS_STATUS_LINK_STATE

OID_GEN_MEDIA_DUPLEX_STATE

OID_GEN_LINK_SPEED

Article • 02/18/2023

As a query, the OID_GEN_LINK_SPEED OID specifies the maximum speed of the NIC.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later miniport drivers
Obsolete. (see Remarks section)

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

The OID_GEN_LINK_STATE is the NDIS 6.0 and later and later equivalent of this OID. However NDIS 6.0 and later miniport drivers must use the **NDIS_STATUS_LINK_STATE** status indication instead to indicate link speed changes.

The unit of measurement is 100 bps, so a value of 100,000 represents a hardware bit rate of 10 Mbps.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_LINK_STATE

OID_GEN_LINK_STATE

OID_GEN_LINK_SPEED_EX

Article • 02/18/2023

As a query, the OID_GEN_LINK_SPEED_EX OID provides the transmit and receive link speeds of an interface.

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

NDIS uses this OID to query the link speed of an NDIS network interface provider. Only NDIS interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

This OID returns the link speeds in an **NDIS_LINK_SPEED** structure.

Miniport drivers supply the link speed during initialization and provide updated link speeds with status indications.

To specify the link speeds in a miniport driver, set the **XmitLinkSpeed** and **RcvLinkSpeed** members of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure that the miniport driver passes to the **NdisMSetMiniportAttributes** function.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_LINK_SPEED

NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES

NdisMSetMiniportAttributes

NDIS Network Interface OIDs

OID_GEN_LINK_STATE

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_LINK_STATE OID to determine the current link state of a miniport adapter. The miniport driver receives the link state in an NDIS_LINK_STATE structure.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

Miniport drivers supply the link state during initialization and provide updates with status indications.

To specify the link state, set the **MediaConnectState**, **MediaDuplexState**, **XmitLinkSpeed**, **RcvLinkSpeed**, **PauseFunctions**, and **AutoNegotiationFlags** members of the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure that the miniport driver passes to the **NdisMSetMiniportAttributes** function.

If a miniport driver does not support this OID, the driver should return NDIS_STATUS_NOT_SUPPORTED. If the miniport driver supports this OID, it returns the connection state, duplex state, and link speeds in an NDIS_LINK_STATE structure.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_LINK_STATE

NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES

NDIS_OBJECT_HEADER

NdisMSetMiniportAttributes

OID_GEN_MEDIA_CONNECT_STATUS_EX

OID_GEN_MEDIA_DUPLEX_STATE

OID_GEN_MAC_OPTIONS

Article • 02/18/2023

As a query, the OID_GEN_MAC_OPTIONS OID specifies a bitmask that defines optional properties of the underlying driver or a NIC.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

NDIS handles this OID for NDIS 6.0 and later miniport drivers.

A protocol that initiates this query can determine which of the flags the underlying driver sets, and can optionally take advantage of them.

The following flags are currently defined:

NDIS_MAC_OPTION_COPY_LOOKAHEAD_DATA
The protocol driver is free to access indicated data by any means. Some fast-copy functions have trouble accessing on-board device memory. Miniport drivers that indicate data out of mapped device memory should never set this flag. If a miniport driver does set this flag, it relaxes the restriction on fast-copy functions.

NDIS_MAC_OPTION_RECEIVE_SERIALIZED
The miniport driver indicates packets in a serial manner. That is, such a driver does not enter a new receive indication until the previous receive, if any, has been completed.

NDIS_MAC_OPTION_TRANSFERS_NOT_PEND
The miniport driver never completes receive indications asynchronously.

A miniport driver that indicates receive operations with the **NdisMIndicateReceiveNetBufferLists** function should set this flag.

NDIS_MAC_OPTION_NO_LOOPBACK
The NIC has no internal loopback support so NDIS will manage loopbacks on behalf of this driver. A miniport driver cannot provide its own software loopback as efficiently as NDIS, so every miniport driver should set this flag unless a NIC has hardware loopback support. WAN miniport drivers must set this flag.

NDIS_MAC_OPTION_FULL_DUPLEX
The miniport driver supports full-duplex transmits and indications on SMP platforms.

Note This flag has been deprecated for use by NDIS 5.0 and later miniport drivers. NDIS 5.0 and later ignores this flag.

NDIS_MAC_OPTION_EOTX_INDICATION
This flag is obsolete.

NDIS_MAC_OPTION_8021P_PRIORITY
The NIC and its driver support 802.1p packet priority. For more information, see Packet Priority. Packet-priority values are received in **NET_BUFFER** structures from higher-layer drivers. The appropriate information is generated in the MAC headers of packets and transmitted over the network. In addition, this NIC and its driver support extracting the appropriate information from the MAC headers of packets received from the network. This information is forwarded in NET_BUFFER structures to higher-layer drivers.

Note NDIS 6.0 and later and later and later miniport drivers must set the NDIS_MAC_OPTION_8021P_PRIORITY flag.

NDIS_MAC_OPTION_SUPPORTS_MAC_ADDRESS_OVERWRITE
NDIS sets this flag when a miniport driver calls the **NdisReadNetworkAddress** function.

NDIS_MAC_OPTION_RECEIVE_AT_DPC
This flag is obsolete.

NDIS_MAC_OPTION_8021Q_VLAN
The miniport driver can assign and remove VLAN identifier (ID) marking in the MAC headers of packets. The driver maintains a configured VLAN ID for each NIC that the driver handles. The driver filters out incoming packets that do not belong to the VLAN to which a NIC is associated and marks outgoing packets with the VLAN ID. During the driver's *MiniportInitializeEx* function for a particular NIC, the driver initially sets the NIC's

VLAN ID to zero. The driver's *MiniportInitializeEx* function then reads the following configuration parameter from the registry, and, if the parameter is present, sets the NIC's VLAN ID to the parameter's value.

syntax

```
VlanId, REG_DWORD
```

NDIS_MAC_OPTION_RESERVED
Reserved for NDIS internal use.

Note A miniport driver that sets the NDIS_MAC_OPTION_8021Q_VLAN flag must also set the NDIS_MAC_OPTION_8021P_PRIORITY flag. In other words, a miniport driver that supports 802.1Q must also support 802.1p.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

NdisReadNetworkAddress

NET_BUFFER

OID_GEN_MACHINE_NAME

Article • 02/18/2023

As a set, the OID_GEN_MACHINE_NAME OID indicates the local computer name to a miniport driver.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Optional.

NDIS 5.1 miniport drivers
Optional.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Optional.

Remarks

The information buffer passed in this request contains an array of Unicode characters that represents the local computer name. The **InformationBufferLength** value that is supplied to the *MiniportOidRequest* function specifies the length of this array in bytes, not including a NULL terminator.

NDIS sets OID_GEN_MACHINE_NAME only once after a miniport driver completes initialization. Under Windows XP, NDIS does not dynamically notify miniport drivers of a change in the computer name. After changing the computer name, a user must restart the computer so that NDIS notifies miniport drivers of the new computer name.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

MiniportOidRequest

OID_GEN_MAX_LINK_SPEED

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_MAX_LINK_SPEED OID to determine the maximum supported transmit and receive link speeds of a miniport adapter.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

The miniport driver supplies the maximum link speed during initialization.

To specify the maximum link speeds, set the **MaxXmitLinkSpeed** and **MaxRcvLinkSpeed** members of the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure that the miniport driver passes to the NdisMSetMiniportAttributes function. If a miniport driver does not support this OID, the driver should return NDIS_STATUS_NOT_SUPPORTED. If the miniport driver supports this OID, it returns the maximum link speeds in an NDIS_LINK_SPEED structure.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_LINK_SPEED

NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES

NdisMSetMiniportAttributes

OID_GEN_MAXIMUM_FRAME_SIZE

Article • 02/18/2023

As a query, the OID_GEN_MAXIMUM_FRAME_SIZE OID specifies the maximum network packet size, in bytes, that the NIC supports. This specification does not include a header.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not Requested.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

The miniport driver supplies the maximum frame size during initialization and during a restart. NDIS handles this OID query for NDIS 6.0 and later miniport drivers.

In response to this query from requesting transports, the miniport driver should indicate the maximum frame size that the transports can send, excluding the header. A miniport driver that emulates another medium type for binding to a transport must ensure that the maximum frame size for a protocol-supplied net packet does not exceed the size limitations for the true network medium. The same is true for a miniport driver that supports a NIC that requires inserting fields in frames. For example, to determine the maximum transfer unit (MTU), transports send this query to a NIC.

If the miniport driver supports 802.1p packet priority and 802.1Q virtual LAN (VLAN) tags, based on prior actions, if the miniport driver expects that frames must traverse old networks before priority values are removed, that miniport driver might indicate a smaller value in response to this query.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_MAXIMUM_LOOKAHEAD

Article • 02/18/2023

As a query, the OID_GEN_MAXIMUM_LOOKAHEAD OID specifies the maximum number of bytes that the NIC can provide as lookahead data. This specification does not include a header.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

NDIS 6.0 and later miniport drivers do not receive this OID request. NDIS handles this OID with a cached value that miniport drivers supply during initialization.

Upper-layer drivers examine lookahead data to determine whether a packet that is associated with the lookahead data is intended for one or more of their clients.

If the underlying driver supports multipacket receive indications, bound protocol drivers are given full net packets on every indication. Consequently, this value is identical to that returned for OID_GEN_RECEIVE_BLOCK_SIZE.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_RECEIVE_BLOCK_SIZE

OID_GEN_MAXIMUM_SEND_PACKETS

Article • 02/18/2023

As a query, the OID_GEN_MAXIMUM_SEND_PACKETS OID specifies the maximum number of send packet descriptors that a miniport driver's *MiniportSendPackets* function can accept.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later miniport drivers
Obsolete.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

NDIS ignores any value returned by a deserialized driver in response to a query of OID_GEN_MAXIMUM_SEND_PACKETS. NDIS does not adjust the size of the array of packet descriptors that it supplies to a deserialized miniport driver's *MiniportSendPackets* function.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

MiniportSendPackets

OID_GEN_MAXIMUM_TOTAL_SIZE

Article • 02/18/2023

As a query, the OID_GEN_MAXIMUM_TOTAL_SIZE OID specifies the maximum total packet length, in bytes, the NIC supports. This specification includes the header.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

The returned length specifies the largest packet size for the underlying medium. Thus, the returned length depends on the particular medium. A protocol driver might use this returned length as a gauge to determine the maximum size packet that a miniport driver could forward to the protocol driver. If the protocol driver preallocates buffers, it allocates buffers accordingly. The returned length also specifies the largest packet that a protocol driver can pass to the **NdisSendNetBufferLists** function.

If the miniport driver for a NIC enables 802.1p packet priority(that is, the miniport driver specifies the NDIS_MAC_OPTION_8021P_PRIORITY bit in the OID_GEN_MAC_OPTIONS OID bitmask), then the miniport driver must specify its maximum total packet length as 4 bytes less than the maximum size of packets received or sent over the network. For example, if a NIC that has 802.1p packet priority enabled receives and sends packets on the wire that are 1514 bytes in length, the miniport driver for the NIC must report its maximum total packet length as 1510 bytes. The miniport driver must never indicate up to the bound protocol driver packets received over the network that are longer than the packet size specified by OID_GEN_MAXIMUM_TOTAL_SIZE. That is, even if the miniport driver receives packets over the network that are not marked with priority values but are still the maximum size that the underlying medium supports, the miniport driver can only indicate up packets that are no longer than the size specified by OID_GEN_MAXIMUM_TOTAL_SIZE.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NdisSendNetBufferLists

OID_GEN_MAC_OPTIONS

OID_GEN_MEDIA_CAPABILITIES

Article • 02/18/2023

The OID_GEN_MEDIA_CAPABILITIES OID is obsolete. NDIS and NDIS drivers do not use this OID.

Requirements

Version	Not supported.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_MEDIA_CONNECT_STATUS

Article • 02/18/2023

As a query, the OID_GEN_MEDIA_CONNECT_STATUS OID requests the connection status of the NIC on the network.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

NDIS handles this OID for NDIS 6.0 and later miniport drivers.

The OID_GEN_MEDIA_CONNECT_STATUS OID requests the connection status of the NIC on the network as one of the following system-defined values:

NdisMediaStateConnected

NdisMediaStateDisconnected

When a miniport driver senses that the network connection has been lost, it must also call the NdisMIndicateStatusEx or NdisMCoIndicateStatusEx function with NDIS_STATUS_MEDIA_DISCONNECT (for NDIS 5.1) or NDIS_STATUS_LINK_STATE with **MediaConnectStateDisconnected** in the MediaConnectState property (for NDIS 6.x). When the connection is restored, it must then call **NdisM(Co)IndicateStatus** with NDIS_STATUS_MEDIA_CONNECT (for NDIS 5.1) or NDIS_STATUS_LINK_STATE with **MediaConnectStateConnected** in the MediaConnectState property (for NDIS 6.x).

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NdisMCoIndicateStatusEx

NdisMIndicateStatusEx

OID_GEN_MEDIA_CONNECT_STATUS_EX

Article • 02/18/2023

As a query, the OID_GEN_MEDIA_CONNECT_STATUS_EX OID returns the connection state of an interface.

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

NDIS uses this OID to query the connection state of an NDIS network interface provider. Only NDIS interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the query succeeds, the interface provider returns NDIS_STATUS_SUCCESS, and the result of the query can be one of the values in the **NET_IF_MEDIA_CONNECT_STATE** enumeration.

Miniport drivers supply the media connect status during initialization and provide updates with status indications.

To specify the connection state in a miniport driver, set the **MediaConnectState** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure that the miniport driver passes to the **NdisMSetMiniportAttributes** function.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES

NdisMSetMiniportAttributes

NET_IF_MEDIA_CONNECT_STATE

NDIS Network Interface OIDs

OID_GEN_MEDIA_DUPLEX_STATE

Article • 02/18/2023

As a query, the OID_GEN_MEDIA_DUPLEX_STATE OID returns the duplex state of an interface.

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

NDIS uses this OID to query the duplex state of an NDIS network interface provider. Only NDIS interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the query succeeds, the interface provider returns NDIS_STATUS_SUCCESS, and the result of the query can be one of the values in the **NET_IF_MEDIA_DUPLEX_STATE** enumeration.

Miniport drivers supply the media duplex state during initialization and provide updates with status indications.

To specify the duplex state in a miniport driver, set the **MediaDuplexState** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure that the miniport driver passes to the **NdisMSetMiniportAttributes** function.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES

NET_IF_MEDIA_DUPLEX_STATE

NdisMSetMiniportAttributes

NDIS Network Interface OIDs

OID_GEN_MEDIA_IN_USE

Article • 02/18/2023

As a query, the OID_GEN_MEDIA_IN_USE OID specifies a complete list of the media types that the NIC currently uses.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Obsolete.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

NDIS 6.0 and later miniport drivers do not receive this OID request. NDIS handles this OID with a cached value that miniport drivers supply during initialization.

This OID provides the same information as the OID_GEN_MEDIA_SUPPORTED OID.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_MEDIA_SUPPORTED

OID_GEN_MEDIA_SENSE_COUNTS

Article • 02/18/2023

As a query, the OID_GEN_MEDIA_SENSE_COUNTS OID returns the number of times the miniport adapter reported a media state change.

Remarks

The OID_GEN_MEDIA_SENSE_COUNTS OID returns the number of times the miniport adapter reported a media state change.

Requirements

Version	Supported for NDIS 5.1 and later drivers in Windows Vista and later versions of Windows and later versions of Windows. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_MEDIA_SUPPORTED

Article • 02/18/2023

As a query, the OID_GEN_MEDIA_SUPPORTED OID specifies the media types that a NIC can support but not necessarily the media types that the NIC currently uses.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Obsolete.

The following media types were added for NDIS 6.0 and later drivers:

- **NdisMediumTunnel**
- **NdisMediumLoopback**
- **NdisMediumNative802_11**

The following media types were added for NDIS 6.20 and later drivers:

- **NdisMediumIP**

NDIS 5.1 miniport drivers
Mandatory. See OID_GEN_MEDIA_SUPPORTED (NDIS 5.1).

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory. See OID_GEN_MEDIA_SUPPORTED (NDIS 5.1).

Remarks

NDIS 6.0 and later miniport drivers do not receive this OID request. NDIS handles this OID with a cached value that miniport drivers supply during initialization.

These media types are listed as a proper subset of the following system-defined values:

NdisMedium802_3
Ethernet (802.3).

Note NDIS 5.*x* miniport drivers that conform to the 802.11 interface must use this media type. For more information about the 802.11 interface, see 802.11 Wireless LAN Miniport Drivers.

NdisMedium802_5
Token Ring (802.5). This media type is not supported for NDIS 6.0 and later drivers.

Note Starting with Windows 8, the operating system will not support this media type for any miniport drivers.

NdisMediumFddi
FDDI. This media type is not supported on Windows Vista and later versions of Windows.

NdisMediumWan
WAN

NdisMediumLocalTalk
LocalTalk

NdisMediumDix
DEC/Intel/Xerox (DIX) Ethernet

NdisMediumArcnetRaw
ARCNET (raw). This media type is not supported on Windows Vista and later versions of Windows.

NdisMediumArcnet878_2
ARCNET (878.2). This media type is not supported on Windows Vista and later versions of Windows.

NdisMediumAtm
ATM. This media type is not supported for NDIS 6.0 and later drivers.

NdisMediumNative802_11
Native 802.11. This media type is used by miniport drivers that conform to the Native 802.11 interface. For more information about this interface, see Native 802.11 Wireless LAN Miniport Drivers.

NdisMediumWirelessWan
Various types of **NdisWireless***Xxx* media. This media type is not available for use beginning with Windows Vista and later versions of Windows.

NdisMediumIrda
Infrared (IrDA).

NdisMediumCoWan
Connection-oriented WAN.

NdisMedium1394
IEEE 1394 (firewire) bus. This media type is not supported on Windows Vista and later versions of Windows.

NdisMediumBpc
Broadcast PC network.

NdisMediumInfiniBand
InfiniBand network.

NdisMediumTunnel
Tunnel network.

NdisMediumLoopback
NDIS loopback network.

NdisMediumIP
A generic medium that is capable of sending and receiving raw IP packets.

NDIS 5. *x* miniport drivers that support wireless LAN (WLAN) or wireless WAN (WWAN) packets appear to the operating system and to NDIS as Ethernet packets. These NDIS drivers must provide support for WWAN or WLAN networks as Ethernet networks. Such drivers declare their medium as **NdisMedium802_3** and emulate Ethernet to higher-level NDIS drivers. Such drivers must also declare in OID_GEN_PHYSICAL_MEDIUM the appropriate physical medium that they support..

For more information about NDIS 5.X WLAN miniport drivers, see 802.11 Wireless LAN Miniport Drivers.

NDIS 6.0 and later miniport drivers that support the WLAN media transfer packets that appear to the operating system and to NDIS as IEEE 802.11 packets. These NDIS drivers must provide support for WLAN networks as Native 802.11 miniport drivers. Such drivers declare their medium as **NdisMediumNative802_11**.

For more information about Native 802.11 miniport drivers, see Native 802.11 Wireless LAN Miniport Drivers.

If the underlying miniport driver returns **NULL** for this query, or if an experimental media type is used, the driver must indicate receive operations using the **NdisMIndicateReceiveNetBufferLists** function. Any protocol that is bound to such an underlying miniport driver receives all such indications, that is, the protocol driver cannot filter receive operations with OID_GEN_CURRENT_PACKET_FILTER.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NdisMIndicateReceiveNetBufferLists

OID_GEN_CURRENT_PACKET_FILTER

OID_GEN_PHYSICAL_MEDIUM

OID_GEN_MINIPORT_RESTART_ATTRIBUTES

Article • 02/18/2023

The OID_GEN_MINIPORT_RESTART_ATTRIBUTES OID identifies general attributes for the propagation of miniport adapter restart attributes in an NDIS driver stack.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

Remarks

The OID_GEN_MINIPORT_RESTART_ATTRIBUTES OID is not used to issue OID query or set requests.

If the **Oid** member in the NDIS_RESTART_ATTRIBUTES structure is OID_GEN_MINIPORT_RESTART_ATTRIBUTES, the **Data** member of the structure contains an NDIS_RESTART_GENERAL_ATTRIBUTES structure.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_RESTART_ATTRIBUTES

NDIS_RESTART_GENERAL_ATTRIBUTES

OID_GEN_MULTICAST_BYTES_RCV

Article • 02/18/2023

As a query, the OID_GEN_MULTICAST_BYTES_RCV OID specifies the number of bytes in multicast/functional packets that are received without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_MULTICAST_BYTES_XMIT

Article • 02/18/2023

As a query, the OID_GEN_MULTICAST_BYTES_XMIT OID specifies the number of bytes in multicast/functional packets that are transmitted without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_MULTICAST_FRAMES_RCV

Article • 02/18/2023

As a query, the OID_GEN_MULTICAST_FRAMES_RCV OID specifies the number of multicast/functional packets that are received without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

The count from this OID, combined with the count from OID_GEN_BROADCAST_FRAMES_RCV, is identical to the *ifInNUcastPkts* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_MULTICAST_FRAMES_XMIT

Article • 02/18/2023

As a query, the OID_GEN_MULTICAST_FRAMES_XMIT OID specifies the number of multicast/functional packets that are transmitted without errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

The count from this OID, combined with the count from OID_GEN_BROADCAST_FRAMES_XMIT, is identical to the *ifOutNUcastPkts* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_NDIS_RESERVED_1

Article • 03/14/2023

The OID_GEN_NDIS_RESERVED_1 OID is reserved for NDIS. NDIS drivers do not use this OID.

Requirements

Version	Reserved for NDIS.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_NDIS_RESERVED_2

Article • 03/14/2023

The OID_GEN_NDIS_RESERVED_2 OID is reserved for NDIS. NDIS drivers do not use this OID.

Requirements

Version	Reserved for NDIS.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_NDIS_RESERVED_5

Article • 03/14/2023

The OID_GEN_NDIS_RESERVED_5 OID is reserved for NDIS. NDIS drivers do not use this OID.

Requirements

Version	Reserved for NDIS.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_NETWORK_LAYER_ADDRESSES

Article • 02/18/2023

As a set, the OID_GEN_NETWORK_LAYER_ADDRESSES OID notifies underlying miniport driver and other layered drivers about the list of network-layer addresses that are associated with bound instances.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Optional.

NDIS 5.1 miniport drivers
Optional.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Optional.

Remarks

A bound instance is the binding between the calling transport and a driver set up by a call to NdisOpenAdapterEx. Transports use TRANSPORT_ADDRESS and TA_ADDRESS structures to notify underlying miniport drivers and other layered drivers about the list of network-layer addresses. Miniport drivers and other layered drivers use compatible NETWORK_ADDRESS_LIST and NETWORK_ADDRESS structures, defined as follows, to set the list of network-layer addresses on a bound interface.

C++

```
typedef struct _NETWORK_ADDRESS_LIST {
  LONG  AddressCount;
  USHORT  AddressType;
  NETWORK_ADDRESS  Address[1];
} NETWORK_ADDRESS_LIST, *PNETWORK_ADDRESS_LIST;
```

The members of this structure contain the following information:

AddressCount
Specifies the number of network-layer addresses listed in the array in the **Address** member.

AddressType
Specifies the protocol type that sends this OID. This member is only valid if the **AddressCount** member is set to zero. The **AddressCount** member is set to zero to notify a miniport driver or other layered driver to clear the list of network-layer addresses on a bound interface. The protocol can be one of the following values:

NDIS_PROTOCOL_ID_DEFAULT
Default protocol

NDIS_PROTOCOL_ID_TCP_IP
TCP/IP protocol

NDIS_PROTOCOL_ID_IPX
NetWare IPX protocol

NDIS_PROTOCOL_ID_NBF
NetBIOS protocol

Address
Array of network-layer addresses of type NETWORK_ADDRESS. The **AddressCount** member specifies the number of elements in this array.

C++

```cpp
typedef struct _NETWORK_ADDRESS {
  USHORT  AddressLength;
  USHORT  AddressType;
  UCHAR   Address[1];
} NETWORK_ADDRESS, *PNETWORK_ADDRESS;
```

The members of this structure contain the following information:

AddressLength
Specifies the size, in bytes, of this network-layer address. The **Address** member contains the array of bytes that specify this address.

AddressType
Specifies the protocol type that sends this OID and this network-layer address. This member is only valid if the **AddressCount** member in the NETWORK_ADDRESS_LIST

structure is set to a nonzero value. The **AddressCount** member in NETWORK_ADDRESS_LIST is set to a nonzero value to notify a miniport driver or other layered driver to change the list of network-layer addresses on a bound interface. Protocol types are defined in the preceding list.

Address
Array of bytes that specify this network-layer address. The **AddressLength** member specifies the number of bytes in this array.

The transport can call the NdisOidRequest function and can pass an NDIS_OID_REQUEST structure that is filled with the OID_GEN_NETWORK_LAYER_ADDRESSES code. This call notifies a bound instance of a change in the addresses that are associated with that instance. In this call, the transport also passes the bound instance in the *NdisBindingHandle* parameter. The bound instance is the binding set up between the transport and the underlying miniport driver or other layered driver. For this call, the transport should fill the **InformationBuffer** member of NDIS_OID_REQUEST with a pointer to a TRANSPORT_ADDRESS structure. TRANSPORT_ADDRESS corresponds to a NETWORK_ADDRESS_LIST structure and should contain the list of network-layer addresses.

Suppose a transport passes addresses through an intermediate driver down to an underlying miniport driver. If the intermediate driver also requires the addresses, it should take note of them before passing them on to the underlying miniport driver. An underlying miniport driver, especially an old driver, can return a status value of NDIS_STATUS_NOT_SUPPORTED or NDIS_STATUS_SUCCESS. The underlying miniport driver propagates the status of the operation back up towards the transport. If the intermediate driver must continue receiving address notifications, and if it is necessary, the intermediate driver should change the status to NDIS_STATUS_SUCCESS.Otherwise, the transport might interpret NDIS_STATUS_NOT_SUPPORTED as an indication that the underlying miniport driver does not require that the transport issue additional address updates. If NDIS_STATUS_SUCCESS is returned, transports are obligated to continue notifying underlying drivers of any change in associated addresses, including addition and deletion of addresses.

A protocol can set the **AddressCount** member of TRANSPORT_ADDRESS to zero to notify a miniport driver or other layered driver to clear the list of network-layer addresses on a bound interface. If **AddressCount** is set to zero, the **AddressType** member in NETWORK_ADDRESS_LIST is valid and the **AddressType** members in NETWORK_ADDRESS structures are not valid. On the other hand, a protocol can set **AddressCount** to a nonzero value to notify a miniport driver or other layered driver to change the list of network-layer addresses on a bound interface. In this case, the

AddressType member in NETWORK_ADDRESS_LIST is not valid and the **AddressType** members in NETWORK_ADDRESS structures are valid.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NdisOidRequest

NdisOpenAdapterEx

OID_GEN_OPERATIONAL_STATUS

Article • 02/18/2023

As a query, use the OID_GEN_OPERATIONAL_STATUS OID to determine the current operational status of a network interface (*ifOperStatus* from RFC 2863).

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

NDIS handles this OID for miniport adapters and filter modules, and only NDIS network interface providers receive this OID query.

If the query succeeds, the interface provider returns NDIS_STATUS_SUCCESS, and the result of the query can be one of the values in the **NET_IF_OPER_STATUS** enumeration.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NET_IF_OPER_STATUS

NDIS Network Interface OIDs

OID_GEN_PCI_DEVICE_CUSTOM_PROPERTIES

Article • 02/18/2023

As a query, overlying drivers use the OID_GEN_PCI_DEVICE_CUSTOM_PROPERTIES OID to get the PCI custom properties of a device.

Remarks

NDIS handles OID_GEN_PCI_DEVICE_CUSTOM_PROPERTIES and miniport drivers do not receive an OID query.

This query is optional for other NDIS drivers.

NDIS returns an NDIS_PCI_DEVICE_CUSTOM_PROPERTIES structure that contains the PCI custom properties.

For non-PCI miniport adapters, NDIS fails OID_GEN_PCI_DEVICE_CUSTOM_PROPERTIES with the NDIS_STATUS_INVALID_DEVICE_REQUEST status code.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PCI_DEVICE_CUSTOM_PROPERTIES

OID_GEN_PHYSICAL_MEDIUM

Article • 02/18/2023

As a query, the OID_GEN_PHYSICAL_MEDIUM OID specifies the types of physical media that the NIC supports. This OID is essentially an extension of OID_GEN_MEDIA_SUPPORTED.

Version Information

Note This OID is supported in NDIS 6.0 and 6.1. For NDIS 6.20 and later, use OID_GEN_PHYSICAL_MEDIUM_EX.

Remarks

NDIS handles this OID for miniport drivers. The miniport driver supplies the physical medium value during initialization.

Miniport drivers report a physical media type to differentiate their physical media from media that they declared to support in the OID_GEN_MEDIA_SUPPORTED OID query. These media types are listed as a proper subset of the following system-defined values from the **NDIS_PHYSICAL_MEDIUM** enumeration:

NdisPhysicalMediumUnspecified The physical medium is none of the preceding mediums. For example, a one-way satellite feed is an unspecified physical medium.

NdisPhysicalMediumWirelessLan Packets are transferred over a wireless LAN network through a miniport driver that conforms to the 802.11 interface. For more information about this interface, see. 802.11 Wireless LAN Miniport Drivers

NdisPhysicalMediumCableModem Packets are transferred over a DOCSIS-based cable network.

NdisPhysicalMediumPhoneLine Packets are transferred over standard phone lines. Includes, for example, HomePNA media. **NdisPhysicalMediumPowerLine** Packets are transferred over wiring that is connected to a power distribution system.

NdisPhysicalMediumDSL Packets are transferred over a Digital Subscriber Line (DSL) network. Includes, for example, ADSL and UADSL (G.Lite).
NdisPhysicalMediumFibreChannel Packets are transferred over a Fibre Channel interconnect.

NdisPhysicalMedium1394 Packets are transferred over an IEEE 1394 bus.

NdisPhysicalMediumWirelessWan Packets are transferred over a Wireless WAN link. Includes, for example, CDPD, CDMA, and GPRS.

NdisPhysicalMediumNative802_11 Packets are transferred over a wireless LAN network through a miniport driver that conforms to the Native 802.11 interface. For more information about this interface, see Native 802.11 Wireless LAN Miniport Drivers.

Note The Native 802.11 interface is supported in NDIS 6.0 and later versions.

NdisPhysicalMediumBluetooth Packets are transferred over a Bluetooth network. Bluetooth is a short-range wireless technology that uses the 2.4 GHz spectrum.

NdisPhysicalMediumInfiniband The Infiniband physical medium. Packets are transferred over an infiniband interconnect.

NdisPhysicalMediumUWB The Ultra Wideband (UWB) physical medium. Packets are transferred over a UWB network. UWB is a radio frequency platform that personal area networks can use to wirelessly communicate over short distances at high speeds.

NdisPhysicalMedium802_3 The Ethernet (802.3) physical medium. Packets are transferred over a wired LAN through a miniport driver that conforms to the 802.3 interface specification. This medium type does not include devices that emulate 802.3.

NdisPhysicalMedium802_5 The Token Ring physical medium. (802.5 is not supported in NDIS 6.0 and later drivers.) Packets are transferred over a Token Ring network through a miniport driver that conforms to the 802.5 interface specification.

NdisPhysicalMediumIrda The infrared (IrDA) physical medium. Packets are transferred over a nonvisible, infrared light spectrum IrDA network.

NdisPhysicalMediumWiredWAN The wired, wide area network (WAN) physical medium. Packets are transferred over a wired WAN.

NdisPhysicalMediumWiredCoWan The wired, connection-oriented WAN physical medium. Packets are transferred over a wired WAN in a connection-oriented environment.

NdisPhysicalMediumOther The physical medium is none of the preceding mediums. **NdisPhysicalMediumOther** specifies a new physical medium type that is not present in the NDIS_PHYSICAL_MEDIUM enumeration.

NDIS supports the OID_GEN_PHYSICAL_MEDIUM OID for miniport adapters that support newer networks, even though those networks transfer packets that appear to the operating system and to NDIS as standard and well known media types.

Newer networks transfer packets that might appear like standard media but that might have new features or slight differences from the standard. This OID was developed so upper-layer drivers and applications could determine the actual networks to which a NIC connects. After retrieving information about underlying networks, upper-layer drivers and applications could use this information to modify how such drivers and applications behave.

To clearly distinguish an 802.3 NIC from an emulated 802.3 NIC for which there is no physical medium type defined, NDIS 6.0 and later versions require 802.3 miniport drivers to report **NdisPhysicalMedium802_3**.

Requirements

Version	Supported in NDIS 6.0 and 6.1. For NDIS 6.20 and later, use OID_GEN_PHYSICAL_MEDIUM_EX instead.
Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_MEDIA_SUPPORTED

OID_GEN_PHYSICAL_MEDIUM_EX

OID_GEN_PHYSICAL_MEDIUM_EX

Article • 02/18/2023

As a query, the OID_GEN_PHYSICAL_MEDIUM_EX OID specifies the types of physical media that a miniport adapter supports.

Remarks

NDIS handles this OID for NDIS 6.0 and later miniport drivers. The miniport driver supplies the physical medium value during initialization.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains an NDIS_PHYSICAL_MEDIUM enumeration value.

Note The difference between OID_GEN_PHYSICAL_MEDIUM_EX and OID_GEN_PHYSICAL_MEDIUM is that the OID_GEN_PHYSICAL_MEDIUM_EX version does not override the **NdisPhysicalMedium802_3** type as **NdisPhysicalMediumUnspecified** whereas OID_GEN_PHYSICAL_MEDIUM still does. We recommend that all 6.x drivers use the EX version. OID_GEN_PHYSICAL_MEDIUM_EX is exposed through a WMI GUID.

Miniport drivers report a physical media type to differentiate their physical media from media that they declared to support in the OID_GEN_MEDIA_SUPPORTED OID query.

NDIS supports the OID_GEN_PHYSICAL_MEDIUM_EX OID for miniport adapters that support newer networks, even though those networks transfer packets that appear to the operating system and to NDIS as standard, well-known media types.

Newer networks transfer packets that might appear like standard media, but that might have new features or slight differences from the standard. This OID exists so upper-layer drivers and applications can determine the actual networks to which a NIC connects. After retrieving information about underlying networks, upper-layer drivers and applications can use this information to modify how such drivers and applications behave.

To clearly distinguish an 802.3 NIC from an emulated 802.3 NIC for which there is no physical medium type defined, NDIS 6.0 and later and later versions require 802.3 miniport drivers to report an **NdisPhysicalMedium802_3** media type.

Requirements

Version	Supported in NDIS 6.20 and later. Not

	requested for miniport drivers. (See Remarks section.)
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

OID_GEN_MEDIA_SUPPORTED

OID_GEN_PHYSICAL_MEDIUM

OID_GEN_PORT_AUTHENTICATION_PARAMETERS

Article • 02/18/2023

As a set, NDIS and overlying drivers use the OID_GEN_PORT_AUTHENTICATION_PARAMETERS OID to set the current state of an NDIS port.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Optional. Mandatory for NDIS ports. (see Remarks section)

Remarks

Miniport drivers that support NDIS ports must support this OID.

If a miniport driver does not support this OID, the miniport driver should return NDIS_STATUS_NOT_SUPPORTED.

If the miniport driver supports this OID, the driver returns NDIS_STATUS_SUCCESS and provides the receive port direction, port control state, and authenticate state in an NDIS_PORT_AUTHENTICATION_PARAMETERS structure.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PORT_AUTHENTICATION_PARAMETERS

OID_GEN_PORT_STATE

Article • 02/18/2023

As a query, overlying drivers use the OID_GEN_PORT_STATE OID to get the current state of the port that is specified in the **PortNumber** member of the NDIS_OID_REQUEST structure.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

NDIS handles this OID and miniport drivers do not receive this OID query.

If the query succeeds, NDIS returns NDIS_STATUS_SUCCESS and returns the port state information in an NDIS_PORT_STATE structure.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PORT_STATE

OID_GEN_PROMISCUOUS_MODE

Article • 02/18/2023

As a query, use the OID_GEN_PROMISCUOUS_MODE OID to determine whether a network interface is promiscuous or not (*ifPromiscuousMode* from RFC 2863).

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the interface provider returns NDIS_STATUS_SUCCESS and if the interface accepts only packets that are addressed to that interface, the result value should be **FALSE**. This value should be **TRUE** if the interface accepts all network packets.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS Network Interface OIDs

OID_GEN_PROTOCOL_OPTIONS

Article • 02/18/2023

As a set, the OID_GEN_PROTOCOL_OPTIONS OID specifies a bitmask that defines optional properties of the protocol driver.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. This OID is for protocol drivers.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

A protocol informs NDIS of its properties, which can optionally take advantage of them. If the protocol driver does not set its flags on a binding, NDIS assumes they are all clear.

The following flags are currently defined:

NDIS_PROT_OPTION_ESTIMATED_LENGTH
Specifies that packets can be indicated at the worst-case estimate of packet size, instead of an exact value, to this protocol.

NDIS_PROT_OPTION_NO_LOOPBACK
Specifies that the protocol does not require loopback support on the binding.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_RCV_CRC_ERROR

Article • 02/18/2023

As a query, the OID_GEN_RCV_CRC_ERROR OID specifies the number of frames that are received with checksum errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

The value for the OID_GEN_RCV_DISCARDS OID includes CRC errors. For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_RCV_DISCARDS

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_RCV_DISCARDS OID to determine the number of receive discards on a miniport adapter.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

NDIS handles this OID for miniport drivers. See the OID_GEN_STATISTICS OID for more information about statistics.

The count is identical to the *ifInDiscards* counter described in RFC 2863.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_RCV_ERROR

Article • 02/18/2023

As a query, the OID_GEN_RCV_ERROR OID specifies the number of frames that a NIC receives but does not indicate to the protocols due to errors.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 drivers
Mandatory.

Remarks

The count is identical to the *ifInErrors* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_RCV_LINK_SPEED

Article • 02/18/2023

As a query, use the OID_GEN_RCV_LINK_SPEED OID to determine the receive link speed of a network interface.

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the interface provider returns NDIS_STATUS_SUCCESS, the result of the query is a ULONG64 value that indicates the receive link speed of the interface, in bits per second.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS Network Interface OIDs

OID_GEN_RCV_NO_BUFFER

Article • 02/18/2023

As a query, the OID_GEN_RCV_NO_BUFFER OID specifies the number of frames that the NIC cannot receive due to lack of NIC receive buffer space. Some NICs do not provide the exact number of missed frames; they provide only the number of times at least one frame is missed.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 drivers
Mandatory.

Remarks

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_RCV_OK

Article • 02/18/2023

As a query, the OID_GEN_RCV_OK OID specifies the number of frames that the NIC receives without errors and indicates to bound protocols.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later drivers
Mandatory.

NDIS 5.1 drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 drivers
Mandatory.

Remarks

OID_GEN_RCV_OK specifies the number of frames that are received without errors. However, the OID_GEN_STATISTICS does not include this information.

NOTE: Statistics OIDs are mandatory for NDIS 6.0 and later miniport drivers unless NDIS handles them. For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_RECEIVE_BLOCK_SIZE

Article • 02/18/2023

As a query. the OID_GEN_RECEIVE_BLOCK_SIZE OID specifies the amount of storage, in bytes, that a single packet occupies in the receive buffer space of the NIC.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

The OID_GEN_RECEIVE_BLOCK_SIZE OID specifies the amount of storage, in bytes, that a single packet occupies in the receive buffer space of a NIC.

The same information can be obtained from the current and maximum *lookahead* size. However, one of these OIDs can be mandatory to verify each other. Also protocol drivers can determine if the underlying driver indicates full-packet receives by comparing the values that driver returns for the OID_GEN_CURRENT_LOOKAHEAD and OID_GEN_RECEIVE_BLOCK_SIZE OIDs.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_CURRENT_LOOKAHEAD

OID_GEN_RECEIVE_BUFFER_SPACE

Article • 02/18/2023

As a query, the OID_GEN_RECEIVE_BUFFER_SPACE OID specifies the amount of memory on the NIC that is available for buffering receive data.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

A protocol driver can use this OID as a guide for advertising its receive window after it establishes sessions with remote nodes.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_RECEIVE_HASH

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_RECEIVE_HASH OID to obtain the current receive hash calculation settings of a miniport adapter. NDIS returns an NDIS_RECEIVE_HASH_PARAMETERS structure that contains the current receive hash settings.

As a set, NDIS and overlying drivers use the OID_GEN_RECEIVE_HASH OID to configure the receive hash calculations on a miniport adapter. The miniport driver receives an NDIS_RECEIVE_HASH_PARAMETERS structure.

Remarks

For NDIS miniport drivers, the query is not requested.

Support for this OID set is optional for miniport drivers, including those that support RSS.

An overlying driver can use the OID_GEN_RECEIVE_HASH OID to enable and configure hash calculations on received frames without enabling RSS.

Note Protocol drivers must disable receive hash calculations before they enable RSS. If RSS is enabled, a protocol driver disables RSS before it enables receive hash calculations. A miniport driver should fail a set request with **NDIS_STATUS_INVALID_OID** or **NDIS_STATUS_NOT_SUPPORTED** to enable receive hash calculations if OID_GEN_RECEIVE_SCALE_PARAMETERS is currently enabled.

Note The secret key is appended after the NDIS_RECEIVE_HASH_PARAMETERS structure members.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_RECEIVE_HASH_PARAMETERS

OID_GEN_RECEIVE_SCALE_CAPABILITIES

Article • 02/18/2023

As a query, overlying drivers can use the OID_GEN_RECEIVE_SCALE_CAPABILITIES OID to query the receive side scaling (RSS) capabilities of a NIC and its miniport driver.

Remarks

NDIS miniport drivers do not receive this OID request. NDIS handles the query for miniport drivers.

The miniport driver returns the RSS capabilities in an NDIS_RECEIVE_SCALE_CAPABILITIES structure.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_RECEIVE_SCALE_CAPABILITIES

OID_GEN_RECEIVE_SCALE_PARAMETERS

Article • 02/18/2023

As a query, NDIS and overlying drivers can use the OID_GEN_RECEIVE_SCALE_PARAMETERS OID to query the current receive side scaling (RSS) parameters of a NIC. NDIS returns an NDIS_RECEIVE_SCALE_PARAMETERS structure that defines the current RSS parameters.

As a set, NDIS and overlying drivers use the OID_GEN_RECEIVE_SCALE_PARAMETERS OID to set the current RSS parameters of a NIC. The miniport driver receives an NDIS_RECEIVE_SCALE_PARAMETERS structure that defines the RSS parameters.

> ⓘ **Note**
>
> In RSSv2, this OID is only used to query current RSS parameters of a given scaling entity. For miniport drivers that support RSSv2, see OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 for setting RSS parameters other than the indirection table.

Remarks

For NDIS miniport drivers, the query is not requested and the set is required for drivers that support RSS. NDIS handles the query for miniport drivers.

The TCP/IP driver configures IPv4 and IPv6 with a single OID set request of OID_GEN_RECEIVE_SCALE_PARAMETERS. That is, when the stack should enable RSS for both IPv4 and IPv6, it sets both of the corresponding flags in the **HashInformation** member of the NDIS_RECEIVE_SCALE_PARAMETERS structure and sends one OID request. Also, IPv4 and IPv6 use the same secret key and the key will always be 40 bytes, even if only IPv4 is enabled.

The underlying miniport adapter must use the most recent OID_GEN_RECEIVE_SCALE_PARAMETERS OID settings it has received. For example, if the miniport gets an OID_GEN_RECEIVE_SCALE_PARAMETERS OID with the IPv4 hash types missing, it must disable IPv4 RSS if it was previously enabled.

Note An overlying driver can use the OID_GEN_RECEIVE_HASH OID to enable and configure hash calculations on received frames without enabling RSS.

Note Protocol drivers must disable receive hash calculations (OID_GEN_RECEIVE_HASH) before they enable RSS. If RSS is enabled, a protocol driver disables RSS before it enables receive hash calculations. A miniport driver should fail a set request with **NDIS_STATUS_INVALID_OID** or **NDIS_STATUS_NOT_SUPPORTED** to enable RSS if OID_GEN_RECEIVE_HASH is currently enabled.

Note The indirection table and secret key are appended after the NDIS_RECEIVE_SCALE_PARAMETERS structure members. For more information about the indirection table and secret key, see **NDIS_RECEIVE_SCALE_PARAMETERS**.

Requirements

Version	Supported in NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_RECEIVE_SCALE_PARAMETERS

OID_GEN_RECEIVE_HASH

OID_GEN_RECEIVE_SCALE_PARAMETERS_V2

Article • 02/18/2023

⚠ **Warning**

Some information in this topic relates to prereleased product, which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

RSSv2 is preview only in Windows 10, version 1809.

The OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 OID is sent to RSSv2-capable miniport drivers to set run-time parameters, other than the indirection table, for a scaling entity. OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 replaces the OID_GEN_RECEIVE_SCALE_PARAMETERS OID from RSSv1 and is not visible to NDIS Light Weight Filters (LWFs) before NDIS 6.80. This OID is a Regular OID and can be issued as a Query or Set request. It is issued at IRQL == PASSIVE_LEVEL. It can target a given VPort, when the *NDIS_OID_REQUEST_FLAGS_VPORT_ID_VALID* flag is set at NIC switch creation. Otherwise, it targets the physical NIC in the Native RSS case.

As a Query, NDIS and overlying drivers can use OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 to query the RSS parameters of a NIC. NDIS returns an NDIS_RECEIVE_SCALE_PARAMETERS_V2 structure that defines the current RSS parameters.

As a Set, the purpose of this OID is to perform the following actions:

- Initially configure the scaling entity (a miniport adapter in Native RSS mode or a VPort in VMQ mode).
- Enable or disable RSS.
- When in RSS mode, perform non-timing-critical management functions such as changing the hash key, hash type and hash function, number of queues, or number of indirection table entries for the scaling entity.

Remarks

Enabling RSS and setting RSS parameters can be performed in one step.. After the upper layer enables RSS using this OID, the initial state of the scaling entity is as follows:

- The primary processor becomes *inactive*.
- The default processor becomes *active*.
- All the ITEs become *active*.
- The miniport driver starts calculation of the RSS hash, setting of the corresponding OOB for all packets, and directing packets to a processor specified by the indirection table entry or default processor parameter.

After RSS is enabled, the upper layer issues the OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OID to move ITEs to different processors. In RSSv2, the **DefaultQueue** and **PrimaryProcessor** are also moved to a different processor using OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES.

During the process of disabling RSS, the upper layer will point all ITEs to the primary processor before invoking this OID to turn RSS off. After this point, receive traffic should target the primary processor. However, miniport drivers should not expect the disabling of RSS before VPort deletion. The upper layer can set the receive filter on the VPort to zero, thus ensuring that no receive traffic is flowing through the VPort, then proceed to delete the VPort without disabling RSS.

The upper layer will ensure that important invariants are not violated before performing management functions. For example:

- Before changing the number of queues, the upper layer will ensure that the indirection table does not reference more processors than configured for a VPort. Before changing the number of indirection table entries for VMMQ-RESTRICTED adapters, the upper layer will ensure that the content of the indirection table is normalized to the power of 2.

Error conditions and status codes

This OID returns the following status codes when an error occurs:

Status code	Error condition
NDIS_STATUS_INVALID_LENGTH	The OID was malformed.
NDIS_STATUS_NO_QUEUES	The number of queues is being changed when RSS is enabled, but the current indirection table references more processors than the new number of queues.

Status code	Error condition
NDIS_STATUS_INVALID_DATA	- The indirection table is being reduced in size, but does not contain a power-of-two repeat pattern. - During an RSS state transition (to *on* or *off*), a processor from a steering parameter that becomes *active* does not belong to the adapter's RSS processor set. Note that *inactive* steering parameters are only tracking writes to the processor and are not enforced. Enforcement happens during RSS state transition when the parameter becomes *active*.
NDIS_STATUS_INVALID_PARAMETER	Other fields, either in the header or the OID itself, contain invalid values.

Requirements

Version: Windows 10, version 1709 **Header**: Ntddndis.h (include Ndis.h)

See also

- Receive Side Scaling Version 2 (RSSv2)
- OID_GEN_RECEIVE_SCALE_PARAMETERS
- NDIS_RECEIVE_SCALE_PARAMETERS_V2
- OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES

OID_GEN_RESET_COUNTS

Article • 02/18/2023

As a query, the OID_GEN_RESET_COUNTS OID returns the number of times the miniport adapter was reset.

Remarks

The OID_GEN_RESET_COUNTS OID returns the number of times the miniport adapter was reset.

Requirements

Version	Supported for NDIS 5.1 and later drivers in Windows Vista and later versions of Windows and later versions of Windows. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

OID_GEN_RNDIS_CONFIG_PARAMETER

Article • 03/14/2023

As a set, the OID_GEN_RNDIS_CONFIG_PARAMETER is used to set device-specific parameters. The host uses it with RNDIS devices only.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For RNDIS devices only.

NDIS 5.1 miniport drivers
Optional.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Optional.

Remarks

The OID_GEN_RNDIS_CONFIG_PARAMETER is used with RNDIS devices. The host uses it to set device-specific parameters. It is not used by miniport drivers. For more information about this OID, see Setting Device-Specific Parameters.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES

Article • 02/18/2023

⚠ **Warning**

Some information in this topic relates to prereleased product, which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

RSSv2 is preview only in Windows 10, version 1809.

The OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES OID is sent to RSSv2-capable miniport drivers to perform moves of individual indirection table entries. This OID is a Synchronous OID, meaning it cannot return NDIS_STATUS_PENDING. It is issued as a Method request only, at IRQL == DISPATCH_LEVEL.

This call uses the *XxxSynchronousOidRequest* entry point, where *Xxx* is either *Miniport* or *Filter* depending on the type of driver receiving the request. This entry point causes a system bug check if it sees an NDIS_STATUS_PENDING return status.

OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES uses the NDIS_RSS_SET_INDIRECTION_ENTRIES structure to instruct a miniport adapter to synchronously perform a set of actions, where each action moves a single entry of the RSS indirection table of a specified VPort to a target specified CPU.

Remarks

This OID must execute and complete in the processor context that issued it. Miniport drivers must fully execute this OID upon returning NDIS_STATUS_SUCCESS to the upper layer. This means that the miniport driver should be prepared to receive back-to-back OID requests to move multiple ITEs on a new processor immediately after the first move finishes with NDIS_STATUS_SUCCESS.

💡 **Tip**

Fully executing this OID means that the miniport driver must be ready to successfully attempt another action to move an ITE. It does not prescribe where in-flight receive traffic is indicated right after the queue move, which can either be on the source CPU or the target CPU.

Upper layer protocols issue OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES to set ITEs and/or the primary and default processor parameters to point to different processors.

This OID can be issued for either *active* or *inactive* traffic steering parameters. For more information about steering parameters, see Receive side scaling version 2 (RSSv2). For parameters/ITEs in the *inactive* state, the miniport driver should validate and cache the target processor until the next relevant RSS state change (enablement or disablement). At that point, cached processor numbers become *active* and are used for directing the traffic. Updates to *active* parameters (which must also be validated) should be taken immediately into effect to direct the traffic.

OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES must be issued to a miniport adapter with the *NDIS_OID_REQUEST_FLAGS_VPORT_ID_VALID* flag cleared. This is because of the possibility of different VPorts being referenced by different elements in the array.

This OID is invoked only at IRQL == DISPATCH_LEVEL.

Miniport drivers should be prepared to handle at least as many indirection table entry move actions as they advertise in the NDIS_NIC_SWITCH_CAPABILITIES structure. This is defined in the **NumberOfIndirectionTableEntriesPerNonDefaultVPort** or **NumberOfIndirectionTableEntriesForDefaultVPort** member of that structure, or **128** in Native RSS mode.

Miniport drivers should attempt to execute as many entries as they can and update the **EntryStatus** member of each NDIS_RSS_SET_INDIRECTION_ENTRY with the result of the operation.

OID handler for OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES

The OID handler for OID_GEN_RSS_SET_INDIRECTION_TABLE_ENTRIES is expected to behave as follows:

- A return of NDIS_STATUS_PENDING is not permitted due to the OID's Synchronous call type.
- Finalize any incoming ITE moves that were destined for the current CPU (previously initiated on remote processors).
- It is strongly recommended for miniport drivers to perform a full parameter validation pass. If not possible, perform one-by-one validation and execution of

array entries. Miniport drivers should specifically check if all the referenced objects are valid:

  - Returning NDIS_STATUS_PENDING in the **EntryStatus** field for an ITE is not permitted.
  - The miniport adapter exists and is in a good state. Else, set the **EntryStatus** field of the entry to NDIS_STATUS_ADAPTER_NOT_FOUND, NDIS_STATUS_ADAPTER_NOT_READY, etc.
  - Each VPort exists and is in a good state. Else, set the **EntryStatus** field of the entry to NDIS_STATUS_INVALID_PORT, NDIS_STATUS_INVALID_PORT_STATE, etc.
  - Each indirection table entry index is within the configured range. This range is either 0xFFFF or is in the [0...NumberOfIndirectionTableEntries - 1] range set by the OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 OID. The 0xFFFF and 0xFFFE entry indices have special meanings: 0xFFFF defines the default processor, while 0xFFFE defines the primary processor. On error, the handler sets the **EntryStatus** field of the entry to NDIS_STATUS_INVALID_PARAMETER.
  - The upper layer and the miniport driver expect that the ITE points to the current processor (actor CPU) before the move. In other words, the ITE cannot be redirected remotely. If this is not true, set the **EntryStatus** field of the entry to NDIS_STATUS_NOT_ACCEPTED.
  - All target processors are valid and are part of the miniport adapter's RSS set. Else, set the **EntryStatus** field of the entry to NDIS_STATUS_INVALID_DATA.

- Either subsequently or as part of the parameter validation pass, validate the resource situation. Validate that the number of queues to be used after a full batch move (evacuation) does not exceed the **NumberOfQueues** set in the NDIS_RECEIVE_SCALE_PARAMETERS_V2 structure during an OID_GEN_RECEIVE_SCALE_PARAMETERS_V2 request. Otherwise, NDIS_STATUS_NO_QUEUES is returned. NDIS_STATUS_NO_QUEUES should be used for all conditions that represent a violation of the configured number of queues. NDIS_STATUS_RESOURCES should only be used to designate transient out-of-memory conditions.
- As part of resource checks, for each scaling entity (for example, VPort), the miniport driver must handle a condition when all ITEs that point to the currrent CPU are moved away from it..

If all of the above checks pass, the miniport driver should be able to unconditionally apply the new configuration and must set the **EntryStatus** field of each entry to NDIS_STATUS_SUCCESS.

In general, the handler for this OID should be very light weight. It should not call NDIS or operating system services other than for possible synchronization operations like spinlocks and **NdisMConfigMSIXTableEntry**.

The miniport driver should not call NDIS to indicate status or PnP events.

The miniport driver should also not use receive/transmit complete indications in the context of this OID handler, as doing so leads to recursion. The upper layer can invoke this OID from the context of receive or transmit indications.

Moving all indirection table entries

Miniport drivers should recognize and handle a special request that moves all indirection table entries away from the current CPU. Because RSSv2 operates with individual ITE moves, miniport drivers must guarantee the atomicity of the overall operation. If it encounters an error in the middle of a batch while processing the corresponding array of move commands, the miniport driver should revert all commands that were already performed and mark all commands as "failed" in the per-command **EntryStatus** field. The upper layer protocol always expects the "move all ITEs" batch to contain either all commands marked as "succeeded," or all commands marked as "failed," and it will assume that traffic obeys the resulting state (either before or after the move). If the upper layer sees only some entries marked as "failed," it will bug check the system and point to the miniport driver as the cause.

To aid the miniport driver's handling of the "move all ITEs" command, and to avoid deadlocks, upper layer protocols group move commands in the batch in pairs of **SwitchId** + **VPortId** fields, such that:

- Commands that the upper layer wants to be executed together, as part of the "move all" command, for the same VPort are placed consecutively in the overall batch.
- The miniport driver should not attempt to execute the overall command batch, which may target different VPorts, in a "move all" fashion. Only the group of commands that target the same VPort (tagged with the same **SwitchId** + **VPortId** pair) need to be executed conforming to the "move all" semantics.
- When the upper layer does not care about "move all" semantics, it might interleave commands to the same VPort with commands to different VPort(s). In this case, if the second group of commands to the same VPort can't be executed because of a "number of queues" violation, the miniport driver marks that group with the corresponding status code (NDIS_STATUS_NO_QUEUES) and the upper layer takes responsibility for recovering.

For example, if the upper layer protocol interleaves a series of commands like this:

- `VPort=1 ITE[0,1]`
- `VPort=2 ITE[0]`

- `VPort=1 ITE[2]`

The miniport driver does not need to attempt to atomically execute all four move commands, or all three move commands for `VPort=1` (`ITE[0,1,2]`). It only needs to execute the `VPort=1 ITE[0,1]` group in a "move all" fashion, then the `VPort=2 ITE[0]` group, then `VPort=1 ITE[2]`. All three command groups might have a different outcome. For example, the groups for `VPort=1 ITE[0,1]` and `VPort=2 ITE[0]` might succeed, and the `VPort=1 ITE[2]` group might fail. The outcome should be reflected in the corresponding **EntryStatus** member of each command structure. This way, the miniport driver does not need to take precautions for safe execution of the overall batch (for example, lock the whole adapter). Only those commands that target a specific VPort need to be serialized, finer-grained per-VPort locking can be used, and certain deadlocks are avoided.

> ⓘ Note
>
> The entire group of the command entries must be marked with the same entry status.

Error conditions and status codes

This OID returns the following status codes when an error occurs:

Status code	Error condition
NDIS_STATUS_INVALID_LENGTH	The OID was malformed.
NDIS_STATUS_INVALID_PARAMETER	Other fields, either in the header or in the OID itself (but not in individual command entries) contain invalid values.

Requirements

Version: Windows 10, version 1709 **Header**: Ntddndis.h (include Ndis.h)

See also

- Receive Side Scaling Version 2 (RSSv2)
- NDIS_RSS_SET_INDIRECTION_ENTRIES
- NDIS_RSS_SET_INDIRECTION_ENTRY
- NDIS_NIC_SWITCH_CAPABILITIES

- OID_GEN_RECEIVE_SCALE_PARAMETERS_V2
- NDIS_RECEIVE_SCALE_PARAMETERS_V2

OID_GEN_STATISTICS

Article • 02/06/2024

As a query, NDIS and overlying drivers use the OID_GEN_STATISTICS OID to obtain statistics of an adapter or a miniport driver.

Note: General statistics OIDs count all traffic through the network adapter including Network Direct Kernel (NDK) traffic. NDK statistics may be counted separately with OID_NDK_STATISTICS.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory.

The NDIS_STATISTICS_INFO structure is defined as follows:

ManagedCPlusPlus

```
    typedef struct _NDIS_STATISTICS_INFO {
         NDIS_OBJECT_HEADER Header;
         ULONG SupportedStatistics;
         ULONG64 ifInDiscards;
         ULONG64 ifInErrors;
         ULONG64 ifHCInOctets;
         ULONG64 ifHCInUcastPkts;
         ULONG64 ifHCInMulticastPkts;
         ULONG64 ifHCInBroadcastPkts;
         ULONG64 ifHCOutOctets;
         ULONG64 ifHCOutUcastPkts;
         ULONG64 ifHCOutMulticastPkts;
         ULONG64 ifHCOutBroadcastPkts;
         ULONG64 ifOutErrors;
         ULONG64 ifOutDiscards;
         ULONG64 ifHCInUcastOctets;
         ULONG64 ifHCInMulticastOctets;
         ULONG64 ifHCInBroadcastOctets;
         ULONG64 ifHCOutUcastOctets;
         ULONG64 ifHCOutMulticastOctets;
         ULONG64 ifHCOutBroadcastOctets;
    } NDIS_STATISTICS_INFO, *PNDIS_STATISTICS_INFO;
```

This structure contains the following members:

Header
The NDIS_OBJECT_HEADER structure for the NDIS_STATISTICS_INFO structure. Set the **Type** member of the structure that **Header** specifies to NDIS_OBJECT_TYPE_DEFAULT, the **Revision** member to NDIS_STATISTICS_INFO_REVISION_1, and the **Size** member to NDIS_SIZEOF_STATISTICS_INFO_REVISION_1.

SupportedStatistics
The set of statistics that the miniport driver supports.

Note NDIS 6.0 and later drivers must support all statistics and must report them when queried for OID_GEN_STATISTICS.

The value is the bitwise OR of the following flags:

NDIS_STATISTICS_FLAGS_VALID_DIRECTED_FRAMES_RCV
The data in the **ifHCInUcastPkts** member is valid.

NDIS_STATISTICS_FLAGS_VALID_MULTICAST_FRAMES_RCV
The data in the **ifHCInMulticastPkts** member is valid.

NDIS_STATISTICS_FLAGS_VALID_BROADCAST_FRAMES_RCV
The data in the **ifHCInBroadcastPkts** member is valid.

NDIS_STATISTICS_FLAGS_VALID_BYTES_RCV
The data in the **ifHCInOctets** member is valid.

NDIS_STATISTICS_FLAGS_VALID_RCV_DISCARDS
The data in the **ifInDiscards** member is valid.

NDIS_STATISTICS_FLAGS_VALID_RCV_ERROR
The data in the **ifInErrors** member is valid.

NDIS_STATISTICS_FLAGS_VALID_DIRECTED_FRAMES_XMIT
The data in the **ifHCOutUcastPkts** member is valid.

NDIS_STATISTICS_FLAGS_VALID_MULTICAST_FRAMES_XMIT
The data in the **ifHCOutMulticastPkts** member is valid.

NDIS_STATISTICS_FLAGS_VALID_BROADCAST_FRAMES_XMIT
The data in the **ifHCOutBroadcastPkts** member is valid.

NDIS_STATISTICS_FLAGS_VALID_BYTES_XMIT
The data in the **ifHCOutOctets** member is valid.

NDIS_STATISTICS_FLAGS_VALID_XMIT_ERROR
The data in the **ifOutErrors** member is valid.

NDIS_STATISTICS_FLAGS_VALID_XMIT_DISCARDS
The data in the **ifOutDiscards** member is valid.

NDIS_STATISTICS_FLAGS_VALID_DIRECTED_BYTES_RCV
The data in the **ifHCInUcastOctets** member is valid.

NDIS_STATISTICS_FLAGS_VALID_MULTICAST_BYTES_RCV
The data in the **ifHCInMulticastOctets** member is valid.

NDIS_STATISTICS_FLAGS_VALID_BROADCAST_BYTES_RCV
The data in the **ifHCInBroadcastOctets** member is valid.

NDIS_STATISTICS_FLAGS_VALID_DIRECTED_BYTES_XMIT
The data in the **ifHCOutUcastOctets** member is valid.

NDIS_STATISTICS_FLAGS_VALID_MULTICAST_BYTES_XMIT
The data in the **ifHCOutMulticastOctets** member is valid.

NDIS_STATISTICS_FLAGS_VALID_BROADCAST_BYTES_XMIT
The data in the **ifHCOutBroadcastOctets** member is valid.

ifInDiscards
The dropped-receive-buffer error count. This is the same value that OID_GEN_RCV_DISCARDS returns.

ifInErrors
The receive error count. This count is the same value that OID_GEN_RCV_ERROR returns.

ifHCInOctets
The sum of the receive-directed byte count, receive-multicast byte count, and receive-broadcast byte count. This sum is the same value that OID_GEN_BYTES_RCV returns.

ifHCInUcastPkts
The number of directed packets that are received without errors. This number is the same value that OID_GEN_DIRECTED_FRAMES_RCV returns.

ifHCInMulticastPkts
The number of multicast/functional packets that are received without errors. This number is the same value that OID_GEN_MULTICAST_FRAMES_RCV returns.

ifHCInBroadcastPkts
The number of broadcast packets that are received without errors. This number is the same value that OID_GEN_BROADCAST_FRAMES_RCV returns.

ifHCOutOctets
The sum of the transmit-directed byte count, transmit-multicast byte count and transmit-broadcast byte count. This sum is the same value that OID_GEN_BYTES_XMIT returns.

ifHCOutUcastPkts
The number of directed packets that are transmitted without errors. This number is the same value that OID_GEN_DIRECTED_FRAMES_XMIT returns.

ifHCOutMulticastPkts
The number of multicast/functional packets that are transmitted without errors. This number is the same value that OID_GEN_MULTICAST_FRAMES_XMIT returns.

ifHCOutBroadcastPkts
The number of broadcast packets that are transmitted without errors. This number is the same value that OID_GEN_BROADCAST_FRAMES_XMIT returns.

ifOutErrors
The transmit error count. This count is the same value that OID_GEN_XMIT_ERROR returns.

ifOutDiscards
The number of packets that is discarded by the interface. This is same as the value that is returned by querying the OID_GEN_XMIT_DISCARDS OID.

ifHCInUcastOctets
The number of bytes in directed packets that are received without errors. This count is the same value that OID_GEN_DIRECTED_BYTES_RCV returns.

ifHCInMulticastOctets
The number of bytes in multicast/functional packets that are received without errors. This count is the same value that OID_GEN_MULTICAST_BYTES_RCV returns.

ifHCInBroadcastOctets
The number of bytes in broadcast packets that are received without errors. This count is the same value that OID_GEN_BROADCAST_BYTES_RCV returns.

ifHCOutUcastOctets
The number of bytes in directed packets that are transmitted without errors. This count is the same value that OID_GEN_DIRECTED_BYTES_XMIT returns.

ifHCOutMulticastOctets
The number of bytes in multicast/functional packets that are transmitted without errors. This count is the same value that OID_GEN_MULTICAST_BYTES_XMIT returns.

ifHCOutBroadcastOctets
The number of bytes in broadcast packets that are transmitted without errors. This count is the same value that OID_GEN_BROADCAST_BYTES_XMIT returns.

Remarks

Miniport drivers must implement the statistics counters and report the correct statistics values. The statistics counters are unsigned 64-bit values. The miniport driver returns the statistics in an NDIS_STATISTICS_INFO structure.

Requirements

Expand table

Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OBJECT_HEADER

OID_GEN_BROADCAST_BYTES_RCV

OID_GEN_BROADCAST_BYTES_XMIT

OID_GEN_BROADCAST_FRAMES_RCV

OID_GEN_BROADCAST_FRAMES_XMIT

OID_GEN_BYTES_RCV

OID_GEN_BYTES_XMIT

OID_GEN_DIRECTED_BYTES_RCV

OID_GEN_DIRECTED_BYTES_XMIT

OID_GEN_DIRECTED_FRAMES_RCV

OID_GEN_DIRECTED_FRAMES_XMIT

OID_GEN_MULTICAST_FRAMES_RCV

OID_GEN_MULTICAST_FRAMES_XMIT

OID_GEN_MULTICAST_BYTES_RCV

OID_GEN_MULTICAST_BYTES_XMIT

OID_GEN_RCV_DISCARDS

OID_GEN_RCV_ERROR

OID_GEN_XMIT_DISCARDS

OID_GEN_XMIT_ERROR

OID_GEN_SUPPORTED_GUIDS

Article • 02/18/2023

As a query, the OID_GEN_SUPPORTED_GUIDS OID requests the miniport driver to return an array of structures of the type NDIS_GUID.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Optional.

NDIS 5.1 miniport drivers
Optional.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Optional.

Remarks

Each structure in the array specifies the mapping of a custom GUID (globally unique identifier) to either a custom OID or to an NDIS_STATUS that the miniport driver sends through the NdisMIndicateStatusEx function.

The NDIS_GUID structure is defined as follows:

C++

```
typedef struct _NDIS_GUID {
    GUID                Guid;
    union {
        NDIS_OID        Oid;
        NDIS_STATUS     Status;
    };
    ULONG               Size;
    ULONG               Flags;
} NDIS_GUID, *PNDIS_GUID;
```

The members of this structure contain the following information:

Guid
Specifies the custom GUID defined for the miniport driver.

Oid
Specifies the custom OID to which **Guid** maps.

Status
Specifies the NDIS_STATUS to which **Guid** maps.

Size
Specifies the size in bytes of each data item in the array returned by the miniport driver. If the fNDIS_GUID_ANSI_STRING or fNDIS_GUID_NDIS_STRING flag is set, **Size** is set to -1. Otherwise, **Size** specifies the size in bytes of the data item that the GUID represents. This member is specified only when the fNDIS_GUID_ARRAY flag is set.

Flags
The following flags can be combined by the OR operator to indicate whether the GUID maps to an OID or to an NDIS_STATUS string and to indicate the type of data that is supplied for the GUID:

fNDIS_GUID_TO_OID
Indicates that the NDIS_GUID structure maps a GUID to an OID.

fNDIS_GUID_TO_STATUS
Indicates that the NDIS_GUID structure maps a GUID to an NDIS_STATUS string.

fNDIS_GUID_ANSI_STRING
Indicates that a null-terminated ANSI string is supplied for the GUID.

fNDIS_GUID_UNICODE_STRING
Indicates that a Unicode string is supplied for the GUID.

fNDIS_GUID_ARRAY
Indicates that an array of data items is supplied for the GUID. The specified **Size** indicates the length of each data item in the array.

fNDIS_GUID_ALLOW_READ
When set, indicates that all users are allowed to use this GUID to obtain information.

fNDIS_GUID_ALLOW_WRITE
When set, indicates that all users are allowed to use this GUID to set information.

Note By default, custom WMI GUIDs supplied by a miniport driver are only accessible to users with administrator privileges. A user with administrator privileges can always read or write to a custom GUID if the miniport driver supports the read or write operation for that GUID. Set the fNDIS_GUID_ALLOW_READ and fNDIS_GUID_ALLOW_WRITE flags to allow all users to access a custom GUID.

Note that all custom GUIDs registered by a miniport driver must set either fNDIS_GUID_TO_OID or fNDIS_GUID_TO_STATUS (never set both). All other flags may be combined by using the OR operator as applicable.

In the following example, an NDIS_GUID structure maps a GUID to OID_802_3_MULTICAST_LIST:

C++

```
NDIS_GUID    NdisGuid = {{0x44795701, 0xa61b, 0x11d0, 0x8d, 0xd4,
                          0x00, 0xc0, 0x4f, 0xc3,
                          0x35, 0x8c},
                          OID_802_3_MULTICAST_LIST,
                          6,
                          fNDIS_GUID_TO_OID | fNDIS_GUID_ARRAY};
```

A GUID is an identifier used by Windows Management Instrumentation (WMI) to obtain or set information. NDIS intercepts a GUID sent by WMI to an NDIS driver, it maps the GUID to an OID, and sends the OID to the driver. The driver returns the data items to NDIS, which then returns the data to WMI.

NDIS also translates changes in NIC status into GUIDs that are recognized by WMI. When a miniport driver reports a change in NIC status using the NdisMIndicateStatusEx function, NDIS translates the NDIS_STATUS indicated by the miniport driver into a GUID that NDIS sends to WMI.

If a miniport driver supports customs GUIDs, it must support OID_GEN_SUPPORTED_GUIDS. This OID returns to NDIS the mapping of custom GUIDs to custom OIDs or NDIS_STATUS strings. After querying the miniport driver using OID_GEN_SUPPORTED_GUIDS, NDIS registers the miniport driver's custom GUIDs with WMI.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NdisMIndicateStatusEx

OID_GEN_SUPPORTED_LIST

Article • 02/18/2023

As a query, the OID_GEN_SUPPORTED_LIST OID specifies an array of OIDs for objects that the miniport driver or a NIC supports. Objects include general, media-specific, and implementation-specific objects.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested.

NDIS 5.1 miniport drivers
Mandatory. See OID_GEN_SUPPORTED_LIST (NDIS 5.1).

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory. See OID_GEN_SUPPORTED_LIST (NDIS 5.1).

Remarks

NDIS 6.0 and later miniport drivers do not receive this OID request. NDIS handles this OID with a cached value that miniport drivers supply during initialization.

To specify the list of supported OIDs during initialization, a miniport driver sets the **SupportedOidList** member of the **NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES** structure and passes the structure to the **NdisMSetMiniportAttributes** function.

NDIS forwards a subset of the provided list to protocol drivers that make this query. That is, NDIS filters any supported statistics OIDs out of the list because protocol drivers never make statistics queries.

If a miniport driver lists an OID in its supported OIDs list, it must fully support the OID. That is, the miniport driver must return valid data when it responds to a query or set request for the OIDs that it includes in the list. For example, the OID_GEN_STATISTICS OID is a required OID for NDIS 6.0 and later miniport drivers. If a miniport driver does not support the statistics in hardware or software and returns incorrect statistics information, the driver cannot specify OID_GEN_STATISTICS in its supported OIDs list.

Duplicates might appear in the supported OIDs list. Drivers are not required to guarantee that there is only one entry for each OID in the list.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_TRANSMIT_BLOCK_SIZE

Article • 02/18/2023

As a query, the OID_GEN_TRANSMIT_BLOCK_SIZE OID specifies the minimum number of bytes that a single net packet occupies in the transmit buffer space of the NIC.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

The OID_GEN_TRANSMIT_BLOCK_SIZE OID specifies the minimum number of bytes that a single net packet occupies in the transmit buffer space of the NIC. For example, a NIC that has a transmit space divided into 256-byte pieces would have a transmit block size of 256 bytes. To calculate the total transmit buffer space on such a NIC, its driver multiplies the number of transmit buffers on the NIC by its transmit block size.

For other NICs, the transmit block size is identical to its maximum packet size.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_TRANSMIT_BUFFER_SPACE

Article • 02/18/2023

As a query, the OID_GEN_TRANSMIT_BUFFER_SPACE OID specifies the amount of memory, in bytes, on the NIC that is available for buffering transmit data.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

A protocol can use this OID as a guide for sizing the amount of transmit data per send.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_TRANSMIT_QUEUE_LENGTH

Article • 02/18/2023

As a query, the OID_GEN_TRANSMIT_QUEUE_LENGTH OID specifies the number of packets that are currently queued for transmission, whether on the NIC or in a driver-internal queue.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later drivers
Optional.

NDIS 5.1 drivers
Optional.

Windows XP
Supported.

NDIS 5.1 drivers
Optional.

Remarks

For queries, the number returned is always the total number of packets currently queued. This number can include unsubmitted send requests queued in the NDIS library.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_TRANSPORT_HEADER_OFFSET

Article • 02/18/2023

As a set, the OID_GEN_TRANSPORT_HEADER_OFFSET OID indicates the size of additional headers for packets that a particular transport sends and receives.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Optional.

NDIS 5.1 miniport drivers
Optional.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Optional.

Remarks

A transport informs miniport drivers and other layered drivers of this header size; these drivers can then use this information when processing packets. For example, a driver could use the sublayer header size obtained from the transport to locate the beginning of higher layer information in packets, such as the start of the IP header; the driver could then parse and adjust the fields of the IP protocol header as appropriate. Transports use a TRANSPORT_HEADER_OFFSET structure, defined as follows, to indicate this header size.

C++

```
typedef struct _TRANSPORT_HEADER_OFFSET {
  USHORT  ProtocolType;
  USHORT  HeaderOffset;
} TRANSPORT_HEADER_OFFSET, *PTRANSPORT_HEADER_OFFSET;
```

The members of this structure contain the following information:

ProtocolType
Specifies the protocol type that sends this OID and that subsequently sends and receives packets using the specified sublayer header size. The protocol is one of the following values:

NDIS_PROTOCOL_ID_DEFAULT
Default protocol

NDIS_PROTOCOL_ID_TCP_IP
TCP/IP protocol

NDIS_PROTOCOL_ID_IPX
NetWare IPX protocol

NDIS_PROTOCOL_ID_NBF
NetBIOS protocol

HeaderOffset
Specifies the size, in bytes, of the sublayer header that precedes the protocol header for packets that the protocol subsequently sends to or receives from the miniport driver or other layered driver. For example, sizeof(Ethernet header) + sizeof(SNAP header).

Typically, transports calculate the header size of packets from information that is retrieved from miniport drivers. To request the maximum total packet size in bytes that a NIC supports, including the header, transports use the OID_GEN_MAXIMUM_TOTAL_SIZE OID. To request the maximum packet size in bytes that a NIC supports, not including a header, transports use the OID_GEN_MAXIMUM_FRAME_SIZE OID. To calculate the maximum header size, transports subtract the maximum frame size from the maximum total size.

If a transport transmits packets that contain sublayer header information, the transport must know the sublayer header size of these packets and must inform underlying miniport drivers and other layered drivers about the size so that the drivers can process the packets. Sending and receiving particular sublayer header information within a packet may be an option that can be set in the registry for a particular protocol. Transports could then obtain information about sublayer headers from the registry and pass the header size down to miniport drivers or other layered drivers.

For example, if a transport handles packets from the Fiber Distributed Data Interface medium, the transport must send a set request to underlying miniport drivers and other layered drivers using OID_GEN_TRANSPORT_HEADER_OFFSET to inform those drivers about the size of the packets' sublayer header. (FDDI is not supported in Windows Vista and later versions of Windows.) These packets from FDDI could contain Logical Link

Control (LLC) information. This LLC information could in turn include an LLC header and other headers such as Sub-Network Access Protocol (SNAP). The transport determines from the registry to use LLC/SNAP and passes the header size of the LLC/SNAP segments of packets to miniport drivers.

This OID is optional for miniport drivers and other layered drivers. Because this OID is optional, drivers are not required to respond to requests that transports make using this OID.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_MAXIMUM_FRAME_SIZE

OID_GEN_MAXIMUM_TOTAL_SIZE

OID_GEN_UNKNOWN_PROTOS

Article • 02/18/2023

As a query, use the OID_GEN_UNKNOWN_PROTOS OID to determine the unknown-protocol packet count of a network interface (*ifInUnknownProtos* from RFC 2863).

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

The unknown-protocol statistics counter specifies the number of packets that were received through the interface that were discarded because the associated protocol was unknown or unsupported.

If the interface provider returns NDIS_STATUS_SUCCESS, the result of the query is a ULONG64 value that specifies the number of packets.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS Network Interface OIDs

OID_GEN_VENDOR_DESCRIPTION

Article • 02/18/2023

As a query, the OID_GEN_VENDOR_DESCRIPTION OID points to a null-terminated Unicode string describing the Network Interface Controller (NIC).

Set requests are not supported.

Remarks

This OID is mandatory for NDIS 6.0 and later miniport drivers.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_VENDOR_DRIVER_VERSION

Article • 02/18/2023

As a query, the OID_GEN_VENDOR_DRIVER_VERSION OID specifies the vendor-assigned version number of the miniport driver.

Set requests are not supported.

Remarks

The low-order half of the return value specifies the minor version; the high-order half specifies the major version.

This OID is mandatory for NDIS 6.0 and later miniport drivers.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_GEN_VENDOR_ID

Article • 02/18/2023

As a query, the OID_GEN_VENDOR_ID OID specifies a three-byte IEEE-registered vendor code, followed by a single byte that the vendor assigns to identify a particular NIC.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Mandatory.

NDIS 5.1 miniport drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Mandatory.

Remarks

The IEEE code uniquely identifies the vendor and is the same as the three bytes appearing at the beginning of the NIC hardware address.

Vendors without an IEEE-registered code should use the value 0xFFFFFF.

Independent hardware vendor's filter drivers or intermediate drivers might query this OID.

Requirements

Header	Ntddndis.h (include Ndis.h)

OID_GEN_VLAN_ID

Article • 02/18/2023

As a query, the OID_GEN_VLAN_ID OID reports the configured VLAN identifier (ID) for a NIC.

As a set, the OID_GEN_VLAN_ID OID specifies the configured VLAN identifier (ID) for an NIC that the miniport driver handles.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Optional.

NDIS 5.1 miniport drivers
Optional.

Windows XP
Supported.

NDIS 5.1 miniport drivers
Optional.

Remarks

The information buffer passed in this request contains an NDIS_VLAN_ID data type. This NDIS_VLAN_ID value contains the VLAN ID in the 12 least significant bits per the IEEE 802.1Q-2005 standard. Higher order bits of the NDIS_VLAN_ID value are reserved and must be set to 0. Note that NDIS defines NDIS_VLAN_ID as a ULONG.

When a transport uses OID_GEN_VLAN_ID in a query, the miniport driver returns the current configured VLAN ID for the NIC. When used in a set, the miniport driver sets the NIC's current configured VLAN ID to the specified value.

During the miniport driver's *MiniportInitializeEx* function for a particular NIC, the driver initially sets the NIC's VLAN ID to zero. The driver's *MiniportInitializeEx* function then reads the following configuration parameter from the registry, and, if the parameter is present, sets the NIC's VLAN ID to the parameter's value.

syntax

```
VlanId, REG_DWORD
```

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

OID_GEN_XMIT_DISCARDS

Article • 02/18/2023

As a query, NDIS and overlying drivers use the OID_GEN_XMIT_DISCARDS OID to determine the number of transmit discards on a miniport adapter.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later miniport drivers
Not requested. (see Remarks section)

Remarks

NDIS handles this OID for miniport drivers. See the OID_GEN_STATISTICS OID for more information about statistics.

The count that this OID returns is the number of packets that is discarded by the interface. The count is identical to the *ifOutDiscards* counter described in RFC 2863.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_XMIT_ERROR

Article • 02/18/2023

As a query, the OID_GEN_XMIT_ERROR OID specifies the number of frames that a NIC fails to transmit.

Version Information

Windows Vista and later versions of Windows
Obsolete.

NDIS 6.0 and later drivers
Not requested. Use OID_GEN_STATISTICS instead.

NDIS 5.1 drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 drivers
Mandatory.

Remarks

The count is identical to the *ifOutErrors* counter described in RFC 2863.

For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_GEN_XMIT_LINK_SPEED

Article • 02/18/2023

As a query, use the OID_GEN_XMIT_LINK_SPEED OID to determine the transmit link speed of a network interface.

Version Information

Windows Vista and later
Supported.

NDIS 6.0 and later miniport drivers
Not requested. For NDIS interface providers only.

Remarks

Only NDIS network interface providers, and therefore not miniport drivers or filter drivers, must support this OID as an OID request.

If the interface provider returns NDIS_STATUS_SUCCESS, the result of the query is a ULONG64 value that indicates the transmit link speed of the interface, in bits per second.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

NDIS Network Interface OIDs

OID_GEN_XMIT_OK

Article • 02/18/2023

As a query, the OID_GEN_XMIT_OK OID specifies the number of frames that are transmitted without errors.

Version Information

Windows Vista and later versions of Windows
Supported.

NDIS 6.0 and later drivers
Mandatory.

NDIS 5.1 drivers
Mandatory.

Windows XP
Supported.

NDIS 5.1 drivers
Mandatory.

Remarks

OID_GEN_XMIT_OK specifies the number of frames that are transmitted without errors. However, the OID_GEN_STATISTICS does not include this information.

NOTE: Statistics OIDs are mandatory for NDIS 6.0 and later miniport drivers unless NDIS handles them. For general information about statistics OIDs, see General Statistics.

Requirements

Header	Ntddndis.h (include Ndis.h)

See also

OID_GEN_STATISTICS

OID_IP4_OFFLOAD_STATS

Article • 02/18/2023

The host stack queries the OID_IP4_OFFLOAD_STATS OID to obtain statistics on IPv4 datagrams that an offload target has processed on offloaded TCP connections. The host stack sets this OID to cause an offload target to reset the counters for such statistics to zero.

In response to a query of OID_IP4_OFFLOAD_STATS, an offload target supplies a filled-in IP_OFFLOAD_STATS structure. The IP_OFFLOAD_STATS structure contains the statistics for IPv4 datagrams processed on offloaded TCP connections.

In response to a set of OID_IP4_OFFLOAD_STATS, an offload target should reset all of its IPv4 statistics counters for offloaded TCP connections to zero.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_IP6_OFFLOAD_STATS

Article • 02/18/2023

The host stack queries the OID_IP6_OFFLOAD_STATS OID to obtain statistics on IPv6 datagrams that an offload target has processed on offloaded TCP connections. The host stack sets this OID to cause an offload target to reset the counters for such statistics to zero.

In response to a query of OID_IP6_OFFLOAD_STATS, an offload target supplies a filled-in IP_OFFLOAD_STATS structure. The IP_OFFLOAD_STATS structure contains the statistics for IPv6 datagrams processed on offloaded TCP connections.

In response to a set of OID_IP6_OFFLOAD_STATS, an offload target should reset all of its IPv6 statistics counters for offloaded TCP connections to zero.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_NDK_CONNECTIONS

Article • 02/18/2023

As a query, NDIS and overlying drivers or user-mode applications use the OID_NDK_CONNECTIONS OID to query the list of active Network Direct connections from the miniport adapter.

NDIS 6.30 and later miniport drivers that provide NDK services must support this OID. Otherwise, this OID is optional.

Remarks

NDIS issues this OID to obtain the list of active Network Direct connections from an adapter. The adapter must return the list of connections with the NDIS_NDK_CONNECTIONS structure at the **InformationBuffer** member of the NDIS_OID_REQUEST structure.

This structure is variable-sized based on the number of connections that are returned. The size of the connection array, as element count, is specified in the **Count** member.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2012
Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NDK_CONNECTIONS

NDIS_OID_REQUEST

OID_NDK_LOCAL_ENDPOINTS

Article • 02/18/2023

As a query, NDIS and overlying drivers or user-mode applications use the OID_NDK_LOCAL_ENDPOINTS OID to the list of active Network Direct listeners and shared endpoints on a miniport adapter.

NDIS 6.30 and later miniport drivers that provide NDK services must support this OID. Otherwise, this OID is optional.

Remarks

NDIS issues this OID to obtain the list of active Network Direct listeners and shared endpoints from an adapter. The adapter is required to return the list of listeners and shared endpoints in the NDIS_NDK_LOCAL_ENDPOINTS structure at **InformationBuffer** member of the NDIS_OID_REQUEST structure.

This structure is variable-sized based on the number of local endpoints that are returned. The size of the local endpoint array, as element count, is specified in the **Count** member.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2012
Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NDK_LOCAL_ENDPOINTS

NDIS_OID_REQUEST

OID_NDK_SET_STATE

Article • 02/18/2023

As a set request, NDIS and overlying drivers use the OID_NDK_SET_STATE OID to set the state of the miniport adapter's NDK functionality.

NDIS 6.30 and later miniport drivers that provide NDK services must support this OID. Otherwise, this OID is optional.

Remarks

NDIS issues this OID with the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure pointing to a **BOOLEAN** and **InformationBufferLength** member equal to sizeof(**BOOLEAN**).

- If the **BOOLEAN** value is **TRUE** and the ***NetworkDirect** keyword value is nonzero, the miniport adapter's NDK functionality must be enabled.

  The miniport driver can read the ***NetworkDirect** keyword value by doing the following:

  1. Call **NdisOpenConfigurationEx** with the NDIS handle that the **NdisMRegisterMiniportDriver** function returned when the miniport driver was initialized. For more information about calling **NdisOpenConfigurationEx**, see Reading the Registry in an NDIS 6.0 Miniport Driver.

  2. Call **NdisReadConfiguration**, passing:

     - "*NetworkDirect" for the *Keyword* parameter

     - **NdisParameterInteger** for the *ParameterType* parameter

- If the **BOOLEAN** value is **FALSE**, the NDK functionality of the miniport adapter must be disabled.

To enable or disable its NDK functionality, the miniport driver's *MiniportOidRequest* callback function should follow the steps in Enabling and Disabling NDK Functionality.

Note An NDK-capable miniport driver must never call **NdisMNetPnPEvent** from the context of its *MiniportOidRequest* function, because doing so could cause a deadlock. Instead, it should call **NdisMNetPnPEvent** from some other context or queue a work item.

An NDK-capable miniport driver's *MiniportOidRequest* function must return **STATUS_SUCCESS** for an OID_NDK_SET_STATE OID request unless a failure occurs. The driver must not return **NDIS_STATUS_PENDING**.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2012
Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NdisMNetPnPEvent

NdisQueueIoWorkItem

NdisReadConfiguration

NDK_ADAPTER

OID_NDK_SET_STATE

OID_NDK_STATISTICS

Article • 02/18/2023

As a query, NDIS and overlying drivers or user-mode applications use the OID_NDK_STATISTICS OID to get the NDK statistics of a miniport adapter.

NDIS 6.30 and later miniport drivers that provide NDK services must support this OID. Otherwise, this OID is optional.

Note NDIS supports this OID with the direct OID request interface. For more information about the direct OID request interface, see NDIS 6.1 Direct OID Request Interface.

Remarks

NDIS issues this OID with the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure pointing to an **NDIS_NDK_STATISTICS_INFO** structure.

The NDK-capable miniport driver must provide the **CounterSet** member, which is a **NDIS_NDK_PERFORMANCE_COUNTERS** structure.

The counters are published to tools such as perfmon (see the NetworkDirect Activity performance counter) and made available programmatically with the Performance Data Helper (PDH) and Performance Library (PERFLIB) programming interfaces. For more information about these interfaces, see Performance Counters.

These counters are also available by calling the Get-NetAdapterStatistics PowerShell cmdlet with the **RdmaStatistics** attribute. For more information about the **RdmaStatistics** attribute, see **MSFT_NetAdapterStatisticsSettingData**.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2012
Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

Kernel Mode Performance Monitoring

NDIS_NDK_PERFORMANCE_COUNTERS

NDIS_NDK_STATISTICS_INFO

NDIS_OID_REQUEST

OID_NIC_SWITCH_ALLOCATE_VF

Article • 02/18/2023

An overlying driver issues an object identifier (OID) method request of OID_NIC_SWITCH_ALLOCATE_VF to allocate resources for a PCI Express (PCIe) Virtual Function (VF). The VF is exposed on a network adapter that supports the single root I/O virtualization (SR-IOV) interface.

Overlying drivers issue this OID method request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

Remarks

The PF miniport driver allocates software resources for a VF when the driver handles an object identifier (OID) method request of OID_NIC_SWITCH_ALLOCATE_VF. Even though the hardware resources have been allocated for a VF, it is considered to be nonoperational until the PF miniport driver successfully completes the OID_NIC_SWITCH_ALLOCATE_VF.

For more information about how to allocate VF resources, see Allocating Resources for a Virtual Function.

Note After an overlying driver requests resource allocation for a VF, that driver is the only component that can request the freeing of the resources for the same VF. The overlying driver must issue an OID set request of OID_NIC_SWITCH_FREE_VF to free the VF resources. Before the overlying driver can be halted, it must free the resources for each VF that was allocated by the driver's OID_NIC_SWITCH_ALLOCATE_VF request.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Status Code	**Description**
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_VF_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_NIC_SWITCH_VF_PARAMETERS). The PF miniport driver must set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_MAKE_RID

OID_NIC_SWITCH_CREATE_SWITCH

OID_NIC_SWITCH_CREATE_VPORT

NDIS_NIC_SWITCH_VF_PARAMETERS

OID_NIC_SWITCH_FREE_VF

OID_NIC_SWITCH_CREATE_SWITCH

Article • 02/18/2023

NDIS issues an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_SWITCH to create a NIC switch on a network adapter. When it handles this OID request, the miniport driver allocates the resources for the NIC switch on the adapter.

NDIS issues this OID method request to the miniport driver of the network adapter's PCI Express (PCIe) Physical Function (PF). This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

Note Overlying drivers, such as protocol or filter drivers, cannot issue OID method requests of OID_NIC_SWITCH_CREATE_SWITCH to the PF miniport driver.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_PARAMETERS** structure.

Remarks

When it receives the OID method request of OID_NIC_SWITCH_CREATE_SWITCH, the PF miniport driver must do the following:

1. If the PF miniport driver supports static switch creation and configuration, it creates the NIC switch when NDIS calls *MiniportInitializeEx*. When the driver handles this OID request, it must verify the configuration parameters in the **NDIS_NIC_SWITCH_PARAMETERS** structure. The parameters must be the same as those used by the driver to create the switch during the call to *MiniportInitializeEx*. If this is not true, the driver must fail the OID request.

   For more information, see Static Creation of a NIC Switch.

2. If the PF miniport driver supports dynamic switch creation and configuration, the driver must validate the configuration values of the **NDIS_NIC_SWITCH_PARAMETERS** structure and create the NIC switch based on these values.

   For more information, see Dynamic Creation of a NIC Switch.

3. The PF miniport driver must allocate the necessary hardware and software resources for the default VPort on the NIC switch.

Note The default VPort is always created through an OID request of OID_NIC_SWITCH_CREATE_SWITCH and deleted through an OID request of OID_NIC_SWITCH_DELETE_SWITCH. OID requests of OID_NIC_SWITCH_CREATE_VPORT and OID_NIC_SWITCH_DELETE_VPORT are used for the creation and deletion of nondefault VPorts on the NIC switch.

4. The PF miniport driver that supports dynamic switch creation and configuration must enable SR-IOV virtualization on the switch by calling **NdisMEnableVirtualization**. This call configures the **NumVFs** member and the **VF Enable** bit in the SR-IOV Extended Capability structure of the adapter's PCI Express (PCIe) configuration space.

   For more information about the SR-IOV configuration space, see the PCI-SIG Single Root I/O Virtualization and Sharing 1.1 specification.

   Note If the PF miniport driver supports static switch creation, it enables SR-IOV virtualization after it creates the switch when *MiniportInitializeEx* is called.

If the PF miniport driver successfully completes the OID method request of OID_NIC_SWITCH_CREATE_SWITCH, it allows the following to occur:

- VFs can be allocated on the NIC switch through OID method requests of OID_NIC_SWITCH_ALLOCATE_VF.
- Nondefault VPorts can be created on the NIC switch through OID method requests of OID_NIC_SWITCH_CREATE_VPORT.

For more information on how to handle this OID request, see Handling the OID_NIC_SWITCH_CREATE_SWITCH Request.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID method request of OID_NIC_SWITCH_CREATE_SWITCH.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the SR-IOV interface or is not enabled to use the interface.

Status Code	Description
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_NIC_SWITCH_PARAMETERS). The PF miniport driver must set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

NDIS_OID_REQUEST

NDIS_NIC_SWITCH_PARAMETERS

NdisMEnableVirtualization

OID_NIC_SWITCH_ALLOCATE_VF

OID_NIC_SWITCH_CREATE_VPORT

OID_NIC_SWITCH_CREATE_VPORT

Article • 02/18/2023

An overlying driver issues an object identifier (OID) method request of OID_NIC_SWITCH_CREATE_VPORT to create a nondefault virtual port (VPort) on a network adapter's NIC switch. This OID method request also attaches the created VPort to the network adapter's PCI Express (PCIe) Physical Function (PF) or a previously allocated PCIe Virtual Function (VF).

Overlying drivers issue this OID method request to the miniport driver for the network adapter's PF. This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure.

Remarks

The overlying driver initializes the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure with the configuration information about the nondefault VPort to be created. The configuration information includes the PCIe function to which the nondefault VPort is attached and the number of queue pairs for the nondefault VPort.

When the PF miniport driver is issued the OID request, the driver allocates the hardware and software resources associated with the specified nondefault VPort. After all the resources are successfully allocated, the PF miniport driver completes the OID successfully by returning NDIS_STATUS_SUCCESS from *MiniportOidRequest*.

If the OID_NIC_SWITCH_CREATE_VPORT request completes successfully, the PF miniport driver and the overlying driver must retain the **VPortId** value of the nondefault VPort for successive operations. The **VPortId** value is used during these operations:

- NDIS and the overlying drivers use the **VPortId** value to identify the nondefault VPort in successive OID requests related to this VPort, such as OID_NIC_SWITCH_VPORT_PARAMETERS and OID_NIC_SWITCH_DELETE_VPORT.

- During send operations, NDIS specifies the **VPortId** value to identify the VPort from which a packet was sent. This value is specified within the out-of-band (OOB) **NDIS_NET_BUFFER_LIST_FILTERING_INFO** data of the NET_BUFFER_LIST structure.

- During receive operations, the PF miniport driver specifies the **VPortId** value to which a packet is to be forwarded. This value is also specified in the OOB

NDIS_NET_BUFFER_LIST_FILTERING_INFO data of the NET_BUFFER_LIST structure.

For more information, see Creating a Virtual Port.

Note The default VPort always exists and is not created though an OID request of OID_NIC_SWITCH_CREATE_VPORT. The default VPort has an identifier of NDIS_DEFAULT_VPORT_ID. When the PF miniport driver creates a NIC switch, the driver automatically attaches the default VPort to the PF of the network adapter.

Return Status Codes

NDIS or the PF miniport driver returns one of the following status codes for the OID method request of OID_NIC_SWITCH_CREATE_SWITCH.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the SR-IOV interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_VPORT_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_NIC_SWITCH_VPORT_PARAMETERS). The PF miniport driver must set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportOidRequest

NDIS_NIC_SWITCH_PARAMETERS

NDIS_NIC_SWITCH_VPORT_PARAMETERS

NDIS_OID_REQUEST

NET_BUFFER_LIST

OID_NIC_SWITCH_ALLOCATE_VF

OID_NIC_SWITCH_DELETE_VPORT

OID_NIC_SWITCH_PARAMETERS

OID_NIC_SWITCH_VPORT_PARAMETERS

OID_NIC_SWITCH_CURRENT_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_NIC_SWITCH_CURRENT_CAPABILITIES to obtain the currently enabled hardware capabilities of the NIC switch in a network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_NIC_SWITCH_CAPABILITIES structure.

Remarks

Starting with NDIS 6.20, miniport drivers supply the currently enabled NIC switch hardware capabilities on the network adapter when its *MiniportInitializeEx* function is called. The driver initializes an NDIS_NIC_SWITCH_CAPABILITIES structure with the NIC switch hardware capabilities and sets the **CurrentNicSwitchCapabilities** member of the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure to a pointer to the **NDIS_NIC_SWITCH_CAPABILITIES** structure. The miniport driver then calls the NdisMSetMiniportAttributes function and sets the *MiniportAttributes* parameter to a pointer to an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

Note Starting with NDIS 6.30, miniport drivers that support the single root I/O virtualization (SR-IOV) interface must register the enabled hardware capabilities of the NIC switch. Drivers register these capabilities by calling NdisMSetMiniportAttributes.

Overlying protocol and filter drivers do not have to issue OID query requests of OID_NIC_SWITCH_CURRENT_CAPABILITIES. NDIS provides the currently enabled NIC switch hardware capabilities of a network adapter to these drivers in the following way:

- NDIS reports the currently enabled NIC switch hardware capabilities of an underlying network adapter to overlying protocol drivers in the **NicSwitchCapabilities** member of the NDIS_BIND_PARAMETERS structure during the bind operation.

- NDIS reports the currently enabled NIC switch hardware capabilities of an underlying network adapter to overlying filter drivers in the **NicSwitchCapabilities** member of the NDIS_FILTER_ATTACH_PARAMETERS structure during the attach operation.

Return Status Codes

NDIS handles the OID query request of the OID_NIC_SWITCH_CURRENT_CAPABILITIES request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_NIC_SWITCH_CURRENT_CAPABILITIES request, it returns one of the following status codes:

Status Code	Description
NDIS_STATUS_SUCCESS	The request completed successfully. The **InformationBuffer** points to an NDIS_NIC_SWITCH_CAPABILITIES structure.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_NIC_SWITCH_CAPABILITIES). The miniport driver must set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_FILTER_ATTACH_PARAMETERS

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NDIS_NIC_SWITCH_CAPABILITIES

NDIS_OID_REQUEST

OID_NIC_SWITCH_DELETE_SWITCH

Article • 02/18/2023

NDIS issues an object identifier (OID) set request of OID_NIC_SWITCH_DELETE_SWITCH to delete a NIC switch from a network adapter.

NDIS issues this OID set request to the miniport driver of the network adapter's PCI Express (PCIe) Physical Function (PF). This OID set request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

Note Overlying drivers, such as protocol or filter drivers, cannot issue this OID method request to the PF miniport driver.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_DELETE_SWITCH_PARAMETERS** structure.

Remarks

An OID set request of OID_NIC_SWITCH_DELETE_SWITCH deletes a NIC switch that was previously created through an OID method request of OID_NIC_SWITCH_CREATE_SWITCH.

When it receives the OID method request of OID_NIC_SWITCH_DELETE_SWITCH, the PF miniport driver must do the following:

1. If the PF miniport driver supports static creation and configuration of NIC switches, it must free the software resources associated with the specified NIC switch. However, the driver can only free the hardware resources for the NIC switch when *MiniportHaltEx* is called.

   For more information about static NIC switch creation, see Static Creation of a NIC Switch.

2. If the PF miniport driver supports the dynamic creation and configuration of NIC switches, it must free the hardware and software resources associated with the specified NIC switch.

   For more information about dynamic NIC switch creation, see Dynamic Creation of a NIC Switch.

3. If the PF miniport driver supports the dynamic creation and all the NIC switches have been deleted, the driver must disable virtualization on the adapter by calling

NdisMEnableVirtualization. To disable virtualization, the network adapter must set the *EnableVirtualization* parameter to FALSE and the *NumVFs* parameter to zero.

NdisMEnableVirtualization clears the **NumVFs** member and the **VF Enable** bit in the SR-IOV Extended Capability structure in the PCI configuration space of the network adapter's PF.

Note If the PF miniport driver supports static creation and configuration of NIC switches, it must only call **NdisMEnableVirtualization** when *MiniportHaltEx* is called.

For more information, see Deleting a NIC Switch.

Return Status Codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the **NdisMOidRequestComplete** function, passing NDIS_STATUS_SUCCESS for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport driver is resetting.
NDIS_STATUS_REQUEST_ABORTED	The miniport driver stopped processing the request. For example, NDIS called the *MiniportResetEx* function.

NDIS returns one of the following status codes for this request:

Term	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the SR-IOV interface or is not enabled to use the interface.

Term	Description
NDIS_STATUS_FILE_NOT_FOUND	One or more of the members of the NDIS_NIC_SWITCH_DELETE_SWITCH_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer is too small. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportHaltEx

NDIS_OID_REQUEST

NDIS_NIC_SWITCH_DELETE_SWITCH_PARAMETERS

OID_NIC_SWITCH_ALLOCATE_VF

OID_NIC_SWITCH_CREATE_SWITCH

OID_NIC_SWITCH_DELETE_VPORT

OID_NIC_SWITCH_FREE_VF

OID_NIC_SWITCH_DELETE_VPORT

Article • 02/18/2023

An overlying driver issues an object identifier (OID) set request of OID_NIC_SWITCH_DELETE_VPORT to delete a nondefault virtual port (VPort) that was previously created on a network adapter's NIC switch. The overlying driver can delete a VPort that it has previously created only by issuing an OID method request of OID_NIC_SWITCH_CREATE_VPORT.

Overlying drivers issue this OID set request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID set request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to the **NDIS_NIC_SWITCH_DELETE_VPORT_PARAMETERS** structure.

Remarks

An overlying driver, such as a protocol or filter driver, can only delete a nondefault VPort that it has previously created. The overlying driver creates a VPort by issuing an OID method request of OID_NIC_SWITCH_CREATE_VPORT.

When the PF miniport driver receives the OID request of OID_NIC_SWITCH_DELETE_VPORT, the driver must free the hardware and software resources that were allocated for the specified VPort.

For more information, see Deleting a Virtual Port.

Note Only nondefault VPorts can be explicitly deleted through OID requests of OID_NIC_SWITCH_DELETE_VPORT. The default VPort is implicitly deleted when the PF miniport driver deletes the default NIC switch. For more information, see Deleting a NIC Switch.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID set request of OID_NIC_SWITCH_DELETE_VPORT.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Status Code	Description
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_DELETE_VPORT_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_NIC_SWITCH_DELETE_VPORT_PARAMETERS). The PF miniport driver must set the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NIC_SWITCH_DELETE_VPORT_PARAMETERS

NDIS_OID_REQUEST

NdisCloseAdapterEx

OID_NIC_SWITCH_CREATE_VPORT

OID_NIC_SWITCH_DELETE_SWITCH

OID_NIC_SWITCH_ENUM_SWITCHES

Article • 02/18/2023

An overlying driver or user-mode application issues an object identifier (OID) query request of OID_NIC_SWITCH_ENUM_SWITCHES to obtain an array. Each element in the array specifies the attributes of a NIC switch that has been created on a network adapter.

After a successful return from this OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer that contains the following:

- An NDIS_NIC_SWITCH_INFO_ARRAY structure that defines the number of elements within the array.

- An array of NDIS_NIC_SWITCH_INFO structures. Each of these structures contains the information about a single NIC switch created on the network adapter.

  Note If the network adapter has no NIC switches, the driver sets the **NumElements** member of the NDIS_NIC_SWITCH_INFO_ARRAY structure to zero and no NDIS_NIC_SWITCH_INFO structures are returned.

Remarks

Overlying drivers and user-mode applications issue OID query requests of OID_NIC_SWITCH_ENUM_SWITCHES to enumerate the NIC switches created on a network adapter.

Note Starting with Windows Server 2012, the single root I/O virtualization (SR-IOV) interface only supports the default NIC switch on the network adapter. Therefore, the returned NDIS_NIC_SWITCH_INFO_ARRAY structure must specify a single NDIS_NIC_SWITCH_INFO element for the default NIC switch, which is referenced by the identifier of NDIS_DEFAULT_SWITCH_ID.

Return Status Codes

NDIS handles the OID query request of the OID_NIC_SWITCH_ENUM_SWITCHES request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_NIC_SWITCH_ENUM_SWITCHES request, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the SR-IOV interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_INFO_ARRAY structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NIC_SWITCH_INFO

NDIS_NIC_SWITCH_INFO_ARRAY

NDIS_OID_REQUEST

OID_NIC_SWITCH_CREATE_SWITCH

OID_NIC_SWITCH_PARAMETERS

OID_NIC_SWITCH_ENUM_VFS

Article • 02/18/2023

An overlying driver or user-mode application issues an object identifier (OID) method request of OID_NIC_SWITCH_ENUM_VFS to obtain an array. Each element in the array specifies the attributes of a PCI Express (PCIe) Virtual Function (VF) that are attached to a NIC switch on a network adapter's NIC switch.

After a successful return from this OID query request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer that contains the following:

- An **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure that defines the number of elements within the array.

- An array of **NDIS_NIC_SWITCH_VF_INFO** structures. Each of these structures contains information about a single VF on a NIC switch of the network adapter. A VF is attached to a NIC switch through OID method requests of OID_NIC_SWITCH_ALLOCATE_VF.

  Note If no VFs are attached to a NIC switch on the network adapter, the **NumElements** member of the **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure is set to zero and no **NDIS_NIC_SWITCH_VF_INFO** structures are returned.

Remarks

Overlying drivers and user-mode applications issue OID method requests of OID_NIC_SWITCH_ENUM_VFS to enumerate the VFs attached to a network adapter's NIC switch.

Before the driver or application issues the OID request, it must initialize an **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure that is passed along with the request. The driver or application must follow these guidelines when initializing the **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure:

- If the NDIS_NIC_SWITCH_VF_INFO_ARRAY_ENUM_ON_SPECIFIC_SWITCH flag is set in the **Flags** member, the driver or application must set the **SwitchId** member to the NIC switch identifier on the SR-IOV network adapter. By setting these members in this way, VF information is returned only for the specified NIC switch on the SR-IOV network adapter.

Note The overlying driver and user-mode application can obtain the NIC switch identifiers by issuing an OID query request of OID_NIC_SWITCH_ENUM_SWITCHES.

- If the **Flags** member is set to zero, the driver or application must set the **SwitchId** member to zero. By setting these members in this way, VF information is returned for all NIC switch on the SR-IOV network adapter.

Note Starting with Windows Server 2012, Windows supports only the default NIC switch on the network adapter. Regardless of the flags set in the **Flags** member, the **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

For more information about NIC switches, see NIC Switches.

Return Status Codes

NDIS handles the OID method request of the OID_NIC_SWITCH_ENUM_VFS request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_NIC_SWITCH_ENUM_VFS request, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the **NDIS_NIC_SWITCH_VF_INFO_ARRAY** structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NIC_SWITCH_VF_INFO

NDIS_NIC_SWITCH_VF_INFO_ARRAY

NDIS_OID_REQUEST

OID_NIC_SWITCH_ALLOCATE_VF

OID_NIC_SWITCH_CREATE_SWITCH

OID_NIC_SWITCH_VF_PARAMETERS

OID_NIC_SWITCH_ENUM_VPORTS

Article • 02/18/2023

An overlying driver or user-mode application issues an object identifier (OID) method request of OID_NIC_SWITCH_ENUM_VPORTS to obtain an array. Each element in the array specifies the attributes of a virtual port (VPort) that has been created on a network adapter's NIC switch.

After a successful return from this OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer that contains the following:

- An NDIS_NIC_SWITCH_VPORT_INFO_ARRAY structure that defines the number of elements within the array.

- An array of NDIS_NIC_SWITCH_VPORT_INFO structures. Each of these structures contains information about a VPort on the network adapter's NIC switch.

  Note If no VPorts have been created on the network adapter, the driver sets the **NumElements** member of the NDIS_NIC_SWITCH_VPORT_INFO_ARRAY structure to zero and no NDIS_NIC_SWITCH_VPORT_INFO structures are returned.

Remarks

Overlying drivers and user-mode applications issue OID query requests of OID_NIC_SWITCH_ENUM_VPORTS to enumerate the VPorts that are allocated on a network adapter's NIC switch.

Before the driver or application issues the OID request, it must initialize an NDIS_NIC_SWITCH_VPORT_INFO_ARRAY structure that is passed along with the request. The driver or application must follow these guidelines when initializing the **NDIS_NIC_SWITCH_VPORT_INFO_ARRAY** structure:

- If the NDIS_NIC_SWITCH_VPORT_INFO_ARRAY_ENUM_ON_SPECIFIC_SWITCH flag is set in the **Flags** member, information is returned for all VPorts created on a specified NIC switch. The NIC switch is specified by the **SwitchId** member of that structure.

  Note Starting with Windows Server 2012, the SR-IOV interface supports only the default NIC switch on the network adapter. Regardless of the flags that are set in the **Flags** member, the **SwitchId** member must be set to NDIS_DEFAULT_SWITCH_ID.

- If the NDIS_NIC_SWITCH_VPORT_INFO_ARRAY_ENUM_ON_SPECIFIC_FUNCTION flag is set in the **Flags** member, information is returned for all VPorts attached to a specified PCI Express (PCIe) Physical Function (PF) or Virtual Function (VF) on the network adapter. The PF or VF is specified by the **AttachedFunctionId** member of that structure.

  If the **AttachedFunctionId** member is set to NDIS_PF_FUNCTION_ID, information is returned for all VPorts, including the default VPort, that are attached to the network adapter's PF. If the **AttachedFunctionId** member is set to a valid VF identifier, information is returned for all VPorts to the specified VF.

  Note Starting with Windows Server 2012, only one nondefault VPort can be attached to a VF. However, multiple VPorts (including the default VPort) can be attached to the PF.

- If the **Flags** member is set to zero, information is returned for all VPorts attached to the PF or VF on the network adapter. In this case, the values of the **SwitchId** and **AttachedFunctionId** are ignored.

For more information, see Enumerating Virtual Ports on a Network Adapter.

Return Status Codes

NDIS handles the OID method request of the OID_NIC_SWITCH_ENUM_VPORTS request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_NIC_SWITCH_ENUM_VPORTS request, it returns one of the following status codes:

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_VF_INFO_ARRAY structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

Status Code	Description
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NIC_SWITCH_VPORT_INFO

NDIS_NIC_SWITCH_VPORT_INFO_ARRAY

NDIS_OID_REQUEST

OID_NIC_SWITCH_CREATE_SWITCH

OID_NIC_SWITCH_PARAMETERS

OID_NIC_SWITCH_FREE_VF

Article • 02/18/2023

An overlying driver issues an object identifier (OID) set request of OID_NIC_SWITCH_FREE_VF to free the resources for a network adapter's PCI Express (PCIe) Virtual Function (VF).

Overlying drivers issue this OID set request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID set request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_FREE_VF_PARAMETERS** structure.

The overlying driver specifies the identifier of the VF to be freed through the **VFId** member of this structure. The driver obtained this identifier from an earlier OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

Remarks

An overlying driver issues an OID set request of OID_NIC_SWITCH_FREE_VF to free the resources for a VF. These resources were previously allocated through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

For more information about how to free VF resources, see Freeing Resources for a Virtual Function.

Note Once an overlying driver requests resource allocation for a VF, that driver is the only component that can request the freeing of the resources for the same VF. The overlying driver must issue an OID set request of OID_NIC_SWITCH_FREE_VF to free the VF resources. Before the overlying driver can be halted, it must free the resources for each VF that was allocated by the driver's OID_NIC_SWITCH_ALLOCATE_VF request.

Return status codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.

Term	Description
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the NdisMOidRequestComplete function, passing **NDIS_STATUS_SUCCESS** for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport driver is resetting.
NDIS_STATUS_REQUEST_ABORTED	The miniport driver stopped processing the request. For example, NDIS called the *MiniportResetEx* function.

NDIS returns one of the following status codes for this request:

Term	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the SR-IOV interface or is not enabled to use the interface.
NDIS_STATUS_FILE_NOT_FOUND	One or more of the members of the NDIS_NIC_SWITCH_FREE_VF_PARAMETERS structure have invalid values. For example, the **VFId** member might specify a VF that either has not been allocated or that has VPorts that have not been deleted.
NDIS_STATUS_INVALID_LENGTH	The information buffer is too small. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NIC_SWITCH_FREE_VF_PARAMETERS

NDIS_OID_REQUEST

NdisCloseAdapterEx

OID_NIC_SWITCH_ALLOCATE_VF

OID_NIC_SWITCH_CREATE_VPORT

OID_NIC_SWITCH_DELETE_VPORT

OID_NIC_SWITCH_DELETE_SWITCH

OID_NIC_SWITCH_HARDWARE_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_NIC_SWITCH_HARDWARE_CAPABILITIES to obtain the hardware capabilities of the NIC switch in the network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_NIC_SWITCH_CAPABILITIES structure.

Remarks

The NDIS_NIC_SWITCH_CAPABILITIES structure contains information about the hardware capabilities of a NIC switch on the network adapter. These capabilities can include the hardware capabilities that are currently disabled by the INF file settings or through the **Advanced** properties page.

Note All the capabilities of the specified NIC switch are returned through an OID query request of OID_NIC_SWITCH_HARDWARE_CAPABILITIES, regardless of whether a capability is enabled or disabled.

Starting with NDIS 6.20, miniport drivers supply the NIC switch hardware capabilities when its *MiniportInitializeEx* function is called. The driver initializes an NDIS_NIC_SWITCH_CAPABILITIES structure with the NIC switch hardware capabilities and sets the **HardwareNicSwitchCapabilities** member of the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure to a pointer to the **NDIS_NIC_SWITCH_CAPABILITIES** structure. The miniport driver then calls the NdisMSetMiniportAttributes function and sets the *MiniportAttributes* parameter to a pointer to an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

Note Starting with NDIS 6.30, miniport drivers that support the single root I/O virtualization (SR-IOV) interface must register the hardware capabilities of the NIC switch. Drivers register these capabilities by calling NdisMSetMiniportAttributes.

Return Status Codes

NDIS handles the OID query request of OID_NIC_SWITCH_HARDWARE_CAPABILITIES request for miniport drivers, and returns one of the following status codes:

Status Code	Description
NDIS_STATUS_SUCCESS	The request completed successfully. The **InformationBuffer** points to an NDIS_NIC_SWITCH_CAPABILITIES structure.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_NIC_SWITCH_CAPABILITIES). NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NDIS_NIC_SWITCH_CAPABILITIES

NDIS_OID_REQUEST

OID_NIC_SWITCH_PARAMETERS

Article • 02/18/2023

An overlying driver issues an object identifier (OID) method request of OID_NIC_SWITCH_PARAMETERS to obtain the current configuration parameters of a specified NIC switch on a network adapter. NDIS handles these OID method requests for the miniport driver.

Overlying drivers issue an OID set request of OID_NIC_SWITCH_PARAMETERS to set the configuration parameters of a specified NIC switch on a network adapter. These OID set requests are issued to the miniport driver of the network adapter's PCI Express (PCIe) Physical Function (PF). These OID set requests are required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_PARAMETERS** structure.

The overlying driver specifies the NIC switch for the OID method or set request by setting the **SwitchId** member of the **NDIS_NIC_SWITCH_PARAMETERS** structure to the switch identifier. The overlying driver obtains the switch identifier through one of the following ways:

- From a previous OID method request of OID_NIC_SWITCH_ENUM_SWITCHES.
- From the **NicSwitchArray** member of the **NDIS_BIND_PARAMETERS** structure. NDIS passes a pointer to this structure in the *BindParameters* parameter of the *ProtocolBindAdapterEx* function.
- From the **NicSwitchArray** member of the **NDIS_FILTER_ATTACH_PARAMETERS** structure. NDIS passes a pointer to this structure in the *AttachParameters* parameter of the *FilterAttach* function.

Note Starting with Windows Server 2012, Windows supports only the default NIC switch on the network adapter. The **SwitchId** member of the **NDIS_NIC_SWITCH_PARAMETERS** structure must be set to NDIS_DEFAULT_SWITCH_ID.

Remarks

The overlying driver issues OID_NIC_SWITCH_PARAMETERS requests in the following way:

- The overlying driver issues an OID method request of OID_NIC_SWITCH_PARAMETERS to obtain the current parameters of a specified NIC switch. For more information, see Querying the Parameters of a NIC Switch.

  Note NDIS handles OID method requests of OID_NIC_SWITCH_PARAMETERS for the PF miniport driver.

- The overlying driver issues an OID set request of OID_NIC_SWITCH_PARAMETERS to change the current parameters of a specified NIC switch. For more information, see Setting the Parameters of a NIC Switch.

  Note The PF miniport driver handles OID set requests of OID_NIC_SWITCH_PARAMETERS.

Return Status Codes

NDIS or the PF miniport driver returns the following status codes for set or method OID requests of OID_NIC_SWITCH_PARAMETERS.

Status Code	**Description**
NDIS_STATUS_SUCCESS	The request completed successfully. The **InformationBuffer** points to an NDIS_NIC_SWITCH_CAPABILITIES structure.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS or the PF miniport driver sets the **DATA.METHOD_INFORMATION.BytesNeeded** member (for OID method requests) or **DATA.SET_INFORMATION.BytesNeeded** member (for OID set requests) in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_REINIT_REQUIRED	The PF miniport driver requires a reinitialization of the network adapter to apply the changes to the NIC switch.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

FilterAttach

NDIS_BIND_PARAMETERS

NDIS_FILTER_ATTACH_PARAMETERS

NDIS_NIC_SWITCH_PARAMETERS

NDIS_OID_REQUEST

OID_NIC_SWITCH_CREATE_SWITCH

OID_NIC_SWITCH_ENUM_SWITCHES

ProtocolBindAdapterEx

OID_NIC_SWITCH_VF_PARAMETERS

Article • 02/18/2023

An overlying driver or user-mode application issues an object identifier (OID) method request of OID_NIC_SWITCH_VF_PARAMETERS to obtain the current configuration parameters of a PCI Express (PCIe) Virtual Function (VF) on a network adapter. Only VFs that have resources allocated through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF can be queried through an OID method request of OID_NIC_SWITCH_VF_PARAMETERS.

NDIS handles the OID method request of OID_NIC_SWITCH_VF_PARAMETERS for miniport drivers.

When the OID method request is made, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

Remarks

The overlying driver or user-mode application specifies the VF to query by setting the **VFId** member of the **NDIS_NIC_SWITCH_VF_PARAMETERS** structure to the identifier of the VF. The overlying driver or application obtains the VF identifier through one of the following ways:

- By issuing an OID method request of OID_NIC_SWITCH_ENUM_VFS.

  If this OID request is completed successfully, the overlying driver or user-mode application receives a list of all VFs allocated on the network adapter. Each element within the list is an **NDIS_NIC_SWITCH_VF_INFO** structure, with the VF identifier specified by the **VFId** member.

- By issuing an OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

  If this OID request is completed successfully, the overlying driver receives the identifier of the newly created VF in the **VFId** member of the returned **NDIS_NIC_SWITCH_VF_PARAMETERS** structure.

  Note Only overlying drivers can obtain the VF identifier in this manner.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an

NDIS_NIC_SWITCH_VF_PARAMETERS structure. This structure contains the configuration parameters for the specified VF.

Return Status Codes

NDIS handles the OID method request of OID_NIC_SWITCH_VF_PARAMETERS for miniport drivers, and returns the following status code for OID method requests of OID_NIC_SWITCH_VF_PARAMETERS.

Status Code	Description
NDIS_STATUS_SUCCESS	The request completed successfully. The **InformationBuffer** member points to an NDIS_NIC_SWITCH_VF_PARAMETERS structure.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_VF_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_NIC_SWITCH_VF_PARAMETERS). NDIS sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NIC_SWITCH_VF_PARAMETERS

NDIS_OID_REQUEST

OID_NIC_SWITCH_ALLOCATE_VF

OID_NIC_SWITCH_ENUM_VFS

NDIS_NIC_SWITCH_VF_INFO

OID_NIC_SWITCH_VF_PARAMETERS

OID_NIC_SWITCH_VPORT_PARAMETERS

Article • 02/18/2023

An overlying driver can obtain the parameters for a virtual port (VPort) on a NIC switch that has been created on a network adapter that supports single root I/O virtualization (SR-IOV). The driver issues an object identifier (OID) method request of OID_NIC_SWITCH_VPORT_PARAMETERS to obtain these parameters.

Overlying drivers issue an OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS to set the configuration parameters of a specified VPort that is attached to the network adapter's NIC switch. These OID set requests are issued to the miniport driver of the network adapter's PCI Express (PCIe) Physical Function (PF). These OID set requests are required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure.

The overlying driver specifies the VPort for the OID method or set request by setting the **VPortId** member of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure to the identifier associated with the VPort. The overlying driver obtains the VPort identifier through one of the following ways:

- From a previous OID method request of OID_NIC_SWITCH_CREATE_VPORT.
- From a previous OID method request of OID_NIC_SWITCH_ENUM_VPORTS.

Remarks

OID_NIC_SWITCH_VPORT_PARAMETERS can be used in either OID method requests or OID set requests.

Handling OID Method Requests of OID_NIC_SWITCH_VPORT_PARAMETERS

Overlying drivers issue an OID method request of OID_NIC_SWITCH_VPORT_PARAMETERS to query the current configuration parameters of a VPort that is attached to the network adapter's NIC switch. Overlying drivers specify the VPort to query by setting the **VPortId** member of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure to the VPort identifier.

NDIS handles the OID method request of OID_NIC_SWITCH_VPORT_PARAMETERS for miniport drivers. NDIS returns information that it obtained from previous OID requests of OID_NIC_SWITCH_CREATE_VPORT and OID_NIC_SWITCH_ENUM_VPORTS.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure. This structure contains the configuration parameters for the specified switch.

For more information, see Querying the Parameters of a Virtual Port.

Handling OID Set Requests of OID_NIC_SWITCH_VPORT_PARAMETERS

Overlying drivers issue an OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS to change the current configuration parameters of a VPort that is attached to a network adapter's NIC switch. This OID request can be used to update the parameters for default as well as nondefault VPorts.

Only a limited subset of configuration parameters for a VPort can be changed. The overlying driver specifies the parameter to change by setting the following members of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure:

1. The **VPortId** member is set to the identifier of the VPort whose parameters will be changed.

2. The appropriate NDIS_NIC_SWITCH_VPORT_PARAMETERS_*Xxx*_CHANGED flags are set in the **Flags** member. Members of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure can only be changed if a corresponding NDIS_NIC_SWITCH_PARAMETERS_*Xxx*_CHANGED flag is defined in Ntddndis.h.

3. The corresponding members of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure are set with the VPort configuration parameters that are to be changed.

After the PF miniport driver receives the OID set request of OID_NIC_SWITCH_VPORT_PARAMETERS, the driver configures the hardware with the configuration parameters. The driver can only change those configuration parameters identified by NDIS_NIC_SWITCH_VPORT_PARAMETERS_*Xxx*_CHANGED flags in the **Flags** member of the **NDIS_NIC_SWITCH_VPORT_PARAMETERS** structure.

For more information, see Setting the Parameters of a Virtual Port.

Return Status Codes

NDIS or the PF miniport driver returns the following status code for set or method OID requests of OID_NIC_SWITCH_VPORT_PARAMETERS.

Status Code	Description
NDIS_STATUS_SUCCESS	The request completed successfully. The **InformationBuffer** points to an NDIS_NIC_SWITCH_CAPABILITIES structure.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_NIC_SWITCH_VPORT_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS or the PF miniport driver sets the **DATA.METHOD_INFORMATION.BytesNeeded** member (for OID method requests) or **DATA.SET_INFORMATION.BytesNeeded** member (for OID set requests) in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_NIC_SWITCH_VPORT_PARAMETERS

NDIS_OID_REQUEST

OID_NIC_SWITCH_CREATE_VPORT

OID_NIC_SWITCH_ENUM_VPORTS

OID_OFFLOAD_ENCAPSULATION

Article • 02/18/2023

As a query request, overlying drivers use the OID_OFFLOAD_ENCAPSULATION OID to obtain the current task offload encapsulation settings of an underlying miniport adapter. NDIS handles this OID query for miniport drivers.

As a set request, overlying drivers use the OID_OFFLOAD_ENCAPSULATION OID to set the task offload encapsulation settings of an underlying miniport adapter. Miniport drivers that support task offload must handle this OID set request.

Remarks

The InformationBuffer member of the NDIS_OID_REQUEST structure contains an NDIS_OFFLOAD_ENCAPSULATION structure.

Miniport drivers

If a miniport driver does not support offload and this OID, the driver should return NDIS_STATUS_NOT_SUPPORTED.

Miniport drivers must use the contents of the NDIS_OFFLOAD_ENCAPSULATION structure to update the currently reported TCP offload capabilities. After the update, the miniport driver must report the current task offload capabilities with the NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG status indication. This status indication ensures that all of the overlying protocol drivers are updated with the new capabilities information.

This OID is used to activate all configured or enabled offloads, or deactivate all offloads (in other words, the hardware starts to perform the offloads). It does not provide fine control over individual offloads. Instead, OID_TCP_OFFLOAD_PARAMETERS is used to configure individual offloads and can also activate them. Generally, most TCP/IP task offloads can be configured and activated with OID_TCP_OFFLOAD_PARAMETERS.

However, this OID's NDIS_OFFLOAD_ENCAPSULATION structure also covers two other encapsulation types that are not covered by OID_TCP_OFFLOAD_PARAMETERS's NDIS_OFFLOAD_PARAMETERS structure: **NDIS_ENCAPSULATION_IEEE_802_3** and **NDIS_ENCAPSULATION_IEEE_LLC_SNAP_ROUTED**. Miniport drivers need to handle this difference in encapsulation types that are covered by the different OIDs.

If this OID is issued by the protocol driver to deactivate all offloads, the **Enabled** member of the NDIS_OFFLOAD_ENCAPSULATION member will be set to NDIS_OFFLOAD_SET_OFF.

Setting encapsulation (protocol drivers)

Protocol drivers set OID_OFFLOAD_ENCAPSULATION after determining the system encapsulation requirements. A protocol driver can determine the capabilities of the underlying miniport adapter from the NDIS_BIND_PARAMETERS structure or by querying OID_TCP_OFFLOAD_CURRENT_CONFIG. The protocol driver must set an encapsulation type that the miniport adapter supports on at least one offload service.

If a miniport driver supports any offload type that supports the requested encapsulation type, the driver must return NDIS_STATUS_SUCCESS in response to a set of OID_OFFLOAD_ENCAPSULATION. Otherwise, the miniport driver should return NDIS_STATUS_INVALID_PARAMETER.

For send operations, a protocol driver can issue send requests by using only those offload types that the miniport adapter supports with the required encapsulation type. Therefore, if an OID set request of OID_OFFLOAD_ENCAPSULATION fails, the protocol driver must not use any offload settings in send requests that are directed to that miniport adapter.

For receive operations, the miniport driver must not start checksum or Internet protocol security (IPsec) offload services until after it receives an OID set request of OID_OFFLOAD_ENCAPSULATION.

Obtaining current encapsulation settings (protocol drivers)

A protocol driver can issue an OID_OFFLOAD_ENCAPSULATION query only after setting the OID_OFFLOAD_ENCAPSULATION OID.

NDIS responds with an NDIS_OFFLOAD_ENCAPSULATION structure that contains the current encapsulation settings.

Protocol drivers must be prepared to handle any NDIS_STATUS_Xxx failure code. If a failure occurs, the protocol driver must not attempt to perform any offload operations that are directed to the affected miniport adapter.

See also

NDIS_BIND_PARAMETERS

NDIS_OFFLOAD_ENCAPSULATION

NDIS_OID_REQUEST

NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG

OID_TCP_OFFLOAD_CURRENT_CONFIG

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_PACKET_COALESCING_FILTER_MATCH_COUNT

Article • 02/18/2023

NDIS issues an OID query request of OID_PACKET_COALESCING_FILTER_MATCH_COUNT to obtain the number of packets that were cached, or *coalesced*, on the network adapter. The network adapter coalesces received packets if the adapter is enabled for NDIS packet coalescing and the packet matches a receive filter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a caller-allocated ULONG64 variable. Before a successful return from the query request, the driver updates the ULONG64 variable with the number of packets that have matched receive filters on the network adapter.

Remarks

Starting with NDIS 6.30, drivers that support NDIS packet coalescing must support OID query requests of OID_PACKET_COALESCING_FILTER_MATCH_COUNT.

Note Drivers that support the single root I/O virtualization (SR-IOV) or virtual machine queue (VMQ) interfaces are not required to support OID query requests of this OID.

A miniport driver that supports packet coalescing must increment a ULONG64 counter for each received packet that was coalesced on the network adapter. Packets are coalesced if they match a receive filter, which overlying drivers download to the miniport driver through OID method requests of OID_RECEIVE_FILTER_SET_FILTER.

The driver returns the value of this counter when it handles an OID query request of OID_PACKET_COALESCING_FILTER_MATCH_COUNT.

The miniport driver must not clear the counter after it handles the OID query request of OID_PACKET_COALESCING_FILTER_MATCH_COUNT. The miniport driver must only clear the counter if the following conditions are true:

- The miniport driver handles an OID set request of OID_PNP_SET_POWER to resume to a full-power state of NdisDeviceStateD0.
- NDIS calls the miniport driver's *MiniportResetEx* function to reset the underlying network adapter.

For more information about packet coalescing, see NDIS Packet Coalescing.

Return status codes

The miniport driver returns one of the following status codes for the OID method request of OID_PACKET_COALESCING_FILTER_MATCH_COUNT:

NDIS_STATUS_SUCCESS
The OID request completed successfully.

NDIS_STATUS_INVALID_LENGTH
The information buffer was too short. The driver sets the **DATA.SET_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.

NDIS_STATUS_FAILURE
The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportResetEx

NDIS_OID_REQUEST

OID_PNP_SET_POWER

OID_RECEIVE_FILTER_SET_FILTER

OID_PD_CLOSE_PROVIDER

Article • 02/18/2023

An NDIS protocol or filter driver sends an object identifier (OID) method request of OID_PD_CLOSE_PROVIDER to the PDPI provider to give up access to the PD capability in a PDPI provider object.

An NDIS protocol or filter driver must call this OID when it receives an unbind notification, a pause indication, a low-power event, or a PD configuration change event that indicates the PD is disabled on the binding.

Before calling this OID, the NDIS protocol or filter driver must ensure that it has closed and freed all PD objects such as queues, counters, and filters that it created over the PD provider instance. The NDIS protocol or filter driver must guarantee that there are no in-progress calls to any of the PD provider dispatch table functions before issuing this OID.

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Ntddndis.h (include Ndis.h)

See also

MiniportOidRequest

NDIS_PD_CLOSE_PROVIDER_PARAMETERS

NDIS_STATUS_PD_CURRENT_CONFIG

OID_PD_OPEN_PROVIDER

OID_PD_OPEN_PROVIDER

Article • 02/18/2023

An NDIS protocol or filter driver sends an object identifier (OID) method request of OID_PD_OPEN_PROVIDER to a PD-capable miniport driver to gain access to the PD capability in the miniport driver's PDPI provider object. All PD-capable miniport drivers must handle this OID request.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer contains the following data:

- An **NDIS_PD_OPEN_PROVIDER_PARAMETERS** structure

Remarks

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Ntddndis.h (include Ndis.h)

See also

MiniportOidRequest

NDIS_PD_OPEN_PROVIDER_PARAMETERS

NDIS_STATUS_PD_CURRENT_CONFIG

OID_PD_CLOSE_PROVIDER

OID_PD_QUERY_CURRENT_CONFIG

Article • 02/18/2023

An NDIS protocol or filter driver sends an object identifier (OID) method request of OID_PD_QUERY_CURRENT_CONFIG to a PD-capable miniport driver to retrieve the PD status and capabilities. All PD-capable miniport drivers must handle this OID request.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_PD_CONFIG structure

Remarks

Requirements

Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	Ntddndis.h (include Ndis.h)

See also

MiniportOidRequest

NDIS_PD_CONFIG

NDIS_OID_REQUEST

OID_PM_ADD_PROTOCOL_OFFLOAD

Article • 02/18/2023

As a set, NDIS protocol drivers use the OID_PM_ADD_PROTOCOL_OFFLOAD OID to add a protocol offload for power management to a network adapter. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_PM_PROTOCOL_OFFLOAD** structure.

Remarks

NDIS 6.20 and later protocol drivers use OID_PM_ADD_PROTOCOL_OFFLOAD OID to add a protocol offload for power management to a network adapter. If the request is successful, the network adapter must generate and transmit the necessary response packets for the offloaded protocol when the network adapter is in a low power state.

A protocol driver can offload a protocol after it successfully binds to an underlying network adapter and as soon as it has the necessary data (such as the IP address of the interface) to offload the protocol. The protocol driver can also offload a protocol in response to some other power management event notifications, such as the rejection of a previously added WOL pattern or an offloaded protocol.

To avoid race conditions in NDIS and other protocol drivers that are bound to the same miniport adapter, after NDIS starts to set a network adapter to a low power state, it will fail any attempt to offload another protocol to that network adapter. For example, if an NDIS protocol driver tries to offload a protocol in the context of processing a **NetEventSetPower** event notification for that network adapter, NDIS will fail the request.

Before NDIS sends this OID request down to the underlying NDIS drivers or completes the request to the overlying driver, it sets the ULONG **ProtocolOffloadId** member of the **NDIS_PM_PROTOCOL_OFFLOAD** structure to a unique value. Protocol drivers and NDIS use this protocol offload identifier with the OID_PM_REMOVE_PROTOCOL_OFFLOAD OID request to remove the protocol offload from the underlying network adapter.

Note The protocol offload identifier is a unique value for each of the protocol offloads that are set on a network adapter. However, the protocol offload identifier is not globally unique across all network adapters.

If NDIS or an underlying network adapter rejects an offload, it generates an **NDIS_STATUS_PM_OFFLOAD_REJECTED** status indication. This can occur after returning NDIS_STATUS_SUCCESS for the OID. The **StatusBuffer** member of the

NDIS_STATUS_INDICATION structure contains the ULONG protocol offload identifier of the rejected protocol offload.

For information on how a Native 802.11 Wireless LAN miniport driver uses this OID, see Adding and Deleting Low Power Protocol Offloads.

The miniport driver returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The requested protocol offload was added successfully. The **ProtocolOffloadId** member of the NDIS_PM_PROTOCOL_OFFLOAD structure contains a protocol offload identifier.

NDIS_STATUS_PENDING
The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_PM_PROTOCOL_OFFLOAD_LIST_FULL
The request failed because the protocol offload list is full and the network adapter cannot add another protocol offload.

NDIS_STATUS_RESOURCES
NDIS or an underlying network adapter could not add the new protocol offload due to lack of resources.

NDIS_STATUS_INVALID_PARAMETER
One or more parameters in the NDIS_PM_PROTOCOL_OFFLOAD structure were invalid.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_NOT_SUPPORTED
The network adapter does not support the requested protocol offload.

NDIS_STATUS_FAILURE
The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later. Mandatory for miniport drivers.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PM_PROTOCOL_OFFLOAD

NDIS_STATUS_INDICATION

NDIS_STATUS_PM_OFFLOAD_REJECTED

OID_PM_REMOVE_PROTOCOL_OFFLOAD

Adding and Deleting Low Power Protocol Offloads

OID_PM_ADD_WOL_PATTERN

Article • 02/18/2023

As a set, NDIS protocol drivers use the OID_PM_ADD_WOL_PATTERN OID to add a power management wake-on-LAN pattern to a network adapter. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_PM_WOL_PATTERN** structure.

Remarks

NDIS 6.20 and later protocol drivers use OID_PM_ADD_WOL_PATTERN to add a Wake on LAN (WOL) pattern to a network adapter. The OID request contains criterion that the network adapter must compare to incoming packets when it is in a low power state. The network adapter must generate a wake up event when it receives a packet that matches the pattern criteria.

A protocol driver can add WOL patterns after it successfully binds to an underlying network adapter and as soon as it has the necessary data (such as the IP address of the interface) to set up the WOL pattern. The protocol driver can also add a WOL pattern in response to some other power management event notifications such as the rejection of a previously added WOL pattern or an offloaded protocol.

To avoid race conditions in NDIS and other protocol drivers that are bound to the same miniport adapter, after NDIS starts to set a network adapter to a low power state, it will fail any attempt to add a new wake up pattern to that network adapter. For example, if an NDIS protocol driver tries to add a new WOL pattern in the context of processing a **NetEventSetPower** event notification for that network adapter, NDIS will fail the request.

Before NDIS sends this OID request down to the underlying NDIS drivers or completes the request to the overlying driver, it sets the ULONG **PatternId** member of the **NDIS_PM_WOL_PATTERN** structure to a unique value. Protocol drivers and NDIS use this pattern identifier with the OID_PM_REMOVE_WOL_PATTERN OID request to remove the WOL pattern from the underlying network adapter.

Note The pattern identifier is a unique value for each of the patterns that are set on a network adapter. However, the pattern identifier is not globally unique across all miniport adapters.

If NDIS or an underlying network adapter removes a WOL pattern, it generates an **NDIS_STATUS_PM_WOL_PATTERN_REJECTED** status indication. The **StatusBuffer** member of the NDIS_STATUS_INDICATION structure contains the ULONG WOL pattern identifier of the rejected WOL pattern.

The miniport driver returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The requested pattern was added successfully. The **PatternId** member of the NDIS_PM_WOL_PATTERN structure contains a pattern identifier.

NDIS_STATUS_PENDING
The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_PM_WOL_PATTERN_LIST_FULL
The request failed because the pattern list is full and the network adapter cannot add another pattern.

NDIS_STATUS_RESOURCES
NDIS or underlying network adapter could not add the new pattern due to lack of resources.

NDIS_STATUS_INVALID_PARAMETER
One or more parameters in the NDIS_PM_WOL_PATTERN structure were invalid.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_NOT_SUPPORTED
The network adapter does not support the requested WOL pattern.

NDIS_STATUS_FAILURE
The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later. Mandatory for miniport drivers.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PM_WOL_PATTERN

NDIS_STATUS_INDICATION

NDIS_STATUS_PM_WOL_PATTERN_REJECTED

OID_PM_REMOVE_WOL_PATTERN

OID_PNP_ADD_WAKE_UP_PATTERN

OID_PM_CURRENT_CAPABILITIES

Article • 02/18/2023

As a query, overlying drivers can use the OID_PM_CURRENT_CAPABILITIES OID to query the currently available power management capabilities of a network adapter. After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_PM_CAPABILITIES structure.

Remarks

NDIS handles the query for miniport drivers. Starting with NDIS 6.20, miniport drivers supply the power management hardware capabilities during initialization. However, NDIS can hide some capabilities from the protocol driver. For example, NDIS might report different capabilities when a user disables some or all of the power management capabilities.

Note that the current power management capabilities that NDIS reports to a protocol driver are not necessarily the same as the hardware capabilities that the miniport driver reported to NDIS.

NDIS reports the power management capabilities of an underlying network adapter to overlying protocol drivers in the **PowerManagementCapabilitiesEx** member of the NDIS_BIND_PARAMETERS structure during the bind operation. Therefore, protocol drivers do not have to query the OID.

NDIS issues an NDIS_STATUS_PM_CAPABILITIES_CHANGE status indication to report changes in the power management capabilities that are available to overlying drivers.

If the underlying network adapter has an NDIS 6.1 or older miniport driver, NDIS translates the power management capabilities of the underlying network adapter to an NDIS_PM_CAPABILITIES structure.

NDIS returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** points to an NDIS_PM_CAPABILITIES structure.

NDIS_STATUS_PENDING
The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_FAILURE
The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later. Not requested for miniport drivers. (See Remarks section.)
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_OID_REQUEST

NDIS_PM_CAPABILITIES

NDIS_STATUS_PM_CAPABILITIES_CHANGE

OID_PM_GET_PROTOCOL_OFFLOAD

Article • 02/18/2023

An overlying driver issues an OID method request of OID_PM_GET_PROTOCOL_OFFLOAD to obtain parameter settings for a low power protocol offload from a network adapter.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure initially contains a pointer to a ULONG protocol offload identifier. After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an NDIS_PM_PROTOCOL_OFFLOAD structure.

Remarks

NDIS 6.20 and later protocol drivers use OID_PM_GET_PROTOCOL_OFFLOAD method OID to retrieve parameter settings for a low power protocol offload from a network adapter.

The information buffer must point to a ULONG-type protocol offload identifier. NDIS set this protocol offload identifier in the **ProtocolOffloadId** member of the NDIS_PM_PROTOCOL_OFFLOAD structure when NDIS sent the prior OID_PM_ADD_PROTOCOL_OFFLOAD OID request to the underlying network adapter.

The miniport driver returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The requested data was retrieved successfully. The information buffer contains the corresponding NDIS_PM_PROTOCOL_OFFLOAD structure.

NDIS_STATUS_PENDING
The request is pending completion. The final status code and results will be passed to the OID request completion handler of the caller.

NDIS_STATUS_INVALID_PARAMETER
The specified protocol offload identifier was not valid.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_NOT_SUPPORTED

The NDIS version of the miniport driver is below 6.20.

NDIS_STATUS_FAILURE

The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later. Mandatory for miniport drivers. (See Remarks section.)
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PM_PROTOCOL_OFFLOAD

OID_PM_ADD_PROTOCOL_OFFLOAD

OID_PM_HARDWARE_CAPABILITIES

Article • 02/18/2023

As a query, overlying drivers can use the OID_PM_HARDWARE_CAPABILITIES OID to query the power management hardware capabilities of a network adapter. After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_PM_CAPABILITIES structure.

Remarks

NDIS handles the query for miniport drivers. Starting with NDIS 6.20, miniport drivers supply the power management hardware capabilities during initialization in the **PowerManagementCapabilitiesEx** member of the NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES structure.

The miniport driver must issue an NDIS_STATUS_PM_CAPABILITIES_CHANGE status indication to report changes in the power management hardware capabilities of a network adapter to NDIS and overlying drivers.

NDIS returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** points to an NDIS_PM_CAPABILITIES structure.

NDIS_STATUS_PENDING
The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_FAILURE
The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later. Not

	requested for miniport drivers. (See Remarks section.)
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_MINIPORT_ADAPTER_GENERAL_ATTRIBUTES

NDIS_OID_REQUEST

NDIS_PM_CAPABILITIES

NDIS_STATUS_PM_CAPABILITIES_CHANGE

OID_PM_PARAMETERS

Article • 02/18/2023

As a query, protocol drivers can use the OID_PM_PARAMETERS OID to query the power management hardware capabilities of a network adapter that are currently enabled. After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_PM_PARAMETERS structure.

As a set, protocol drivers can use the OID_PM_PARAMETERS OID to enable or disable the current hardware capabilities of a network adapter. The protocol driver provides a pointer to an NDIS_PM_PARAMETERS structure in the **InformationBuffer** member of the NDIS_OID_REQUEST structure.

Remarks

Starting with NDIS 6.20, overlying protocol and filter drivers use OID_PM_PARAMETERS to query and set the power management hardware capabilities of a network adapter that are currently enabled.

When an overlying driver queries the OID_PM_PARAMETERS OID, NDIS completes the request without forwarding it to the miniport driver. NDIS stores the requested settings and combines them with the settings from other such requests. Before NDIS transitions the network adapter to the low power state, NDIS sends a set request to the miniport driver that contains the combined settings from all of the requests that NDIS stored.

The capabilities that are currently enabled can be a subset of the capabilities that the hardware supports. For more information about the capabilities that the hardware supports, see OID_PM_HARDWARE_CAPABILITIES.

Note If NDIS sets the NDIS_PM_SELECTIVE_SUSPEND_ENABLED flag in the **WakeUpFlags** member of NDIS_PM_PARAMETERS structure, it issues the OID set request of OID_PM_PARAMETERS directly to the miniport driver. This allows NDIS to bypass the processing by filter drivers in the networking driver stack.

NDIS or the miniport driver returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The request completed successfully.

NDIS_STATUS_PENDING
The request is pending completion. NDIS will pass the final status code and results to

the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_INVALID_PARAMETER
The request failed because it tried to enable a capability that the underlying network adapter does not support.

NDIS_STATUS_FAILURE
The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PM_PARAMETERS

OID_PM_HARDWARE_CAPABILITIES

OID_PM_PROTOCOL_OFFLOAD_LIST

Article • 02/18/2023

As a query, overlying drivers can use the OID_PM_PROTOCOL_OFFLOAD_LIST OID to enumerate the protocol offloads that are set on an underlying network adapter. After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a list of NDIS_PM_PROTOCOL_OFFLOAD structures that describe the currently active protocol offloads.

Remarks

NDIS handles the query for miniport drivers. NDIS drivers can use the OID_PM_PROTOCOL_OFFLOAD_LIST OID to get a list of protocol offloads that are set on an underlying network adapter.

For each NDIS_PM_PROTOCOL_OFFLOAD structure in the list, NDIS sets the **NextProtocolOffloadOffset** member to the offset from the beginning of the OID information buffer (that is, the beginning of the buffer that the **InformationBuffer** member of the NDIS_OID_REQUEST structure points to) to the beginning of the next NDIS_PM_PROTOCOL_OFFLOAD structure in the list. The offset in the **NextProtocolOffloadOffset** member of the last structure in the list is zero.

If there are no protocol offloads that are set on the network adapter, NDIS sets the **DATA.QUERY_INFORMATION.BytesWritten** member of the NDIS_OID_REQUEST structure to zero and returns NDIS_STATUS_SUCCESS. The data within the **DATA.QUERY_INFORMATION.InformationBuffer** member is not modified by NDIS.

NDIS returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** contains a pointer to a list of protocol offloads, if any.

NDIS_STATUS_PENDING
The request is pending completion. The final status code and results will be passed to the OID request completion handler of the caller.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the

DATA.QUERY_INFORMATION.BytesNeeded member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_FAILURE
The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later. Not requested for miniport drivers. (See Remarks section.)
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PM_PROTOCOL_OFFLOAD

OID_PM_REMOVE_PROTOCOL_OFFLOAD

Article • 02/18/2023

As a set request, NDIS and protocol drivers use the OID_PM_REMOVE_PROTOCOL_OFFLOAD OID to remove a power management protocol offload from a network adapter. The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a **ULONG** protocol offload identifier.

Remarks

NDIS and protocol drivers use the OID_PM_REMOVE_PROTOCOL_OFFLOAD OID to remove a protocol offload from the underlying network adapter.

The **DATA.SET_INFORMATION.InformationBuffer** member of the NDIS_OID_REQUEST structure must point to a **ULONG** value for a previously added protocol offload identifier. NDIS sets this protocol offload identifier in the **ProtocolOffloadId** member of the NDIS_PM_PROTOCOL_OFFLOAD structure when NDIS sent the prior OID_PM_ADD_PROTOCOL_OFFLOAD OID request to the underlying network adapter.

Remarks for miniport driver writers

NDIS ensures that the buffer size is at least **sizeof**(**ULONG**) and contains a valid protocol offload ID. Therefore, a miniport driver's *MiniportOidRequest* function should return NDIS_STATUS_SUCCESS for this request.

Note If the miniport driver is resetting, its *MiniportOidRequest* function should return NDIS_STATUS_NOT_ACCEPTED.

Return status codes

NDIS returns one of the following status codes for this request:

NDIS_STATUS_SUCCESS
The protocol offload was removed successfully.

NDIS_STATUS_PENDING
The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_INVALID_LENGTH
The information buffer is too small. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required, in bytes.

NDIS_STATUS_FILE_NOT_FOUND
The protocol offload identifier in the OID request is not valid.

Requirements

Version	Supported in NDIS 6.20 and later. Mandatory for miniport drivers.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PM_PROTOCOL_OFFLOAD

OID_PM_ADD_PROTOCOL_OFFLOAD

OID_PM_REMOVE_WOL_PATTERN

Article • 02/18/2023

As a set, NDIS and protocol drivers use the OID_PM_REMOVE_WOL_PATTERN OID to remove a power management wake on LAN (WOL) pattern from a network adapter. The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a ULONG pattern identifier.

Remarks

NDIS and protocol drivers use OID_PM_REMOVE_WOL_PATTERN to remove a wake on LAN (WOL) pattern from the underlying network adapter.

The **DATA.SET_INFORMATION.InformationBuffer** member of the **NDIS_OID_REQUEST** structure must point to a ULONG value for a previously added WOL pattern identifier. NDIS set this pattern identifier in the **PatternId** member of the **NDIS_PM_WOL_PATTERN** structure when NDIS sent the prior OID_PM_ADD_WOL_PATTERN OID request to the underlying network adapter.

Return Status Codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the **NdisMOidRequestComplete** function, passing **NDIS_STATUS_SUCCESS** for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport driver is resetting.
NDIS_STATUS_REQUEST_ABORTED	The miniport driver stopped processing the request. For example, NDIS called the *MiniportResetEx* function.

NDIS returns one of the following status codes for this request:

Term	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The NDIS version of the miniport driver is less than NDIS 6.20.
NDIS_STATUS_FILE_NOT_FOUND	The pattern identifier in the OID request is invalid.
NDIS_STATUS_INVALID_LENGTH	The information buffer is too small. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

Requirements

Version	Supported in NDIS 6.20 and later. Mandatory for miniport drivers.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PM_WOL_PATTERN

OID_PM_ADD_WOL_PATTERN

NDIS_STATUS_PM_WOL_PATTERN_REJECTED

OID_PM_WOL_PATTERN_LIST

Article • 02/18/2023

As a query, overlying drivers can use the OID_PM_WOL_PATTERN_LIST OID to enumerate the wake on LAN patterns that are set on an underlying network adapter. After a successful return from the query, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a list of NDIS_PM_WOL_PATTERN structures that describe the currently added WOL patterns.

Remarks

NDIS handles the query for miniport drivers. NDIS drivers can use the OID_PM_WOL_PATTERN_LIST OID to get a list of wake on LAN patterns that are set on an underlying network adapter.

For each NDIS_PM_WOL_PATTERN structure in the list, NDIS sets the **NextWoLPatternOffset** member to the offset from the beginning of the OID information buffer (that is, the beginning of the buffer that the **InformationBuffer** member of the NDIS_OID_REQUEST structure points to) to the beginning of the next **NDIS_PM_WOL_PATTERN** structure in the list. The offset in the **NextWoLPatternOffset** member of the last structure in the list is zero.

For offsets in an NDIS_PM_WOL_PATTERN structure other than **NextWoLPatternOffset** (for example, **NameBufferOffset**), NDIS provides offsets that are relative to the beginning of each **NDIS_PM_WOL_PATTERN** structure.

If there are no WOL patterns that are set on the network adapter, NDIS sets the **DATA.QUERY_INFORMATION.BytesWritten** member of the NDIS_OID_REQUEST structure to zero and returns **NDIS_STATUS_SUCCESS** for the request. The data within the **DATA.QUERY_INFORMATION.InformationBuffer** member is not modified by NDIS.

NDIS returns one of the following status codes for the request:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** contains a pointer to a list of WOL patterns, if any.

NDIS_STATUS_PENDING
The request is pending completion. The final status code and results will be passed to the OID request completion handler of the caller.

NDIS_STATUS_BUFFER_TOO_SHORT
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_FAILURE
The request failed for reasons other than the preceding reasons.

Requirements

Version	Supported in NDIS 6.20 and later. Not requested for miniport drivers. (See Remarks section.)
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_PM_WOL_PATTERN

OID_PM_ADD_WOL_PATTERN

OID_PM_REMOVE_WOL_PATTERN

OID_PNP_WAKE_UP_PATTERN_LIST

OID_PNP_ADD_WAKE_UP_PATTERN

Article • 02/18/2023

The OID_PNP_ADD_WAKE_UP_PATTERN OID is sent by a protocol driver to a miniport driver to specify a wake-up pattern. The wake-up pattern, along with its mask, is described by an **NDIS_PM_PACKET_PATTERN** structure.

A protocol that enables pattern-match wake-up for a miniport driver (see OID_PNP_ENABLE_WAKE_UP) uses OID_PNP_ADD_WAKE_UP_PATTERN to specify a wake-up pattern. The wake-up pattern can be stored in host memory or on the network adapter, depending on the capabilities of the network adapter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains the following:

- An **NDIS_PM_PACKET_PATTERN** structure that provides information about the pattern and its mask.

- A mask that indicates which bytes of an incoming packet should be compared with corresponding bytes in the pattern. The mask starts with the first byte of the packet. The mask immediately follows the **NDIS_PM_PACKET_PATTERN** structure in the *InformationBuffer*. For more information about how this mask works, see the Network Device Class Power Management Reference specification.

- A wake-up pattern, which begins **PatternOffset** bytes from the beginning of the *InformationBuffer*. For more information about wake-up patterns, see the Network Device Class Power Management Reference specification.

The number of wake-up patterns that the miniport driver can accept from a protocol might depend on the availability of resources, such as the host memory that the miniport driver has allocated for such patterns, or the available storage in the network adapter. If a miniport driver cannot add a wake-up pattern due to insufficient resources, the miniport driver returns **NDIS_STATUS_RESOURCES** in response to OID_PNP_ADD_WAKE_UP_PATTERN.

If a protocol driver tries to add a duplicate pattern, the miniport driver should return **NDIS_STATUS_INVALID_DATA** in response to OID_PNP_ADD_WAKE_UP_PATTERN.

An intermediate driver in which the upper edge receives this OID request must always propagate the request to the underlying miniport driver by calling **NdisRequest** or **NdisCoRequest**.

Requirements

Version	Supported in NDIS 6.0 and NDIS 6.1. For NDIS 6.20 and later, use OID_PM_ADD_WOL_PATTERN instead.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PM_PACKET_PATTERN

OID_PM_ADD_WOL_PATTERN

OID_PNP_CAPABILITIES

Article • 02/18/2023

The OID_PNP_CAPABILITIES OID requests a miniport driver to return the wake-up capabilities of its network adapter or requests an intermediate driver to return the intermediate driver's wake-up capabilities. The wake-up capabilities are formatted as an **NDIS_PNP_CAPABILITIES** structure, which is defined as follows:

C++

```
    typedef struct _NDIS_PNP_CAPABILITIES {
         ULONG Flags;
         NDIS_PM_WAKE_UP_CAPABILITIES WakeUpCapabilities;
    } NDIS_PNP_CAPABILITIES, *PNDIS_PNP_CAPABILITIES;
```

The members of this structure contain the following information:

Flags

NDIS_DEVICE_WAKE_UP_ENABLE

NDIS sets this flag if the underlying miniport driver supports one or more wake-up capabilities. Protocol drivers can test this flag to determine whether an underlying miniport driver has wake-up capabilities. Miniport drivers should not access this flag.

WakeUpCapabilities

An **NDIS_PM_WAKE_UP_CAPABILITIES** structure that specifies the wake-up capabilities of the miniport driver's network adapter. The **NDIS_PM_WAKE_UP_CAPABILITIES** structure is defined as follows:

C++

```
typedef struct _NDIS_PM_WAKE_UP_CAPABILITIES {
         NDIS_DEVICE_POWER_STATE MinMagicPacketWakeUp;
         NDIS_DEVICE_POWER_STATE MinPatternWakeUp;
         NDIS_DEVICE_POWER_STATE MinLinkChangeWakeUp;
       } NDIS_PM_WAKE_UP_CAPABILITIES, *PNDIS_PM_WAKE_UP_CAPABILITIES;
```

The members of this structure contain the following information:

MinMagicPacketWakeUp

Specifies the lowest device power state from which the miniport driver's network adapter can signal a wake-up on receipt of a magic packet. (A *magic packet* is a packet that contains 16 contiguous copies of the receiving network adapter's Ethernet address.)

The device power state is specified as one of the following NDIS_DEVICE_POWER_STATE values:

NdisDeviceStateUnspecified
The network adapter does not support magic-packet wake-ups.

NdisDeviceStateD0
The network adapter can signal a magic-packet wake-up from device power state D0. Because D0 is the fully powered state, this does not cause a wake-up but can be used as a run-time event.

NdisDeviceStateD1
The network adapter can signal a magic-packet wake-up from device power states D1 and D0.

NdisDeviceStateD2
The network adapter can signal a magic-packet wake-up from device states D2, D1, and D0.

NdisDeviceStateD3
The network adapter can signal a magic-packet wake-up from device power states D3, D2, D1, and D0.

MinPatternWakeUp
Specifies the lowest device power state from which the miniport driver's network adapter can signal a wake-up event on receipt of a network frame that contains a pattern specified by the protocol driver. The power state is specified as one of the following NDIS_DEVICE_POWER_STATE values:

NdisDeviceStateUnspecified
The network adapter does not support pattern-match wake-ups.

NdisDeviceStateD0
The network adapter can signal a pattern-match wake-up from device power state D0. Because D0 is the fully powered state, this does not cause a wake-up but can be used as a run-time event.

NdisDeviceStateD1
The network adapter can signal a pattern-match wake-up from device power states D1 and D0.

NdisDeviceStateD2
The network adapter can signal a pattern-match wake-up from device power states D2, D1, and D0.

NdisDeviceStateD3
The network adapter can signal a pattern-match wake-up from device power states D3, D2, D1, and D0.

MinLinkChangeWakeUp
Reserved. NDIS ignores this member.

For Miniport Drivers

After the miniport driver completes initialization, both the protocol driver and NDIS can query the miniport driver with this OID to determine the following:

- Whether the miniport driver is PM-aware.
- The network adapter's capabilities of indicating network wake-up events.

If the miniport driver returns **NDIS_STATUS_SUCCESS** to a query of OID_PNP_CAPABILITIES, NDIS considers the miniport driver to be PM-aware. If the miniport driver returns **NDIS_STATUS_NOT_SUPPORTED**, NDIS considers the miniport driver to be a legacy miniport driver that is not PM-aware.

When calling **NdisMSetAttributesEx**, a miniport driver that does not support wake-up capabilities but that can save and restore its network adapter state across a power-state transition can set the **NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND** flag. Setting this flag prevents NDIS from calling the driver's *MiniportHalt* function before the system transitions to a low-power (sleeping) state. However, if the miniport driver returns **NDIS_STATUS_NOT_SUPPORTED** in response to a query OID_PNP_CAPABILITIES, NDIS ignores the **NDIS_ATTRIBUTE_NO_HALT_ON_SUSPEND** flag and halts the network adapter if the system goes into a low-power state.

A miniport driver's network adapter can support any combination of wake-up events, including no wake-up events. A miniport driver can still support power management even if its network adapter cannot not signal wake-up events. In this case, the only power management OIDs that the miniport driver supports in addition to OID_PNP_CAPABILITIES are OID_PNP_QUERY_POWER and OID_PNP_SET_POWER.

If a miniport driver's network adapter does not support a particular wake-up event, the miniport driver should indicate an **NDIS_DEVICE_POWER_STATE** value of **NdisDeviceStateUnspecified** for the wake-up event in the **NDIS_PM_WAKE_UP_CAPABILITIES** structure.

OID_PNP_CAPABILITIES only indicates the wake-up capabilities of a miniport driver' s network adapter; it does not enable such capabilities. OID_PNP_ENABLE_WAKE_UP is used to enable a network adapter's wake-up capabilities.

For Intermediate Drivers

If the underlying network adapter is PM-aware, the intermediate driver should return **NDIS_STATUS_SUCCESS** to a query of OID_PNP_CAPABILITIES. In the **NDIS_PM_WAKE_UP_CAPABILITIES** structure returned by this OID, the intermediate driver should specify a device power state of **NdisDeviceStateUnspecified** for each wake-up capability (**MinMagicPacketWakeUp** or **MinPatternWakeUp**). Such a response indicates that the intermediate driver is PM-aware but does not manage a physical device.

If the underlying network adapter is not PM-aware, the intermediate driver should return **NDIS_STATUS_NOT_SUPPORTED** to a query of OID_PNP_CAPABILITIES.

Note For information about how NDIS 6.20 and later miniport drivers report power management capabilities, see Reporting Power Management Capabilities.

Requirements

Version	Supported in NDIS 6.0 and NDIS 6.1. For NDIS 6.20 and later, use OID_PM_CURRENT_CAPABILITIES instead.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_DEVICE_POWER_STATE

NdisMSetAttributesEx

OID_PM_CURRENT_CAPABILITIES

OID_PNP_ENABLE_WAKE_UP

OID_PNP_QUERY_POWER

OID_PNP_SET_POWER

Reporting Power Management Capabilities

OID_PNP_ENABLE_WAKE_UP

Article • 02/18/2023

As a set, the OID_PNP_ENABLE_WAKE_UP OID specifies the wake-up capabilities that a miniport driver should enable in a network adapter.

As a query, OID_PNP_ENABLE_WAKE_UP obtains the current wake-up capabilities that are enabled for a network adapter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure is a bitmask of flags that can be used to enable a combination of wake-up events:

NDIS_PNP_WAKE_UP_MAGIC_PACKET
When set, specifies that the miniport driver should enable a network adapter to signal a wake-up event on receipt of a magic packet. (A *magic packet* is a packet that contains 16 contiguous copies of the receiving network adapter's Ethernet address.) When cleared, specifies that the miniport driver should disable the network adapter from signaling such a wake-up event.

NDIS_PNP_WAKE_UP_PATTERN_MATCH
When set, specifies that the miniport driver should enable a network adapter to signal a wake-up event on receipt of a packet that contains a pattern specified by the protocol with OID_PNP_ADD_WAKE_UP_PATTERN. When cleared, specifies that the miniport driver should disable the network adapter from signaling such a wake-up event.

NDIS_PNP_WAKE_UP_LINK_CHANGE
Reserved. NDIS ignores this flag.

A protocol driver uses the network adapter's wake-up capabilities in **NDIS_BIND_PARAMETERS** to enable the associated network adapter's wake-up capabilities. A protocol driver can also query this OID to determine which wake-up capabilities are enabled for a network adapter.

NDIS does not immediately enable the wake-up capabilities that a protocol driver specifies. Instead, NDIS keeps tracks of the wake-up capabilities that the protocol driver enabled and, just before the network adapter transitions to a low-power state, NDIS sends an OID_PNP_ENABLE_WAKE_UP set request to the miniport driver to enable the appropriate wake-up events.

Before the network adapter transitions to a low-power state (that is, before NDIS sends the miniport driver an OID_PNP_SET_POWER request), NDIS sends the miniport driver an OID_PNP_ENABLE_WAKE_UP request to enable the appropriate wake-up capabilities.

The miniport driver must take the appropriate device-dependent steps to enable or disable wake-up events on the network adapter.

The miniport driver should clear the wake-up capabilities that NDIS set with OID_PNP_ENABLE_WAKE_UP when the system is resumed. The wake-up capabilities should not be persisted across resumes. If wake-up capabilities are enabled, NDIS explicitly sets OID_PNP_ENABLE_WAKE_UP before the miniport transitions to the low-power state.

An intermediate driver in which the upper edge receives this OID request must always propagate the request to the underlying miniport driver by calling the **NdisOidRequest** or **NdisCoOidRequest** function.

Requirements

Version	Supported in NDIS 6.0 and 6.1. For NDIS 6.20 and later, use OID_PM_PARAMETERS instead).
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_OID_REQUEST

NdisCoOidRequest

NdisOidRequest

OID_PM_PARAMETERS

OID_PNP_ADD_WAKE_UP_PATTERN

OID_PNP_QUERY_POWER

Article • 02/18/2023

The OID_PNP_QUERY_POWER OID requests the miniport driver to indicate whether it can transition its network adapter to the low-power state specified in the *InformationBuffer*. The low-power state is specified as one of the following NDIS_DEVICE_POWER_STATE values:

NdisDeviceStateD1
This specifies a device state of D1.

NdisDeviceStateD2
This specifies a device state of D2.

NdisDeviceStateD3
This specifies a device state of D3.

An OID_PNP_QUERY_POWER request is not used to request a transition to a device state of D0. NDIS simply sends an OID_PNP_SET_POWER request that specifies a device state of D0.

By returning NDIS_STATUS_SUCCESS to this OID request, the miniport driver guarantees that it will transition the network adapter to the specified device power state on receipt of a subsequent OID_PNP_SET_POWER request. The miniport driver, in this case, must do nothing to jeopardize the transition.

Miniport drivers must always return NDIS_STATUS_SUCCESS to this OID request. Any other return code is an error.

An OID_PNP_QUERY_POWER request is always followed by an OID_PNP_SET_POWER request. The OID_PNP_SET_POWER request may immediately follow the OID_PNP_QUERY_POWER request or may arrive at an unspecified interval after the OID_PNP_QUERY_POWER request. A device state of D0 specified in the OID_PNP_SET_POWER request effectively cancels the OID_PNP_QUERY_POWER request.

An intermediate driver must always return NDIS_STATUS_SUCCESS to a query of OID_PNP_QUERY_POWER. An intermediate driver should never propagate an OID_PNP_QUERY_POWER request to an underlying miniport driver.

Requirements

Version	Supported for NDIS 5.1, and NDIS 6.0 and later.

Header	Ntddndis.h (include Ndis.h)

OID_PNP_REMOVE_WAKE_UP_PATTERN

Article • 02/18/2023

The OID_PNP_REMOVE_WAKE_UP_PATTERN OID requests the miniport driver to delete a wake-up pattern that it previously received in an OID_PNP_ADD_WAKE_UP_PATTERN request. The wake-up pattern, along with its mask, is described by an **NDIS_PM_PACKET_PATTERN** structure.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains the following:

- An **NDIS_PM_PACKET_PATTERN** structure that provides information about the pattern and its mask.
- A mask that indicates which bytes of an incoming packet should be compared with corresponding bytes in the pattern. The mask starts with the first byte of the packet. The mask immediately follows the **NDIS_PM_PACKET_PATTERN** structure in the **InformationBuffer**.
- A wake-up pattern, which begins **PatternOffset** bytes from the beginning of the **InformationBuffer**.

An intermediate driver in which the upper edge receives this OID request must always propagate the request to the underlying miniport driver by calling Ndis(Co)Request.

Requirements

Version	Supported in NDIS 6.0 and 6.1. For NDIS 6.20 and later, use OID_PM_REMOVE_WOL_PATTERN instead.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PM_PACKET_PATTERN

OID_PNP_ADD_WAKE_UP_PATTERN

OID_PM_REMOVE_WOL_PATTERN

OID_PNP_SET_POWER

Article • 02/18/2023

The OID_PNP_SET_POWER OID notifies a miniport driver that its underlying network adapter will be transitioning to the device power state specified in the *InformationBuffer*. The device power state is specified as one of the following NDIS_DEVICE_POWER_STATE values:

- **NdisDeviceStateD0**
- **NdisDeviceStateD1**
- **NdisDeviceStateD2**
- **NdisDeviceStateD3**

An OID_PNP_SET_POWER request may be preceded by an OID_PNP_QUERY_POWER request.

Starting with NDIS 6.30, NDIS will not pause and restart the NDIS drivers in the driver stack during power-state transitions if the following conditions are true:

- The underlying miniport driver sets the **NDIS_MINIPORT_ATTRIBUTES_NO_PAUSE_ON_SUSPEND** flag in the NDIS_MINIPORT_ADAPTER_REGISTRATION_ATTRIBUTES structure. The driver passes a pointer to this structure in its call to the NdisMSetMiniportAttributes function.

- All overlying filter drivers that are attached to the miniport driver support NDIS 6.30 or later versions of NDIS.

- All overlying protocol drivers that are bound to the miniport driver support NDIS 6.30 or later versions of NDIS.

Transitioning to a Low-Power State (D1-D3)

When the miniport driver handles a set request of OID_PNP_SET_POWER to transition to a low-power state, it must do the following:

- Fully prepare the network adapter for the indicated network device power state. The task that is performed by the miniport driver to accomplish this is device-dependent.

- Wait for calls to the NdisMIndicateReceiveNetBufferLists function to return.

- Wait for send requests processed by the network adapter to complete. Once completed, the miniport driver must call the NdisMSendNetBufferListsComplete function. The driver should set the **Status** member in each NET_BUFFER_LIST structure to the appropriate NDIS_STATUS_*Xxx* value.

- Complete all pending send requests by calling the NdisMSendNetBufferListsComplete function. The driver must set the **Status** member in each NET_BUFFER_LIST structure to **NDIS_STATUS_LOW_POWER_STATE**.

- Reject all new send requests made to its *MiniportSendNetBufferLists* function immediately by calling the NdisMSendNetBufferListsComplete function. The driver must set the **Status** member in each NET_BUFFER_LIST structure to **NDIS_STATUS_LOW_POWER_STATE**.

The miniport driver that supports NDIS 6.30 and later versions of NDIS must also do the following:

- Not wait for the completion of pending receive indications through calls to its *MiniportReturnNetBufferLists* function. Also, the miniport driver must not alter the NET_BUFFER_LIST structure or data for any packets that are waiting to be completed.

- Handle the OID_PNP_SET_POWER request to a low-power state from either the Paused or Running adapter states. For more information about these states, see Miniport Adapter States and Operations.

Before the network adapter transitions to the D3 state, the miniport driver must turn off everything under the miniport driver's control by performing the following tasks:

- Disable interrupts and the DMA engine on the network adapter.

- Stop the receive engine on the network adapter.

- Do not deallocate or modify receive descriptors and packet buffers that are associated with pending receive indications.

- Cancel all NDIS timers.

Note A miniport driver cannot access the network adapter after the bus driver has transitioned the network adapter to the D3 state.

Transitioning to the Full-Power State (D0)

When the miniport driver handles a set request of OID_PNP_SET_POWER to transition to a full-power state, it must restore the receive engine of the network adapter to the same state that the receive engine was in before the adapter was transitioned to the low-power state.

Note The miniport driver must not access or change any receive buffers that are associated with pending receive indications.

NDIS calls the miniport driver's *MiniportRestart* function after the transition to a full-power state only if NDIS called the driver's *MiniportPause* function before the transition to a low-power state.

Note An intermediate driver must always return **NDIS_STATUS_SUCCESS** to a query of OID_PNP_SET_POWER. An intermediate driver should never propagate an OID_PNP_SET_POWER request to an underlying miniport driver.

Return status codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the **NdisMOidRequestComplete** function, passing NDIS_STATUS_SUCCESS for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport driver is resetting.

Requirements

Version	Supported for NDIS 5.1, and NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

MiniportPause

MiniportRestart

MiniportReturnNetBufferLists

MiniportSendNetBufferLists

NDIS_DEVICE_POWER_STATE

NdisMIndicateReceiveNetBufferLists

NdisMSendNetBufferListsComplete

NET_BUFFER_LIST

OID_PNP_WAKE_UP_ERROR

Article • 02/18/2023

The optional OID_PNP_WAKE_UP_ERROR OID indicates the number of false wake-ups that are signaled by the miniport driver's network adapter. A false wake-up occurs when the network adapter wakes up the system when it shouldn't have. For example, the network adapter could erroneously wake up the system due to an inexact pattern match.

The data type for this OID is a ULONG value.

An intermediate driver in which the upper edge receives this OID request must always propagate the request to the underlying miniport driver by calling Ndis(Co)Request.

Requirements

Version	Supported for NDIS 5.1, and NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

OID_PNP_WAKE_UP_OK

Article • 02/18/2023

The optional OID_PNP_WAKE_UP_OK OID indicates the number of valid wake-ups that are signaled by the miniport driver's NIC. A valid wake-up occurs when the NIC wakes up the system in response to a valid pattern match or magic packet.

The data type for this OID is a ULONG value.

An intermediate driver in which the upper edge receives this OID request must always propagate the request to the underlying miniport driver by calling Ndis(Co)Request.

Requirements

Version	Supported for NDIS 5.1, and NDIS 6.0 and later.
Header	Ntddndis.h (include Ndis.h)

OID_PNP_WAKE_UP_PATTERN_LIST

Article • 02/18/2023

The OID_PNP_WAKE_UP_PATTERN_LIST OID is used by a protocol to query a list of the wake-up patterns currently set for the miniport driver's network adapter. A protocol specifies a wake-up pattern with OID_PNP_ADD_WAKE_UP_PATTERN.

OID_PNP_WAKE_UP_PATTERN_LIST is handled by NDIS rather than the miniport driver.

NDIS returns to the protocol a description of each wake-up pattern set in the miniport driver. Each wake-up pattern, along with its mask, is described by an **NDIS_PM_PACKET_PATTERN** structure.

For each wake-up pattern, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains the following:

- An **NDIS_PM_PACKET_PATTERN** structure that provides information about the pattern and its mask.
- A mask that indicates which bytes of an incoming packet should be compared with corresponding bytes in the pattern. The mask starts with the first byte of the packet. The mask immediately follows the **NDIS_PM_PACKET_PATTERN** structure in the **InformationBuffer**.
- A wake-up pattern, which begins **PatternOffset** bytes from the beginning of the **InformationBuffer**.

An intermediate driver in which the upper edge receives this OID request must always propagate the request to the underlying miniport driver by calling Ndis(Co)Request.

Requirements

Version	Supported in NDIS 6.0 and 6.1. For NDIS 6.20 and later, use OID_PM_WOL_PATTERN_LIST instead.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_PM_PACKET_PATTERN

NDIS_OID_REQUEST

OID_PM_WOL_PATTERN_LIST

OID_PNP_ADD_WAKE_UP_PATTERN

OID_PNP_REMOVE_WAKE_UP_PATTERN

OID_QOS_CURRENT_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_QOS_CURRENT_CAPABILITIES to obtain the currently enabled NDIS Quality of Service (QoS) hardware capabilities of a network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_CAPABILITIES structure.

Note This OID query request is handled by NDIS for miniport drivers that support the IEEE 802.1 Data Center Bridging (DCB) interface.

Remarks

Miniport drivers register the currently-enabled NDIS QoS hardware capabilities of a network adapter when its *MiniportInitializeEx* function is called. The driver registers these capabilities by following these steps:

1. The driver initializes an NDIS_QOS_CAPABILITIES structure with the enabled QoS hardware capabilities.
2. The driver sets the **CurrentQosCapabilities** member of the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure to a pointer to the NDIS_QOS_CAPABILITIES structure.
3. The miniport driver then calls the NdisMSetMiniportAttributes function and sets the *MiniportAttributes* parameter to a pointer to an NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

Note NDIS does not report the currently-enabled NDIS QoS hardware capabilities of a network adapter to overlying protocol and filter drivers during the bind or attach operations.

For more information on how to register NDIS QoS capabilities, see Registering NDIS QoS Capabilities.

Return Status Codes

NDIS handles the OID query request of OID_QOS_CURRENT_CAPABILITIES request for miniport drivers, and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_QOS_CAPABILITIES). NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NdisMSetMiniportAttributes

NDIS_OID_REQUEST

NDIS_QOS_CAPABILITIES

OID_QOS_HARDWARE_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_QOS_HARDWARE_CAPABILITIES to obtain the NDIS Quality of Service (QoS) hardware capabilities of a network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_CAPABILITIES structure.

Note This OID query request is handled by NDIS for miniport drivers that support the IEEE 802.1 Data Center Bridging (DCB) interface.

Remarks

The NDIS_QOS_CAPABILITIES structure contains information about the NDIS QoS hardware capabilities of a network adapter. These capabilities can include hardware capabilities that are currently disabled by INF file settings or through the **Advanced** properties page.

Note All the NDIS QoS hardware capabilities of a network adapter are returned through an OID query request of OID_QOS_HARDWARE_CAPABILITIES, regardless of whether a capability is enabled or disabled.

Miniport drivers registers the NDIS QoS hardware capabilities of a network adapter when its *MiniportInitializeEx* function is called. The driver registers these capabilities by following these steps:

1. The driver initializes an NDIS_QOS_CAPABILITIES structure with the NDIS QoS hardware capabilities.

2. The driver sets the **HardwareQosCapabilities** member of the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure to a pointer to the NDIS_QOS_CAPABILITIES structure.

3. The miniport driver then calls the NdisMSetMiniportAttributes function and sets the *MiniportAttributes* parameter to a pointer to an NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

Note NDIS does not report the NDIS QoS hardware capabilities of a network adapter to overlying protocol and filter drivers during the bind or attach operations.

For more information on how to register NDIS QoS capabilities, see Registering NDIS QoS Capabilities.

Return Status Codes

NDIS handles the OID query request of OID_QOS_HARDWARE_CAPABILITIES request for miniport drivers, and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_QOS_CAPABILITIES). NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NdisMSetMiniportAttributes

NDIS_OID_REQUEST

NDIS_QOS_CAPABILITIES

OID_QOS_OFFLOAD_CREATE_SQ

Article • 02/18/2023

Overlying drivers issue OID set requests of OID_QOS_OFFLOAD_CREATE_SQ to create a new Scheduler Queue (SQ) on the miniport adapter. The caller sets the **InformationBuffer** member of the NDIS_OID_REQUEST structure to contain a pointer to an NDIS_QOS_SQ_PARAMETERS structure. **NDIS_QOS_SQ_PARAMETERS** contains the parameters of the new SQ.

Remarks

NDIS_QOS_SQ_ID is a ULONG value that NDIS allocates and assigns for an SQ. This identifier is unique per miniport adapter. The value **NDIS_QOS_DEFAULT_SQ_ID** is not a valid SQ ID and means that no SQ is to be used.

Return Status Codes

NDIS handles the OID set request of OID_QOS_OFFLOAD_CREATE_SQ for miniport drivers and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_INVALID_PARAMETER	The length of the **InformationBuffer** is less than NDIS_SIZEOF_QOS_SQ_PARAMETERS_REVISION_1 or the **SqId** field of NDIS_QOS_SQ_PARAMETERS in the **InformationBuffer** is **NDIS_QOS_DEFAULT_SQ_ID**.
NDIS_STATUS_Xxx	The request failed for other reasons.

Requirements

Requirement	Value
Version	Supported in NDIS 6.85 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_QOS_SQ_PARAMETERS

NDIS_QOS_OFFLOAD_CAPABILITIES

OID_QOS_HARDWARE_CAPABILITIES

OID_QOS_OFFLOAD_UPDATE_SQ

OID_QOS_OFFLOAD_CURRENT_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an OID query request of OID_QOS_OFFLOAD_CURRENT_CAPABILITIES to obtain the currently enabled Quality of Service (QoS) offload hardware capabilities of a miniport adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_OFFLOAD_CAPABILITIES structure.

Remarks

The NDIS_QOS_OFFLOAD_CAPABILITIES structure specifies the hardware and current Hardware Quality of Service (QoS) offload capabilities of a miniport adapter.

Return Status Codes

NDIS handles the OID query request of OID_QOS_OFFLOAD_CURRENT_CAPABILITIES for miniport drivers and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_BUFFER_TOO_SHORT	The length of the information buffer is not sufficient for the returned data.
NDIS_STATUS_Xxx	The request failed for other reasons.

Requirements

Requirement	Value
Version	Supported in NDIS 6.85 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_QOS_OFFLOAD_CAPABILITIES

OID_QOS_HARDWARE_CAPABILITIES

OID_QOS_OFFLOAD_DELETE_SQ

Article • 02/18/2023

Overlying drivers issue OID set requests of OID_QOS_OFFLOAD_DELETE_SQ to delete a Scheduler Queue (SQ) on the miniport adapter. The caller should set the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure to contain a pointer to an **NDIS_QOS_SQ_ID**.

Remarks

The caller must ensure there are no resources actively referencing this SQ before deleting it.

Return Status Codes

NDIS handles the OID set request of OID_QOS_OFFLOAD_DELETE_SQ for miniport drivers and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_Xxx	The request failed for other reasons.

Requirements

Requirement	Value
Version	Supported in NDIS 6.85 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

OID_QOS_OFFLOAD_CREATE_SQ

OID_QOS_OFFLOAD_UPDATE_SQ

OID_QOS_OFFLOAD_ENUM_SQS

Article • 02/18/2023

Overlying drivers issue OID method requests of OID_QOS_OFFLOAD_ENUM_SQS to obtain a list of all Scheduler Queues (SQs), with their parameters, that are currently present on a miniport adapter.

After a successful return from the OID method request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_SQ_ARRAY structure. Each element of the array is an NDIS_QOS_SQ_PARAMETERS structure.

Remarks

Return Status Codes

NDIS handles the OID method request of OID_QOS_OFFLOAD_ENUM_SQS for miniport drivers and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_PARAMETER	The length of the **InformationBuffer** is less than NDIS_SIZEOF_QOS_SQ_ARRAY_REVISION_1.
NDIS_STATUS_BUFFER_TOO_SHORT	The length of the information buffer is not sufficient for the returned data.
NDIS_STATUS_Xxx	The request failed for other reasons.

Requirements

Requirement	Value
Version	Supported in NDIS 6.85 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_QOS_SQ_PARAMETERS

NDIS_QOS_SQ_ARRAY

OID_QOS_OFFLOAD_HARDWARE_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an OID query request of OID_QOS_OFFLOAD_HARDWARE_CAPABILITIES to obtain the Quality of Service (QoS) offload hardware capabilities of a miniport adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_OFFLOAD_CAPABILITIES structure.

Remarks

The NDIS_QOS_OFFLOAD_CAPABILITIES structure specifies the hardware and current Hardware Quality of Service (QoS) offload capabilities of a miniport adapter.

Return Status Codes

NDIS handles the OID query request of OID_QOS_OFFLOAD_HARDWARE_CAPABILITIES for miniport drivers and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_BUFFER_TOO_SHORT	The length of the information buffer is not sufficient for the returned data.
NDIS_STATUS_Xxx	The request failed for other reasons.

Requirements

Requirement	Value
Version	Supported in NDIS 6.85 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_QOS_OFFLOAD_CAPABILITIES

OID_QOS_OFFLOAD_CURRENT_CAPABILITIES

OID_QOS_OFFLOAD_SQ_STATS

Article • 02/18/2023

Overlying drivers issue OID method requests of OID_QOS_OFFLOAD_SQ_STATS to obtain a list of all Scheduler Queues (SQs), with their stat counters, that are currently present on a miniport adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_SQ_ARRAY structure. Each element of the array is an NDIS_QOS_SQ_STATS structure.

If the **NDIS_OID_REQUEST** buffer of the OID query contains a valid VPortId, then the returned stats are specific to the specified vPort. Otherwise, the stats specify the total stats across all vPorts associated with each SQ.

Remarks

Return Status Codes

NDIS handles the OID method request of OID_QOS_OFFLOAD_SQ_STATS for miniport drivers and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_PARAMETER	The length of the **InformationBuffer** is less than NDIS_SIZEOF_QOS_SQ_ARRAY_REVISION_1.
NDIS_STATUS_BUFFER_TOO_SHORT	The length of the information buffer is not sufficient for the returned data.
NDIS_STATUS_Xxx	The request failed for other reasons.

Requirements

Requirement	Value
Version	Supported in NDIS 6.85 and later.

Requirement	Value
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_QOS_SQ_PARAMETERS

NDIS_QOS_SQ_ARRAY

NDIS_QOS_SQ_STATS

OID_QOS_OFFLOAD_UPDATE_SQ

Article • 02/18/2023

Overlying drivers issue OID set requests of OID_QOS_OFFLOAD_UPDATE_SQ to update a Scheduler Queue (SQ) on the miniport adapter. The caller should set the **InformationBuffer** member of the NDIS_OID_REQUEST structure to contain a pointer to an NDIS_QOS_SQ_PARAMETERS structure.

The caller should set the **SqId** field of **NDIS_QOS_SQ_PARAMETERS** to the current SQ ID of the SQ it wants to update. The caller should set the rest of the fields to the full set of parameters it desires on this SQ, except the **SqType** field which cannot be updated.

Remarks

Return Status Codes

NDIS handles the OID set request of OID_QOS_OFFLOAD_UPDATE_SQ for miniport drivers and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_PARAMETER	The length of the **InformationBuffer** is less than NDIS_SIZEOF_QOS_SQ_PARAMETERS_REVISION_1.
NDIS_STATUS_Xxx	The request failed for other reasons.

Requirements

Requirement	Value
Version	Supported in NDIS 6.85 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_QOS_SQ_PARAMETERS

OID_QOS_OFFLOAD_CREATE_SQ

OID_QOS_OFFLOAD_DELETE_SQ

OID_QOS_OPERATIONAL_PARAMETERS

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_QOS_OPERATIONAL_PARAMETERS to obtain the current NDIS Quality of Service (QoS) operational parameters for a network adapter. The miniport driver configures the network adapter with the operational NDIS QoS parameters in order to perform QoS packet transmission.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_PARAMETERS structure.

Note This OID query request is handled by NDIS for miniport drivers that support the IEEE 802.1 Data Center Bridging (DCB) interface.

Remarks

When NDIS handles the OID query request of OID_QOS_OPERATIONAL_PARAMETERS successfully, it returns the operational NDIS QoS parameters that it had cached from the previous NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication that was issued by the miniport driver. The driver issues this status indication to report on the initial set of operational NDIS QoS parameters. The driver also issues this status indication whenever the operational NDIS QoS parameters change.

NDIS returns an NDIS_QOS_PARAMETERS structure that is initialized in the following way:

- If the miniport driver previously issued an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication, NDIS caches the NDIS_QOS_PARAMETERS data and returns this data for the OID query request of OID_QOS_OPERATIONAL_PARAMETERS.

- If the miniport driver did not issue an NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE status indication, NDIS returns an NDIS_QOS_PARAMETERS structure with all the members (with the exception of the **Header** member) set to zero.

For more information on operational NDIS QoS parameters, see Overview of NDIS QoS Parameters.

Return Status Codes

NDIS returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_QOS_PARAMETERS). NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NdisMOidRequestComplete

NDIS_OID_REQUEST

NDIS_QOS_CAPABILITIES

NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE

NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE

OID_QOS_PARAMETERS

OID_QOS_PARAMETERS

Article • 02/18/2023

The Data Center Bridging (DCB) component (Msdcb.sys) issues an object identifier (OID) method request of OID_QOS_PARAMETERS to configure the local NDIS Quality of Service (QoS) parameters on a network adapter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_QOS_PARAMETERS** structure.

Note This OID method request is mandatory for miniport drivers that support NDIS QoS for the IEEE 802.1 Data Center Bridging (DCB) interface.

Remarks

Miniport drivers obtain the local NDIS QoS parameters through an OID method request of OID_QOS_PARAMETERS. These parameters define how the network adapter prioritizes transmit, or *egress*, packets. For more information about these parameters, see Overview of NDIS QoS Parameters.

Note Only the DCB component can issue an OID method request of OID_QOS_PARAMETERS. An overlying protocol or filter driver must not issue this OID. For more information on the DCB component, see NDIS QoS Architecture for Data Center Bridging.

The DCB component issues an OID_QOS_PARAMETERS request under the following conditions:

- The system administrator installs or uninstalls the Microsoft DCB server feature.

  For more information about the DCB server feature, see System-Provided DCB Components.

- The system administrator enables or disables the DCB server feature while the feature is still installed.

- The system administrator changes any of the DCB server feature parameters.

- The operating system starts or restarts while the DCB server feature is installed.

When the miniport driver handles the OID method request of OID_QOS_PARAMETERS, it must follow these guidelines:

- The miniport driver copies the data within the NDIS_QOS_PARAMETERS structure to its cache of local NDIS QoS parameters. The driver then resolves its operational NDIS QoS parameters based on its cache of local NDIS QoS parameters and its cache of NDIS QoS parameters that it received from a remote peer.

  For more information about how the miniport driver resolves its operational parameters, see Resolving Operational NDIS QoS Parameters.

- The miniport driver must not modify any data that is contained within the NDIS_QOS_PARAMETERS structure. The driver must complete the OID method request and return the original data within the **NDIS_QOS_PARAMETERS** structure.

- The **NDIS_QOS_PARAMETERS_WILLING** flag specifies whether the miniport driver enables or disables the local Data Center Bridging Exchange (DCBX) Willing state. The driver handles this flag in the following way:

  - If this flag is set, the miniport driver must enable the local DCBX Willing state. This allows the driver to be remotely configured with QoS settings. In this case, the driver resolves its operational QoS parameters based on the remote QoS parameters. The miniport driver can also resolve its operational QoS parameters based on any proprietary QoS settings that are defined by the independent hardware vendor (IHV).

  - If this flag is not set, the miniport driver must disable the local DCBX Willing state. This allows the driver to resolve its operational QoS parameters from its local QoS parameters instead of remote QoS parameters. The miniport driver must also disable or override any local QoS parameter for which the related **NDIS_QOS_PARAMETERS_*Xxx*_CONFIGURED** flag is not set.

    For example, the miniport driver can override an unconfigured local QoS parameter with its proprietary settings for the QoS parameter that is defined by the IHV. If there are no proprietary settings for local QoS parameters that are not specified with an **NDIS_QOS_PARAMETERS_*Xxx*_CONFIGURED** flag, the driver must disable the use of these QoS parameters on the network adapter.

    Note The driver can also override configured local QoS parameters if they compromise the QoS parameters used by protocols or technologies that are enabled on the network adapter. For example, the driver can override the local QoS parameters if the network adapter is enabled for remote boot through the Fibre Channel over Ethernet (FCoE) protocol.

  For more information about the local DCBX Willing state, see Managing the Local DCBX Willing State.

For more information on how the miniport driver overrides local QoS parameters, see Managing NDIS QoS Parameters.

Note Overriding the local QoS parameters should not cause the miniport driver to fail the OID method request of OID_QOS_PARAMETERS.

For more information on how the miniport driver manages the local QoS parameters, see Setting Local NDIS QoS Parameters.

Return Status Codes

The miniport driver returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_PENDING	The OID request is pending completion. When the miniport driver calls **NdisMOidRequestComplete**, NDIS will pass the final status code and results to the OID request completion handler of the caller after the request is completed.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_PARAMETER	One or more members of the **NDIS_QOS_PARAMETERS** structure contain incorrect values.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than **sizeof**(**NDIS_QOS_PARAMETERS**). NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NdisMOidRequestComplete

NDIS_OID_REQUEST

NDIS_QOS_CAPABILITIES

NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE

NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE

OID_QOS_REMOTE_PARAMETERS

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_QOS_REMOTE_PARAMETERS to obtain the NDIS Quality of Service (QoS) parameters for a remote peer. The miniport driver uses these remote QoS parameters to resolve its operational NDIS QoS parameters. The driver configures the network adapter with the operational parameters in order to perform QoS packet transmission.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_QOS_PARAMETERS structure.

Note This OID query request is valid only for miniport drivers that support the IEEE 802.1 Data Center Bridging (DCB) interface.

Remarks

When NDIS handles the OID request of OID_QOS_REMOTE_PARAMETERS successfully, it returns the remote NDIS QoS parameters that it had cached from the previous NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication that was issued by the miniport driver. The driver issues this status indication to report on the initial set of remote NDIS QoS parameters. The driver also issues this status indication whenever the remote NDIS QoS parameters change.

NDIS returns an NDIS_QOS_PARAMETERS structure that is initialized in the following way:

- If the miniport driver previously issued an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication, NDIS caches the NDIS_QOS_PARAMETERS data and returns this data for the OID query request of OID_QOS_REMOTE_PARAMETERS.

- If the miniport driver did not issue an NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE status indication, NDIS returns an NDIS_QOS_PARAMETERS structure with all the members (with the exception of the **Header** member) set to zero.

For more information on remote NDIS QoS parameters, see Overview of NDIS QoS Parameters.

Return Status Codes

NDIS returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver does not support the NDIS QoS interface.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_QOS_PARAMETERS). NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NdisMOidRequestComplete

NDIS_OID_REQUEST

NDIS_QOS_CAPABILITIES

NDIS_STATUS_QOS_OPERATIONAL_PARAMETERS_CHANGE

NDIS_STATUS_QOS_REMOTE_PARAMETERS_CHANGE

OID_QOS_PARAMETERS

OID_RECEIVE_FILTER_ALLOCATE_QUEUE

Article • 02/18/2023

Overlying drivers issue object identifier (OID) method requests of OID_RECEIVE_FILTER_ALLOCATE_QUEUE to allocate a queue that has an initial set of configuration parameters.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_RECEIVE_QUEUE_PARAMETERS structure. After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure that has a new queue identifier.

Remarks

The OID method request of OID_RECEIVE_FILTER_ALLOCATE_QUEUE is optional for NDIS 6.20 and later miniport drivers. It is mandatory for miniport drivers that support the virtual machine queue (VMQ) interface.

The overlying driver initializes the NDIS_RECEIVE_QUEUE_PARAMETERS structure with its requested queue configuration. NDIS assigns a queue identifier in the **QueueId** member of the **NDIS_RECEIVE_QUEUE_PARAMETERS** structure and passes the method request to the miniport driver.

Note The overlying driver can set the **NDIS_RECEIVE_QUEUE_PARAMETERS_PER_QUEUE_RECEIVE_INDICATION** and **NDIS_RECEIVE_QUEUE_PARAMETERS_LOOKAHEAD_SPLIT_REQUIRED** flags in the **Flags** member of the NDIS_RECEIVE_QUEUE_PARAMETERS structure. The other flags are not used for queue allocation.

After a miniport driver is issued an OID request of OID_RECEIVE_FILTER_ALLOCATE_QUEUE and handles it successfully, the queue is in the Paused state.

The overlying driver must use the queue identifier that NDIS provides in subsequent OID requests, for example, to modify the queue parameters or free the queue. The queue identifier is also included in the out-of-band (OOB) data on all NET_BUFFER_LIST structures that are associated with the queue. Drivers use the NET_BUFFER_LIST_RECEIVE_QUEUE_ID macro to retrieve the queue identifier in a **NET_BUFFER_LIST** structure.

When NDIS receives an OID request to allocate a receive queue, it verifies the queue parameters. After NDIS allocates the necessary resources and the queue identifier, it submits the OID request to the underlying miniport driver. The queue identifier is unique to the associated network adapter.

If the miniport driver can successfully allocate the necessary software and hardware resources for the receive queue, it completes the OID request by returning **NDIS_STATUS_SUCCESS**.

The miniport driver must retain the queue identifiers for the allocated receive queues. NDIS uses the queue identifier of a receive queue for subsequent calls to the miniport driver in order to set a receive filter on the receive queue, change the receive queue parameters, or free the receive queue.

After an overlying driver allocates one or more receive queues and optionally sets the initial filters, it must issue OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE set OID requests to notify the miniport driver that the allocation is complete for the current batch of receive queues.

The miniport driver must not retain any packets in a receive queue if there are no filters set on that queue. If either a queue never had any filters set or all the filters were cleared, the queue should be empty and any packets should be discarded. That is, the packets are not indicated up the driver stack or retained in the queue.

Overlying drivers use OID requests of OID_RECEIVE_FILTER_FREE_QUEUE to free queues that they allocate.

Return Status Codes

Either NDIS or the miniport driver returns one of the following status codes for the OID method request of OID_RECEIVE_FILTER_ALLOCATE_QUEUE.

Status code	Description
NDIS_STATUS_SUCCESS	The queue was allocated successfully. The information buffer contains the updated NDIS_RECEIVE_QUEUE_PARAMETERS structure.
NDIS_STATUS_PENDING	The request is pending completion. The final status code and results will be passed to an OID request completion handler of the caller.
NDIS_STATUS_INVALID_PARAMETER	One or more of the parameters that the overlying driver provided were not valid.

Status code	Description
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_NOT_SUPPORTED	The NDIS version of the miniport driver is earlier than version 6.20.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NET_BUFFER_LIST

NET_BUFFER_LIST_RECEIVE_QUEUE_ID

OID_RECEIVE_FILTER_FREE_QUEUE

OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE

NDIS_RECEIVE_QUEUE_PARAMETERS

OID_RECEIVE_FILTER_CLEAR_FILTER

Article • 02/18/2023

Overlying drivers issue OID set requests of OID_RECEIVE_FILTER_CLEAR_FILTER to clear a receive filter on a network adapter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS** structure.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.
- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.
- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

The OID set request of OID_RECEIVE_FILTER_CLEAR_FILTER is mandatory for miniport drivers that support the NDIS packet coalescing, SR-IOV, or VMQ interface.

An overlying driver, such as an NDIS protocol or filter driver, uses the OID_RECEIVE_FILTER_CLEAR_FILTER set request to clear a previously set filter. Only the driver that set the receive filter can clear it.

The overlying driver clears a receive filter by setting the **FilterId** member of the **NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS** structure to the identifier for the filter. The driver obtained the filter identifier from an earlier OID method request of OID_RECEIVE_FILTER_SET_FILTER.

Additional Instructions for NDIS Packet Coalescing

The following point applies to miniport and overlying drivers that support NDIS packet coalescing:

- An overlying driver must clear all the receive filters that it set on the miniport driver before it unbinds or detaches from the driver.

Additional Guidelines for the SR-IOV Interface

The following points apply to miniport and overlying drivers that support the SR-IOV interface:

- An overlying driver must clear all the filters that it set on a SR-IOV VPort before it frees the VPort. The overlying driver must also clear all the filters that it set on the default VPort before it closes its binding to the network adapter.
- A miniport driver must not indicate packets on a nondefault VPort if it has completed the OID request of OID_RECEIVE_FILTER_CLEAR_FILTER to clear the last filter on the VPort.

  Note A miniport driver also must not indicate packets on a nondefault VPort if it has completed an OID request of OID_NIC_SWITCH_DELETE_VPORT to free the VPort.

Additional Guidelines for the VMQ Interface

The following points apply to miniport and overlying drivers that support the VMQ interface:

- An overlying driver must clear all the filters that it set on a VMQ receive queue before it frees the queue. The overlying driver must also clear all the filters that it set on the default or drop queues before it closes its binding to the network adapter.
- A miniport driver must not indicate packets on a receive queue if it has completed the OID request of OID_RECEIVE_FILTER_CLEAR_FILTER to clear the last filter on the receive queue.

  Note A miniport driver also must not indicate packets on a receive queue if it has completed an OID request of OID_RECEIVE_FILTER_FREE_QUEUE to free the receive queue.

Return status codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the NdisMOidRequestComplete function, passing **NDIS_STATUS_SUCCESS** for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport adapter has been surprise removed.

NDIS returns one of the following status codes for this request:

NDIS_STATUS_SUCCESS
The specified filter was cleared successfully.

NDIS_STATUS_PENDING
The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_FILE_NOT_FOUND
The filter identifier is not valid.

NDIS_STATUS_INVALID_LENGTH
The information buffer is too small. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_RECEIVE_FILTER_CLEAR_PARAMETERS

OID_NIC_SWITCH_DELETE_VPORT

OID_RECEIVE_FILTER_FREE_QUEUE

OID_RECEIVE_FILTER_SET_FILTER

OID_RECEIVE_FILTER_CURRENT_CAPABILITIES

Article • 02/18/2023

Overlying drivers issue OID query requests of OID_RECEIVE_FILTER_CURRENT_CAPABILITIES to obtain the currently enabled receive filtering capabilities of a network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.
- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.
- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

Starting with NDIS 6.20, miniport drivers register the currently enabled receive filtering hardware capabilities of the network adapter when its *MiniportInitializeEx* function is called. Miniport drivers register these capabilities by following these steps:

1. The driver initializes an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure with the currently enabled receive filtering hardware capabilities.
2. The driver initializes an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure and sets the **CurrentReceiveFilterCapabilities** member to a pointer to the **NDIS_RECEIVE_FILTER_CAPABILITIES** structure.
3. The miniport driver calls the **NdisMSetMiniportAttributes** function and sets the *MiniportAttributes* parameter to a pointer to an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

Overlying protocol and filter drivers do not have to issue OID query requests of OID_RECEIVE_FILTER_CURRENT_CAPABILITIES. NDIS provides the currently enabled receive filtering capabilities to these drivers in the following way:

- NDIS provides the currently enabled receive filtering capabilities of an underlying network adapter to overlying protocol drivers in the **ReceiveFilterCapabilities** member of the NDIS_BIND_PARAMETERS structure during the bind operation.
- NDIS provides the currently enabled receive filtering capabilities of an underlying network adapter to overlying filter drivers in the **ReceiveFilterCapabilities** member of the NDIS_FILTER_ATTACH_PARAMETERS structure during the attach operation.

Return status codes

NDIS handles the OID query request of OID_RECEIVE_FILTER_CURRENT_CAPABILITIES for miniport drivers, and returns one of the following status codes:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** points to an NDIS_RECEIVE_FILTER_CAPABILITIES structure.

NDIS_STATUS_PENDING
The request is pending completion. NDIS passes the final status code and results to the OID request completion handler of the caller after the request has completed.

NDIS_STATUS_INVALID_LENGTH
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_NOT_SUPPORTED
The network adapter does not support receive filtering.

NDIS_STATUS_FAILURE
The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NDIS_OID_REQUEST

NDIS_RECEIVE_FILTER_CAPABILITIES

OID_RECEIVE_FILTER_ENUM_FILTERS

Article • 02/18/2023

An overlying driver issues an OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS to obtain a list of all the filters that are configured on a network adapter.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_RECEIVE_FILTER_INFO_ARRAY structure.

After a successful return from the OID method request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer is formatted to contain the following:

- An NDIS_RECEIVE_FILTER_INFO_ARRAY structure that specifies a list of receive filters that are currently configured on a miniport driver.
- An array of NDIS_RECEIVE_FILTER_INFO structures. Each structure specifies the parameters of a receive filter that is currently configured on a miniport driver.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.
- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.
- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

Overlying drivers or applications issue OID method requests of OID_RECEIVE_FILTER_ENUM_FILTERS to enumerate the receive filters that were set on a network adapter. This includes receive filters that were set on an SR-IOV virtual port (VPort) or a VMQ receive queue.

Additional Guidelines for the NDIS Packet Coalescing Interface

Starting with Windows Server 2012, NDIS packet coalescing only supports the default receive queue of a network adapter.

To enumerate the packet coalescing receive filters, the overlying driver must set the **QueueId** member of the NDIS_RECEIVE_FILTER_INFO_ARRAY structure to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

Additional Guidelines for the SR-IOV Interface

Starting with Windows Server 2012, the SR-IOV interface only supports the default receive queue of a virtual port (VPort).

To enumerate the VPort receive filters, the overlying driver must set the **QueueId** member of the NDIS_RECEIVE_FILTER_INFO_ARRAY structure to NDIS_DEFAULT_RECEIVE_QUEUE_ID.

Additional Guidelines for the VMQ Interface

An overlying driver can issue OID method requests of OID_RECEIVE_FILTER_ENUM_FILTERS to enumerate the receive filters that were set on a VMQ receive queue. When the overlying driver initializes the NDIS_RECEIVE_FILTER_INFO_ARRAY structure, it sets the **QueueId** member to one of the following values:

- The queue identifier value for a nondefault receive queue. The overlying driver obtained the queue identifier input value from an earlier OID method request of OID_RECEIVE_FILTER_ALLOCATE_QUEUE or an OID query request of OID_RECEIVE_FILTER_ENUM_QUEUES.
- The queue identifier value of NDIS_DEFAULT_RECEIVE_QUEUE_ID, which specifies the default receive queue.

Return status codes

NDIS handles the OID method request of OID_RECEIVE_FILTER_ENUM_FILTERS for miniport drivers, and returns one of the following status codes:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** points to an NDIS_RECEIVE_FILTER_INFO_ARRAY structure.

NDIS_STATUS_PENDING
The request is pending completion. NDIS passes the final status code and results to the OID request completion handler of the caller after the request has completed.

NDIS_STATUS_INVALID_LENGTH
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_FAILURE
The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_RECEIVE_FILTER_INFO

NDIS_RECEIVE_FILTER_INFO_ARRAY

OID_RECEIVE_FILTER_ALLOCATE_QUEUE

OID_RECEIVE_FILTER_ENUM_QUEUES

OID_RECEIVE_FILTER_SET_FILTER

OID_RECEIVE_FILTER_ENUM_QUEUES

Article • 02/18/2023

Overlying drivers and user-mode applications issue object identifier (OID) query requests of OID_RECEIVE_FILTER_ENUM_QUEUES to obtain a list of all the receive queues that are allocated on a network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_RECEIVE_QUEUE_INFO_ARRAY structure that is followed by an NDIS_RECEIVE_QUEUE_INFO structure for each filter.

Remarks

NDIS obtained the information from an internal cache of the data that it received from the OID_RECEIVE_FILTER_ALLOCATE_QUEUE and OID_RECEIVE_FILTER_QUEUE_PARAMETERS OID requests.

Overlying drivers and user-mode applications issue OID query requests of OID_RECEIVE_FILTER_ENUM_QUEUES to enumerate the receive queues on a network adapter.

If a protocol driver issues the request, the request type inside the NDIS_OID_REQUEST structure is set to **NdisRequestQueryInformation** and this OID returns an array of all the receive queues that the protocol driver allocated on the network adapter. If a user-mode application issued the request, the request type inside the **NDIS_OID_REQUEST** structure is set to **NdisRequestQueryStatistics**, and this OID returns an array of information for all the receive queues on the network adapter.

Return Status Codes

NDIS handles the OID query request of OID_RECEIVE_FILTER_ENUM_QUEUES for miniport drivers, and returns one of the following status codes.

Status code	Description
NDIS_STATUS_SUCCESS	The request completed successfully. The **InformationBuffer** points to an NDIS_RECEIVE_QUEUE_INFO_ARRAY structure.

Status code	Description
NDIS_STATUS_PENDING	The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request has completed.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_RECEIVE_QUEUE_INFO

NDIS_RECEIVE_QUEUE_INFO_ARRAY

OID_RECEIVE_FILTER_ALLOCATE_QUEUE

OID_RECEIVE_FILTER_QUEUE_PARAMETERS

OID_RECEIVE_FILTER_FREE_QUEUE

Article • 02/18/2023

NDIS protocol drivers issue object identifier (OID) set requests of OID_RECEIVE_FILTER_FREE_QUEUE to free a receive queue.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_QUEUE_FREE_PARAMETERS** structure with a queue identifier of type **NDIS_RECEIVE_QUEUE_ID**.

Remarks

The OID set request of OID_RECEIVE_FILTER_FREE_QUEUE is optional for NDIS 6.20 and later miniport drivers. It is mandatory for miniport drivers that support the virtual machine queue interface.

After an overlying driver issues the OID_RECEIVE_FILTER_ALLOCATE_QUEUE OID to allocate a receive queue, it issues the OID_RECEIVE_FILTER_FREE_QUEUE OID to free the receive queue.

When NDIS requests a miniport driver to free a VMQ receive queue, it follows these steps:

1. The network adapter stops the DMA transfer of data to receive buffers that are associated with the receive queue, after which the queue must enter the DMA Stopped state. The network adapter probably stopped the DMA activity when it received the OID_RECEIVE_FILTER_CLEAR_FILTER OID request to clear the last set filter on the receive queue.

2. The miniport driver generates an **NDIS_STATUS_RECEIVE_QUEUE_STATE** status indication with the **QueueState** member of the **NDIS_RECEIVE_QUEUE_STATE** structure set to **NdisReceiveQueueOperationalStateDmaStopped** to notify NDIS that the DMA transfer has been stopped.

3. The miniport driver waits for all the indicated receive packets for that queue to be returned to the miniport driver.

4. The miniport driver frees all the shared memory that it allocated for the network adapter's receive buffers that are associated with the queue by calling **NdisFreeSharedMemory**.

5. The miniport driver completes the OID_RECEIVE_FILTER_FREE_QUEUE OID request to free the receive queue.

Miniport drivers call the **NdisFreeSharedMemory** function to free shared memory for a queue. If the miniport driver allocated the shared memory for a nondefault queue, the driver frees the shared memory in the context of the OID_RECEIVE_FILTER_FREE_QUEUE OID while it is freeing the queue. Miniport drivers free shared memory that they allocated for the default queue in the context of the *MiniportHaltEx* function.

An overlying driver must free all the filters that it set on a queue before it frees the queue. Also, an overlying driver must free all the receive queues that it allocated on a network adapter before it calls the **NdisCloseAdapterEx** function to close a binding to the network adapter. NDIS frees all the queues that are allocated on a network adapter before it calls the miniport driver's *MiniportHaltEx* function.

Return Status Codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the **NdisMOidRequestComplete** function, passing NDIS_STATUS_SUCCESS for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport driver is resetting.
NDIS_STATUS_REQUEST_ABORTED	The miniport driver stopped processing the request. For example, NDIS called the *MiniportResetEx* function.

NDIS returns one of the following status codes for this request:

Status code	Description
NDIS_STATUS_SUCCESS	The requested queue was freed successfully.

Status code	Description
NDIS_STATUS_PENDING	The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler for the caller after the request has completed.
NDIS_STATUS_INVALID_PARAMETER	The queue identifier is invalid.
NDIS_STATUS_INVALID_LENGTH	The information buffer is too short. NDIS sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportHaltEx

NDIS_OID_REQUEST

NDIS_RECEIVE_QUEUE_FREE_PARAMETERS

NDIS_STATUS_RECEIVE_QUEUE_STATE

NdisCloseAdapterEx

NdisFreeSharedMemory

OID_RECEIVE_FILTER_ALLOCATE_QUEUE

OID_RECEIVE_FILTER_GLOBAL_PARAMETERS

Article • 02/18/2023

Overlying drivers issue OID query requests of OID_RECEIVE_FILTER_GLOBAL_PARAMETERS to obtain the global receive filtering parameters of a network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_GLOBAL_PARAMETERS** structure.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.
- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.
- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

Starting with NDIS 6.20, protocol drivers use OID_RECEIVE_FILTER_GLOBAL_PARAMETERS to query the current global configuration parameters for receive filtering on a network adapter. For example, protocol drivers can use this OID to determine whether types of receive filters or receive queues are enabled or disabled.

Return status codes

NDIS handles the OID query request of OID_RECEIVE_FILTER_GLOBAL_PARAMETERS for miniport drivers, and returns one of the following status codes:

NDIS_STATUS_SUCCESS
The request completed successfully.

NDIS_STATUS_PENDING
The request is pending completion. NDIS passes the final status code and results to the

OID request completion handler of the caller after the request is complete.

NDIS_STATUS_INVALID_LENGTH
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_INVALID_PARAMETER
The request failed because it tried to enable a capability that the underlying network adapter does not support.

NDIS_STATUS_FAILURE
The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_RECEIVE_FILTER_GLOBAL_PARAMETERS

OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES

Article • 02/18/2023

Overlying drivers issue OID query requests of OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES to obtain the receive filtering hardware capabilities of a network adapter.

After a successful return from the OID query request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an**NDIS_RECEIVE_FILTER_CAPABILITIES** structure.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.
- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.
- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

The **NDIS_RECEIVE_FILTER_CAPABILITIES** structure contains information about the receive filtering hardware capabilities of a network adapter. These capabilities can include hardware capabilities that are currently disabled by INF file settings or through the **Advanced** properties page.

Note All the receive filtering hardware capabilities of a network adapter are returned through an OID query request of OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES, regardless of whether a capability is enabled or disabled.

Starting with NDIS 6.20, miniport drivers register the currently enabled receive filtering hardware capabilities of the network adapter when its *MiniportInitializeEx* function is called. Miniport drivers register these capabilities by following these steps:

1. The driver initializes an **NDIS_RECEIVE_FILTER_CAPABILITIES** structure with the receive filtering hardware capabilities.

2. The driver initializes an NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure and sets the **CurrentReceiveFilterCapabilities** member to a pointer to the NDIS_RECEIVE_FILTER_CAPABILITIES structure.

3. The miniport driver calls the NdisMSetMiniportAttributes function and sets the *MiniportAttributes* parameter to a pointer to an NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure.

Return status codes

NDIS handles the OID query request of OID_RECEIVE_FILTER_HARDWARE_CAPABILITIES for miniport drivers, and returns one of the following status codes:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** points to an NDIS_RECEIVE_FILTER_CAPABILITIES structure.

NDIS_STATUS_PENDING
The request is pending completion. NDIS passes the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_INVALID_LENGTH
The information buffer was too short. NDIS set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_NOT_SUPPORTED
The network adapter does not support receive filtering.

NDIS_STATUS_FAILURE
The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NDIS_OID_REQUEST

NDIS_RECEIVE_FILTER_CAPABILITIES

OID_RECEIVE_FILTER_MOVE_FILTER

Article • 02/18/2023

An overlying driver issues an object identifier (OID) set request of OID_RECEIVE_FILTER_MOVE_FILTER to move a previously configured receive filter. Receive filters are moved from one virtual port (VPort) to a different VPort.

Overlying drivers issue this OID set request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID set request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS structure.

Remarks

NDIS validates the members of the NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS structure before it forwards the OID set request to the PF miniport driver.

The PF miniport driver must handle this OID set request atomically. The driver must be able to configure the network adapter to simultaneously remove the filter from a receive queue and VPort and set it on a different receive queue and VPort.

For more information, see Moving a Receive Filter to a Virtual Port.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID set request of OID_RECEIVE_FILTER_MOVE_FILTER.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS structure have invalid values.

Status Code	Description
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is less than sizeof(NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS). The PF miniport driver must set the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_RECEIVE_FILTER_MOVE_FILTER_PARAMETERS

OID_RECEIVE_FILTER_PARAMETERS

Article • 02/18/2023

An overlying driver issues an OID method request of OID_RECEIVE_FILTER_PARAMETERS to obtain the current configuration parameters of a filter on a network adapter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. NDIS uses the **FilterId** member in the input structure to identify the filter.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer. This buffer is formatted to contain the following:

- An **NDIS_RECEIVE_FILTER_PARAMETERS** structure that specifies the parameters for an NDIS receive filter.
- An array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures that specifies the filter test criterion for a field in a network packet header.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.
- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.
- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

Overlying drivers issue OID method requests of OID_RECEIVE_FILTER_PARAMETERS to obtain the configuration parameters for a receive filter that was set on a network adapter. This includes a receive filter that was set on a VMQ receive queue or SR-IOV virtual port (VPort), as well as a packet coalescing filter that was downloaded to the miniport driver.

The overlying driver obtained the filter identifier from an earlier OID method request of OID_RECEIVE_FILTER_SET_FILTER or from OID requests of OID_RECEIVE_FILTER_ENUM_FILTERS.

Return status codes

NDIS handles the OID request of OID_RECEIVE_FILTER_PARAMETERS for miniport drivers, and returns one of the following status codes:

NDIS_STATUS_SUCCESS
The request completed successfully. The **InformationBuffer** points to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure.

NDIS_STATUS_PENDING
The request is pending completion. NDIS passes the final status code and results to the OID request completion handler of the caller after the request is complete.

NDIS_STATUS_INVALID_PARAMETER
The overlying driver or application provided an invalid filter identifier. A filter identifier is not valid if it is zero or if it specifies an undefined filter.

NDIS_STATUS_INVALID_LENGTH
The information buffer was too short. NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.

NDIS_STATUS_FAILURE
The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

OID_RECEIVE_FILTER_ENUM_FILTERS

NDIS_RECEIVE_FILTER_PARAMETERS

OID_RECEIVE_FILTER_SET_FILTER

OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE

Article • 02/18/2023

NDIS protocol drivers issue object identifier (OID) method requests of OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE to notify the miniport driver that an allocation has completed for the current batch of receive queues.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_RECEIVE_QUEUE_ALLOCATION_COMPLETE_ARRAY structure that is followed by an NDIS_RECEIVE_QUEUE_ALLOCATION_COMPLETE_PARAMETERS structure for each queue. After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to the same array of structures, and the **CompletionStatus** member of each **NDIS_RECEIVE_QUEUE_ALLOCATION_COMPLETE_PARAMETERS** structure contains the completion status for each queue.

Remarks

The OID method request of OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE is optional for NDIS 6.20 and later miniport drivers. It is mandatory for miniport drivers that support the virtual machine queue (VMQ) interface.

After allocating one or more receive queues and optionally setting the initial filters, the protocol driver must issue the OID method request of OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE in order to notify the miniport driver that the allocation has completed for the current batch of receive queues. This allows the miniport driver to balance the hardware resources among multiple receive queues; if necessary, it can allocate resources such as shared memory for the receive queues.

After a miniport driver receives an OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE request and it has filters that are set on the queue, the queue is in the Running state. In this state, the miniport driver can start indications of packets in the queue by calling NdisMIndicateReceiveNetBufferLists.

Return Status Codes

The miniport driver returns one of the following status codes for the OID method request of OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE.

Status code	Description
NDIS_STATUS_SUCCESS	The queue allocation has completed. The information buffer contains the updated NDIS_RECEIVE_QUEUE_ALLOCATION_COMPLETE_ARRAY structure and parameter structures with the completion status for the queue allocation.
NDIS_STATUS_PENDING	The request is pending completion. The final status code and results will be passed to the OID request completion handler of the caller.
NDIS_STATUS_INVALID_PARAMETER	One or more of the parameters that the overlying driver provided were not valid.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_NOT_SUPPORTED	The NDIS version of the miniport driver is earlier than version 6.20.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NdisMIndicateReceiveNetBufferLists

NDIS_OID_REQUEST

NDIS_RECEIVE_QUEUE_ALLOCATION_COMPLETE_ARRAY

NDIS_RECEIVE_QUEUE_ALLOCATION_COMPLETE_PARAMETERS

OID_RECEIVE_FILTER_QUEUE_PARAMETERS

Article • 02/18/2023

Overlying drivers issue object identifier (OID) method requests of OID_RECEIVE_FILTER_QUEUE_PARAMETERS to obtain the current configuration parameters of a receive queue. The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_RECEIVE_QUEUE_PARAMETERS structure with a queue identifier of type **NDIS_RECEIVE_QUEUE_ID**. After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_QUEUE_PARAMETERS** structure.

Overlying drivers issue OID set requests of OID_RECEIVE_FILTER_QUEUE_PARAMETERS to change the current configuration parameters of a queue. The overlying driver provides a pointer to an NDIS_RECEIVE_QUEUE_PARAMETERS structure in the **InformationBuffer** member of the NDIS_OID_REQUEST structure.

Remarks

Overlying drivers issue OID set requests of OID_RECEIVE_FILTER_QUEUE_PARAMETERS to change the parameters of one or more receive queues. The OID set request is optional for NDIS 6.20 and later miniport drivers. However, the OID request is mandatory for miniport drivers that support the virtual machine queue (VMQ) interface.

Note Only the overlying driver that allocated the queue can change the configuration parameters by issuing OID set requests of OID_RECEIVE_FILTER_QUEUE_PARAMETERS.

The overlying driver obtained the queue identifier input value from an earlier OID_RECEIVE_FILTER_ALLOCATE_QUEUE method OID request.

After the overlying driver allocates a queue, it can change the configuration parameters that have a corresponding change flag (NDIS_RECEIVE_QUEUE_PARAMETER_*Xxx*_CHANGED) in the **Flags** member of the NDIS_RECEIVE_QUEUE_PARAMETERS structure. However, after the queue has been allocated, the overlying driver cannot change the configuration parameters that do not have a corresponding change flag.

Return Status Codes

NDIS handles the OID method request of OID_RECEIVE_FILTER_QUEUE_PARAMETERS for miniport drivers, and returns one of the following status codes.

Status code	Description
NDIS_STATUS_SUCCESS	The request completed successfully.
NDIS_STATUS_PENDING	The request is pending completion. NDIS will pass the final status code and results to the OID request completion handler of the caller after the request has completed.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_INVALID_PARAMETER	The request failed because it tried to enable a capability that the underlying network adapter does not support.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_RECEIVE_QUEUE_PARAMETERS

OID_RECEIVE_FILTER_ALLOCATE_QUEUE

OID_RECEIVE_FILTER_QUEUE_PARAMETERS

OID_RECEIVE_FILTER_SET_FILTER

Article • 02/18/2023

An overlying driver issues an OID method request of OID_RECEIVE_FILTER_SET_FILTER to set a filter on a network adapter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An **NDIS_RECEIVE_FILTER_PARAMETERS** structure that specifies the parameters for an NDIS receive filter.

- An array of **NDIS_RECEIVE_FILTER_FIELD_PARAMETERS** structures that specifies the filter test criterion for a field in a network packet header.

After a successful return from the OID method request, the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_RECEIVE_FILTER_PARAMETERS** structure. If the overlying driver is creating a new receive filter, NDIS updates this structure with a new filter identifier.

Remarks

NDIS receive filters are used in the following NDIS interfaces:

- NDIS Packet Coalescing. For more information about how to use receive filters in this interface, see Managing Packet Coalescing Receive Filters.

- Single Root I/O Virtualization (SR-IOV). For more information about how to use receive filters in this interface, see Setting a Receive Filter on a Virtual Port.

- Virtual Machine Queue (VMQ). For more information about how to use receive filters in this interface, see Setting and Clearing VMQ Filters.

The OID method request of OID_RECEIVE_FILTER_SET_FILTER is mandatory for miniport drivers that support the NDIS packet coalescing, SR-IOV, or VMQ interface.

The overlying driver initializes the **NDIS_RECEIVE_FILTER_PARAMETERS** structure with its requested filter configuration. NDIS assigns a filter identifier in the **FilterId** member of the NDIS_RECEIVE_FILTER_PARAMETERS structure and passes the method request to the underlying miniport driver.

Each filter that is set on a receive queue has a unique filter identifier for a network adapter. That is, the filter identifiers are not duplicated on different queues that the network adapter manages. When NDIS receives an OID request to set a filter on a receive queue, it verifies the filter parameters. After NDIS allocates the necessary resources and the filter identifier, it submits the OID request to the underlying network adapter. If the network adapter can successfully allocate the necessary software and hardware resources for the filter, it completes the OID request with a return status of NDIS_STATUS_SUCCESS.

Note Starting with NDIS 6.30, packet coalescing receive filter are only supported on the default receive queue of the network adapter. This receive queue has an identifier of NDIS_DEFAULT_RECEIVE_QUEUE_ID.

The miniport driver must retain the filter identifiers for the allocated receive filters. NDIS uses the identifier of a filter in later OID requests to change the receive filter parameters or clear the receive filter.

After a miniport driver receives an OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE request and it has filters that are set on the queue, the queue is in the *Running* state. In this state, the miniport driver can start indications of packets in the queue by calling **NdisMIndicateReceiveNetBufferLists**.

Additional Guidelines for the SR-IOV Interface

The following points apply to miniport drivers that support the SR-IOV interface:

- For the SR-IOV interface, a receive queue is created on a default or nondefault virtual port (VPort).

  Note Starting with Windows Server 2012, the SR-IOV interface only supports the default receive queue of a VPort.

  After an SR-IOV VPort is allocated through an OID set request of OID_NIC_SWITCH_CREATE_VPORT, overlying drivers can set filters on the VPort with OID requests of OID_RECEIVE_FILTER_SET_FILTER.

  Note Only the overlying driver that allocated the VPort can set a filter on that VPort.

- Because the default VPort always exists, overlying drivers can always set a filter on the default VPort.

- When the VPort is created, no receive filters are set on it. In this case, the miniport driver must not indicate any receive packets on that VPort before the miniport driver receives an OID request of OID_RECEIVE_FILTER_SET_FILTER for the VPort.

After this OID request is issued, the miniport driver can indicate packets on that VPort.

Note If the miniport driver indicates packets on a VPort while it is processing an OID request of OID_RECEIVE_FILTER_SET_FILTER, it must complete the OID request and return an NDIS_STATUS_SUCCESS status code.

Additional Guidelines for the VMQ Interface

The following points apply to miniport drivers that support the VMQ interface:

- After a VMQ receive queue is allocated, overlying drivers can set filters on the receive queue with OID requests of OID_RECEIVE_FILTER_SET_FILTER.

  Note Only the protocol driver that allocated a receive queue can set a filter on that queue.

- Because the default queue always exists, overlying drivers can always set a filter on the default queue. If the network adapter supports a drop queue, overlying drivers can set a filter on the drop queue.

  Overlying drivers do not own the default or drop queues. Therefore, all protocol drivers that are bound to a network adapter use the default or drop queue.

- When the receive queue is created, no receive filters are set on it. In this case, the miniport driver must not indicate any receive packets on that receive queue before the miniport driver receives an OID request of OID_RECEIVE_FILTER_SET_FILTER for the receive queue. After this OID request is issued, the miniport driver can indicate packets on that receive queue.

  Note If the miniport driver indicates packets on a queue while it is processing an OID request of OID_RECEIVE_FILTER_SET_FILTER, it must complete the OID request and return an NDIS_STATUS_SUCCESS status code.

Return status codes

The miniport driver returns one of the following status codes for the OID method request of OID_RECEIVE_FILTER_SET_FILTER:

NDIS_STATUS_SUCCESS
The filter was set on the queue successfully. The information buffer contains the updated NDIS_RECEIVE_FILTER_PARAMETERS structure.

NDIS_STATUS_PENDING

The request is pending completion. The final status code and results will be passed to the OID request completion handler of the caller.

NDIS_STATUS_INVALID_PARAMETER

One or more of the parameters that the overlying driver provided was not valid.

NDIS_STATUS_INVALID_LENGTH

The information buffer was too short. NDIS sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.

NDIS_STATUS_NOT_SUPPORTED

The NDIS version of the miniport driver is an earlier version than 6.20.

NDIS_STATUS_FAILURE

The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.20 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NdisMIndicateReceiveNetBufferLists

NDIS_OID_REQUEST

NDIS_RECEIVE_FILTER_PARAMETERS

NET_BUFFER_LIST

NET_BUFFER_LIST_RECEIVE_FILTER_ID

OID_NIC_SWITCH_CREATE_VPORT

OID_RECEIVE_FILTER_CLEAR_FILTER

OID_RECEIVE_FILTER_QUEUE_ALLOCATION_COMPLETE

OID_SRIOV_BAR_RESOURCES

Article • 02/18/2023

NDIS issues an object identifier (OID) method request of OID_SRIOV_BAR_RESOURCES to determine the memory resources that were allocated to a PCI Express (PCIe) Base Address Register (BAR) of a PCIe Virtual Function (VF).

NDIS issues this OID method request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following structures:

- An NDIS_SRIOV_BAR_RESOURCES_INFO structure that specifies the VF and BAR for which the PF miniport driver returns resource information.

- A CM_PARTIAL_RESOURCE_DESCRIPTOR structure which follows the NDIS_SRIOV_BAR_RESOURCES_INFO structure. The **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure contains information about the memory resources that were allocated to the specified BAR.

Remarks

NDIS issues an OID method request of OID_SRIOV_BAR_RESOURCES to obtain the system physical address and length of the memory resources that were allocated to a VF BAR. Before it issues the OID method request, NDIS formats the NDIS_SRIOV_BAR_RESOURCES_INFO structure in the following way:

- NDIS sets the **VFId** member of the NDIS_SRIOV_BAR_RESOURCES_INFO structure to the identifier associated with the VF.

- NDIS sets the **BarIndex** member of the NDIS_SRIOV_BAR_RESOURCES_INFO structure to the BAR index for the specified VF. The BAR index is the offset of the register within the table of BARs in the PCI configuration space.

- NDIS sets the **BarResourcesOffset** member of the NDIS_SRIOV_BAR_RESOURCES_INFO structure to the offset, in units of bytes, from the beginning of the **NDIS_SRIOV_BAR_RESOURCES_INFO** structure to a CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

Note Overlying drivers, such as protocol or filter drivers, cannot issue OID method requests of OID_SRIOV_BAR_RESOURCES to the PF miniport driver.

When the PF miniport driver receives the OID method request, the driver returns the resources for the specified BAR by formatting the CM_PARTIAL_RESOURCE_DESCRIPTOR structure within the **InformationBuffer** member of the NDIS_OID_REQUEST structure. The driver formats the **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure with the system hardware resources associated with the BAR for the specified VF.

Note The driver must format the structure for a resource type of **CmResourceTypeMemory**.

Return Status Codes

The PF miniport driver returns one of the following status codes for the method request of OID_SRIOV_BAR_RESOURCES.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_BAR_RESOURCES_INFO structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer is less than (sizeof(NDIS_SRIOV_BAR_RESOURCES_INFO) + sizeof(CM_PARTIAL_RESOURCE_DESCRIPTOR). The PF miniport driver must set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

CM_PARTIAL_RESOURCE_DESCRIPTOR

NDIS_OID_REQUEST

NDIS_SRIOV_BAR_RESOURCES_INFO

OID_SRIOV_CURRENT_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_SRIOV_CURRENT_CAPABILITIES to obtain the current single root I/O virtualization (SR-IOV) capabilities of a network adapter.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to the NDIS_SRIOV_CAPABILITIES structure.

Remarks

Starting with NDIS 6.30, miniport drivers supply the enabled SR-IOV hardware capabilities on the network adapter when its *MiniportInitializeEx* function is called. The driver initializes an NDIS_SRIOV_CAPABILITIES structure with the currently enabled SR-IOV hardware capabilities and sets the **CurrentSriovCapabilities** member of the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure to a pointer to the **NDIS_SRIOV_CAPABILITIES** structure. The miniport driver then calls the NdisMSetMiniportAttributes function and sets the *MiniportAttributes* parameter to a pointer to an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

Overlying protocol and filter drivers do not have to issue OID query requests of OID_SRIOV_CURRENT_CAPABILITIES. NDIS provides the currently enabled SR-IOV capabilities of a network adapter to these drivers in the following way:

- NDIS reports the currently enabled SR-IOV capabilities of an underlying network adapter to overlying protocol drivers in the **SriovCapabilities** member of the NDIS_BIND_PARAMETERS structure during the bind operation.

- NDIS reports the currently enabled SR-IOV capabilities of an underlying network adapter to overlying filter drivers in the **SriovCapabilities** member of the NDIS_FILTER_ATTACH_PARAMETERS structure during the attach operation.

Return Status Codes

NDIS handles the OID query request of the OID_SRIOV_CURRENT_CAPABILITIES request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_SRIOV_CURRENT_CAPABILITIES request, it returns one of the following status codes:

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. The miniport driver must set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_FILTER_ATTACH_PARAMETERS

NDIS_OID_REQUEST

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NDIS_SRIOV_CAPABILITIES

NdisMSetMiniportAttributes

OID_SRIOV_HARDWARE_CAPABILITIES

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_SRIOV_HARDWARE_CAPABILITIES to obtain the single root I/O virtualization (SR-IOV) hardware capabilities of the network adapter.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to the NDIS_SRIOV_CAPABILITIES structure.

Remarks

The NDIS_SRIOV_CAPABILITIES structure contains information about the hardware capabilities of the network adapter, such as whether the adapter supports SR-IOV and whether the miniport driver is managing the adapter's PCI Express (PCIe) Physical Function (PF) or Virtual Function (VF). These capabilities can include the hardware capabilities that are currently disabled by the INF file settings or through the **Advanced** properties page.

Note All the SR-IOV capabilities of the network adapter are returned through an OID query request of OID_SRIOV_HARDWARE_CAPABILITIES, regardless of whether a capability is enabled or disabled.

Starting with NDIS 6.30, miniport drivers supply the SR-IOV hardware capabilities when its *MiniportInitializeEx* function is called. The driver initializes an NDIS_SRIOV_CAPABILITIES structure with the SR-IOV hardware capabilities and sets the **HardwareSriovCapabilities** member of the NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES structure to a pointer to the **NDIS_SRIOV_CAPABILITIES** structure. The miniport driver then calls the NdisMSetMiniportAttributes function and sets the *MiniportAttributes* parameter to a pointer to an **NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES** structure.

Return Status Codes

NDIS handles the OID query request of the OID_SRIOV_HARDWARE_CAPABILITIES request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_SRIOV_HARDWARE_CAPABILITIES request, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. The miniport driver must set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_BIND_PARAMETERS

NDIS_FILTER_ATTACH_PARAMETERS

NDIS_OID_REQUEST

NDIS_MINIPORT_ADAPTER_HARDWARE_ASSIST_ATTRIBUTES

NDIS_SRIOV_CAPABILITIES

NdisMSetMiniportAttributes

OID_SRIOV_PF_LUID

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_SRIOV_PF_LUID to receive the locally unique identifier (LUID) associated with the PCI Express (PCIe) Physical Function (PF) of the network adapter.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to the NDIS_SRIOV_PF_LUID_INFO structure.

Remarks

NDIS generates a LUID for the PF before NDIS calls the *MiniportInitializeEx* function of the miniport driver for the PF. This LUID is valid until NDIS calls the *MiniportHaltEx* function of the driver.

Note The value of the **Luid** member differs from the **NetLuid** member of the NDIS_MINIPORT_INIT_PARAMETERS structure. This structure is passed to the miniport driver through the *MiniportInitParameters* parameter of *MiniportInitializeEx*.

Return Status Codes

NDIS handles the OID query request of OID_SRIOV_PF_LUID request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_SRIOV_PF_LUID request, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. The miniport driver must set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

MiniportInitializeEx

NDIS_OID_REQUEST

NDIS_SRIOV_PF_LUID_INFO

OID_SRIOV_PROBED_BARS

Article • 02/18/2023

NDIS issues an object identifier (OID) query request of OID_SRIOV_PROBED_BARS to obtain the values of a network adapter's PCI Express (PCIe) Base Address Registers (BARs). This function returns the BAR values that were reported by the network adapter following a query performed by the PCI bus driver. This query determines the memory or I/O address space that is required by the network adapter.

NDIS issues OID query requests of OID_SRIOV_PROBED_BARS to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID query request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer is formatted to contain the following:

- An NDIS_SRIOV_PROBED_BARS_INFO structure that contains the parameters for a read operation on the PCI BARs of a network adapter.

- An array of ULONG values for each BAR of the PCIe network adapter. The maximum number of elements within this array is PCI_TYPE0_ADDRESSES.

Remarks

The PCI bus driver, which runs in the management operating system of the Hyper-V parent partition, queries the memory or I/O address space requirements of each PCI Base Address Register (BAR) of the network adapter. The PCI bus driver performs this query when it first detects the adapter on the bus.

Through this PCI BAR query, the PCI bus driver determines the following:

- Whether a PCI BAR is supported by the network adapter.

- If a BAR is supported, how much memory or I/O address space is required for the BAR.

The virtual PCI (VPCI) bus driver runs in the guest operating system of a Hyper-V child partition. When a PCI Express (PCIe) Virtual Function (VF) is attached to the child partition, the VPCI bus driver will expose a virtual network adapter for the VF (*VF network adapter*). Before it does this, the VPCI bus driver must perform a PCI BAR query to determine the required memory or address space that is required by the VF network adapter.

Because access to the PCI configuration space is a privileged operation, it can only be performed by components that run in the management operating system of a Hyper-V parent partition. When the VPCI bus driver queries the PCI BARs, NDIS issues an OID query request of OID_SRIOV_PROBED_BARS to the PF miniport driver. The results returned by this OID query request are forwarded to the VPCI bus driver so that it can determine how much memory address space would be needed by the VF network adapter.

Note OID requests of OID_SRIOV_PROBED_BARS can only be issued by NDIS. The OID request must not be issued by overlying drivers, such as protocol of filter drivers.

The OID_SRIOV_PROBED_BARS query request contains an NDIS_SRIOV_PROBED_BARS_INFO structure. When the PF miniport driver handles this OID, the driver must return the PCI BAR values within the array referenced by the **BaseRegisterValuesOffset** member of the **NDIS_SRIOV_PROBED_BARS_INFO** structure. For each offset within the array, the PF miniport driver must set the array element to the ULONG value of the BAR at the same offset within the physical adapter's PCI configuration space.

Each BAR value returned by the driver must be the same value that would follow a PCI BAR query as performed by the PCI driver that runs in the management operating system. The PF miniport driver can call NdisMQueryProbedBars to determine this information.

For more information about the BARs of a PCI device, see the *PCI Local Bus Specification*.

For more information on how to query PCI BAR registers for a VF, see the Querying the PCI Base Address Registers of a Virtual Function.

Return Status Codes

The PF miniport driver returns one of the following status codes for the query request of OID_SRIOV_PROBED_BARS:

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.

Status Code	Description
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_PROBED_BARS_INFO structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer is less than (sizeof(NDIS_SRIOV_PROBED_BARS_INFO) + PCI_TYPE0_ADDRESSES). The PF miniport driver must set the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SRIOV_PROBED_BARS_INFO

NdisMQueryProbedBars

OID_SRIOV_READ_VF_CONFIG_BLOCK

Article • 02/18/2023

An overlying driver issues an object identifier (OID) method request of OID_SRIOV_READ_VF_CONFIG_BLOCK to read data from a specified PCI Express (PCIe) Virtual Function (VF) configuration block.

Overlying drivers issue this OID method request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An NDIS_SRIOV_READ_VF_CONFIG_BLOCK_PARAMETERS structure that contains the offset, in units of bytes, from the beginning of this structure to a location within the buffer that contains the data that is read from the VF configuration block.

- Additional buffer space for the data to be read from the specified VF configuration block.

Remarks

A VF configuration block is used for backchannel communication between the PF and VF miniport drivers. The IHV can define one or more VF configuration blocks for the miniport drivers. Each VF configuration block has an IHV-defined format, length, and block ID.

Note Data from each VF configuration block is used only by the PF and VF miniport drivers.

Before it issues the OID method request of OID_SRIOV_READ_VF_CONFIG_BLOCK, the overlying driver must set the members of NDIS_SRIOV_READ_VF_CONFIG_BLOCK_PARAMETERS structure in the following way:

- Set the **VFId** member to the identifier of the VF from which the information is to be read.

- Set the **BlockId** member to the identifier of the VF configuration block from which the information is to be read.

- Set the **Length** member to the number of bytes to read from the configuration block.

- Set the **BufferOffset** member to the offset within the buffer (referenced by **InformationBuffer** member) that will contain the data that is read from the specified VF configuration block. This offset is specified in units of bytes from the beginning of the NDIS_SRIOV_READ_VF_CONFIG_BLOCK_PARAMETERS structure.

When it handles the OID method request of OID_SRIOV_READ_VF_CONFIG_BLOCK, the PF miniport driver must follow these guidelines:

- The PF miniport driver must verify that the VF, specified by the **VFId** member of the NDIS_SRIOV_READ_VF_CONFIG_BLOCK_PARAMETERS structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.

- The PF miniport driver must verify that the **BlockId** member of the NDIS_SRIOV_READ_VF_CONFIG_BLOCK_PARAMETERS structure specifies a valid VF configuration block. If not, the driver must fail the OID request.

For more information about backchannel communication within the single root I/O virtualization (SR-IOV) interface, see SR-IOV PF/VF Backchannel Communication.

Return Status Codes

The PF miniport driver returns one of the following status codes for the method request of OID_SRIOV_READ_VF_CONFIG_BLOCK.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_READ_VF_CONFIG_BLOCK_PARAMETERS structure have invalid values.

Status Code	Description
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. The miniport driver must set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SRIOV_READ_VF_CONFIG_BLOCK_PARAMETERS

OID_NIC_SWITCH_ALLOCATE_VF

OID_SRIOV_READ_VF_CONFIG_SPACE

OID_SRIOV_READ_VF_CONFIG_SPACE

Article • 02/18/2023

An overlying driver issues an object identifier (OID) method request of OID_SRIOV_READ_VF_CONFIG_SPACE to read data from the PCI Express (PCIe) configuration space for a specified PCIe Virtual Function (VF) on the network adapter.

After a successful return from this OID method request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS structure that contains the parameters for a read operation of the PCI configuration space of a VF.
- Additional buffer space for the data to be read from the PCI configuration space.

Remarks

The VF miniport driver runs in the guest operating system of a Hyper-V child partition. Because of this, the VF miniport driver cannot directly access hardware resources, such as the VF's PCI configuration space. Only the miniport driver for the PCIe Physical Function (PF) can access the PCI configuration space for a VF. The PF miniport driver runs in the management operating system of a Hyper-V parent partition and has privileged access to the VF resources.

In order to read the VF PCI configuration space, overlying drivers that run in the management operating system issue the OID method request of OID_SRIOV_READ_VF_CONFIG_SPACE to the PF miniport driver. This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

For example, the virtualization stack that runs in the management operating system issues the OID method request of OID_SRIOV_READ_VF_CONFIG_SPACE when the VF miniport driver calls NdisMGetBusData to read from its VF PCI configuration space.

When it handles the OID method request of OID_SRIOV_READ_VF_CONFIG_SPACE, the PF miniport driver must follow these guidelines:

- The miniport driver must verify that the VF, specified by the **VFId** member of the NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS structure, has resources that have been previously allocated. The miniport driver allocates resources for a VF through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.

- The miniport driver must verify that the buffer (referenced by the **InformationBuffer** member of the NDIS_OID_REQUEST structure) is large enough to return the requested PCIe configuration space data. If this is not true, the driver must fail the OID request.

- The miniport driver typically calls NdisMGetVirtualFunctionBusData to query the requested PCIe configuration space. However, the miniport driver can also return PCIe configuration space data for the VF that the driver has cached from previous read or write operations of the PCIe configuration space.

  Note If an independent hardware vendor (IHV) provides a virtual bus driver (VBD) as part of its SR-IOV driver package, its miniport driver must not call NdisMGetVirtualFunctionBusData. Instead, the driver must interface with the VBD through a private communication channel, and request that the VBD call *ReadVfConfigBlock*. This function is exposed from the GUID_VPCI_INTERFACE_STANDARD interface that is supported by the underlying virtual PCI (VPCI) bus driver.

If the PF miniport driver can successfully complete the OID request, the driver must copy the requested PCI configuration space data to the buffer referenced by the **InformationBuffer** member of the NDIS_OID_REQUEST structure. The driver copies the data to the buffer at the offset specified by **BufferOffset** member of NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS structure.

For more information, see Querying the PCI Configuration Data of a Virtual Function.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID method request of OID_SRIOV_READ_VF_CONFIG_SPACE.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.

Status Code	Description
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. The miniport driver must set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

GUID_VPCI_INTERFACE_STANDARD

NDIS_OID_REQUEST

NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS

NdisMGetBusData

NdisMGetVirtualFunctionBusData

OID_NIC_SWITCH_ALLOCATE_VF

ReadVfConfigBlock

OID_SRIOV_RESET_VF

Article • 02/18/2023

Overlying drivers issue an object identifier (OID) set request of OID_SRIOV_RESET_VF to reset a specified PCI Express (PCIe) Virtual Function (VF) on a network adapter that supports single root I/O virtualization. Overlying drivers issue this OID set request to the miniport driver of the PCI Express (PCIe) Physical Function (PF) of the network adapter.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SRIOV_RESET_VF_PARAMETERS** structure. The overlying driver specifies the identifier of the VF to be reset through the **VFId** member of this structure.

Remarks

A VF can be reset through a PCI Express (PCIe) Function Level Reset (FLR). Because the FLR request is a privileged operation, it can only be performed by the PF miniport driver that runs in the management operating system of a Hyper-V parent partition. Overlying drivers that run in the management operating system are notified of the FLR request and issue the OID set request of OID_SRIOV_RESET_VF to the PF miniport driver.

When it handles this OID request, the PF miniport driver must follow these guidelines:

- The PF miniport driver must verify that the VF, specified by the **VFId** member of the **NDIS_SRIOV_RESET_VF_PARAMETERS** structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.

- The reset operation must only affect the specified VF. The operation must not affect other VFs or the PF on the same network adapter.

For more information, see Resetting a Virtual Function.

Return Status Codes

The PF miniport driver returns one of the following status codes for the set request of OID_SRIOV_RESET_VF.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Status Code	Description
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_RESET_VF_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. The PF miniport driver must set the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SRIOV_RESET_VF_PARAMETERS

OID_NIC_SWITCH_ALLOCATE_VF

OID_SRIOV_SET_VF_POWER_STATE

Article • 02/18/2023

An overlying driver issues an object identifier (OID) set request of OID_SRIOV_SET_VF_POWER_STATE to change the power state of a specified PCI Express (PCIe) Virtual Function (VF) on the network adapter. Since changing the power state is a privileged operation, overlying drivers issue this OID set request to the miniport driver of the PCIe Physical Function (PF) on the network adapter. The PF miniport driver then sets the specified power state on the VF.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SRIOV_SET_VF_POWER_STATE_PARAMETERS** structure.

Remarks

When the PF miniport driver is issued this OID set request, it must follow these guidelines:

- The PF miniport driver must verify that the VF, specified by the **VFId** member of the **NDIS_SRIOV_SET_VF_POWER_STATE_PARAMETERS** structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If the specified VF is not in an allocated state, the driver must fail the OID request.
- The power state operation must only affect the specified VF. The operation must not affect other VFs or the PF on the same network adapter.

For more information, see Setting the Power State of a Virtual Function.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID set request of OID_SRIOV_SET_VF_POWER_STATE.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.

Status Code	Description
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_SET_VF_POWER_STATE_PARAMETERS structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. The PF miniport driver must set the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SRIOV_SET_VF_POWER_STATE_PARAMETERS

OID_NIC_SWITCH_ALLOCATE_VF

OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK

Article • 02/18/2023

NDIS issues an object identifier (OID) method request of OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK to notify the miniport driver of a PCI Express (PCIe) Virtual Function (VF) that data within one or more configuration blocks has changed. NDIS issues this OID when the miniport driver for a PCIe Physical Function (PF) calls NdisMInvalidateConfigBlock.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SRIOV_VF_INVALIDATE_CONFIG_BLOCK_INFO structure. This structure specifies one or more Virtual Function (VF) configuration blocks whose data has been changed (*invalidated*) by the PF miniport driver.

Remarks

A VF configuration block is used for backchannel communication between the PF and VF miniport drivers. The IHV can define one or more VF configuration blocks for the device. Each VF configuration block has an IHV-defined format, length, and block ID.

Note Data from each VF configuration block is used only by the PF and VF miniport drivers.

VF configuration data is exchanged between the following drivers:

- The VF driver, which runs in the guest operating system. This operating system runs within a Hyper-V child partition.
- The PF driver, which runs in the management operating system. This operating system runs within the Hyper-V parent partition.

In order to handle notifications of invalid VF configuration data, NDIS and the miniport drivers perform the following steps:

1. In the guest operating system, NDIS issues an I/O control request of IOCTL_VPCI_INVALIDATE_BLOCK request. When this IOCTL is completed, NDIS is notified that VF configuration data has changed.
2. In the management operating system, the following steps occur:

a. The PF miniport driver calls the NdisMInvalidateConfigBlock function to notify NDIS that VF configuration data has changed and is no longer valid. The driver sets the *BlockMask* parameter to a ULONGLONG bitmask that specifies which VF configuration blocks have changed. Each bit in the bitmask corresponds to a VF configuration block. If the bit is set to one, the data in the corresponding VF configuration block has changed.

b. NDIS signals the virtualization stack, which runs in the management operating system, about the change to VF configuration block data. The virtualization stack caches the *BlockMask* parameter data.

   Note Each time that the PF miniport driver calls NdisMInvalidateConfigBlock, the virtualization stack ORs the *BlockMask* parameter data with the current value in its cache.

c. The virtualization stack notifies the virtual PCI (VPCI) driver, which runs in the guest operating system, about the invalidation of VF configuration data. The virtualization stack sends the cached *BlockMask* parameter data to the VPCI driver.

3. In the Guest operating system, the following steps occur:

   a. The VPCI driver saves the cached *BlockMask* parameter data in the **BlockMask** member of the VPCI_INVALIDATE_BLOCK_OUTPUT structure that is associated with the IOCTL_VPCI_INVALIDATE_BLOCK request.

   b. The VPCI driver successfully completes the IOCTL_VPCI_INVALIDATE_BLOCK request. When this happens, NDIS issues an OID method request of OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK to the VF miniport driver. An NDIS_SRIOV_VF_INVALIDATE_CONFIG_BLOCK_INFO is passed along in the OID request. This structure contains the cached *BlockMask* parameter data.

      NDIS also issues another IOCTL_VPCI_INVALIDATE_BLOCK request to handle successive notifications of changes to VF configuration data.

   c. When the VF driver handles the OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK request, it reads data from the specified VF configuration blocks.

For more information about backchannel communication within the single root I/O virtualization (SR-IOV) interface, see SR-IOV PF/VF Backchannel Communication.

Return Status Codes

The miniport driver returns one of the following status codes for the OID method request of OID_SRIOV_VF_INVALIDATE_CONFIG_BLOCK.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_VF_INVALIDATE_CONFIG_BLOCK_INFO structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the size of the NDIS_SRIOV_VF_INVALIDATE_CONFIG_BLOCK_INFO structure.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

IOCTL_VPCI_INVALIDATE_BLOCK

NDIS_OID_REQUEST

NDIS_SRIOV_VF_INVALIDATE_CONFIG_BLOCK_INFO

NdisMInvalidateConfigBlock

OID_SRIOV_READ_VF_CONFIG_SPACE

VPCI_INVALIDATE_BLOCK_OUTPUT

OID_SRIOV_VF_SERIAL_NUMBER

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_SRIOV_VF_SERIAL_NUMBER to determine the serial number of the PCI Express (PCIe) Virtual Function (VF) network adapter. This virtual network adapter is exposed in the guest operating system of a Hyper-V child partition to which the VF is attached.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SRIOV_VF_SERIAL_NUMBER_INFO** structure.

Remarks

The overlying driver uses the serial number to map the VF network adapter to an instance of a VF on the physical network adapter. The serial number is generated by the virtualization stack before resources for the VF are allocated through an OID set request of OID_NIC_SWITCH_ALLOCATE_VF.

Return Status Codes

NDIS handles the OID query request of the OID_SRIOV_VF_SERIAL_NUMBER request for miniport drivers. The drivers will not be issued this OID request.

When NDIS handles the OID_SRIOV_VF_SERIAL_NUMBER request, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SRIOV_VF_SERIAL_NUMBER_INFO

OID_NIC_SWITCH_ALLOCATE_VF

OID_SRIOV_VF_VENDOR_DEVICE_ID

Article • 02/18/2023

An overlying driver issues an object identifier (OID) method request of OID_SRIOV_VF_VENDOR_DEVICE_ID to query the PCI Express (PCIe) device identifier (DeviceID) and vendor identifier (VendorID) for a PCI Express (PCIe) Virtual Function (VF) network adapter. This virtual network adapter is exposed in the Hyper-V child partition that is attached to the VF.

Overlying drivers issue this OID method request to the miniport driver of the PCI Express (PCIe) Physical Function (PF) of the network adapter. This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SRIOV_VF_VENDOR_DEVICE_ID_INFO structure.

Remarks

Before it issues this OID method request, the overlying driver must initialize an NDIS_SRIOV_VF_VENDOR_DEVICE_ID_INFO structure and must set the **VFId** member to the identifier of the VF from which the information is to be read.

When it handles this OID request, the PF miniport driver must verify that the specified VF has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.

For more information, see Querying the PCI Vendor and Device Identifiers for a Virtual Function.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID method request of OID_SRIOV_VF_VENDOR_DEVICE_ID.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Status Code	Description
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the NDIS_SRIOV_VF_VENDOR_DEVICE_ID_INFO structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SRIOV_VF_VENDOR_DEVICE_ID_INFO

OID_NIC_SWITCH_ALLOCATE_VF

OID_SRIOV_WRITE_VF_CONFIG_BLOCK

Article • 02/18/2023

An overlying driver issues an object identifier (OID) set request of OID_SRIOV_WRITE_VF_CONFIG_BLOCK to write data to a PCI Express (PCIe) Virtual Function (VF) configuration block.

Overlying drivers issue this OID set request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An NDIS_SRIOV_WRITE_VF_CONFIG_BLOCK_PARAMETERS structure that contains the offset, in units of bytes, from the beginning of this structure to a location within the buffer that contains the data that is written to the VF configuration block.
- Additional buffer space for the data to be written to the specified VF configuration block.

Remarks

A VF configuration block is used for backchannel communication between the PF and VF miniport drivers. The IHV can define one or more VF configuration blocks for the miniport drivers. Each VF configuration block has an IHV-defined format, length, and block ID.

Note Data from each VF configuration block is used only by the PF and VF miniport drivers.

Before it issues the OID set request of OID_SRIOV_WRITE_VF_CONFIG_BLOCK, the overlying driver must set the members of NDIS_SRIOV_WRITE_VF_CONFIG_BLOCK_PARAMETERS structure in the following way:

- Set the **VFId** member to the identifier of the VF for which the information is to be written.
- Set the **BlockId** member to the identifier of the configuration block from which the information is to be written.

- Set the **Length** member to the number of bytes to write to the VF configuration block.
- Set the **BufferOffset** member to the offset within the buffer (referenced by **InformationBuffer** member) that contains the data that is to be written from the specified VF configuration block. This offset is specified in units of bytes from the beginning of the **NDIS_SRIOV_WRITE_VF_CONFIG_BLOCK_PARAMETERS** structure.

When it handles the OID set request of OID_SRIOV_WRITE_VF_CONFIG_BLOCK, the PF miniport driver must follow these guidelines:

- The PF miniport driver must verify that the VF, specified by the **VFId** member of the **NDIS_SRIOV_WRITE_VF_CONFIG_BLOCK_PARAMETERS** structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF during an OID method request of OID_NIC_SWITCH_ALLOCATE_VF. If resources for the specified VF have not been allocated, the driver must fail the OID request.
- The PF miniport driver must verify that the **BlockId** member of the **NDIS_SRIOV_WRITE_VF_CONFIG_BLOCK_PARAMETERS** structure specifies a valid VF configuration block. If not, the driver must fail the OID request.

For more information about backchannel communication within the single root I/O virtualization (SR-IOV) interface, see SR-IOV PF/VF Backchannel Communication.

Return Status Codes

The miniport driver returns one of the following status codes for the OID set request of OID_SRIOV_WRITE_VF_CONFIG_BLOCK:

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the **NDIS_SRIOV_WRITE_VF_CONFIG_BLOCK_PARAMETERS** structure have invalid values.

Status Code	Description
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SRIOV_WRITE_VF_CONFIG_BLOCK_PARAMETERS

OID_NIC_SWITCH_ALLOCATE_VF

OID_SRIOV_READ_VF_CONFIG_SPACE

OID_SRIOV_WRITE_VF_CONFIG_SPACE

Article • 02/18/2023

An overlying driver issues an object identifier (OID) set request of OID_SRIOV_WRITE_VF_CONFIG_SPACE to write data to the PCI Express (PCIe) configuration space for a specified PCIe Virtual Function (VF) on the network adapter.

Overlying drivers issue this OID set request to the miniport driver for the network adapter's PCIe Physical Function (PF). This OID method request is required for PF miniport drivers that support the single root I/O virtualization (SR-IOV) interface.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a caller-allocated buffer. This buffer is formatted to contain the following:

- An NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS structure that contains the parameters for a write operation of the PCI configuration space of a VF.
- Additional buffer space that contains the data to be written to the PCI configuration space.

Remarks

The VF miniport driver runs in the guest operating system of a Hyper-V child partition. Because of this, the VF miniport driver cannot directly access hardware resources, such as the VF's PCI configuration space. Only the PF miniport driver, which runs in the management operating system of a Hyper-V parent partition, can access the PCI configuration space for a VF.

The overlying driver, such as the virtualization stack, issues the OID set request of OID_SRIOV_WRITE_VF_CONFIG_SPACE when the VF miniport driver calls NdisMSetBusData to write to its PCI configuration space.

When it handles the OID method request of OID_SRIOV_WRITE_VF_CONFIG_SPACE, the PF miniport driver must follow these guidelines:

- The PF miniport driver must verify that the VF, specified by the **VFId** member of the NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS structure, has resources that have been previously allocated. The PF miniport driver allocates resources for a VF through an OID method request of OID_NIC_SWITCH_ALLOCATE_VF.

If resources for the specified VF have not been allocated, the driver must fail the OID request.

- The PF miniport driver calls **NdisMSetVirtualFunctionBusData** to write to the requested PCI configuration space. However, the PF miniport driver can also return PCI configuration space data for the VF that the driver has cached from previous read or write operations of the PCI configuration space.

  Note If an independent hardware vendor (IHV) provides a virtual bus driver (VBD) as part of its SR-IOV driver package, its PF miniport driver must not call **NdisMSetVirtualFunctionBusData**. Instead, the driver must interface with the VBD through a private communication channel, and request that the VBD call *SetVirtualFunctionData*. This function is exposed from the GUID_VPCI_INTERFACE_STANDARD interface that is supported by the underlying virtual PCI (VPCI) bus driver.

If the PF miniport driver can successfully complete the OID request, the driver must copy the requested PCI configuration space data to the buffer referenced by the **InformationBuffer** member of the **NDIS_OID_REQUEST** structure. The driver copies the data to the buffer at the offset specified by **BufferOffset** member of the **NDIS_SRIOV_READ_VF_CONFIG_SPACE_PARAMETERS** structure.

For more information, see Setting the PCI Configuration Data of a Virtual Function.

Return Status Codes

The PF miniport driver returns one of the following status codes for the OID set request of OID_SRIOV_WRITE_VF_CONFIG_SPACE.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The PF miniport driver either does not support the single root I/O virtualization (SR-IOV) interface or is not enabled to use the interface.
NDIS_STATUS_INVALID_PARAMETER	One or more of the members of the **NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS** structure have invalid values.
NDIS_STATUS_INVALID_LENGTH	The information buffer was too short. NDIS sets the **DATA.SET_INFORMATION.BytesNeeded** member in the **NDIS_OID_REQUEST** structure to the minimum buffer size that is required.

Status Code	Description
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

GUID_VPCI_INTERFACE_STANDARD

NDIS_OID_REQUEST

NDIS_SRIOV_WRITE_VF_CONFIG_SPACE_PARAMETERS

NdisMSetBusData

NdisMSetVirtualFunctionBusData

OID_NIC_SWITCH_ALLOCATE_VF

OID_SRIOV_READ_VF_CONFIG_SPACE

SetVirtualFunctionData

OID_SWITCH_FEATURE_STATUS_QUERY

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) method request of OID_SWITCH_FEATURE_STATUS_QUERY to obtain custom status information from an extension about the extensible switch. This information is known as *feature status* information. The format of this information is defined by the independent software vendor (ISV).

After a successful return from this OID method request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_FEATURE_STATUS_PARAMETERS structure that specifies the parameters for the type of feature status information to be returned.
- An NDIS_SWITCH_FEATURE_STATUS_CUSTOM structure that contains the feature status information for the extensible switch.

Remarks

For guidelines on how to handle an OID set request of OID_SWITCH_FEATURE_STATUS_QUERY, see Managing Custom Switch Feature Status Information.

Return Status Codes

The extension returns one of the following status codes for the OID method request of OID_SWITCH_FEATURE_STATUS_QUERY.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Status Code	Description
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to return the feature status information as well as the NDIS_SWITCH_FEATURE_STATUS_CUSTOM and NDIS_SWITCH_FEATURE_STATUS_PARAMETERS structures. The underlying miniport edge of the extensible switch sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PROPERTY_TYPE

NDIS_SWITCH_FEATURE_STATUS_CUSTOM

NDIS_SWITCH_FEATURE_STATUS_PARAMETERS

OID_SWITCH_NIC_ARRAY

Article • 02/18/2023

A Hyper-V extensible switch extension issues an object identifier (OID) query request of OID_SWITCH_NIC_ARRAY to obtain an array. Each element in the array specifies the configuration parameters of a virtual network adapter that is associated with an extensible switch port.

If the OID query request is completed successfully, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_NIC_ARRAY structure that defines the number of elements in the array. This structure also specifies the offset to the first element in the array.
- An array of NDIS_SWITCH_NIC_PARAMETERS structures. Each of these structures contains information about a network adapter that is connected to an extensible switch port.

  Note If no network adapters are connected to extensible switch ports, the underlying miniport edge of the extensible switch sets the **NumElements** member of the NDIS_SWITCH_NIC_ARRAY structure to zero. In this case, no NDIS_SWITCH_NIC_PARAMETERS structures are returned.

Remarks

The OID_SWITCH_NIC_ARRAY OID must only be issued when the Hyper-V extensible switch has completed activation. Please see Querying the Hyper-V Extensible Switch Configuration for more details.

When the extension processes the returned NDIS_SWITCH_NIC_PARAMETERS structure, it must not assume that the various string members of the NDIS_SWITCH_PORT_PARAMETERS structure, such as **NicFriendlyName**, are NULL-terminated. The data types for these string members are type-defined by the IF_COUNTED_STRING structure. The driver must determine the string length from the value of the **Length** member of this structure.

Note If the string is null-terminated, the **Length** member must not include the terminating null character.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_NIC_ARRAY and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to return the NDIS_SWITCH_NIC_ARRAY and its array of NDIS_SWITCH_NIC_PARAMETERS elements. The underlying miniport edge of the extensible switch sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_ARRAY

NDIS_SWITCH_NIC_PARAMETERS

Querying the Hyper-V Extensible Switch Configuration

OID_SWITCH_NIC_CONNECT

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_CONNECT to notify underlying extensible switch extensions that a network connection between an extensible switch port and a network adapter is completely established. The protocol edge previously notified extensions that this connection is being established when it issued an OID set request of OID_SWITCH_NIC_CREATE.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_PARAMETERS** structure.

Remarks

The **PortId** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the extensible switch port for which the connect notification is being made. The extensible switch extension can obtain the parameter information for this port and other extensible switch ports in the following ways:

- By issuing OID query requests of OID_SWITCH_PORT_ARRAY. The extension issues this OID on *FilterAttach* only when OID_SWITCH_PARAMETERS returns an **NDIS_SWITCH_PARAMETERS** structure with **IsActive** set to TRUE. If **IsActive** is FALSE, the extension issues the OID when the **NetEventSwitchActivate** **NET_PNP_EVENT** is issued by the extension miniport adapter.

- By inspecting the various OID sets requests of OID_SWITCH_PORT_CREATE and OID_SWITCH_PORT_DELETE.

The **Index** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the index of a network adapter for which the connection notification is being made. The network adapter with the specified **Index** value is connected to the extensible switch port specified by the **PortId** member. For more information on these index values, see Network Adapter Index Values.

When it receives the OID set request of OID_SWITCH_NIC_CONNECT, the extension must follow these guidelines:

- When the OID_SWITCH_NIC_CONNECT request completes with NDIS_STATUS_SUCCESS, the network connection and extensible switch port are fully operational. The extension can generate or forward packet traffic to the port's network connection. The extension can also issue extensible switch OIDs or status indications that use the port as the source port. The extension can also call *ReferenceSwitchPort* to increment the extensible switch reference counter for the port.

- The extension must not modify the **NDIS_SWITCH_NIC_PARAMETERS** structure that is associated with the OID request.

- The extension must always call **NdisFOidRequest** to forward this OID request to underlying extensions. The extension must not complete the OID request itself.

- The extensible switch external network adapter can bind to one or more underlying physical adapters. For every physical network adapter that is bound to the external network adapter, the protocol edge of the extensible switch issues a separate OID set request of OID_SWITCH_NIC_CONNECT. Each OID set request specifies a different network adapter connection index value. For more information on these values, see Network Adapter Index Values.

  The extension must maintain the connection state for each underlying physical adapter that is bound to the external network adapter. For more information about the different configurations in which physical network adapters can be bound to the external network adapter, see Types of Physical Network Adapter Configurations.

Note The extension must not issue its own OID set requests of OID_SWITCH_NIC_CONNECT.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID set request of OID_SWITCH_NIC_CONNECT and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.

Header	Ntddndis.h (include Ndis.h)

See also

NdisFReturnNetBufferLists

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_PARAMETERS

NdisFOidRequest

OID_SWITCH_NIC_CREATE

OID_SWITCH_PORT_ARRAY

ReferenceSwitchPort

OID_SWITCH_NIC_CREATE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_CREATE to notify underlying extensible switch extensions that a new connection is being established between an extensible switch port and an external or virtual network adapter. After the connection is fully established, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_PARAMETERS** structure.

Remarks

The **PortId** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the extensible switch port for which the creation notification is being made. The extensible switch extension can obtain the parameter information for this and other ports on the extensible switch by issuing OID query requests of OID_SWITCH_PORT_ARRAY.

The **Index** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the index of a network adapter for which the creation notification is being made. The network adapter with the specified **Index** value is connected to the extensible switch port specified by the **PortId** member. For more information on these index values, see Network Adapter Index Values.

When it receives the OID set request of OID_SWITCH_NIC_CREATE, the extension must follow these guidelines:

- The extension must not modify the **NDIS_SWITCH_NIC_PARAMETERS** structure that is associated with the OID request.

- The OID_SWITCH_NIC_CREATE request only notifies the extension that a new extensible switch connection is being brought up and that packet traffic may soon begin to occur over the specified port. However, the extension cannot use the port until the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT. Until that OID is issued, the extension must not do the following:

  - Generate any packet traffic to the network adapter connection on the extensible switch port for which the OID_SWITCH_NIC_CREATE OID request was issued.

  - Forward or originate OID requests of OID_SWITCH_NIC_REQUEST to an underlying network adapter for which the OID_SWITCH_NIC_CREATE OID request was issued.
  - Forward or originate NDIS status indications of **NDIS_STATUS_SWITCH_NIC_STATUS** from an underlying network adapter for which the OID_SWITCH_NIC_CREATE OID request was issued.
  - Call *ReferenceSwitchNic* to increment the extensible switch reference counter for the specified network adapter connection on the extensible switch port.

  Note The extension may intercept send or receive packets for the specified port between the OID requests of OID_SWITCH_NIC_CREATE and OID_SWITCH_NIC_CONNECT. In this case, the extension should forward the send or receive packet requests instead of canceling them.

- The extension can veto the creation notification by returning NDIS_STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot satisfy its configured policies on the specified port, the extension should veto the creation notification.

  If the extension returns other NDIS_STATUS_*Xxx* status codes, the creation notification is also vetoed. However, returning status codes for transitory scenarios, such as returning NDIS_STATUS_RESOURCES, could result in a retry of the creation notification.

  If the extension does not veto the OID request, it should monitor the status when the request is completed. The extension should do this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

  Note The extension can only veto the OID request if the **Index** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies a network adapter index value of zero.

- If the extension does not veto the creation notification, it must call **NdisFOidRequest** to forward this OID request to underlying extensions in the extensible switch driver stack.

  Note The extension should monitor the completion status of this OID request. The extension does this to detect whether underlying extensions in the extensible switch driver stack have vetoed the creation notification.

- If the extension calls **NdisFOidRequest** to forward this OID request, the extension will not immediately receive any packet traffic to or from the extensible switch port. In addition, the extension cannot immediately inject send or receive traffic for the extensible switch port.
- The extension can only forward packet traffic to the extensible switch port after the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_NIC_CONNECT.

  Note In some situations, packet traffic may be forwarded by the extensible switch to the port before an OID set request of OID_SWITCH_NIC_CONNECT is issued.
- The extensible switch external network adapter can bind to one or more underlying physical adapters. For every physical network adapter that is bound to the external network adapter, the protocol edge of the extensible switch issues a separate OID set request of OID_SWITCH_NIC_CREATE. Each OID set request specifies a different network adapter connection index value. For more information on these index values, see Network Adapter Index Values.

  The extension must maintain the connection state for each underlying physical adapter. For more information about the different configurations in which physical network adapters can be bound to the external network adapter, see Types of Physical Network Adapter Configurations.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Note The extension must not issue its own OID set requests of OID_SWITCH_NIC_CREATE.

Return Status Codes

If the extension completes the OID set request of OID_SWITCH_NIC_CREATE, it returns one of the following status codes.

Status Code	**Description**
NDIS_STATUS_DATA_NOT_ACCEPTED	The extension vetoed the creation notification.
NDIS_STATUS_RESOURCES	The extension vetoed the creation notification due to a low resource condition.
NDIS_STATUS_*Xxx*	The extension vetoed the creation notification for other reasons.

Note If the extension completes the OID set request, it must not return NDIS_STATUS_SUCCESS.

If the extension does not complete the OID set request of OID_SWITCH_NIC_CREATE, the request is completed by the underlying miniport edge of the extensible switch. The underlying miniport edge returns the following status code for this OID set request:

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_PARAMETERS

NdisFOidRequest

OID_SWITCH_NIC_CONNECT

OID_SWITCH_PORT_ARRAY

ReferenceSwitchPort

OID_SWITCH_NIC_DELETE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_DELETE to the extensible switch driver stack. This OID request notifies underlying extensible switch extensions about the deletion of a connection between an extensible switch port and a network adapter. The protocol edge of the extensible switch previously notified extensions that this connection is being deleted when it issued an OID set request of OID_SWITCH_NIC_DISCONNECT.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_PARAMETERS** structure.

Remarks

The **PortId** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the port for which the deletion notification is being made. The extensible switch extension can obtain the parameter information for this and other ports on the extensible switch by issuing OID query requests of OID_SWITCH_PORT_ARRAY.

The **Index** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the index of a network adapter for which the deletion notification is being made. The network adapter with the specified **Index** value is connected to the extensible switch port specified by the **PortId** member. For more information on these index values, see Network Adapter Index Values.

Before the protocol edge of the extensible switch issues the OID_SWITCH_NIC_DELETE request, it guarantees that all pending send or receive packet requests for the specified network adapter connection have been completed. The protocol edge also guarantees that all pending OID requests for the adapter connection have been completed, and the extensible switch reference counters for the adapter connection have a zero value.

Note If the extension had incremented an extensible switch reference counter for the network adapter by calling *ReferenceSwitchNic*, the OID_SWITCH_NIC_DELETE request is not issued while the reference counter is nonzero. The extension decrements the extensible switch reference counter by calling *DereferenceSwitchNic*.

The extension must follow these guidelines for handling OID set requests of OID_SWITCH_NIC_DELETE:

- The extension must not modify the **NDIS_SWITCH_NIC_PARAMETERS** structure that is associated with the OID request.

- The extension must always forward this OID set request to underlying extensions. The extension must not complete the request.

- The extension must not issue its own OID set requests of OID_SWITCH_NIC_DELETE.

- The extensible switch external network adapter can bind to one or more underlying physical adapters. For every physical network adapter that is bound to the external network adapter, the protocol edge of the extensible switch issues a separate OID set request of OID_SWITCH_NIC_DELETE. Each OID set request specifies a different network adapter connection index value. For more information on these index values, see Network Adapter Index Values.

  The extension must maintain the connection state for each underlying physical adapter. For more information about the different configurations in which physical network adapters can be bound to the external network adapter, see Types of Physical Network Adapter Configurations.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_NIC_DELETE and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

DereferenceSwitchNic

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_PARAMETERS

OID_SWITCH_NIC_DISCONNECT

OID_SWITCH_PORT_ARRAY

ReferenceSwitchNic

OID_SWITCH_NIC_DISCONNECT

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_DISCONNECT to notify underlying extensible switch extensions that a connection between an extensible switch port and a network adapter is being torn down. After the connection is completely torn down, the protocol edge of the extensible switch will issue an OID set request of OID_SWITCH_NIC_DELETE.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_PARAMETERS** structure.

Remarks

The **Index** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the index of a network adapter for which the disconnect notification is being made. The network adapter with the specified **Index** value is connected to the extensible switch port specified by the **PortId** member. For more information on these index values, see Network Adapter Index Values.

The extension must follow these guidelines when it handles OID set requests of OID_SWITCH_NIC_DISCONNECT:

- The extension must not modify the **NDIS_SWITCH_NIC_PARAMETERS** structure that is associated with the OID request.

- The OID_SWITCH_NIC_DISCONNECT request only notifies the extension that the extensible switch connection is being torn down between the specified network adapter and extensible switch port. After the extension handles this OID request, it must not do the following:

  - Generate any packet traffic to the network adapter connection on the extensible switch port for which the OID_SWITCH_NIC_DISCONNECT OID request was issued.

  - Call *ReferenceSwitchNic* to increment the extensible switch reference counter for the specified network adapter connection on the extensible switch port.

  - Forward or originate OID requests of OID_SWITCH_NIC_REQUEST to an underlying network adapter for which the OID_SWITCH_NIC_DISCONNECT OID request was issued.

Note If the extension called *ReferenceSwitchNic* to increment the extensible switch reference counter before the OID_SWITCH_NIC_DISCONNECT is issued, the extension can still forward or originate OID requests.

  - Forward or originate NDIS status indications of **NDIS_STATUS_SWITCH_NIC_STATUS** from an underlying network adapter for which the OID_SWITCH_NIC_DISCONNECT OID request was issued.

    Note If the extension called *ReferenceSwitchNic* to increment the extensible switch reference counter before the OID_SWITCH_NIC_DISCONNECT is issued, the extension can still forward or originate NDIS status indications.

    Note If the extension previously called *ReferenceSwitchNic* to increment the extensible switch reference counter, it does not need to synchronize its calls to originate or forward OID requests or NDIS status indications with its code that manages Hyper-V extensible switch OID requests. After the extension increments the reference counter, the extensible switch interface will not issue an OID set request of OID_SWITCH_NIC_DELETE.

- The extension must always forward this OID set request to underlying extensions. The extension must not complete the request.

- The extensible switch external network adapter can bind to one or more underlying physical adapters. For every physical network adapter that is bound to the external network adapter, the protocol edge of the extensible switch issues a separate OID set request of OID_SWITCH_NIC_DISCONNECT. Each OID set request specifies a different network adapter connection index value. For more information on these index values, see Network Adapter Index Values.

  The extension must maintain the connection state for each underlying physical adapter. For more information about the different configurations in which physical network adapters can be bound to the external network adapter, see Types of Physical Network Adapter Configurations.

Note The extension must not issue its own OID set requests of OID_SWITCH_NIC_DISCONNECT.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_NIC_DISCONNECT and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_PARAMETERS

OID_SWITCH_NIC_DELETE

OID_SWITCH_PORT_ARRAY

ReferenceSwitchPort

OID_SWITCH_NIC_REQUEST

Article • 02/18/2023

An object identifier (OID) method request of OID_SWITCH_NIC_REQUEST is used to encapsulate and forward OID requests to the Hyper-V extensible switch external network adapter. This allows the encapsulated OID request to be delivered to the driver for the underlying physical network adapter that is bound to the external network adapter.

This OID request is also used to encapsulate OID requests that were issued to other network adapters that are connected to extensible switch ports. In this case, the encapsulated OID request is forwarded through the extensible switch driver stack for inspection by extensions.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SWITCH_NIC_OID_REQUEST structure. This structure specifies the forwarding information for the OID request. This structure also contains a pointer to the original **NDIS_OID_REQUEST** structure of the OID request that is being forwarded.

Remarks

When OID requests arrive at the Hyper-V extensible switch interface, it encapsulates them in order to forward them down the extensible switch control path. These OID requests include the following:

- Hardware offload OID requests, including requests for Internet Protocol security (IPsec), virtual machine queue (VMQ), and single root I/O virtualization (SR-IOV). These OID requests are issued by an overlying protocol or filter driver that runs in the management operating system of the Hyper-V parent partition.

  When these OID requests arrive at the extensible switch interface, the protocol edge of the extensible switch encapsulates the OID request within an NDIS_SWITCH_NIC_OID_REQUEST structure. The protocol edge sets the members of this structure in the following way:

  - The **DestinationPortId** and **DestinationNicIndex** members are set to the corresponding values for the external network adapter.

  - If the OID request was originated from a Hyper-V child partition, the **SourcePortId** and **SourceNicIndex** members are set to the corresponding values for the port and network adapter that are used by the partition. Otherwise, the **SourcePortId** and **SourceNicIndex** members are set to zero.

  Note The extension must retain the values of these members if it forwards or redirects the OID request.

  - The **OidRequest** member is set to a pointer to the **NDIS_OID_REQUEST** structure for the encapsulated OID request.

The protocol edge then issues the OID_SWITCH_NIC_REQUEST request to forward the encapsulated OID request down the extensible switch control path to the external network adapter.

An underlying forwarding extension can redirect encapsulated hardware offload OID requests to a physical network adapter that is bound to the external network adapter. For example, if the extension supports physical network adapters from an extensible switch team that are bound to the external network adapter, it can forward the OID_SWITCH_NIC_REQUEST request to a physical adapter in the load balancing failover (LBFO) team that supports the hardware offload. For more information on this procedure, see Managing Hardware Offload OID Requests to Physical Network Adapters.

For more information about extensible switch teams, see Types of Physical Network Adapter Configurations.

- Multicast OID requests, including OID_802_3_ADD_MULTICAST_ADDRESS and OID_802_3_DELETE_MULTICAST_ADDRESS. These OID requests are issued by overlying protocol and filter drivers that run in either the management operating system or the guest operating system of a Hyper-V child partition.

  When these OID requests arrive at the extensible switch interface, the protocol edge of the extensible switch encapsulates the OID request within an **NDIS_SWITCH_NIC_OID_REQUEST** structure. The protocol edge also sets the **SourcePortId** and **SourceNicIndex** members to the corresponding values for the port and network adapter from which the OID request originated. The protocol edge then issues the OID_SWITCH_NIC_REQUEST request to forward the encapsulated OID request down the extensible switch control path for inspection by underlying extensions.

  Note In this case, the protocol edge sets the **DestinationPortId** and **DestinationNicIndex** members to zero. This specifies that the encapsulated OID request is to be delivered to extensions in the control path.

Underlying forwarding extensions can inspect these encapsulated OID requests and retain the multicast address information that they specify. For example, the extension may need this information if it originates multicast packets that it forwards to an extensible switch port.

For more information, see Forwarding OID Requests from a Hyper-V Child Partition.

A forwarding extension can also issue an OID_SWITCH_NIC_REQUEST in order to forward encapsulated OID requests to a physical network adapter that is bound to the external network adapter. This allows the extension to originate its own OID request or redirect an existing OID request to a physical network adapter that is bound to the external network adapter. In order to do this, the extension must follow these steps:

1. The extension calls *ReferenceSwitchNic* to increment a reference counter for the index of the destination physical network adapter. This guarantees that the extensible switch interface will not delete the physical network adapter connection while its reference counter is nonzero.

   Note The extensible switch interface could disconnect the physical network adapter connection while its reference counter is nonzero. For more information, see Hyper-V Extensible Switch Port and Network Adapter States.

2. The extension encapsulates the OID request by initializing an **NDIS_SWITCH_NIC_OID_REQUEST** structure in the following way:

   - The **DestinationPortId** member must be set to the identifier of the extensible switch port to which the external network adapter is connected.

   - The **DestinationNicIndex** member must be set to the nonzero index value of the underlying physical network adapter.

   - If the extension is originating on behalf of a Hyper-V child partition, the **SourcePortId** and **SourceNicIndex** members are set to the corresponding values for the port and network adapter that are used by the partition. Otherwise, the **SourcePortId** and **SourceNicIndex** members are set to zero.

     For example, if the extension is managing hardware offload resources for a child partition, it must set the **SourcePortId** and **SourceNicIndex** members to specify which partition the encapsulated hardware offload OID request is for.

   - The **OidRequest** member must be set to a pointer to an initialized **NDIS_OID_REQUEST** structure for the encapsulated OID request.

3. The extension calls **NdisFOidRequest** to forward the OID request to the specified destination extensible switch port and network adapter.

4. When NDIS calls the *FilterOidRequestComplete* function, the extension calls *DereferenceSwitchNic* to clear the reference counter for the index of the destination physical network adapter.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_NIC_REQUEST and returns one of the following status codes.

Status Code	**Description**
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_*Xxx*	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_INDICATION

NDIS_SWITCH_NIC_OID_REQUEST

OID_SWITCH_NIC_RESTORE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_RESTORE to notify the extensible switch extension about run-time data that can be restored for an extensible switch port and its network adapter connection.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SWITCH_NIC_SAVE_STATE structure. This structure is allocated by the protocol edge of the extensible switch.

Remarks

When it receives the OID set request of OID_SWITCH_NIC_RESTORE, the extensible switch extension must first determine whether it owns the run-time data. The extension does this by comparing the value of the **ExtensionId** member of the NDIS_SWITCH_NIC_SAVE_STATE structure to the GUID value that the extension uses to identify itself.

If the extension owns the run-time data for an extensible switch port, it restores this data in the following way:

1. The extension copies the run-time data in the **SaveData** member to extension-allocated storage.

   Note The value of the **PortId** member of the NDIS_SWITCH_NIC_SAVE_STATE structure may be different from the **PortId** value at the time that the run-time data was saved. This can occur if run-time data was saved during a Live Migration from one host to another. However, the configuration of the extensible switch port is retained during the Live Migration. This enables the extension to restore the run-time data to the extensible switch port by using the new **PortId** value.

2. The extension completes the OID set request with NDIS_STATUS_SUCCESS.

If the extension does not own the specified run-time data, the extension calls NdisFOidRequest to forward this OID set request to underlying extensions in the extensible switch driver stack. In this case, the extension must not modify the NDIS_SWITCH_NIC_SAVE_STATE structure that is associated with the OID request.

If the OID_SWITCH_NIC_RESTORE set request is received by the miniport edge of the extensible switch, it completes the OID request with NDIS_STATUS_SUCCESS. This notifies the protocol edge of the extensible switch that no extension owns the run-time data.

For more information about how to restore run-time data, see Restoring Hyper-V Extensible Switch Run-Time Data.

Note If the extension fails the OID set request, the extensible switch will fail the entire restore operation. As a result, the extension should avoid failing the OID request if it is possible. For example, if the extension cannot allocate the resource necessary to restore the run-time data, it should fail the OID request if it cannot function properly without restoring the run-time data. However, if the extension can recover from the failure condition, it should not fail the OID set request.

Return Status Codes

If the extension completes the OID set request of OID_SWITCH_NIC_RESTORE, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_*Xxx*	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_SAVE_STATE

NdisFOidRequest

OID_SWITCH_NIC_RESTORE_COMPLETE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_RESTORE_COMPLETE to notify Hyper-V extensible switch extensions about the completion of the operation to restore run-time data. Through this operation, the extension restores its run-time data for a port and its associated network adapter connection.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_SAVE_STATE** structure. This structure is allocated by the protocol edge of the extensible switch.

Remarks

When it receives the OID set request of OID_SWITCH_NIC_RESTORE_COMPLETE, the extension must follow these guidelines:

- The extension must not modify the **NDIS_SWITCH_NIC_SAVE_STATE** structure that is associated with the OID request.
- The extension must call **NdisFOidRequest** to forward this OID set request to underlying extensions in the extensible switch driver stack. The extension must not fail the OID request.

OID set requests of OID_SWITCH_NIC_RESTORE_COMPLETE are ultimately handled by the underlying miniport edge of the extensible switch. After this OID method request has been received by the miniport edge, it completes the OID request with NDIS_STATUS_SUCCESS. This notifies the protocol edge of the extensible switch that all extensions in the extensible switch driver stack have completed the save operation.

For more information on how to save run-time data for an extensible switch port, see Saving Hyper-V Extensible Switch Run-Time Data.

Return Status Codes

If the extension completes the OID set request of OID_SWITCH_NIC_RESTORE_COMPLETE, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_SAVE_STATE

OID_SWITCH_NIC_SAVE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) method request of OID_SWITCH_NIC_SAVE during an operation to save run-time data for an extensible switch port and its network adapter connection. The extension returns this data so that run-time data can be saved and restored at a later time. After the run-time data is saved, it is restored through OID set requests of OID_SWITCH_NIC_RESTORE.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_SAVE_STATE** structure. This structure is allocated by the protocol edge of the extensible switch.

Remarks

When it receives the OID method request of OID_SWITCH_NIC_SAVE, the extensible switch extension saves run-time data by doing the following:

- The extension saves the data within the **NDIS_SWITCH_NIC_SAVE_STATE** structure starting from *SaveDataOffset* bytes from the start of the structure.

- If the *SaveDataSize* provided is not large enough to hold the required save data, the extension sets the method structure's *BytesNeeded* field to NDIS_SIZEOF_NDIS_SWITCH_NIC_SAVE_STATE_REVISION_1 plus the amount of buffer necessary to hold the save data, and completes the OID with NDIS_STATUS_BUFFER_TOO_SHORT. The OID will be reissued with the required size.

- The extension populates the *ExtensionId* and *ExtensionFriendlyName* fields with its own identifier and name, and completes the OID method request with NDIS_STATUS_SUCCESS. This causes the protocol edge of the extensible switch to issue another OID method request to allow the extension to either return more save data, or allow other extensions down the stack to save their own data.

Note If the extension does not have run-time data to save, it must call **NdisFOidRequest** to forward this OID method request to underlying extensions in the extensible switch driver stack. For more information about this procedure, see Filtering OID Requests in an NDIS Filter Driver.

The Hyper-V extensible switch populates the *Header*, *PortId*, *NicIdex*, *SaveDataSize* and *SaveDataOffset* fields of the structure before issuing the OID. The extension cannot modify these fields.

OID method requests of OID_SWITCH_NIC_SAVE are ultimately handled by the underlying miniport edge of the extensible switch. After this OID method request has been received by the miniport edge of the extensible switch, it completes the OID request with NDIS_STATUS_SUCCESS. This notifies the protocol edge of the extensible switch that all extensions in the extensible switch driver stack have been queried for run-time data. The protocol edge of the extensible switch then issues an OID set request of OID_SWITCH_NIC_SAVE_COMPLETE to complete the save operation.

For more information on how to save run-time data for an extensible switch port, see Saving Hyper-V Extensible Switch Run-Time Data.

Return Status Codes

The extensible switch extension returns one of the following status codes for the OID method request of OID_SWITCH_NIC_SAVE.

Status Code	Description
NDIS_STATUS_BUFFER_TOO_SHORT	The length of the information buffer is too small for the NDIS_SWITCH_NIC_SAVE_STATE and its associated run-time data The extensible switch extension must set the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_SUCCESS	The extension returns this status if it is returning run-time data to save.
NDIS_STATUS_*Xxx*	The request failed for other reasons.

The underlying miniport edge of the extensible switch returns the following status code for the OID method request of OID_SWITCH_NIC_SAVE.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_SAVE_STATE

NdisFOidRequest

OID_SWITCH_NIC_RESTORE

OID_SWITCH_NIC_SAVE_COMPLETE

OID_SWITCH_NIC_SAVE_COMPLETE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_SAVE_COMPLETE to notify Hyper-V extensible switch extensions about the completion of the operation to save run-time data. Through this operation, the extension saves run-time data for a port and its associated network adapter connection.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_SAVE_STATE** structure.

Remarks

When it receives the OID set request of OID_SWITCH_NIC_SAVE_COMPLETE, the extension must follow these guidelines:

- The extension must not modify the **NDIS_SWITCH_NIC_SAVE_STATE** structure that is associated with the OID request.
- The extension must call **NdisFOidRequest** to forward this OID set request to underlying extensions in the extensible switch driver stack. The extension must not fail the OID request.

OID set requests of OID_SWITCH_NIC_SAVE_COMPLETE are ultimately handled by the underlying miniport edge of the extensible switch. After this OID method request has been received by the miniport edge, it completes the OID request with NDIS_STATUS_SUCCESS. This notifies the protocol edge of the extensible switch that all extensions in the extensible switch driver stack have completed the save operation.

For more information on how to save run-time data for an extensible switch port, see Saving Hyper-V Extensible Switch Run-Time Data.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_NIC_SAVE_COMPLETE and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_SAVE_STATE

OID_SWITCH_NIC_UPDATED

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_NIC_UPDATED to the extensible switch driver stack. This OID request notifies underlying extensible switch extensions about the update of the parameters of a network adapter. The OID will only be issued for NICs that have already been connected, and have not yet begun the disconnect process. These run-time configuration changes can include **NicFriendlyName**, **NetCfgInstanceId**, **MTU**, **NumaNodeId**, **PermanentMacAddress**, **VMMacAddress**, **CurrentMacAddress**, and **VFAssigned**.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_NIC_PARAMETERS** structure.

Remarks

The **PortId** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the port for which the update notification is being made. The extensible switch extension can obtain the parameter information for this and other ports on the extensible switch by issuing OID query requests of OID_SWITCH_PORT_ARRAY.

The **Index** member of the **NDIS_SWITCH_NIC_PARAMETERS** structure specifies the index of a network adapter for which the update notification is being made. The network adapter with the specified **Index** value is connected to the extensible switch port specified by the **PortId** member. For more information on these index values, see Network Adapter Index Values.

The extension must follow these guidelines for handling OID set requests of OID_SWITCH_NIC_UPDATED:

- The extension must not modify the **NDIS_SWITCH_NIC_PARAMETERS** structure that is associated with the OID request.
- The extension must always forward this OID set request to underlying extensions. The extension must not complete the request.
- The extension must not issue its own OID set requests of OID_SWITCH_NIC_UPDATED.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_NIC_UPDATED and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

DereferenceSwitchNic

NDIS_OID_REQUEST

NDIS_SWITCH_NIC_PARAMETERS

OID_SWITCH_NIC_DISCONNECT

OID_SWITCH_PORT_ARRAY

ReferenceSwitchNic

OID_SWITCH_PARAMETERS

Article • 02/18/2023

A Hyper-V extensible switch extension issues an object identifier (OID) query request of OID_SWITCH_PARAMETERS to obtain the configuration data of the extensible switch.

If the OID query request completes successfully, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SWITCH_PARAMETERS structure.

Remarks

When the extension processes the returned NDIS_SWITCH_PARAMETERS structure, it must not assume that the various string members of the **NDIS_SWITCH_PARAMETERS** structure, such as **SwitchName**, are null-terminated. The data types for these string members are type-defined by the IF_COUNTED_STRING structure. The extension must determine the string length from the value of the **Length** member of this structure.

Note If the string is null-terminated, the **Length** member must not include the terminating null character.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_PARAMETERS and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to return the OID_SWITCH_PARAMETERS structure for an OID query request. The underlying miniport edge of the extensible switch sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PARAMETERS

NdisFOidRequest

OID_SWITCH_PORT_ARRAY

Article • 02/18/2023

A Hyper-V extensible switch extension issues an object identifier (OID) query request of OID_SWITCH_PORT_ARRAY to obtain an array. Each element in the array specifies the configuration parameters for an extensible switch port.

If the OID query request completes successfully, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PORT_ARRAY structure that defines the number of elements within the array.
- An array of NDIS_SWITCH_PORT_PARAMETERS structures. Each of these structures contains information about a port on the extensible switch.

  Note If no ports have been created on the extensible switch, the driver sets the **NumElements** member of the NDIS_SWITCH_PORT_ARRAY structure to zero and no NDIS_SWITCH_PORT_PARAMETERS structures are returned.

Remarks

The OID_SWITCH_PORT_ARRAY OID must only be issued when the Hyper-V extensible switch has completed activation. Please see Querying the Hyper-V Extensible Switch Configuration for more details.

When the extension handles the returned NDIS_SWITCH_PORT_PARAMETERS structure, it must not assume that the various string members of the **NDIS_SWITCH_PORT_PARAMETERS** structure, such as **PortName**, are null-terminated. The data types for these string members are type-defined by the IF_COUNTED_STRING structure. The driver must determine the string length from the value of the **Length** member of this structure.

Note If the string is null-terminated, the **Length** member must not include the terminating null character.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_PORT_ARRAY and returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to return the NDIS_SWITCH_PORT_ARRAY and its array of NDIS_SWITCH_PORT_PARAMETERS elements. The underlying miniport edge of the extensible switch sets the **DATA.QUERY_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_ARRAY

NDIS_SWITCH_PORT_PARAMETERS

Querying the Hyper-V Extensible Switch Configuration

OID_SWITCH_PORT_CREATE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_CREATE to notify extensible switch extensions about the creation of an extensible switch port.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SWITCH_PORT_PARAMETERS structure.

Remarks

The **PortId** member of the NDIS_SWITCH_PORT_PARAMETERS structure specifies the port for which the creation notification is being made.

The extensible switch extension must follow these guidelines for handling OID set requests of OID_SWITCH_PORT_CREATE:

- The extension must not modify the NDIS_SWITCH_PORT_PARAMETERS structure that is associated with the OID request.

- The extension can veto the creation notification by returning NDIS_STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot allocate resources to enforce its configured policies on the port, the driver should veto the creation notification.

  If the extension returns other NDIS_STATUS_*Xxx* error status codes, the creation notification is also vetoed. However, returning status codes for transitory scenarios, such as returning NDIS_STATUS_RESOURCES, could result in a retry of the creation notification.

  If the extension does not veto the OID request, it should monitor the status when the request is completed. The extension should do this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

  For more information on port policies, see Managing Hyper-V Extensible Switch Policies.

- If the extension calls **NdisFOidRequest** to forward this OID set request, the extension should monitor the completion status of this OID request. The extension does this to detect whether underlying extensions in the extensible switch driver stack have vetoed the port creation notification.

- After the OID request is forwarded and completes successfully, the extension can issue OIDs requests for the port, such as OID_SWITCH_PORT_PROPERTY_ENUM, until an OID request of OID_SWITCH_PORT_TEARDOWN is issued. This OID request notifies the extension that the port will begin the deletion process from the extensible switch.

- Extensions cannot forward packets to the specified port in the **NDIS_SWITCH_PORT_PARAMETERS** structure until an OID set request of OID_SWITCH_NIC_CONNECT is issued and is completed successfully.

Note Extensions must not issue OID set requests of OID_SWITCH_PORT_CREATE.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Return Status Codes

If the extension completes the OID set request of OID_SWITCH_PORT_CREATE, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_DATA_NOT_ACCEPTED	The extension vetoed the creation notification.
NDIS_STATUS_RESOURCES	The extension vetoed the creation notification due to a low resource condition.
NDIS_STATUS_*Xxx*	The extension vetoed the creation notification for other reasons.

Note If the extension completes the OID set request, it must not return NDIS_STATUS_SUCCESS.

If the extension does not complete the OID set request of OID_SWITCH_PORT_CREATE, the request is completed by the underlying miniport edge of the extensible switch. The underlying miniport edge returns the following status code for this OID set request.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_PARAMETERS

NdisFOidRequest

OID_SWITCH_NIC_CONNECT

OID_SWITCH_PORT_ARRAY

OID_SWITCH_PORT_PROPERTY_ENUM

OID_SWITCH_PORT_DELETE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_DELETE to notify extensible switch extensions about the deletion of an extensible switch port.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SWITCH_PORT_PARAMETERS structure.

Remarks

The **PortId** member of the NDIS_SWITCH_PORT_PARAMETERS structure specifies the extensible switch port for which the delete notification is being made.

If a network adapter is connected to the specified port, the protocol edge of the extensible switch will delete the connection before it deletes the port. In this case, the protocol edge will follow these steps before it deletes the port:

- The protocol edge issues an OID set request of OID_SWITCH_NIC_DISCONNECT to notify the extension that the connection between a network adapter and the extensible switch port is being deleted.

- After all pending packets for the specified extensible switch port have been canceled or completed, the protocol edge issues an OID set request of OID_SWITCH_NIC_DELETE to notify the extension that the connection between a network adapter and the extensible switch port has been deleted.

  At this point, the protocol edge can start to delete the port.

The protocol edge of the extensible switch follows these steps when it deletes an extensible switch port:

1. The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_TEARDOWN. This OID request notifies underlying extensible switch extensions about the start of the deletion process for an extensible switch port.

2. The protocol edge issues an OID set request of OID_SWITCH_PORT_DELETE after all OID requests to the extensible switch port have completed.

   Note If the extension had previously called *ReferenceSwitchPort* to increment the port's reference counter, it must call *DereferenceSwitchPort* before the protocol edge issues the OID_SWITCH_NIC_DELETE request.

The extension must follow these guidelines for handling OID set requests of OID_SWITCH_PORT_DELETE:

- The extension must not modify the **NDIS_SWITCH_PORT_PARAMETERS** structure that is associated with the OID request.

- The extension must always forward this OID set request to underlying extensions. The extension must not fail the request.

- After the OID_SWITCH_PORT_DELETE request is completed with NDIS_STATUS_SUCCESS, the extension will not receive any packets or OID requests for the deleted port. The extension cannot forward packets to the deleted port. The extension also cannot issue OID requests nor call the *ReferenceSwitchPort* function for the deleted port.

Note Extensible switch extensions must not issue OID set requests of OID_SWITCH_PORT_DELETE.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID set request of OID_SWITCH_PORT_DELETE and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

DereferenceSwitchPort

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_PARAMETERS

NdisFOidRequest

OID_SWITCH_NIC_DELETE

OID_SWITCH_PORT_ARRAY

ReferenceSwitchPort

OID_SWITCH_PORT_FEATURE_STATUS_QUERY

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) method request of OID_SWITCH_PORT_FEATURE_STATUS_QUERY to obtain custom status information from an extension about an extensible switch port. This information is known as *feature status* information. The format of this information is defined by the independent software vendor (ISV).

After a successful return from this OID method request, the **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PORT_FEATURE_STATUS_PARAMETERS structure that specifies the parameters for the type of feature status information to be returned.
- An NDIS_SWITCH_PORT_FEATURE_STATUS_CUSTOM structure that contains the feature status information for the extensible switch port.

Remarks

For guidelines on how to handle an OID set request of OID_SWITCH_PORT_FEATURE_STATUS_QUERY, see Managing Custom Port Feature Status Information.

Return Status Codes

The extension returns one of the following status codes for the OID method request of OID_SWITCH_PORT_FEATURE_STATUS_QUERY.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Status Code	Description
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to return the feature status information as well as the NDIS_SWITCH_PORT_FEATURE_STATUS_CUSTOM and NDIS_SWITCH_PORT_FEATURE_STATUS_PARAMETERS structures. The underlying miniport edge of the extensible switch sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_FEATURE_STATUS_CUSTOM

NDIS_SWITCH_PORT_FEATURE_STATUS_PARAMETERS

NDIS_SWITCH_PORT_PROPERTY_TYPE

OID_SWITCH_PORT_PROPERTY_ADD

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_PROPERTY_ADD to notify extensible switch extensions about the addition of a policy property for an extensible switch port.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PORT_PROPERTY_PARAMETERS structure that specifies the identification and type of parameters for a port policy.
- A property buffer that contains the parameters for a port policy. The property buffer contains a structure that is based on the **PropertyType** member of the NDIS_SWITCH_PORT_PROPERTY_PARAMETERS structure. For example, if the **PropertyType** member is set to **NdisSwitchPortPropertyTypeVlan**, the property buffer contains an NDIS_SWITCH_PORT_PROPERTY_VLAN structure.

Remarks

A forwarding extension can handle the OID set request of OID_SWITCH_PORT_PROPERTY_ADD. All other types of extensions must call NdisFOidRequest to forward the OID request to the next extension in the extensible switch driver stack.

The extension can veto the addition of the port property by returning NDIS_STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot allocate resources to enforce its configured policies on the port, it should veto the addition request.

Note If the extension returns other NDIS_STATUS_*Xxx* error status codes, the creation notification is also vetoed. However, returning status codes for transitory scenarios, such as returning NDIS_STATUS_RESOURCES, could result in a retry of the creation notification.

If the extension does not veto the OID request, it should monitor the status when the request is completed. The extension should do this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

For guidelines on how to handle an OID set request of OID_SWITCH_PORT_PROPERTY_ADD, see Managing Port Policies.

Return Status Codes

If the forwarding extension completes the OID set request of OID_SWITCH_PORT_PROPERTY_ADD, it returns one of the following status codes:

Status Code	Description
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to process the NDIS_SWITCH_PORT_PROPERTY_PARAMETERS structure and the data in the structure's property buffer. The extension sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_DATA_NOT_ACCEPTED	The forwarding extension has vetoed the port policy addition notification.
NDIS_STATUS_NOT_SUPPORTED	The forwarding extension does not support the port policy.
NDIS_STATUS_*Xxx*	The OID request failed for other reasons.

If the extension does not complete the OID set request of OID_SWITCH_PORT_PROPERTY_ADD, the request is completed by the underlying miniport edge of the extensible switch. The miniport edge returns the following status code:

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_PROPERTY_CUSTOM

NDIS_SWITCH_PORT_PROPERTY_PARAMETERS

NDIS_SWITCH_PORT_PROPERTY_VLAN

NdisFOidRequest

OID_SWITCH_PORT_PROPERTY_DELETE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_PROPERTY_DELETE to notify extensible switch extensions about the deletion of a policy property for an extensible switch port.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer that contains an NDIS_SWITCH_PORT_PROPERTY_DELETE_PARAMETERS structure.

Remarks

A forwarding extension can handle the OID set request of OID_SWITCH_PORT_PROPERTY_DELETE. All other types of extensions must call NdisFOidRequest to forward the OID request to the next extension in the extensible switch driver stack.

For guidelines on how to handle an OID set request of OID_SWITCH_PORT_PROPERTY_DELETE, see Managing Port Policies.

Return Status Codes

If the forwarding extension completes the OID set request of OID_SWITCH_PORT_PROPERTY_DELETE, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The forwarding extension does not support the port policy.
NDIS_STATUS_*Xxx*	The OID request failed for other reasons.

If the forwarding extension does not complete the OID set request of OID_SWITCH_PORT_PROPERTY_DELETE, the request is completed by the underlying miniport edge of the extensible switch. The miniport edge returns the following status code.

Status Code	Description

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_PROPERTY_CUSTOM

NDIS_SWITCH_PORT_PROPERTY_PARAMETERS

NDIS_SWITCH_PORT_PROPERTY_VLAN

NdisFOidRequest

OID_SWITCH_PORT_PROPERTY_ENUM

Article • 02/18/2023

The Hyper-V extensible switch extension issues an object identifier (OID) method request of OID_SWITCH_PORT_PROPERTY_ENUM to obtain an array. This array contains the provisioned port policies that match the specified criteria. Each element in the array specifies the properties of a policy for a specified extensible switch port.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PORT_PROPERTY_ENUM_PARAMETERS structure that specifies the parameters for the policy enumeration of a specified port.

- An array of NDIS_SWITCH_PORT_PROPERTY_ENUM_INFO structures. Each of these structures contains information about the properties of an extensible switch port policy.

  Note If the **NumProperties** member of the NDIS_SWITCH_PORT_PROPERTY_ENUM_PARAMETERS structure is set to zero, no NDIS_SWITCH_PORT_PROPERTY_ENUM_INFO structures are returned.

Remarks

Before it issues an OID method request of OID_SWITCH_PORT_PROPERTY_ENUM, the extensible switch extension must follow these guidelines:

- The extension can only issue the OID_SWITCH_PORT_PROPERTY_ENUM request after the protocol edge of the extensible switch issues an OID_SWITCH_PORT_CREATE request and before it issues an OID_SWITCH_PORT_TEARDOWN request.

- The extension must call *ReferenceSwitchPort* before it calls **NdisFOidRequest** to issue the OID_SWITCH_PORT_PROPERTY_ENUM request. This ensures that the specified port will not be deleted until after the OID request is completed.

  After the OID request is completed, the extension must call *DereferenceSwitchPort*. The extension must call this function regardless of whether the OID request was completed with NDIS_STATUS_SUCCESS.

The OID_SWITCH_PORT_PROPERTY_ENUM OID must only be issued when the Hyper-V extensible switch has completed activation. Please see Querying the Hyper-V Extensible

Switch Configuration for more details.

Note If the extension receives the OID method request of OID_SWITCH_PORT_PROPERTY_ENUM, it must not complete the OID request. Instead, it must call **NdisFOidRequest** to forward the OID request down the extensible switch driver stack.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_PORT_PROPERTY_ENUM and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

DereferenceSwitchPort

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_PROPERTY_ENUM_INFO

NDIS_SWITCH_PORT_PROPERTY_ENUM_PARAMETERS

NdisFOidRequest

Querying the Hyper-V Extensible Switch Configuration

ReferenceSwitchPort

OID_SWITCH_PORT_PROPERTY_UPDATE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_PROPERTY_UPDATE to notify extensible switch extensions about the update of a property for an extensible switch port policy.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PORT_PROPERTY_PARAMETERS structure that specifies the identification and type of a port property.
- A property buffer that contains the parameters for a port policy. The property buffer contains a structure that is based on the **PropertyType** member of the NDIS_SWITCH_PORT_PROPERTY_PARAMETERS structure. For example, if the **PropertyType** member is set to **NdisSwitchPortPropertyTypeVlan**, the property buffer contains an NDIS_SWITCH_PORT_PROPERTY_VLAN structure.

Remarks

A forwarding extension can handle the OID set request of OID_SWITCH_PORT_PROPERTY_UPDATE. All other types of extensions must call NdisFOidRequest to forward the OID request to the next extension in the extensible switch driver stack.

The extension can veto the update of the port property by returning NDIS_STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot allocate resources to enforce its updated policies on the port, it should veto the update request.

Note If the extension returns other NDIS_STATUS_*Xxx* error status codes, the update notification is also vetoed. However, returning status codes for transitory scenarios, such as returning NDIS_STATUS_RESOURCES, could result in a retry of the creation notification.

If the extension does not veto the OID request, it should monitor the status when the request is completed. The extension should do this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

For guidelines on how to handle an OID set request of OID_SWITCH_PORT_PROPERTY_UPDATE, see Managing Port Policies.

Return Status Codes

If the forwarding extension completes the OID set request of OID_SWITCH_PORT_PROPERTY_UPDATE, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to process the NDIS_SWITCH_PORT_PROPERTY_PARAMETERS structure and the data in the structure's property buffer. The extension sets the **DATA.SET_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_DATA_NOT_ACCEPTED	The forwarding extension has vetoed the port policy deletion notification.
NDIS_STATUS_NOT_SUPPORTED	The forwarding extension does not support the port policy.
NDIS_STATUS_*Xxx*	The OID request failed for other reasons.

If the extension does not complete the OID set request of OID_SWITCH_PORT_PROPERTY_UPDATE, the request is completed by the underlying miniport edge of the extensible switch. The miniport edge returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_PROPERTY_CUSTOM

NDIS_SWITCH_PORT_PROPERTY_PARAMETERS

NDIS_SWITCH_PORT_PROPERTY_VLAN

NdisFOidRequest

OID_SWITCH_PORT_TEARDOWN

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_TEARDOWN to notify underlying extensible switch extensions that an extensible switch port will begin the deletion process. This process is started when the protocol driver issues an OID set request of OID_SWITCH_PORT_DELETE.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to an **NDIS_SWITCH_PORT_PARAMETERS** structure.

Remarks

The **PortId** member of the **NDIS_SWITCH_PORT_PARAMETERS** structure specifies the extensible switch port for which the connect notification is being made. The extensible switch extension must update any cached information about the port that it obtained in the following ways:

- By issuing OID query requests of OID_SWITCH_PORT_ARRAY. The extension issues this OID on *FilterAttach* only when OID_SWITCH_PARAMETERS returns an **NDIS_SWITCH_PARAMETERS** structure with **IsActive** set to TRUE. If **IsActive** is FALSE, the extension issues the OID when the **NetEventSwitchActivate** **NET_PNP_EVENT** is issued by the extension miniport.

- By inspecting the various OID sets requests of OID_SWITCH_PORT_CREATE and OID_SWITCH_PORT_DELETE.

The protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_TEARDOWN to notify the extension that a port is in the process of being deleted from the extensible switch. Before this OID request is issued, the protocol edge of the extensible switch had previously issued the following OIDs if the port had an active network adapter connection:

- OID_SWITCH_NIC_DISCONNECT, which notified underlying extensions that the network adapter is no longer connected to the port that is specified in the **NDIS_SWITCH_PORT_PARAMETERS** structure.

- OID_SWITCH_NIC_DELETE, which notified underlying extensions that the network connection between the network adapter and extensible switch port has been deleted.

The protocol edge issues this OID set request after all pending packets for the specified extensible switch port have been canceled or completed.

After the extension completes this OID set request and the reference counter for the extensible switch port is zero, the protocol edge of the extensible switch issues an OID set request of OID_SWITCH_PORT_DELETE. This OID request deletes the port from the extensible switch.

Note An extension increments the reference counter for an extensible switch port by calling *ReferenceSwitchPort*. An extension decrements the reference counter by calling *DereferenceSwitchPort*.

The extension must follow these guidelines for handling OID set requests of OID_SWITCH_PORT_TEARDOWN:

- The extension must always forward this OID set request to underlying extensions. The extension must not fail the request.

  Note The extension must not modify the **NDIS_SWITCH_PORT_PARAMETERS** structure that is associated with the OID request.

- After the extension forwards this OID request, it cannot forward packets to the deleted port. The extension also cannot issue OID requests nor call the *ReferenceSwitchPort* function for the deleted port.

Note The extension must not issue OID set requests of OID_SWITCH_PORT_TEARDOWN.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID set request of OID_SWITCH_PORT_TEARDOWN and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.

Header	Ntddndis.h (include Ndis.h)

See also

DereferenceSwitchPort

FilterAttach

NDIS_OID_REQUEST

NDIS_SWITCH_PARAMETERS

NDIS_SWITCH_PORT_PARAMETERS

NdisFOidRequest

NET_PNP_EVENT

OID_SWITCH_NIC_DELETE

OID_SWITCH_PARAMETERS

OID_SWITCH_PORT_ARRAY

ReferenceSwitchPort

OID_SWITCH_PORT_UPDATED

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PORT_UPDATED to notify extensible switch extensions about the update of an extensible switch port. The OID will only be issued for ports that have already been created, and have not yet begun the teardown/delete process. Currently, only the **PortFriendlyName** field is subject to update after creation.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to an NDIS_SWITCH_NIC_SAVE_STATE structure.

Remarks

The **PortId** member of the NDIS_SWITCH_PORT_PARAMETERS structure specifies the extensible switch port for which the update notification is being made.

The extension must follow these guidelines for handling OID set requests of OID_SWITCH_PORT_UPDATED:

- The extension must not modify the NDIS_SWITCH_PORT_PARAMETERS structure that is associated with the OID request.

- The extension must always forward this OID set request to underlying extensions. The extension must not fail the request.

Note Extensible switch extensions must not issue OID set requests of OID_SWITCH_PORT_UPDATED.

For more information about the states of extensible switch ports and network adapter connections, see Hyper-V Extensible Switch Port and Network Adapter States.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID set request of OID_SWITCH_PORT_UPDATED and returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

DereferenceSwitchNic

NDIS_OID_REQUEST

NDIS_SWITCH_PORT_PARAMETERS

NdisFOidRequest

OID_SWITCH_NIC_DELETE

OID_SWITCH_PORT_ARRAY

ReferenceSwitchNic

OID_SWITCH_PROPERTY_ADD

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PROPERTY_ADD to notify extensible switch extensions about the addition of a switch policy property

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PROPERTY_PARAMETERS structure that specifies the identification and type of an extensible switch policy.

- A property buffer that contains the parameters for an extensible switch policy. The property buffer contains a structure that is based on the **PropertyType** member of the NDIS_SWITCH_PROPERTY_PARAMETERS structure.

  Note Starting with Windows Server 2012, the **PropertyType** member must be set to **NdisSwitchPropertyTypeCustom** and the property buffer must contain an NDIS_SWITCH_PROPERTY_CUSTOM structure.

Remarks

A forwarding extension can handle the OID set request of OID_SWITCH_PROPERTY_ADD. All other types of extensions must call NdisFOidRequest to forward the OID request to the next extension in the extensible switch driver stack.

The extension can veto the addition of the switch property by returning NDIS_STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot allocate resources to enforce its updated policies on the switch, it should veto the addition request.

Note If the extension returns other NDIS_STATUS_*Xxx* error status codes, the creation notification is also vetoed. However, returning status codes for transitory scenarios, such as returning NDIS_STATUS_RESOURCES, could result in a retry of the creation notification.

If the extension does not veto the OID request, it should monitor the status when the request is completed. The extension should do this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

For guidelines on how to handle an OID set request of OID_SWITCH_PROPERTY_ADD, see Managing Switch Policies.

Return Status Codes

If the forwarding extension completes the OID set request of OID_SWITCH_PROPERTY_ADD, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_DATA_NOT_ACCEPTED	The extension has vetoed the switch policy addition notification.
NDIS_STATUS_FAILURE	The OID request failed for other reasons.

If the extension does not complete the OID set request of OID_SWITCH_PROPERTY_ADD, the request is completed by the underlying miniport edge of the extensible switch. The miniport edge returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PROPERTY_CUSTOM

NDIS_SWITCH_PROPERTY_PARAMETERS

NdisFOidRequest

OID_SWITCH_PROPERTY_DELETE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PROPERTY_DELETE to notify extensible switch extensions about the deletion of a switch policy property.

The **InformationBuffer** member of the **NDIS_OID_REQUEST** structure contains a pointer to a buffer that contains an **NDIS_SWITCH_PROPERTY_DELETE_PARAMETERS** structure.

Remarks

A forwarding extension can handle the OID set request of OID_SWITCH_PROPERTY_DELETE. All other types of extensions must call **NdisFOidRequest** to forward the OID request to the next extension in the extensible switch driver stack.

For guidelines on how to handle an OID set request of OID_SWITCH_PROPERTY_DELETE, see Managing Switch Policies.

Return Status Codes

If the forwarding extension completes the OID set request of OID_SWITCH_PROPERTY_DELETE, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The forwarding extension does not support the switch policy.
NDIS_STATUS_*Xxx*	The OID request failed for other reasons.

If the forwarding extension does not complete the OID set request of OID_SWITCH_PROPERTY_DELETE, the request is completed by the underlying miniport edge of the extensible switch. The miniport edge returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PROPERTY_CUSTOM

NDIS_SWITCH_PROPERTY_PARAMETERS

NdisFOidRequest

OID_SWITCH_PROPERTY_ENUM

Article • 02/18/2023

The Hyper-V extensible switch extension issues an object identifier (OID) method request of OID_SWITCH_PROPERTY_ENUM to obtain an array. This array contains the provisioned switch policies that match the specified criteria. Each element in the array specifies the properties of an extensible switch policy.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PROPERTY_ENUM_PARAMETERS structure that specifies the parameters for the extensible switch policy enumeration.

- An array of NDIS_SWITCH_PROPERTY_ENUM_INFO structures. Each of these structures contains information about an extensible switch policy.

  Note If the extension has not been provisioned with instances of the specified extensible switch policy, the extension sets the **NumProperties** member of the NDIS_SWITCH_PROPERTY_ENUM_PARAMETERS structure to zero and no NDIS_SWITCH_PROPERTY_ENUM_INFO structures are returned.

Remarks

The OID_SWITCH_PROPERTY_ENUM OID must only be issued when the Hyper-V extensible switch has completed activation. Please see Querying the Hyper-V Extensible Switch Configuration for more details.

Unlike OID query requests of OID_SWITCH_PORT_PROPERTY_ENUM, the extension does not have to call any *ReferenceSwitchXxx* or *DereferenceSwitchXxx* functions when it issues the OID_SWITCH_PROPERTY_ENUM request down the extensible switch driver stack.

Note If the extension receives the OID method request of OID_SWITCH_PROPERTY_ENUM, it must not complete the OID request. Instead, it must call NdisFOidRequest to forward the OID request down the extensible switch driver stack.

Return Status Codes

The underlying miniport edge of the extensible switch completes the OID query request of OID_SWITCH_PROPERTY_ENUM and returns one of the following status codes.

Status Code	**Description**
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_INVALID_LENGTH	The length of the information buffer is too small to return the NDIS_SWITCH_PROPERTY_ENUM_PARAMETERS structure and its array of NDIS_SWITCH_PROPERTY_ENUM_INFO elements. The underlying miniport edge of the extensible switch sets the **DATA.METHOD_INFORMATION.BytesNeeded** member in the NDIS_OID_REQUEST structure to the minimum buffer size that is required.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PROPERTY_ENUM_INFO

NDIS_SWITCH_PROPERTY_ENUM_PARAMETERS

Querying the Hyper-V Extensible Switch Configuration

OID_SWITCH_PROPERTY_UPDATE

Article • 02/18/2023

The protocol edge of the Hyper-V extensible switch issues an object identifier (OID) set request of OID_SWITCH_PROPERTY_UPDATE to notify extensible switch extensions about the update to parameters for an extensible switch policy property.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains a pointer to a buffer. This buffer contains the following data:

- An NDIS_SWITCH_PROPERTY_PARAMETERS structure that specifies the identification and type of an extensible switch policy.

- A property buffer that contains the parameters for an extensible switch policy. The property buffer contains a structure that is based on the **PropertyType** member of the NDIS_SWITCH_PROPERTY_PARAMETERS structure.

  Note Starting with Windows Server 2012, the **PropertyType** member must be set to **NdisSwitchPropertyTypeCustom** and the property buffer must contain an NDIS_SWITCH_PROPERTY_CUSTOM structure.

Remarks

A forwarding extension can handle the OID set request of OID_SWITCH_PROPERTY_UPDATE. All other types of extensions must call NdisFOidRequest to forward the OID request to the next extension in the extensible switch driver stack.

The extension can veto the update of the switch property by returning NDIS_STATUS_DATA_NOT_ACCEPTED for the OID request. For example, if an extension cannot allocate resources to enforce its updated policies on the switch, it should veto the update request.

Note If the extension returns other NDIS_STATUS_*Xxx* error status codes, the creation notification is also vetoed. However, returning status codes for transitory scenarios, such as returning NDIS_STATUS_RESOURCES, could result in a retry of the creation notification.

If the extension does not veto the OID request, it should monitor the status when the request is completed. The extension should do this to determine whether the OID request was vetoed by underlying extensions in the extensible switch control path or by the extensible switch interface.

For guidelines on how to handle an OID set request of OID_SWITCH_PROPERTY_UPDATE, see Managing Switch Policies.

Return Status Codes

If the extension completes the OID set request of OID_SWITCH_PROPERTY_UPDATE, it returns one of the following status codes.

Status Code	Description
NDIS_STATUS_DATA_NOT_ACCEPTED	The extension has vetoed the switch policy update notification.
NDIS_STATUS_FAILURE	The OID request failed for other reasons.

If the extension does not complete the OID set request of OID_SWITCH_PROPERTY_UPDATE, the request is completed by the underlying miniport edge of the extensible switch. The miniport edge returns the following status code.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_SWITCH_PROPERTY_CUSTOM

NDIS_SWITCH_PROPERTY_PARAMETERS

NdisFOidRequest

OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG

Article • 02/18/2023

As a query request, administrative applications (or possibly overlying drivers) can use the OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG OID to determine the currently-enabled connection offload capabilities of an underlying miniport adapter. A system administrator can use this OID through the Microsoft Windows Management Instrumentation (WMI) interface.

Set requests are not supported.

Remarks

NDIS handles this OID for miniport drivers. Miniport drivers report miniport adapter connection offload settings to NDIS. For information about passing connection offload configuration settings to NDIS from a miniport driver and from NDIS to overlying drivers, see NDIS_TCP_CONNECTION_OFFLOAD.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains an NDIS_TCP_CONNECTION_OFFLOAD structure.

In response to OID_TCP_CONNECTION_OFFLOAD_CURRENT_CONFIG, the **Encapsulation** member of NDIS_TCP_CONNECTION_OFFLOAD defines the current packet encapsulation configuration of the miniport adapter. NDIS provides a bitwise OR of the flags that are provided in the **Encapsulation** member. The other members of NDIS_TCP_CONNECTION_OFFLOAD contain settings for various connection offload services. For more information about encapsulation and other capabilities, see NDIS_TCP_CONNECTION_OFFLOAD and NDIS_OFFLOAD_PARAMETERS.

See also

NDIS_OFFLOAD_PARAMETERS
NDIS_OID_REQUEST
NDIS_TCP_CONNECTION_OFFLOAD

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES

Article • 02/18/2023

As a query request, the OID_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES OID reports the current connection offload hardware capabilities of an underlying miniport adapter. User-mode applications (or possibly overlying drivers) can query this OID to determine the connection offload hardware capabilities of an underlying miniport adapter. A system administrator can use this OID through the Microsoft Windows Management Instrumentation (WMI) interface.

Set requests are not supported.

Remarks

NDIS handles this OID for miniport drivers. Miniport drivers report miniport adapter connection offload hardware capabilities to NDIS. For information about passing connection offload hardware capabilities to NDIS from a miniport driver and from NDIS to overlying drivers, see NDIS_TCP_CONNECTION_OFFLOAD.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains an NDIS_TCP_CONNECTION_OFFLOAD structure.

In response to OID_TCP_CONNECTION_OFFLOAD_HARDWARE_CAPABILITIES, the **Encapsulation** member of NDIS_TCP_CONNECTION_OFFLOAD defines the current packet encapsulation hardware capabilities of the miniport adapter. NDIS provides a bitwise OR of the flags that are provided in the **Encapsulation** member. The other members of NDIS_TCP_CONNECTION_OFFLOAD contain settings for various connection offload services. For more information about encapsulation and other capabilities, see NDIS_TCP_CONNECTION_OFFLOAD and NDIS_OFFLOAD_PARAMETERS.

See also

NDIS_OFFLOAD_PARAMETERS
NDIS_OID_REQUEST
NDIS_TCP_CONNECTION_OFFLOAD

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_CONNECTION_OFFLOAD_PARAMETERS

Article • 02/18/2023

As a query request, overlying drivers can use the OID_TCP_CONNECTION_OFFLOAD_PARAMETERS OID to determine the current connection offload settings of an underlying miniport adapter. NDIS handles this OID query for miniport drivers.

As a set request, NDIS and overlying drivers use the OID_TCP_CONNECTION_OFFLOAD_PARAMETERS OID to set the connection offload configuration parameters of an underlying miniport adapter. Miniport drivers that support connection offload must handle this OID set request. Otherwise, the OID_TCP_CONNECTION_OFFLOAD_PARAMETERS OID is optional for miniport drivers.

Remarks

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains an NDIS_TCP_CONNECTION_OFFLOAD_PARAMETERS structure.

> ⓘ Note
>
> Do not confuse OID_TCP_CONNECTION_OFFLOAD_PARAMETERS with the **OID_TCP_OFFLOAD_PARAMETERS** OID that administrative applications use to enable or disable TCP offload features.

See also

NDIS_OID_REQUEST
OID_TCP_OFFLOAD_PARAMETERS

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_OFFLOAD_CURRENT_CONFIG

Article • 02/18/2023

As a query request, administrative applications (or possibly overlying drivers) use the OID_TCP_OFFLOAD_CURRENT_CONFIG OID to determine the current task offload configuration settings of an underlying miniport adapter. A system administrator can use this OID through the Microsoft Windows Management Instrumentation (WMI) interface.

Set requests are not supported.

Remarks

NDIS handles this OID for miniport drivers. Miniport drivers report miniport adapter offload capabilities to NDIS. For information about passing task offload configuration settings to NDIS from a miniport driver and from NDIS to overlying drivers, see NDIS_OFFLOAD.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains an NDIS_OFFLOAD structure. The **NDIS_OFFLOAD** structure includes the following miniport adapter capabilities:

- The header information, which includes the task offload version.
- The checksum offload information, in an NDIS_TCP_IP_CHECKSUM_OFFLOAD structure.
- The large send offload version 1 (LSOV1) information, in an NDIS_TCP_LARGE_SEND_OFFLOAD_V1 structure.
- The Internet protocol security (IPsec) information, in an NDIS_IPSEC_OFFLOAD_V1 structure.
- The large send offload version 2 (LSOV2) information, in an NDIS_TCP_LARGE_SEND_OFFLOAD_V2 structure.

In response to OID_TCP_OFFLOAD_CURRENT_CONFIG, the **Encapsulation** members of the structures in the preceding list define the packet encapsulation capabilities of the miniport adapter. NDIS provides a bitwise OR of the flags that are provided in the **Encapsulation** members of these structures. The other structure members contain settings for various offload services. For more information about encapsulation and other capabilities, see NDIS_TCP_IP_CHECKSUM_OFFLOAD, NDIS_TCP_LARGE_SEND_OFFLOAD_V1, NDIS_IPSEC_OFFLOAD_V1, and NDIS_TCP_LARGE_SEND_OFFLOAD_V2.

Miniport adapters must support Ethernet encapsulation for all of the types of task offload that they support. The other types of encapsulation are optional.

Miniport drivers should automatically enable all of the task offload capabilities during initialization.

See also

NDIS_IPSEC_OFFLOAD_V1
NDIS_OFFLOAD
NDIS_OID_REQUEST
NDIS_TCP_IP_CHECKSUM_OFFLOAD
NDIS_TCP_LARGE_SEND_OFFLOAD_V2 NDIS_IPSEC_OFFLOAD_V1

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES

Article • 02/18/2023

As a query request, the OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES OID reports the task offload hardware capabilities of a miniport adapter's hardware. User-mode applications (or possibly overlying drivers) can query this OID to determine the task offload hardware capabilities of an underlying miniport adapter. A system administrator can use this OID through the Windows Management Instrumentation (WMI) interface.

Set requests are not supported.

Remarks

NDIS handles this OID for miniport drivers. Miniport drivers report miniport adapter hardware capabilities to NDIS. For information about reporting task offload hardware capabilites to NDIS from a miniport driver and from NDIS to overlying drivers, see NDIS_OFFLOAD.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains an NDIS_OFFLOAD structure. NDIS returns NDIS_STATUS_BUFFER_TOO_SHORT if the buffer is not big enough.

After determining the miniport adapter's hardware capabilities, the overlying applications or drivers can use the OID_TCP_OFFLOAD_PARAMETERS OID to enable capabilities that are currently reported as not enabled by the OID_TCP_OFFLOAD_CURRENT_CONFIG OID.

See also

NDIS_OFFLOAD
NDIS_OID_REQUEST
OID_TCP_OFFLOAD_CURRENT_CONFIG
OID_TCP_OFFLOAD_PARAMETERS

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_OFFLOAD_PARAMETERS

Article • 02/18/2023

Query requests are not supported.

As a set request, the OID_TCP_OFFLOAD_PARAMETERS OID sets the current TCP offload configuration of a miniport adapter. Protocol drivers or user-mode applications can set this OID to change the current TCP offload configuration. A system administrator can use this OID through the Microsoft Windows Management Instrumentation (WMI) interface.

Remarks

OID_TCP_OFFLOAD_PARAMETERS is required for miniport drivers that support TCP offloads and optional for other miniport drivers. If a miniport driver does not support this OID, the driver should return NDIS_STATUS_NOT_SUPPORTED.

The **InformationBuffer** member of the NDIS_OID_REQUEST structure contains an NDIS_OFFLOAD_PARAMETERS structure. If the contents of **InformationBuffer** are invalid, the miniport driver should return NDIS_STATUS_INVALID_DATA in response to this OID.

While NDIS processes this OID and before it passes the OID to the miniport driver, NDIS updates the miniport adapter's offload standardized keywords with the new settings.

Miniport drivers must use the contents of the NDIS_OFFLOAD_PARAMETERS structure to update the currently reported TCP offload capabilities. After the update, the miniport driver must report the current task offload capabilities with the NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG status indication. This status indication ensures that all of the overlying protocol drivers are updated with the new capabilities information.

This OID is a more comprehensive OID that instructs miniport drivers to turn certain offloads on or off. Most TCP/IP task offloads can be configured and activated with this OID. For some offloads, such as Rx Checksum or Rx IPSec, this OID serves as a configuration change and doesn't mean the offload will be operational immediately. To activate those offloads, the miniport driver must wait until it receives an OID_OFFLOAD_ENCAPSULATION Set request.

Before setting OID_TCP_OFFLOAD_PARAMETERS, the overlying applications or drivers can use the OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES OID to determine what capabilities a miniport adapter's hardware can support. Use

OID_TCP_OFFLOAD_PARAMETERS to enable capabilities that are reported as not enabled by the OID_TCP_OFFLOAD_CURRENT_CONFIG OID.

See also

NDIS_OFFLOAD_PARAMETERS
NDIS_OID_REQUEST
NDIS_STATUS_TASK_OFFLOAD_CURRENT_CONFIG
OID_TCP_OFFLOAD_CURRENT_CONFIG
OID_TCP_OFFLOAD_HARDWARE_CAPABILITIES

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_RSC_STATISTICS

Article • 02/18/2023

As a query, NDIS and overlying drivers or user-mode applications use the OID_TCP_RSC_STATISTICS OID to get the receive-segment coalescing (RSC) statistics of a miniport adapter.

NDIS 6.30 and later miniport drivers that provide RSC services must support this OID. Otherwise, this OID is optional.

Remarks

The **InformationBuffer** member of NDIS_OID_REQUEST structure contains an NDIS_RSC_STATISTICS_INFO structure.

The miniport driver must maintain the statistics in the members of the NDIS_RSC_STATISTICS_INFO structure as follows:

- The driver must increment the coalesced packet count in the **CoalescedPkts** member by one every time a packet is added to a single coalesced unit (SCU).
- The driver must increment the coalesced octet count in the **CoalescedOctets** member by the size of the TCP payload of the packet every time a packet is added to a SCU.
- The driver must increment the coalesced events count **CoalesceEvents** member by one every time a SCU is finalized. All such SCUs should have a non-zero **CoalescedSegCount** value.
- The driver must increment the abort count in the **Aborts** member by one every time it encounters an exception other than the IP datagram length being exceeded. This count should include the cases where a packet is not coalesced because of hardware resources.

Requirements

Version	Supported for NDIS 6.30 and later drivers in Windows 8.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_OID_REQUEST

NDIS_RSC_STATISTICS_INFO

OID_TCP_TASK_IPSEC_ADD_SA

Article • 02/18/2023

The OID_TCP_TASK_IPSEC_ADD_SA OID is set by the transport protocol to request that a miniport driver add one or more security associations (SAs) to a NIC.

The information for each SA is formatted as an OFFLOAD_IPSEC_ADD_SA structure.

The first seven members of the OFFLOAD_IPSEC_ADD_SA structure (**SrcAddr**, **SrcMask**, **DestAddr**, **DestMask**, **Protocol**, **SrcPort**, and **DestPort**) constitute a filter that specifies the source and destination, as well as the IP protocols, to which the SAs apply. This filter pertains to a transport-mode connection--that is, an end-to-end connection between two hosts. If the specified connection is made through a tunnel, the source and destination addresses of the tunnel are specified by **SrcTunnelAddr** and **DestTunnelAddr**, respectively.

If a filter parameter is set to zero, that parameter is not used to filter packets for the specified SAs. For example, if **SrcAddr** is set to zero, the specified SAs can apply to a packet that contains any source address. To take this to the extreme, if all the filter parameters are set to zero, the specified SAs apply to any source host sending any type of packet to any destination host.

The TCP/IP transport can specify an IP protocol in the **Protocol** member to indicate that the specified SAs apply only to packets of the specified protocol type. If **Protocol** is set to zero, the specified SAs apply to all packets sent from the specified source to the specified destination.

OFFLOAD_SECURITY_ASSOCIATION structure

An OFFLOAD_SECURITY_ASSOCIATION structure specifies a single security association (SA). The OFFLOAD_SECURITY_ASSOCIATION structure is an element in the **SecAssoc** variable-length array. **SecAssoc** contains one or two OFFLOAD_SECURITY_ASSOCIATION structures.

An SA specified for use in processing authentication headers (AH) will have an operation type of **AUTHENTICATE** and will have an **IntegrityAlgo** (integrity algorithm). The SA will not have an a **ConfAlgo** (confidentiality algorithm). In this case, **ConfAlgo** will contain zeros.

An SA specified for use in processing encapsulating security payloads (ESPs) will have an operation type of **ENCRYPT** and may have an **IntegrityAlgo** (integrity algorithm) and/or a **ConfAlgo** (confidentiality algorithm).

OFFLOAD_ALGO_INFO structure

The OFFLOAD_ALGO_INFO structure, which is a member of an OFFLOAD_SECURITY_ASSOCIATION structure, specifies an algorithm used for a security association (SA).

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_TASK_IPSEC_ADD_UDPESP_SA

Article • 02/18/2023

A transport protocol sets OID_TCP_TASK_IPSEC_ADD_UDPESP_SA to request a miniport driver to add one or more security associations (SAs) for UDP-encapsulated ESP packets to a NIC.

The information for each SA is formatted as an OFFLOAD_IPSEC_ADD_UDPESP_SA structure. Note that this structure is almost identical to the OFFLOAD_IPSEC_ADD_SA structure used in the OID_TCP_TASK_IPSEC_ADD_SA request. The only difference is that the OFFLOAD_IPSEC_ADD_UDPESP_SA structure contains the **EncapTypeEntry** and the **EncapTypeEntryOffldHandle** members.

The first seven members of the OFFLOAD_IPSEC_ADD_UDPESP_SA structure (**SrcAddr**, **SrcMask**, **DestAddr**, **DestMask**, **Protocol**, **SrcPort**, and **DestPort**) constitute a filter that specifies the source and destination, as well as the IP protocols, to which the SAs apply. This filter pertains to a transport-mode connection--that is, an end-to-end connection between two hosts. If the specified connection is made through a tunnel, the source and destination addresses of the tunnel are specified by **SrcTunnelAddr** and **DestTunnelAddr**, respectively.

If a filter parameter is set to zero, that parameter is not used to filter packets for the specified SAs. For example, if **SrcAddr** is set to zero, the specified SAs can apply to a packet that contains any source address. To take this to the extreme, if all the filter parameters are set to zero, the specified SAs apply to any source host sending any type of packet to any destination host.

The TCP/IP transport can specify an IP protocol in the **Protocol** member to indicate that the specified SAs apply only to packets of the specified protocol type. If **Protocol** is set to zero, the specified SAs apply to all packets sent from the specified source to the specified destination.

OFFLOAD_SECURITY_ASSOCIATION structure

An OFFLOAD_SECURITY_ASSOCIATION structure specifies a single security association (SA). The OFFLOAD_SECURITY_ASSOCIATION structure is an element in the **SecAssoc** variable-length array. **SecAssoc** contains one or two OFFLOAD_SECURITY_ASSOCIATION structures.

An SA specified for use in processing authentication headers (AH) will have an operation type of **AUTHENTICATE** and will have an **IntegrityAlgo** (integrity algorithm). The SA will not have an a **ConfAlgo** (confidentiality algorithm). In this case, **ConfAlgo** will contain zeros.

An SA specified for use in processing encapsulating security payloads (ESPs) will have an operation type of **ENCRYPT** and may have an **IntegrityAlgo** (integrity algorithm) and/or a **ConfAlgo** (confidentiality algorithm).

OFFLOAD_ALGO_INFO structure

The OFFLOAD_ALGO_INFO structure, which is a member of an OFFLOAD_SECURITY_ASSOCIATION structure, specifies an algorithm used for a security association (SA).

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_TASK_IPSEC_DELETE_SA

Article • 02/18/2023

The OID_TCP_TASK_IPSEC_DELETE_SA OID is set by a transport protocol to request that a miniport driver delete a security association (SA) from a NIC. The SA information is formatted as an OFFLOAD_IPSEC_DELETE_SA structure.

On receiving this request, the miniport driver should delete the specified SA from the NIC and free any system resources allocated for the SA.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA

Article • 02/18/2023

A transport protocol sets OID_TCP_TASK_IPSEC_DELETE_UDPESP_SA to request that a miniport driver delete a UDP-ESP security association (SA) and, possibly, a parser entry from a NIC parser entry list. The SA and parser entry information is formatted as an OFFLOAD_IPSEC_DELETE_UDPESP_SA structure.

If the **EncapTypeEntryOffldHandle** is **NULL**, the miniport should delete the specified SA from the NIC and free any system resources allocated for the SA. If the **EncapTypeEntryOffldHandle** is non-**NULL**, the miniport should also delete the specified parser entry from the NIC's parser entry list.

Note that a transport protocol could request a miniport to delete an SA and/or parser entry before the miniport has completed adding that SA and/or parser entry. The miniport must therefore serialize the deletion operation with the addition operation.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA

Article • 02/18/2023

[The IPsec Task Offload feature is deprecated and should not be used.]

As a set, the TCP/IP transport uses the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA OID to request that a miniport driver add the specified security associations (SAs) to a NIC.

Note NDIS supports this OID with the direct OID request interface. For more information about the direct OID request interface, see NDIS 6.1 Direct OID Request Interface.

Note This OID is supported in NDIS 6.1 and 6.20. For NDIS 6.30 and later drivers see OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX.

Remarks

All NDIS 6.1 and 6.20 miniport drivers that support IPsec offload version 2 (IPsecOV2) must support this OID.

After TCP/IP transport determines that a NIC can perform IPsecOV2 operations, the TCP/IP transport requests the miniport driver to add SAs. The transport cannot offload IPsecOV2 operations to the NIC before the transport adds an SA.

The miniport driver receives an **IPSEC_OFFLOAD_V2_ADD_SA** structure that contains a pointer to the next IPSEC_OFFLOAD_V2_ADD_SA structure in a linked list. The miniport driver configures the NIC for IPsecOV2 processing on the SAs. With a successful set to OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA, the miniport driver supplies the handles that identify the offloaded SAs in the **OffloadHandle** member of IPSEC_OFFLOAD_V2_ADD_SA. (For example, the transport uses the handle in the send path to indicate which offloaded SA to use). If any of the SAs in the linked list were offloaded, the set request is successful.

The miniport driver can return a failure status for the OID request, for example, when the NIC runs out of capacity to offload more SAs. Also, the miniport driver might return a failure status because it needs to avoid a race condition. In this case, the NIC configuration changes and excludes a particular algorithm.

If the request fails, none of the SAs in the linked list were offloaded. If failure occurs for a particular SA in the linked list, the miniport driver should set the **OffloadHandle** member in the corresponding IPSEC_OFFLOAD_V2_ADD_SA structure to **NULL**.

The miniport driver reports the maximum number of SAs that a NIC can support in the **SaOffloadCapacity** member of the **NDIS_IPSEC_OFFLOAD_V2** structure during initialization. If necessary, the TCP/IP transport can set the OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA OID to request that the miniport driver delete an SA from the NIC.

Requirements

Version	Supported in NDIS 6.1 and 6.20. For NDIS 6.30 and later, use OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX.
Header	Ntddndis.h (include Ndis.h)

See also

IPSEC_OFFLOAD_V2_ADD_SA

NDIS_IPSEC_OFFLOAD_V2

OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX

OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA

OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX

Article • 02/18/2023

[The IPsec Task Offload feature is deprecated and should not be used.]

As a set, the TCP/IP transport uses the OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX OID to request that a miniport driver add the specified security associations (SAs) to a NIC.

Note NDIS supports this OID with the direct OID request interface. For more information about the direct OID request interface, see NDIS 6.1 Direct OID Request Interface.

Remarks

All NDIS 6.30 miniport drivers that support IPsec offload version 2 (IPsecOV2) must support this OID.

After TCP/IP transport determines that a NIC can perform IPsecOV2 operations, the TCP/IP transport requests the miniport driver to add SAs. The transport cannot offload IPsecOV2 operations to the NIC before the transport adds an SA.

The miniport driver configures the NIC for IPsecOV2 processing on the SAs. With a successful set to OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA_EX, the miniport driver supplies the handle that identifies the offloaded SA in the **OffloadHandle** member of the **IPSEC_OFFLOAD_V2_ADD_SA_EX** structure. (For example, the transport uses the handle in the send path to indicate which offloaded SA to use). If an SA was offloaded, the set request is successful.

The miniport driver can return a failure status for the OID request, for example, when the NIC runs out of capacity to offload more SAs. Also, the miniport driver might return a failure status because it needs to avoid a race condition. In this case, the NIC configuration changes and excludes a particular algorithm.

If the request fails, SAs were not offloaded. If failure occurs for an SA, the miniport driver should set the **OffloadHandle** member in the corresponding IPSEC_OFFLOAD_V2_ADD_SA_EX structure to **NULL**.

The miniport driver reports the maximum number of SAs that a NIC can support in the **SaOffloadCapacity** member of the **NDIS_IPSEC_OFFLOAD_V2** structure during initialization. If necessary, the TCP/IP transport can set the OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA OID to request that the miniport driver delete an SA from the NIC.

This OID is essentially identical to the previous version, OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA. The only difference is the updated **IPSEC_OFFLOAD_V2_ADD_SA_EX** structure.

Requirements

Version	Supported in NDIS 6.30 and later.
Header	Ntddndis.h (include Ndis.h)

See also

IPSEC_OFFLOAD_V2_ADD_SA_EX

NDIS_IPSEC_OFFLOAD_V2

OID_TCP_TASK_IPSEC_OFFLOAD_V2_ADD_SA

OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA

OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA

Article • 02/18/2023

[The IPsec Task Offload feature is deprecated and should not be used.]

As a set, the TCP/IP transport uses the OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA OID to request that a miniport driver delete the specified security associations (SAs) from a NIC.

Note NDIS supports this OID with the direct OID request interface. For more information about the direct OID request interface, see NDIS 6.1 Direct OID Request Interface.

Remarks

All NDIS 6.1 miniport drivers that support IPsec offload version 2 (IPsecOV2) must support this OID.

When a miniport driver receives this request, the driver should delete the specified SAs from the NIC and free any system resources that were allocated for the SAs.

The miniport driver receives an **IPSEC_OFFLOAD_V2_DELETE_SA** structure that contains a handle to an SA bundle and a pointer to the next **IPSEC_OFFLOAD_V2_DELETE_SA** structure in a linked list.

The miniport driver can set **SaDeleteReq** in the **NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO** structure for a receive **NET_BUFFER_LIST** structure. The TCP/IP transport subsequently issues OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA once to delete the inbound SA that the packet was received over and once again to delete the outbound SA that corresponds to the deleted inbound SA. The NIC must not remove either of these SAs before receiving the corresponding OID_TCP_TASK_IPSEC_OFFLOAD_V2_DELETE_SA request.

Return status codes

The miniport driver's *MiniportOidRequest* function returns one of the following values for this request:

Term	Description

Term	Description
NDIS_STATUS_SUCCESS	The miniport driver completed the request successfully.
NDIS_STATUS_PENDING	The miniport driver will complete the request asynchronously. After the miniport driver has completed all processing, it must succeed the request by calling the NdisMOidRequestComplete function, passing **NDIS_STATUS_SUCCESS** for the *Status* parameter.
NDIS_STATUS_NOT_ACCEPTED	The miniport driver is resetting.
NDIS_STATUS_REQUEST_ABORTED	The miniport driver stopped processing the request. For example, NDIS called the *MiniportResetEx* function.

Requirements

Version	Supported in NDIS 6.1 and later.
Header	Ntddndis.h (include Ndis.h)

See also

IPSEC_OFFLOAD_V2_DELETE_SA

NDIS_IPSEC_OFFLOAD_V2_NET_BUFFER_LIST_INFO

NET_BUFFER_LIST

OID_TCP_TASK_IPSEC_OFFLOAD_V2_UPDATE_SA

Article • 02/18/2023

[The IPsec Task Offload feature is deprecated and should not be used.]

As a set, the TCP/IP transport uses the OID_TCP_TASK_IPSEC_OFFLOAD_V2_UPDATE_SA OID to request that a miniport driver update the specified security associations (SAs) on a NIC.

Note NDIS supports this OID with the direct OID request interface. For more information about the direct OID request interface, see NDIS 6.1 Direct OID Request Interface.

Remarks

All NDIS 6.1 miniport drivers that support IPsec offload version 2 (IPsecOV2) must support this OID.

When a miniport driver receives this request, the driver should update the specified SAs on the NIC. The miniport driver can fail this request if the SA is not found or the ESN is not supported. In this case, the returned status should be NDIS_STATUS_INVALID_PARAMETER.

The miniport driver receives an **IPSEC_OFFLOAD_V2_UPDATE_SA** structure that contains information about the update and a pointer to the next IPSEC_OFFLOAD_V2_UPDATE_SA structure in a linked list.

Requirements

Version	Supported in NDIS 6.1 and later.
Header	Ntddndis.h (include Ndis.h)

See also

IPSEC_OFFLOAD_V2_UPDATE_SA

OID_TCP_TASK_OFFLOAD

Article • 02/18/2023

The host stack queries the OID_TCP_TASK_OFFLOAD OID to obtain the TCP offload capabilities of a miniport driver's NIC or of an offload target. After determining the offload capabilities that a NIC or an offload target supports, the host stack sets this OID to enable one or more of the reported capabilities. The host stack can also disable all of a NIC's or an offload target's TCP offload capabilities by setting OID_TCP_TASK_OFFLOAD. Only one protocol at a time can enable the TCP offload capabilities of a particular NIC.

Querying offload capabilities

When the host stack queries OID_TCP_TASK_OFFLOAD, it supplies in the *InformationBuffer* an NDIS_TASK_OFFLOAD_HEADER structure. This structure specifies the following:

- The offload version supported by the host stack.
- The encapsulation format for send and receive packets processed by the host stack.
- The size of the encapsulation header in such packets.

With this information, a miniport driver or its NIC can locate the beginning of the first IP header in a transmit packet, which is a prerequisite for performing an offload task. An offload target needs to know the encapsulation format to process receive packets. In response to a query of OID_TCP_TASK_OFFLOAD, a miniport driver or offload target returns, in the *InformationBuffer*, the NDIS_TASK_OFFLOAD_HEADER structure followed immediately by one or more NDIS_TASK_OFFLOAD structures. Each NDIS_TASK_OFFLOAD structure describes an offload capability supported by the miniport driver's NIC or by the offload target. If the miniport driver's NIC or the offload target supports multiple versions of a particular offload capability, it should return one NDIS_TASK_OFFLOAD structure for each version.

Each NDIS_TASK_OFFLOAD structure has a **Task** member that specifies the particular offload capability to which the structure applies. Each NDIS_TASK_OFFLOAD structure also has a **TaskBuffer** that contains information pertinent to the specified offload capability. The information in the **TaskBuffer** is formatted as one of the following structures:

- NDIS_TASK_TCP_IP_CHECKSUM
  Specifies checksum offload capabilities.

- NDIS_TASK_IPSEC
  Specifies Internet Protocol security (IPsec) offload capabilities.
- NDIS_TASK_TCP_LARGE_SEND
  Specifies large TCP packet segmentation capabilities.
- NDIS_TASK_TCP_CONNECTION_OFFLOAD
  Specifies TCP chimney offload capabilities. For more information on NDIS_TASK_TCP_CONNECTION_OFFLOAD, see TCP Chimney Offload.

> ⓘ Note
>
> If an intermediate driver modifies the contents of packets that it forwards to an underlying miniport driver such that TCP offload functions cannot be performed on the packets, the intermediate driver should respond to OID_TCP_TASK_OFFLOAD queries with a status of NDIS_STATUS_NOT_SUPPORTED instead of passing the OID request to the underlying miniport driver or offload target.

Enabling offload capabilities

After querying a NIC's or an offload target's offload capabilities, the host stack enables one or more of these capabilities by setting OID_TCP_TASK_OFFLOAD. When setting OID_TCP_TASK_OFFLOAD, the host stack supplies, in the *InformationBuffer*, an NDIS_TASK_OFFLOAD_HEADER structure followed immediately by an NDIS_TASK_OFFLOAD structure for each offload capability that the host stack is enabling.

The **Task** in each NDIS_TASK_OFFLOAD structure indicates the offload capability that the host stack is enabling. The host stack also enables specific aspects of a particular offload capability by setting members of the structure in the **TaskBuffer** of each NDIS_TASK_OFFLOAD structure.

Changing offload capabilities

To change the offload capabilities that are enabled for a NIC or an offload target, the host stack sets OID_TCP_TASK_OFFLOAD. The miniport driver or offload target must enable only those offload capabilities specified by the most recent set of OID_TCP_TASK_OFFLOAD. The miniport driver or offload target must disable all other offload capabilities. Note that before disabling a specific TCP chimney offload capability, the host stack terminates the offload of any offloaded TCP connections that use that capability.

An offload target can use pause or resume offload indications to change its reported TCP offload capabilities:

- An offload target makes a pause indication by calling the NdisMIndicateStatusEx function with the NDIS_STATUS_INDICATION->**StatusCode** member set to NDIS_STATUS_OFFLOAD_PAUSE.
- An offload target makes a resume indication by calling the **NdisMIndicateStatusEx** function with the NDIS_STATUS_INDICATION->**StatusCode** member set to NDIS_STATUS_OFFLOAD_RESUME.

After an offload target requests the host stack to resume offloading state objects, the host stack queries OID_TCP_TASK_OFFLOAD again to obtain the offload target's TCP offload revised capabilities. For more information, see NDIS_STATUS_OFFLOAD_RESUME.

Disabling offload capabilities

To disable all offload capabilities supported by a NIC or an offload target, the host stack sets OID_TCP_TASK_OFFLOAD. In the *InformationBuffer*, the host stack supplies an NDIS_TASK_OFFLOAD_HEADER structure with the **OffsetFirstTask** member of this structure set to zero.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP4_OFFLOAD_STATS

Article • 02/18/2023

The host stack queries the OID_TCP4_OFFLOAD_STATS OID to obtain statistics on TCP segments that an offload target has processed on offloaded TCP connections that convey IPv4 datagrams. The host stack sets this OID to cause an offload target to reset the counters for such statistics to zero.

In response to a query of OID_TCP4_OFFLOAD_STATS, an offload target supplies a filled-in TCP_OFFLOAD_STATS structure.

In response to a set of OID_TCP4_OFFLOAD_STATS, an offload target should reset to zero all of its TCP statistics counters for offloaded TCP connections that convey IPv4 datagrams.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TCP6_OFFLOAD_STATS

Article • 02/18/2023

The host stack queries the OID_TCP6_OFFLOAD_STATS OID to obtain statistics on TCP segments that an offload target has processed on offloaded TCP connections that convey IPv6 datagrams. The host stack sets this OID to cause an offload target to reset the counters for such statistics to zero.

In response to a query of OID_TCP6_OFFLOAD_STATS, an offload target supplies a filled-in TCP_OFFLOAD_STATS structure.

In response to a set of OID_TCP6_OFFLOAD_STATS, an offload target should reset to zero all of its TCP statistics counters for offloaded TCP connections that convey IPv6 datagrams.

Requirements

Version: Windows Vista and later **Header**: Ntddndis.h (include Ndis.h)

OID_TIMESTAMP_CAPABILITY

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_TIMESTAMP_CAPABILITY to obtain the hardware timestamping capabilities of the NIC and software timestamping capabilities of the miniport driver.

The **RequestType** member of the **NDIS_OID_REQUEST** structure will be **NdisRequestQueryInformation**.

NDIS handles this OID for the miniport driver based on the information the miniport driver provided in the **NDIS_STATUS_TIMESTAMP_CAPABILITY** status indication.

Requirements

Requirement	**Value**
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
NDIS Version	NDIS 6.82 and later
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_TIMESTAMP_CAPABILITY

OID_TIMESTAMP_CURRENT_CONFIG

OID_TIMESTAMP_GET_CROSSTIMESTAMP

NDIS_OID_REQUEST

OID_TIMESTAMP_CURRENT_CONFIG

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_TIMESTAMP_CURRENT_CONFIG to obtain the current timestamping configuration of the NIC.

The **RequestType** member of the NDIS_OID_REQUEST structure will be **NdisRequestQueryInformation**.

NDIS handles this OID for the miniport driver based on the information the miniport driver provided in the NDIS_STATUS_TIMESTAMP_CAPABILITY status indication.

Requirements

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
NDIS Version	NDIS 6.82 and later
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG

OID_TIMESTAMP_CAPABILITY

OID_TIMESTAMP_GET_CROSSTIMESTAMP

NDIS_OID_REQUEST

OID_TIMESTAMP_GET_CROSSTIMESTAMP

Article • 02/18/2023

An overlying driver issues an object identifier (OID) query request of OID_TIMESTAMP_GET_CROSSTIMESTAMP to obtain a cross timestamp from the NIC hardware. A cross timestamp is the set of a NIC hardware timestamp and system timestamp(s) obtained very close to each other. Precision Time Protocol (PTP) version 2 applications use the information provided in this OID to establish a relation between the NIC's hardware clock and a system clock.

The miniport driver must support this OID if it sets the **CrossTimestamp** field to **TRUE** in the NDIS_TIMESTAMP_CAPABILITIES structure as part of the current configuration. For more details on reporting the current configuration, see the NDIS_STATUS_TIMESTAMP_CURRENT_CONFIG status indication. If the cross timestamping ability is disabled, then the OID should be completed with an appropriate error code (for example, NDIS_STATUS_NOT_SUPPORTED).

The **RequestType** member of the NDIS_OID_REQUEST structure will be **NdisRequestQueryInformation**.

When a miniport driver receives the OID request of OID_TIMESTAMP_GET_CROSSTIMESTAMP, the driver completes the OID by filling the **InformationBuffer** in the **QUERY_INFORMATION** with an NDIS_HARDWARE_CROSSTIMESTAMP structure. The **Type** field in the **Header** field of the **NDIS_HARDWARE_CROSSTIMESTAMP** structure should be set to **NDIS_OBJECT_TYPE_DEFAULT** and the **Revision** field to **NDIS_HARDWARE_CROSSTIMESTAMP_REVISION_1**. The driver should fill the **SystemTimestamp1**, **HardwareClockTimestamp** and **SystemTimestamp2** fields with following timestamps taken as close to each other as possible and in the following order:

1. **SystemTimestamp1**: Performance counter value (QPC) obtained by calling KeQueryPerformanceCounter.

2. **HardwareClockTimestamp**: The NIC hardware clock's current value. This should be the raw hardware clock value of the NIC.

3. **SystemTimestamp2**: Another performance counter value (QPC) obtained by calling **KeQueryPerformanceCounter**.

Here's an example of how a miniport driver handles OID_TIMESTAMP_GET_CROSSTIMESTAMP:

C++

```
{
. . .
    NDIS_HARDWARE_CROSSTIMESTAMP crossTimestamp;
    LARGE_INTEGER timeStamp;

    RtlZeroMemory(&crossTimestamp, sizeof(crossTimestamp));

    timeStamp = KeQueryPerformanceCounter(NULL);
    crossTimestamp.SystemTimestamp1 = timeStamp.QuadPart;
    crossTimestamp.HardwareClockTimestamp =
FunctionToRetrieveHardwareTimestampFromNetworkCard();
    timeStamp = KeQueryPerformanceCounter(NULL);
    crossTimestamp.SystemTimestamp2 = timeStamp.QuadPart;
    crossTimestamp.Header.Type = NDIS_OBJECT_TYPE_DEFAULT;
    crossTimestamp.Header.Size =
NDIS_SIZEOF_HARDWARE_CROSSTIMESTAMP_REVISION_1;
    crossTimestamp.Header.Revision =
NDIS_HARDWARE_CROSSTIMESTAMP_REVISION_1;

// Complete the OID by filling the query information buffer with the
crossTimestamp
}
```

The **Flags** field in the **NDIS_HARDWARE_CROSSTIMESTAMP** structure is reserved for future use. The miniport driver must not change its value.

The miniport driver and hardware are free to optimize the collection of these timestamps depending on any advanced hardware capabilities. However, the **SystemTimestamp1** and **SystemTimestamp2** values returned on OID completion must accurately correspond to the performance counter (QPC) value at the time of capture. The **HardwareClockTimestamp** must correspond to the NIC's hardware clock value at the point of capture. If a particular implementation can more accurately determine two timestamps rather than three (for example, one system timestamp and the corresponding NIC hardware clock timestamp), then it should set the **SystemTimestamp2** field to the same value as **SystemTimestamp1**.

The miniport driver should not set the **SystemTimestamp1**, **HardwareClockTimestamp**, or **SystemTimestamp2** values to **zero**.

Return Status Codes

The miniport driver returns one of the following status codes for the OID query request of OID_TIMESTAMP_GET_CROSSTIMESTAMP.

Status Code	Description
NDIS_STATUS_SUCCESS	The OID request completed successfully.
NDIS_STATUS_NOT_SUPPORTED	The miniport driver either does not support cross timestamping or the cross timestamping ability is disabled.
NDIS_STATUS_FAILURE	The request failed for other reasons.

Requirements

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
NDIS Version	NDIS 6.82 and later
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_TIMESTAMP_CAPABILITY

OID_TIMESTAMP_CURRENT_CONFIG

OID_TIMESTAMP_CAPABILITY

NDIS_OID_REQUEST

OID_TUNNEL_INTERFACE_RELEASE_OID

Article • 02/18/2023

The OID_TUNNEL_INTERFACE_RELEASE_OID object identifier (OID) is reserved for system use. Do not use it in your driver.

Requirements

Version: Windows 7 and later

Header: Ntddndis.h (include Ndis.h)

OID_TUNNEL_INTERFACE_SET_OID

Article • 02/18/2023

The OID_TUNNEL_INTERFACE_SET_OID object identifier (OID) is reserved for system use. Do not use it in your driver.

Requirements

Version: Windows 7 and later

Header: Ntddndis.h (include Ndis.h)

OID_WAN_CO_GET_COMP_INFO

Article • 02/18/2023

The OID_WAN_CO_GET_COMP_INFO OID requests the miniport driver to return information about the capabilities of the NIC or of its driver, in particular whether either supports compression. If so, the values returned are used to negotiate compression with the Point-to-Point Protocol (PPP) Compression Control Protocol. The protocol subsequently negotiates a PPP compression scheme with an OID_WAN_CO_SET_COMP_INFO request. This compression information is specific to a virtual connection (VC).

Compression information is returned in an NDIS_WAN_CO_GET_COMP_INFO structure, defined as follows:

ManagedCPlusPlus

```
    typedef struct _NDIS_WAN_CO_GET_COMP_INFO {
         OUT NDIS_WAN_COMPRESS_INFO SendCapabilities;
         OUT NDIS_WAN_COMPRESS_INFO RecvCapabilities;
    } NDIS_WAN_CO_GET_COMP_INFO,   *PNDIS_WAN_CO_GET_COMP_INFO;
```

The members of this structure contain the following information:

SendCapabilities
Specifies a structure containing information about compression capabilities for sending data.

RecvCapabilities
Specifies a structure containing information about compression capabilities for receiving data.

Remarks

For specifics of the NDIS_WAN_COMPRESS_INFO structure, see OID_WAN_GET_COMP_INFO.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.

Header	Ntddndis.h (include Ndis.h)

See also

OID_WAN_GET_COMP_INFO

OID_WAN_CO_SET_COMP_INFO

OID_WAN_CO_GET_INFO

Article • 02/18/2023

The OID_WAN_CO_GET_INFO OID requests the miniport driver to return information that applies to all virtual connections (VCs) on its NIC. This information is returned in an NDIS_WAN_CO_INFO structure, defined as follows.

ManagedCPlusPlus

```
    typedef struct _NDIS_WAN_CO_INFO {
         OUT ULONG MaxFrameSize;
         OUT ULONG MaxSendWindow;
         OUT ULONG FramingBits;
         OUT ULONG DesiredACCM;
    } NDIS_WAN_CO_INFO, *PNDIS_WAN_CO_INFO;
```

The members of this structure contain the following information:

MaxFrameSize
Specifies the maximum frame size for any net packet that the miniport driver can send and receive. This value should exclude the miniport driver's own framing overhead and/or the PPP HDLC overhead. Typically this value is around 1500.

However, all CoNDIS WAN miniport drivers should use an internal **MaxFrameSize** that is 32 bytes larger than the value they return for this OID. For example, a CoNDIS WAN miniport driver that returns 1500 for this OID should internally accept and send up to 1532. Such a miniport driver can readily support future bridging and additional protocols.

MaxSendWindow
Specifies the maximum number of outstanding packets that the CoNDIS WAN miniport driver can handle on a VC. This member must be set to at least one.

The NDISWAN driver uses the value of this member as a limit on how many packets it submits in send requests to the miniport driver's *MiniportCoSendPackets* function before NDISWAN holds send packets. These packets are queued until the miniport driver completes an outstanding send. A miniport driver can adjust this value dynamically and on a per-VC basis using the **SendWindow** member in the WAN_CO_LINKPARAMS structure that the miniport driver passes to NdisMCoIndicateStatus. NDISWAN uses the current **SendWindow** value as its limit on outstanding sends. If the miniport driver sets **SendWindow** to zero, NDISWAN must stop sending packets for the particular VC. That is, the miniport driver specifies that the send window is shut down, which, in effect, specifies that it cannot accept any packets from NDISWAN.

Because a CoNDIS WAN miniport driver must queue packets internally, the value of **MaxSendWindow** is theoretically **max**(ULONG). However, this driver-determined value should reflect the link speed or hardware capabilities of the NIC. For example, if a miniport driver's NIC always has room for at least four packets, the miniport driver sets **MaxSendWindow** to four so that any incoming packet to *MiniportCoSendPackets* can be placed on the hardware immediately.

FramingBits
A 32-bit value that specifies a bitmask specifying the types of framing the miniport driver supports. The miniport driver can specify a combination of the following values, using the binary OR operator:

RAS_FRAMING
Set only if the miniport driver can detect older RAS framing. Only legacy drivers that supported earlier RAS framing set this flag.

RAS_COMPRESSION
Set only if the miniport driver supports the older RAS compression scheme.

PPP_FRAMING
Should always be set. Indicates the miniport driver can detect and support PPP framing for its medium type.

PPP_COMPRESS_ADDRESS_CONTROL
Set if the miniport driver supports PPP address and control-field compression.

NDISWAN will remove the address and control field if this LCP option is negotiated. Some WAN medium types, such as X.25, do not support this option.

PPP_COMPRESS_PROTOCOL_FIELD
Set if the miniport driver supports PPP protocol field compression.

NDISWAN will remove one byte from the protocol field when applicable if this LCP option is negotiated.

PPP_ACCM_SUPPORTED
Set if the miniport driver supports Asynchronous Control Character Mapping. This bit is only valid for asynchronous media, such as modems. If this bit is set the **DesiredACCM** member should be valid.

PPP_MULTILINK_FRAMING
Set if the miniport driver supports multiple-link framing as specified in IETF RFC 1717.

PPP_SHORT_SEQUENCE_HDR_FORMAT
Set if the miniport driver supports header format for multiple-link framing as specified in

IETF RFC 1717.

SLIP_FRAMING
Set if the miniport driver can detect and support SLIP framing (asynchronous miniport drivers only).

SLIP_VJ_COMPRESSION
Set if the miniport driver can support Van Jacobsen TCP/IP header compression for SLIP. NDISWAN supports SLIP_VJ_COMPRESSION (with 16 slots). Asynchronous media (serial miniport drivers) that support SLIP framing should set this bit.

Asynchronous media need not write any code to support VJ header compression. NDISWAN will take care of it.

SLIP_VJ_AUTODETECT
Set if the miniport driver can auto-detect Van Jacobsen TCP/IP header compression for SLIP. NDISWAN will auto-detect VJ header compression. Asynchronous media (serial miniport drivers) should set this bit if they support SLIP framing.

TAPI_PROVIDER
Set if the miniport driver supports the TAPI Service Provider OIDs. Unless this bit is set, TAPI OID calls will not be made to the miniport driver.

MEDIA_NRZ_ENCODING
Set if the miniport driver supports NRZ encoding, the PPP default for some media types such as ISDN. This value is reserved for future use.

MEDIA_NRZI_ENCODING
Set if the miniport driver supports NRZI encoding. This value is reserved for future use.

MEDIA_NLPID
Set if the miniport driver has and can set the NLPID in its frame. This value is reserved for future use.

RFC_1356_FRAMING
Set if the miniport driver supports IETF RFC 1356 X.25 and ISDN framing. This value is reserved for future use.

RFC_1483_FRAMING
Set if the miniport driver supports IETF RFC 1483 ATM adaptation layer-5 encapsulation. This value is reserved for future use.

RFC_1490_FRAMING
Set if the miniport driver supports IETF RFC 1490 Frame Relay framing. This value is reserved for future use.

NBF_PRESERVE_MAC_ADDRESS
Set if the miniport driver supports IETF framing as specified in the draft "The PPP NETBIOS Frames Control Protocol (NBFCP)."

SHIVA_FRAMING
Superseded by NBF_PRESERVE_MAC_ADDRESS.

PASS_THROUGH_MODE
Set if the miniport driver does its own framing. If this flag is set, NDISWAN passes frames, uninterpreted and unmodified.

Miniport drivers must be in the default PPP framing mode until each miniport driver receives an OID_WAN_CO_SET_LINK_INFO request. The miniport driver must auto-detect any framing that it claims to support.

For example, miniport drivers that support old RAS framing must auto-detect RAS framing from PPP framing. If a miniport driver detects a framing scheme other than the default, that miniport driver should automatically switch its framing into the newly detected framing.

A subsequent query with OID_WAN_CO_GET_LINK_INFO should indicate the detected framing. If no framing is yet detected, the **FramingBits** should be zero in the returned NDIS_WAN_CO_GET_LINK_INFO information.

If the WAN miniport driver is called subsequently with OID_WAN_CO_SET_LINK_INFO in which the **FramingBits** member is zero, the miniport driver should attempt to auto-detect the framing upon reception of each frame.

DesiredACCM
The Asynchronous Control Character Map is negotiated. This member is relevant only for asynchronous media types.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

See also

NdisMCoIndicateStatus

OID_WAN_CO_GET_LINK_INFO

OID_WAN_CO_SET_LINK_INFO

WAN_CO_LINKPARAMS

OID_WAN_CO_GET_LINK_INFO

Article • 02/18/2023

The OID_WAN_CO_GET_LINK_INFO OID requests the miniport driver to return PPP framing information about the current state of a virtual connection (VC). This information is returned in an NDIS_WAN_CO_GET_LINK_INFO structure, defined as follows.

ManagedCPlusPlus

```
typedef struct _NDIS_WAN_CO_GET_LINK_INFO {
     OUT ULONG MaxSendFrameSize;
     OUT ULONG MaxRecvFrameSize;
     OUT ULONG SendFramingBits;
     OUT ULONG RecvFramingBits;
     OUT ULONG SendCompressionBits;
     OUT ULONG RecvCompressionBits;
     OUT ULONG SendACCM;
     OUT ULONG RecvACCM;
} NDIS_WAN_CO_GET_LINK_INFO,   *PNDIS_WAN_CO_GET_LINK_INFO;
```

The members of this structure contain the following information:

MaxSendFrameSize
Specifies the maximum buffer size, in bytes, that the miniport driver can accept for transmission on this VC. The miniport driver's *MiniportCoSendPackets* function can reject any incoming send packet that is larger than this size.

MaxRecvFrameSize
Specifies the largest packet that will be received from the network. The miniport driver can drop any packets that are larger.

SendFramingBits
Specifies send-framing bits indicating the type of framing that should be sent. If the miniport driver detects incompatibilities between **SendFramingBits** and **RecvFramingBits**, it returns NDIS_STATUS_INVALID_DATA.

The proper NLPID and framing format should be used based on the framing bits wherever applicable.

RecvFramingBits
Specifies receive-framing bits indicating the type of framing that should be received.

SendCompressionBits
Reserved.

RecvCompressionBits
Reserved.

SendACCM
For asynchronous media types, logical bits 0-31 indicate the respective byte to be byte stuffed. That is, if bit 0 is set to 1, then ASCII character 0x00 should be byte stuffed, and so forth.

RecvACCM
As described for **SendACCM**.

Remarks

Possible values for **SendFramingBits** and **RecvFramingBits** include any the driver returned in response to the OID_WAN_CO_GET_LINK_INFO query.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

See also

OID_WAN_CO_GET_LINK_INFO

OID_WAN_CO_GET_STATS_INFO

Article • 02/18/2023

The OID_WAN_CO_GET_STATS_INFO OID requests the miniport driver to return statistics information that is specific to a virtual connection (VC). A WAN miniport driver is expected to keep statistics and to return these statistics for this OID in an NDIS_WAN_CO_GET_STATS_INFO structure, defined as follows:

ManagedCPlusPlus

```
    typedef struct _NDIS_WAN_CO_GET_STATS_INFO {
         OUT ULONG BytesSent;
         OUT ULONG BytesRcvd;
         OUT ULONG FramesSent;
         OUT ULONG FramesRcvd;
         OUT ULONG CRCErrors;
         OUT ULONG TimeoutErrors;
         OUT ULONG AlignmentErrors;
         OUT ULONG SerialOverrunErrors;
         OUT ULONG FramingErrors;
         OUT ULONG BufferOverrunErrors;
         OUT ULONG BytesTransmittedUncompressed;
         OUT ULONG BytesReceivedUncompressed;
         OUT ULONG BytesTransmittedCompressed;
         OUT ULONG BytesReceivedCompressed;
    } NDIS_WAN_CO_GET_STATS_INFO,   *PNDIS_WAN_CO_GET_STATS_INFO;
```

The members of this structure contain the following information:

BytesSent
Specifies the number of bytes transmitted.

BytesRcvd
Specifies the number of bytes received.

FramesSent
Specifies the number of frames (WAN packets) sent.

FramesRcvd
Specifies the number of frames received.

CRCErrors
Specifies the number of CRC errors encountered for this VC. CRC errors are caused by the failure of a cyclic redundancy check. A CRC error indicates that one or more bytes in the frame received were found garbled on arrival.

TimeoutErrors
Specifies the number of time-out errors encountered for this VC. Time-out errors occur when an expected byte is not received in time.

AlignmentErrors
Specifies the number of alignment errors encountered for this VC. Alignment errors occur when a byte received is different from the byte expected. This typically happens when a byte is lost or when a time-out error occurs.

SerialOverrunErrors
Specifies the number of serial overruns encountered for this VC. Serial overruns occur when the WAN NIC cannot handle the rate at which data is received.

FramingErrors
Specifies the number of framing errors encountered for this VC. A framing error occurs when an asynchronous byte is received with an invalid start or stop bit.

BufferOverrunErrors
Specifies the number of buffer overruns encountered for this VC. Buffer overruns occur when the WAN miniport driver cannot handle the rate at which data is received.

BytesTransmittedUncompressed
Specifies the number of bytes of uncompressed data transmitted. A miniport driver returns a nonzero value only if it supports compression.

BytesReceivedUncompressed
Specifies the number of bytes of uncompressed data received. A miniport driver returns a nonzero value only if it supports compression.

BytesTransmittedCompressed
Specifies the number of bytes of compressed data transmitted. A miniport driver returns a nonzero value only if it supports compression.

BytesReceivedCompressed
Specifies the number of bytes of compressed data received. A miniport driver returns a nonzero value only if it supports compression.

Remarks

If the underlying driver or its NIC does not support compression, the driver returns zero for the **Bytes..Uncompressed/Compressed** members.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

OID_WAN_CO_SET_COMP_INFO

Article • 02/18/2023

The OID_WAN_CO_SET_COMP_INFO OID notifies the miniport driver of the PPP compression scheme selected by a protocol to which the miniport driver already returned information with a OID_WAN_CO_GET_COMP_INFO query. This PPP compression scheme is specific to a virtual connection (VC).

The protocol supplies a specification for the PPP compression scheme it selected in an NDIS_WAN_CO_SET_COMP_INFO structure, defined as follows:

ManagedCPlusPlus

```
    typedef struct _NDIS_WAN_CO_SET_COMP_INFO {
         IN NDIS_WAN_COMPRESS_INFO SendCapabilities;
         IN NDIS_WAN_COMPRESS_INFO RecvCapabilities;
    } NDIS_WAN_CO_SET_COMP_INFO,   *PNDIS_WAN_CO_SET_COMP_INFO;
```

The members of this structure contain the following information:

SendCapabilities
Specifies a structure containing information about compression capabilities for sending data.

RecvCapabilities
Specifies a structure containing information about compression capabilities for receiving data.

Remarks

For specifics of the NDIS_WAN_COMPRESS_INFO structure, see OID_WAN_GET_COMP_INFO.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

See also

OID_WAN_CO_GET_COMP_INFO

OID_WAN_CO_SET_LINK_INFO

Article • 02/18/2023

The OID_WAN_CO_SET_LINK_INFO OID requests the miniport driver to set PPP framing information for a specific virtual connection (VC). A protocol uses an NDIS_WAN_CO_SET_LINK_INFO structure, defined as follows, to indicate this PPP framing information.

ManagedCPlusPlus

```
    typedef struct _NDIS_WAN_CO_SET_LINK_INFO {
         IN ULONG MaxSendFrameSize;
         IN ULONG MaxRecvFrameSize;
         IN ULONG SendFramingBits;
         IN ULONG RecvFramingBits;
         IN ULONG SendCompressionBits;
         IN ULONG RecvCompressionBits;
         IN ULONG SendACCM;
         IN ULONG RecvACCM;
    } NDIS_WAN_CO_SET_LINK_INFO,   *PNDIS_WAN_CO_SET_LINK_INFO;
```

The members of this structure contain the following information:

MaxSendFrameSize
Specifies the largest buffer, in bytes, the protocol will send for this VC. This value must be less than or equal to that returned by the miniport driver for the OID_WAN_CO_GET_LINK_INFO query.

The miniport driver's *MiniportCoSendPackets* function can reject any send packets submitted for this link that are larger than this value.

MaxRecvFrameSize
Specifies the largest network packet that the protocol will receive subsequently. This value must be less than or equal to that returned by the miniport driver for the OID_WAN_CO_GET_LINK_INFO query. The miniport driver can drop any received packets for this VC that are larger.

SendFramingBits
Specifies send-framing bits indicating the type of framing that should be sent. If the miniport driver detects incompatibilities between **SendFramingBits** and **RecvFramingBits**, it returns NDIS_STATUS_INVALID_DATA.

The proper NLPID and framing format should be used based on the framing bits wherever applicable.

RecvFramingBits

Specifies receive-framing bits indicating the type of framing that should be received.

SendCompressionBits

Reserved.

RecvCompressionBits

Reserved.

SendACCM

For asynchronous media types, logical bits 0-31 indicate the respective byte to be byte stuffed. That is, if bit 0 is set to one then ASCII character 0x00 should be byte stuffed, and so forth.

RecvACCM

As described for **SendACCM**.

Remarks

Possible values for **SendFramingBits** and **RecvFramingBits** include any the underlying driver returned in response to the OID_WAN_CO_GET_INFO query.

Requirements

Version	Supported for NDIS 6.0 and NDIS 5.1 drivers in Windows Vista. Supported for NDIS 5.1 drivers in Windows XP.
Header	Ntddndis.h (include Ndis.h)

See also

OID_WAN_CO_GET_INFO

OID_WAN_CO_GET_LINK_INFO

OID_WWAN_AUTH_CHALLENGE

Article • 02/18/2023

OID_WWAN_AUTH_CHALLENGE sends an authentication challenge to the MB device, or Subscriber Identity Module (SIM) card, to obtain the response from the SIM.

Set requests are not supported.

This is an optional OID. When miniport drivers implement it, they must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_AUTHENTICATION_RESPONSE status notification containing an NDIS_WWAN_AUTHENTICATION_RESPONSE structure to provide the authentication keys requested based on challenges by the caller when completing query requests.

Remarks

When processing this OID, miniport drivers can access the SIM card, but should not access the provider network. This OID must work even in Radio OFF or Airplane Mode.

OID_WWAN_AUTH_CHALLENGE supports both second-generation and third-generation mobile networks. SIM specifies an authentication mechanism that is based on the GSM authentication and key agreement primitives, which is a second-generation mobile network standard. AKA and AKA' uses the third-generation Authentication and Key Agreement mechanism, specified for Universal Mobile Telecommunications System (UMTS) in [TS33.102] and for CDMA2000 in [S.S0055-A] depending on the capabilities of the device.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support returning one or all authentication methods.

Requirements

Version	Supported starting with Windows 8.
Header	Ntddndis.h (include Ndis.h)

OID_WWAN_BASE_STATIONS_INFO

Article • 02/18/2023

OID_WWAN_BASE_STATIONS_INFO retrieves information about the serving and neighboring cells known to the modem. For more info about cellular base station information query, see MB base stations information query support.

For query requests, OID_WWAN_BASE_STATIONS_INFO uses the NDIS_WWAN_BASE_STATIONS_INFO_REQ structure, which in turn contains a WWAN_BASE_STATIONS_INFO structure that specifies aspects of the cell information, such as the maximum number of neighbor cell measurements, to send in response. Modem miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_BASE_STATIONS_INFO notification containing an NDIS_WWAN_BASE_STATIONS_INFO structure, which in turn contains a WWAN_BASE_STATIONS_INFO structure that provides information about both serving and neighboring base stations.

Set requests are not applicable.

Unsolicited events are not applicable.

Requirements

Version: Windows 10, version 1709 **Header**: Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_BASE_STATIONS_INFO_REQ

NDIS_STATUS_WWAN_BASE_STATIONS_INFO

NDIS_WWAN_BASE_STATIONS_INFO

WWAN_BASE_STATIONS_INFO

MB base stations information query support

OID_WWAN_CONNECT

Article • 02/18/2023

OID_WWAN_CONNECT activates or deactivates a particular packet context and reads the activation state of a context.

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_CONTEXT_STATE status notification containing an NDIS_WWAN_CONTEXT_STATE structure that indicates the Packet Data Protocol (PDP) context state of the MB device regardless of completing set or query requests.

Callers requesting to set the Packet Data Protocol (PDP) context state of the MB device provide an NDIS_WWAN_SET_CONTEXT_STATE structure to the miniport driver with the appropriate information.

Remarks

For more information about using this OID, see WWAN Packet Context Management.

This object activates or deactivates a particular packet context and reads the activation state of a context. The miniport driver must send appropriate event notifications whenever the activation state changes.

This object is called only if the miniport driver is in a register state of **WwanRegisterStateHome**, **WwanRegisterStatePartner**, or **WwanRegisterStateRoaming**. When packet service is active, the device must also be in an attach state of **WwanPacketServiceStateAttached**.

Both set and query operations are supported for this object.

- Processing of a set request requires network access but not SIM access.
- Processing of a query request does not require access to network or the SIM.

The data structure for this OID is NDIS_WWAN_SET_CONTEXT_STATE. The miniport driver issues a status indication of NDIS_STATUS_WWAN_CONTEXT_STATE for both set and query requests.

In this version of the driver model, the miniport driver attempts context activation only as instructed by the MB Service. (Miniport drivers may activate a context initiated by the network in later versions.) Miniport drivers must not automatically activate a context even after losing registration or a signal. If the access string is not provided in the activation request, a miniport driver should not attempt to provide a default string. Instead, it must proceed with activating the context with a blank access string.

On the other hand, the miniport driver may deactivate a context as instructed by the MB service. This may occur when network connectivity has been lost for a period that exceeds the threshold of temporary loss of signal, or as part of a graceful shutdown or state cleanup.

Since only one activated context is supported in this version, activating or deactivating a particular context amounts to setting up or tearing down the layer-2 MB connection.

On set requests, the MB service furnishes both **ConnectionId** and **ActivationCommand** parameters in the WWAN_CONTEXT_STATE data structure. It instructs the miniport driver to activate or deactivate a packet context identified by **ConnectionId**, based on the **ActivationCommand** parameter value *WwanActivationCommandActivate* or *WwanActivationCommandDeactivate*.

- If the service or subscription requires activation, the miniport driver should return error code WWAN_STATUS_SERVICE_NOT_ACTIVATED. The PDP-activation may not happen until the service or subscription is activated. All the emergency services might be available subject to the support from the device and the operator. The operating system might call the OID_WWAN_SERVICE_ACTIVATION in response to this error code.

- If the miniport driver receives a context activation request while another packet context is currently activated, it returns error code WWAN_STATUS_MAX_ACTIVATED_CONTEXTS.

- If the miniport driver receives a context deactivation request but the context identified by **ConnectionId** is not currently activated, it returns error code WWAN_STATUS_CONTEXT_NOT_ACTIVATED.

The miniport driver uses the following logic to determine the validity of AccessString, UserName, and Password settings from a set request:

- If **ActivationCommand** is *WwanActivationCommandDeactivate*, the miniport driver should ignore the settings of these three parameters. The rest of the cases only consider the case when **ActivationCommand** is *WwanActivationCommandActivate*.

Context activation persists across user logon and logoff. It is not per logon user.

On query requests, the MB Service uses this object to find out the activation state.

For response to query requests, miniport driver sends the NDIS_STATUS_WWAN_CONTEXT_STATE notification.

Important Note:

In rare, but specific circumstances, the MB Service on Windows 7 may attempt to auto-connect before connectivity to the Internet has been determined for pre-existing connections or during a momentary disruption in Internet connectivity of pre-existing connections. This could result in simultaneous MB and WLAN/Ethernet connections. For example, this can occur during system boot when MB and other connections are attempted simultaneously and the Network List Manager service is still attempting to determine the Internet connectivity of other connections using active and passive methods. It could also occur due to temporary outages in network infrastructure like a corporate proxy server or an ISP network. Thus, the MB Service may attempt to auto-connect to the internet regardless of whether the "Auto-connect only if no alternate Internet connection is available" option is selected.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

WWAN Packet Context Management

OID_WWAN_CREATE_MAC

Article • 02/18/2023

OID_WWAN_CREATE_MAC requests the miniport driver to create a new NDIS port. Context activation requests for the additional PDP context will be sent on this new NDIS port.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later completing the request with the **NDIS_WWAN_MAC_INFO** structure that indicates the NDIS port number and MAC address associated with the port.

Query requests are not supported.

Remarks

Miniport drivers must process requests to create (activate) new NDIS ports asynchronously in order to prevent deadlocks.

Requirements

Version	Available in Windows 8.1 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_MAC_INFO

OID_WWAN_DELETE_MAC

OID_WWAN_DELETE_MAC

Article • 02/18/2023

OID_WWAN_DELETE_MAC requests the miniport driver to delete the NDIS port specified in the NDIS_WWAN_MAC_INFO parameter. The NDIS port should have been created earlier using OID_WWAN_CREATE_MAC.

Miniport drivers must process the set request asynchronously, initially returning NDIS_STATUS_PENDING to the original request, and later completing the request with NDIS_STATUS_SUCCESS.

Query requests are not supported.

Remarks

Miniport drivers must process requests to delete (deactivate) NDIS ports asynchronously in order to prevent deadlocks.

OID_WWAN_DELETE_MAC requests sent to delete the default port will fail with the NDIS status error code NDIS_STATUS_INVALID_PORT.

Upon receiving an OID_WWAN_DELETE_MAC request, miniport drivers should deactivate the PDP context associated with the port, if it has not already been deactivated. This is because a surprise removal event could occur. Deactivating the PDP context at such time will ensure that the modem and the miniport driver remain in a good state.

When the driver receives a surprise removal, the driver blocks and cancels all further OIDs. This means that the driver filters out OID_WWAN_DELETE_MAC even though Windows sends a call with OID_WWAN_DELETE_MAC as part of the *FILTER_DETACH* call.

Requirements

Version	Available in Windows 8.1 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

OID_WWAN_CREATE_MAC

OID_WWAN_DEVICE_CAPS

Article • 02/18/2023

OID_WWAN_DEVICE_CAPS returns the capabilities of the MB device, including the cellular technology it supports, the classes of packet data it supports, the radio frequencies it supports, the type of voice service it provides, and whether it uses a Subscriber Identity Module (SIM card). The supported cellular technology and whether the device uses a SIM are particularly important because network provider selection and SIM user interfaces depend on the values of these two capabilities. The manufacturer and firmware revision are returned as optional fields.

Set requests are not supported.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a **NDIS_STATUS_WWAN_DEVICE_CAPS** status notification containing a **NDIS_WWAN_DEVICE_CAPS** structure that indicates the capabilities of the MB device when completing query requests.

Remarks

Starting with Windows 8, the MB driver model has been updated to version 2.0. Windows 8 miniport drivers should set the **Header.Revision** member of the **NDIS_WWAN_DEVICE_CAPS** structure to **NDIS_WWAN_DEVICE_CAPS_REVISION_2** for *query* requests. Windows 7 miniport drivers should set the **Header.Revision** member of the **NDIS_WWAN_DEVICE_CAPS** structure to **NDIS_WWAN_DEVICE_CAPS_REVISION_1** for *query* requests.

For more information about using this OID, see WWAN Driver Initialization Procedure.

Miniport drivers can access device memory when processing query operations, but should not access the provider network or the Subscriber Identity Module (SIM card).

Many "world-wide" MB devices today support multiple frequency bands because the frequency bands for 2.5G/3G vary from country to country. The list of all the radio frequencies specified in the 3GPP standards (for GSM-based networks) and 3GPP2 standards (for CDMA-based networks) is shown in the following tables. Both standards adopt a similar band classification scheme.

3GPP (GSM-based) Frequency Band Classes

3GPP band	Designated spectrum	Industry name	Uplink (MS to BTS)	Downlink (BTS to MS)	Regions
Band I	UMTS2100	IMT	1920-1980	2110-2170	Europe, Korea, Japan, China
Band II	UMTS1900	PCS1900	1850-1910	1930-1990	North America, LATAM
Band III	UMTS1800	DCS1800	1710-1785	1805-1880	Europe, China
Band IV	AWS	AWS, 1.7/2.1	1710-1755	2110-2155	North America, LATAM
Band V	UMTS850	GSM850	824-849	869-894	North America, LATAM
Band VI	UMTS800	UMTS800	830-840	875-885	Japan
Band VII	UMTS2600	UMTS2600	2500-2570	2620-2690	Europe
Band VIII	UMTS900	EGSM900	880-915	925-960	Europe, China
Band IX	UMTS1700	UMTS1700	1750-1785	1845-1880	Japan
Band X			1710-1770	2110-2170	

3GPP2 (CDMA-based) Frequency Band Classes

3GPP band Industry name Uplink (MS to BTS) Downlink (BTS to MS) Band 0

800MHz Cellular

824.025-844.995

869.025-889.995

Band I

1900MHz Band

1850-1910

1930-1990

Band II

TACS Band

872.025-914.9875

917.0125-959.9875

Band III

JTACS Band

887.0125-924.9875

832.0125-869.9875

Band IV

Korean PCS Band

1750 - 1780

1840 - 1870

Band V

450 MHz Band

410 - 483.475

420 - 493.475

Band VI

2 GHz Band

1920 - 1979.950

2110 - 2169.950

Band VII

700 MHz Band

776 - 794

746 - 764

Band VIII

1800 MHz Band

1710 - 1784.950

1805 - 1879.95

Band IX

900 MHz Band

880 - 914.950

925 - 959.950

Band X

Secondary 800 MHz Band

806 - 900.975

851 - 939.975

Band XI

400 MHz European PAMR Band

410 - 483.475

420 - 493.475

Band XII

800 MHz PAMR Band

870.125 - 875.9875

915.0125 - 920.9875

Band XIII

2.5GHz IMT2000 Extension Band

2500 - 2570

2620 - 2690

Band XIV

US PCS 1.9GHz Band

1850 - 1915

1930 - 1995

Band XV

AWS Band

1710 - 1755

2110 - 2155

Band XVI

US 2.5GHz Band

2502 - 2568

2624 - 2690

Band XVII

US 2.5 GHz Forward Link Only Band

2624-2690

The unit for radio frequency bands in both tables is megahertz (MHz).

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_DEVICE_CAPS

WWAN Driver Initialization Procedure

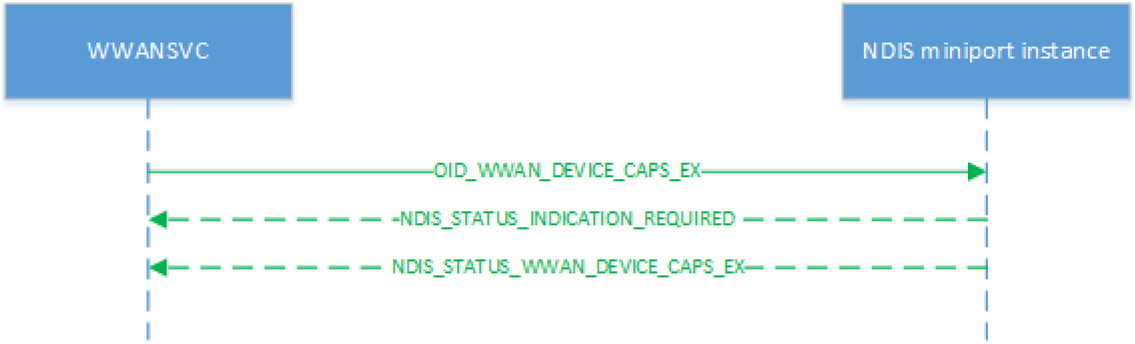
OID_WWAN_DEVICE_CAPS_EX

Article • 02/18/2023

OID_WWAN_DEVICE_CAPS_EX is similar to OID_WWAN_DEVICE_CAPS but is a per-executor OID, unlike OID_WWAN_DEVICE_CAPS which is a per-device OID. This OID serves to indicate the hardware's device/executor capability, including the capability on extended optional features such as LTE attach APN configuration.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an **NDIS_STATUS_WWAN_DEVICE_CAPS_EX** status notification containing an **NDIS_WWAN_DEVICE_CAPS_EX** structure, which in turn contains a **WWAN_DEVICE_CAPS_EX** structure, to provide information about the device's capability.

The following diagram illustrates a query request.


WWANSVC
NDIS miniport instance
OID_WWAN_DEVICE_CAPS_EX
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_DEVICE_CAPS_EX

Set requests are not applicable.

Remarks

It is critical for the driver to report service extension capability as a whole including from the driver to the actual device. If a driver supports a service but it is not supported by the underlying hardware, then the service capabilities should be marked as FALSE.

OID_WWAN_DEVICE_CAPS_EX is also used to retrieve each executor's capability. This OID is the same in structure as existing OID_WWAN_DEVICE_CAPS but with the addition of **Executor ID**. A miniport driver should report the highest OID version it supports.

Just as with OID_WWAN_DEVICE_CAPS, the parameters in this OID are not expected to change due to SIM cards but rather represent the modem's RF capability of the selected executor. A physical hardware modem may have multiple executors and thus may have multiple interfaces that support OID_WWAN_DEVICE_CAPS_EX.

For possible future updates, if the OS's requested version is newer than the device-supported version, the device should return the newest version of the OID structure it supports. If the OS's requested version is older than the latest one supported by the device, then the device should return the version matching the OS's specification. It is a requirement for IHVs to make sure all revisions of OID_WWAN_DEVICE_CAPS_EX are supported for backwards compatibility and legacy support.

Unlike other OIDs new to Windows 10 Version 1703 that are only required if the modem supports multi-SIM/multi-executors, this OID must be implemented for modems that would like to support any Microsoft-defined service extensions starting in Windows 10 Version 1703.

Versions of Windows prior to Windows 10 Version 1703 may still use the existing OID_WWAN_DEVICE_CAPS; their behavior with multi-executor capable modems is not a supported scenario. IHVs must define this behavior.

Windows 10, version 1903

Starting in Windows 10, version 1903, OID_WWAN_DEVICE_CAPS_EX has been upgraded to revision 2. A miniport driver must use revision 2 of this OID and the data structures it contains if the miniport driver supports 5G.

When the host queries capabilities using this OID, the miniport driver must check if the underlying hardware supports 5G cellular capabilities. If it does, the miniport driver sets the bitmask in the **WwanDataClass** field of the **WWAN_DEVICE_CAPS_EX** structure according to hardware capabilties.

Additionally, in the **WwanOptionalServiceCaps** field of the **WWAN_DEVICE_CAPS_EX** structure, a new optional service bit is defined that covers support of all new 5G-related extensions.

For more info about 5G data class support, see MB 5G data class support.

Requirements

Version	Windows 10, version 1703
Header	Ntddndis.h (include Ndis.h)

See also

OID_WWAN_DEVICE_CAPS

NDIS_STATUS_WWAN_DEVICE_CAPS_EX

NDIS_WWAN_DEVICE_CAPS_EX

WWAN_DEVICE_CAPS_EX

OID_WWAN_DEVICE_RESET

Article • 02/18/2023

OID_WWAN_DEVICE_RESET is sent by the mobile broadband host to a modem miniport adapter to reset the modem device.

Query requests are not applicable.

For Set requests, OID_WWAN_DEVICE_RESET uses the NDIS_WWAN_SET_DEVICE_RESET structure. Modem miniport drivers must respond to set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_DEVICE_RESET_STATUS notification containing an NDIS_WWAN_DEVICE_RESET_STATUS structure that represents the reset status of the modem device. This response does not contain a payload, but is always a status code from the modem such as *WWAN_STATUS_SUCCESS* or *WWAN_STATUS_BUSY*.

Unsolicited events are not applicable.

Requirements

Version: Windows 10, version 1709 **Header**: Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_DEVICE_RESET_STATUS

NDIS_WWAN_DEVICE_RESET_STATUS

NDIS_WWAN_SET_DEVICE_RESET

MB modem reset operations

OID_WWAN_DEVICE_SERVICE_COMMAND

Article • 02/18/2023

OID_WWAN_DEVICE_SERVICE_COMMAND allows miniport drivers to implement vendor specific commands.

Both query and set requests are supported.

Miniport drivers must process query and set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE status notification containing a vendor-defined structure (NDIS_WWAN_DEVICE_SERVICE_COMMAND) to provide responses when they have completed the transaction.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support the specified device service or operation.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_DEVICE_SERVICE_RESPONSE

NDIS_WWAN_DEVICE_SERVICE_COMMAND

OID_WWAN_DEVICE_SERVICE_SESSION

Article • 02/18/2023

OID_WWAN_DEVICE_SERVICE_SESSION directs a miniport driver to open or close a device service session.

Query requests are not supported.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION status notification containing a NDIS_WWAN_SET_DEVICE_SERVICE_SESSION structure that describes the result of the operation.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support specified device service or operation.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SET_DEVICE_SERVICE_SESSION

NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION

OID_WWAN_DEVICE_SERVICE_SESSION_WRITE

Article • 02/18/2023

OID_WWAN_DEVICE_SERVICE_SESSION_WRITE directs the miniport driver to write data to the MB device for a device service session.

Query requests are not supported.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION_WRITE_COMPLETE status notification containing a NDIS_WWAN_DEVICE_SERVICE_SESSION_WRITE_COMPLETE structure that describes the completion status of the operation.

Miniport drivers should return NDIS_STATUS_ADAPTER_NOT_OPEN if the device service session is not open.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_DEVICE_SERVICE_SESSION_WRITE_COMPLETE

NDIS_WWAN_DEVICE_SERVICE_SESSION_WRITE

OID_WWAN_DEVICE_SERVICES

Article • 02/18/2023

OID_WWAN_DEVICE_SERVICES returns the list of device services supported by the miniport driver.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_DEVICE_SERVICES status notification containing a NDIS_WWAN_DEVICE_SERVICES structure that indicates the supported device service GUIDs.

Set requests are not supported.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

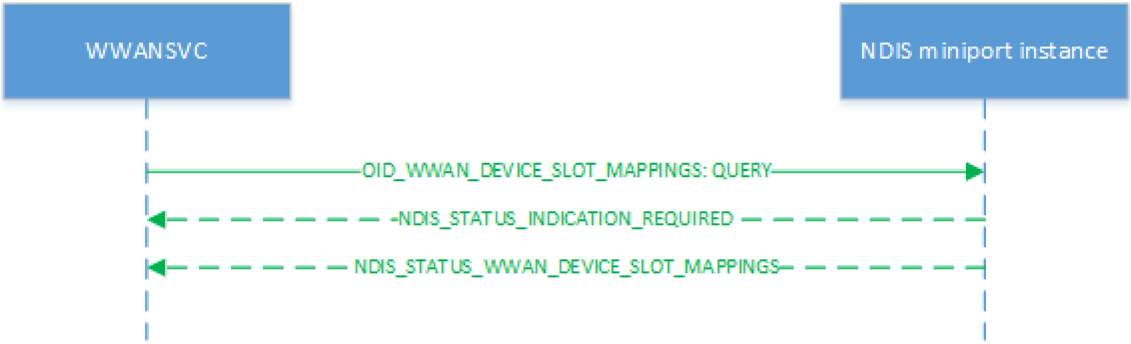
OID_WWAN_DEVICE_SLOT_MAPPING_INFO

Article • 02/18/2023

OID_WWAN_DEVICE_SLOT_MAPPING_INFO sets or returns the device-slot mappings of the MB device (i.e. the executor-slot mappings).

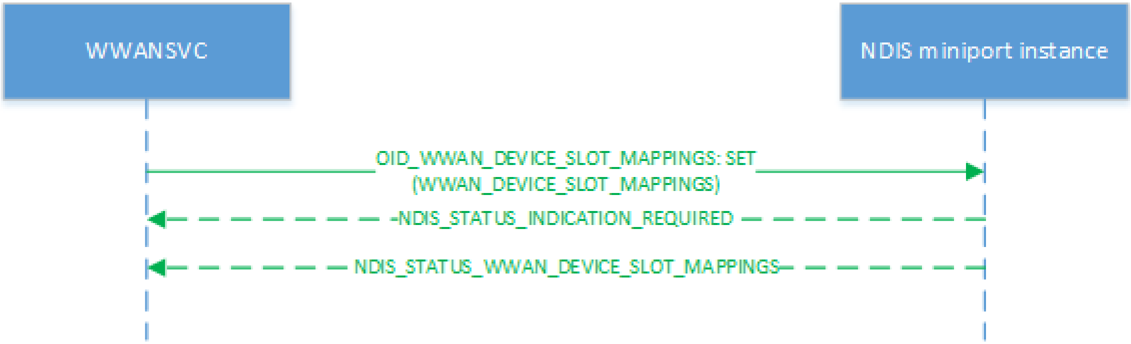
Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO status notification containing an NDIS_WWAN_DEVICE_SLOT_MAPPING_INFO structure to provide information on the executor-to-slot mappings.

The following diagram illustrates a query request.


WWANSVC
NDIS miniport instance
OID_WWAN_DEVICE_SLOT_MAPPINGS: QUERY
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPINGS

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO status notification containing an NDIS_WWAN_DEVICE_SLOT_MAPPING_INFO structure, which in turn contains a WWAN_DEVICE_SLOT_MAPPING_INFO structure to indicate the current mapping status. This holds true even if the set request failed. The structure for set requests for OID_WWAN_DEVICE_SLOT_MAPPING_INFO is NDIS_WWAN_SET_DEVICE_SLOT_MAPPING_INFO.

The following diagram illustrates a set request.


WWANSVC
NDIS miniport instance
OID_WWAN_DEVICE_SLOT_MAPPINGS: SET
(WWAN_DEVICE_SLOT_MAPPINGS)
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPINGS

Remarks

The host expects that on first boot, the modem would have a default mapping between slots and executors. The host performs a SET operation with OID_WWAN_DEVICE_SLOT_MAPPING_INFO to define the slot that is bound to each executor. The host expects the modem to maintain the mapping across reboots and removals/insertions. This OID is not executor-specific and may be sent to any NDIS instance on the device. It may also query the current mapping as shown above.

Requirements

Version	Windows 10, version 1703
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_DEVICE_SLOT_MAPPING_INFO

NDIS_WWAN_DEVICE_SLOT_MAPPING_INFO

NDIS_WWAN_SET_DEVICE_SLOT_MAPPING_INFO

OID_WWAN_DRIVER_CAPS

Article • 02/18/2023

OID_WWAN_DRIVER_CAPS returns the version of the MB driver model supported by the miniport driver.

Set requests are not supported.

Miniport drivers process OID_WWAN_DRIVER_CAPS synchronously and should immediately return with the response buffer containing an **NDIS_WWAN_DRIVER_CAPS** structure that describes the version of the MB driver model implemented by the miniport driver when completing query requests.

Remarks

For more information about using this OID, see MB Miniport Driver Initialization.

Miniport drivers should not access the provider network, or the Subscriber Identity Module (SIM card), when processing query operations.

The current version of the MB driver model version is defined by the WWAN_MAJOR_VERSION and WWAN_MINOR_VERSION #define tokens. If the miniport driver returns a version of the MB driver model that the MB Service does not support, the MB Service will ignore the device.

When the MB Service is initialized or restarted, the miniport driver may already have been loaded. In this case, the MB Service queries the version of the MB driver model implement by miniport driver before it proceeds to issue any other OIDs. This occurs at the beginning of any session.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED in the case of any initialization error. If a miniport driver returns NDIS_STATUS_NOT_SUPPORTED, the MB Service will ignore the device and will not proceed with any other OIDs.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

MB Miniport Driver Initialization

NDIS_WWAN_DRIVER_CAPS

OID_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS

Article • 02/18/2023

OID_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS returns a list of commands supported for a device service.

Set requests are not supported.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS status notification containing a NDIS_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS structure that describes the result of the operation.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support specified device service or operation.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS

NDIS_WWAN_ENUMERATE_DEVICE_SERVICE_COMMANDS

OID_WWAN_ENUMERATE_DEVICE_SERVICES

Article • 02/18/2023

OID_WWAN_ENUMERATE_DEVICE_SERVICES returns the list of device services supported by the miniport driver.

Set requests are not supported.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS status notification containing a NDIS_WWAN_SUPPORTED_DEVICE_SERVICES structure that provides the list of supported device service GUIDs.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_DEVICE_SERVICE_SUPPORTED_COMMANDS

NDIS_WWAN_SUPPORTED_DEVICE_SERVICES

OID_WWAN_HOME_PROVIDER

Article • 02/18/2023

OID_WWAN_HOME_PROVIDER is used to set and retrieve information about the home provider of the cellular service subscription. For GSM-based devices and CDMA-based device with U-RIM, this information should be stored on the Subscriber Identity Module (SIM card). For CDMA-based devices without U-RIM, this information should be stored in auxiliary device memory.

Windows 8 supports both *set* and *query* requests. Windows 7 supports only *query* requests.

Miniport drivers must process both *set* and *query* requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request and later sending a NDIS_STATUS_WWAN_HOME_PROVIDER status notification, for a *query*, or NDIS_STATUS_WWAN_SET_HOME_PROVIDER_COMPLETE status notification, for *set*, containing an NDIS_WWAN_HOME_PROVIDER structure to return information about the home network provider with the **Provider.ProviderState** member of the NDIS_WWAN_HOME_PROVIDER structure set to WWAN_PROVIDER_STATE_HOME.

Set operations are only required to be supported by multi-carrier capable devices. The MB service will only *set* the home provider to multi-carrier providers reported by the miniport via OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS or OID_WWAN_VISIBLE_PROVIDERS. *Set* operations have an input buffer of NDIS_WWAN_SET_HOME_PROVIDER.

Remarks

A *set* operation should not require the user to unlock the device regardless if the current SIM or target SIM is in a locked state.

Current Provider SIM	Target Provider SIM	Result of *set* HOME_PROVIDER
-	Locked	Target PIN not required for setting home provider
Locked	-	Source PIN not required for setting home provider
Locked	Locked	Source and Target PIN not required for setting home provider

For more information about using this OID, see WWAN Provider Operations.

Miniport drivers can access the Subscriber Identity Module (SIM card) when processing query operations, but should not access the provider network.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_HOME_PROVIDER

NDIS_STATUS_WWAN_HOME_PROVIDER

WWAN Provider Operations

OID_WWAN_LTE_ATTACH_CONFIG

Article • 02/18/2023

OID_WWAN_LTE_ATTACH_CONFIG enables the operating system to query or set the default LTE attach context of the inserted SIM's provider (MCC/MNC pair).

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_LTE_ATTACH_CONFIG status notification containing an **NDIS_WWAN_LTE_ATTACH_CONTEXTS** structure that describes the LTE attach configuration.

For Set requests, this OID's payload contains an **NDIS_WWAN_SET_LTE_ATTACH_CONTEXT** structure that describes LTE attach context information for the modem to set.

Remarks

After each Query or Set request, the miniport driver should return an NDIS_STATUS_WWAN_LTE_ATTACH_CONFIG notification that describes the LTE attach configuration.

For more information about using this OID, see MBIM_CID_MS_LTE_ATTACH_CONFIG.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB LTE Attach Operations

NDIS_STATUS_WWAN_LTE_ATTACH_CONFIG

NDIS_WWAN_LTE_ATTACH_CONTEXTS

NDIS_WWAN_SET_LTE_ATTACH_CONTEXT

OID_WWAN_LTE_ATTACH_STATUS

Article • 02/18/2023

OID_WWAN_LTE_ATTACH_STATUS is used to inform the OS of the last used default LTE attach context.

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_LTE_ATTACH_STATUS notification containing an **NDIS_WWAN_LTE_ATTACH_STATUS** structure that describes the last used default LTE attach context.

Set requests are not applicable.

Remarks

For more information about using this OID, see MBIM_CID_MS_LTE_ATTACH_STATUS.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB LTE Attach Operations

NDIS_STATUS_WWAN_LTE_ATTACH_STATUS

NDIS_WWAN_LTE_ATTACH_STATUS

OID_WWAN_MODEM_CONFIG_INFO

Article • 02/18/2023

OID_WWAN_MODEM_CONFIG_INFO retrieves information about the modem configuration information.

MBB drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_MODEM_CONFIG_INFO status notification containing an **NDIS_WWAN_MODEM_CONFIG_INFO** structure, which in turn contains a **WWAN_MODEM_CONFIG_INFO** structure, to provide information about the modem's configuration.

Set requests are not applicable.

Remarks

The MBB driver may not have valid information yet from the modem during early queries. The non-valid information will be set to zero.

Requirements

Version	the next major update to Windows 10
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_MODEM_CONFIG_INFO

NDIS_WWAN_MODEM_CONFIG_INFO

WWAN_MODEM_CONFIG_INFO

OID_WWAN_MODEM_LOGGING_CONFIG

Article • 02/18/2023

OID_WWAN_MODEM_LOGGING_CONFIG is used to configure logs that are collected by the modem and how often they will be sent from the modem to the host over Data Service Stream (DSS).

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_MODEM_LOGGING_CONFIG status notification containing an **NDIS_WWAN_MODEM_LOGGING_CONFIG** structure that describes the current modem logging configuration.

Set payloads contain an **NDIS_WWAN_SET_MODEM_LOGGING_CONFIG** structure specifying how to configure modem logging. Miniport drivers must process Set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_MODEM_LOGGING_CONFIG status notification containing an **NDIS_WWAN_MODEM_LOGGING_CONFIG** structure that describes the modem logging configuration after the Set request.

Remarks

Logging must be configured before a logging session is started. This is an optional OID for miniport drivers to support. However, if the miniport driver supports modem logging via the DSS channel, it must specify that it supports this OID.

For more information about usage of this OID, see MB modem logging with DSS.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB modem logging with DSS

NDIS_STATUS_WWAN_MODEM_LOGGING_CONFIG

NDIS_WWAN_SET_MODEM_LOGGING_CONFIG

NDIS_WWAN_MODEM_LOGGING_CONFIG

OID_WWAN_MPDP

Article • 02/18/2023

OID_WWAN_MPDP sets or queries information about Multiple Packet Data Protocol (MPDP) interfaces for the MB device representing the primary PDP context/EPS bearer.

For query requests, the miniport driver responds to the MB service asynchronously by initially returning NDIS_STATUS_INDICATION_REQUIRED. After the query request is complete, the driver sends an NDIS_STATUS_WWAN_MPDP_LIST notification that contains a list of child interfaces for the primary PDP context, formatted in an **NDIS_WWAN_MPDP_LIST** structure.

For set requests, like with query requests the miniport driver responds to the MB service asynchronously by initially returning NDIS_STATUS_INDICATION_REQUIRED. The set request contains an **NDIS_WWAN_SET_MPDP_STATE** structure, which in turn contains an **NDIS_WWAN_MPDP_INFO** structure with information for the operation.

If the **Operation** member of the **NDIS_WWAN_MPDP_INFO** structure is set to **WwanMPDPOperationCreateChildInterface**, the client driver creates a new child interface for the primary PDP context. The status result of this operation, along with the GUID of the newly created child interface if the operation was successful, are returned to the MB service in an **NDIS_WWAN_MPDP_STATE** structure contained in an NDIS_STATUS_WWAN_MPDP_STATE notification.

If the **Operation** member of the **NDIS_WWAN_MPDP_INFO** structure is set to **WwanMPDPOperationDeleteChildInterface**, the miniport driver deletes the corresponding child interface it previously created and returns information about the deletion operation to the MB service in an **NDIS_WWAN_MPDP_STATE** structure contained in an NDIS_STATUS_WWAN_MPDP_STATE notification.

Requirements

Version: Windows 10, version 1809

Header: Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_MPDP_LIST

NDIS_WWAN_MPDP_LIST

NDIS_WWAN_SET_MPDP_STATE

NDIS_WWAN_MPDP_INFO

NDIS_STATUS_WWAN_MPDP_STATE

NDIS_WWAN_MPDP_STATE

EvtMbbDeviceCreateAdapter

OID_WWAN_NETWORK_BLACKLIST

Article • 02/18/2023

ⓘ Important

Bias-free communication

Microsoft supports a diverse and inclusive environment. This article contains references to terminology that the Microsoft **style guide for bias-free communication** recognizes as exclusionary. The word or phrase is used in this article for consistency because it currently appears in the software. When the software is updated to remove the language, this article will be updated to be in alignment.

OID_WWAN_NETWORK_BLACKLIST gets or sets information about network blacklists for a mobile broadband (MBB) device.

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_NETWORK_BLACKLIST status notification containing an **NDIS_WWAN_NETWORK_BLACKLIST** structure that describes the current network blacklists.

For Set requests, this OID's payload contains an **NDIS_WWAN_SET_NETWORK_BLACKLIST** structure that specifies a list of MNC/MCC combinations that should be ignored by the modem.

Remarks

After each Query or Set request, the miniport driver should return an **NDIS_WWAN_NETWORK_BLACKLIST** structure that contains information about the current network blacklist information.

For more information about usage of this OID, see MBIM_CID_MS_NETWORK_BLACKLIST.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB Network Blacklist Operations

NDIS_STATUS_WWAN_NETWORK_BLACKLIST

NDIS_WWAN_NETWORK_BLACKLIST

NDIS_WWAN_SET_NETWORK_BLACKLIST

OID_WWAN_NETWORK_IDLE_HINT

Article • 02/18/2023

OID_WWAN_NETWORK_IDLE_HINT sends a hint to the network interface regarding whether data is expected to be active or idle on the interface. The network service uses heuristics to determine when to send this request to the interface, typically when it estimates that for a period of time there will be a reduction in network traffic or if the system is entering an idle state (such as connected standby). The network interface can use this as an input to its heuristics to implement procedures such as "fast dormancy".

Query requests are not supported.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later completing the request with the NDIS_WWAN_NETWORK_IDLE_HINT structure that indicates the network idle hint.

Requirements

Version	Available in Windows 10 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_NETWORK_IDLE_HINT

OID_WWAN_NETWORK_PARAMS

Article • 02/18/2023

The host may send an OID_WWAN_NETWORK_PARAMS query request to retrieve network configuration data and/or policy information from an MB device.

Set requests are not valid.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_NETWORK_PARAMS_STATE status notification.

Remarks

For more information about usage of this OID, see MBIM_CID_MS_NETWORK_PARAMS in the MBIMEx 3.0 specification.

Requirements

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022. NDIS 6.84 and later.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_NETWORK_PARAMS_STATE

OID_WWAN_NITZ

Article • 02/18/2023

OID_WWAN_NITZ is used to query the current network time with Network Identity and Time Zone (NITZ).

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_NITZ_INFO status notification containing an **NDIS_WWAN_NITZ_INFO** structure that describes the current network time and time zone.

Set requests are not applicable.

Remarks

For more information about usage of this OID, see MB NITZ support.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB NITZ support

NDIS_STATUS_WWAN_NITZ_INFO

NDIS_WWAN_NITZ_INFO

OID_WWAN_PACKET_SERVICE

Article • 02/18/2023

OID_WWAN_PACKET_SERVICE is used to instruct miniport drivers to perform packet service attach/detach actions on the current registered provider's network for both GSM-based and CDMA-based MB devices. In addition to the packet service attach/detach status, this OID is used to determine data class availability and the currently used data class information.

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an **NDIS_STATUS_WWAN_PACKET_SERVICE** status notification containing an **NDIS_WWAN_PACKET_SERVICE_STATE** structure to provide information about the current packet service state regardless of completing set or query requests.

Callers requesting to set the current packet service state provide an **NDIS_WWAN_SET_PACKET_SERVICE** structure to the miniport driver with the appropriate information.

Remarks

See WWAN Packet Service Attach Operations for more information about using this OID.

Miniport drivers can access the provider network when processing query or set operations, but should not access the Subscriber Identity Module (SIM card).

CDMA-based devices should use this as an opportunity to release the network resource allocation if possible.

Some SIM cards enable the MB device to register only on the packet domain and not the circuit-switched domain. Once a data call ends, the VAN UI sends an OID_WWAN_PACKET_SERVICE set request to detach packet service. This causes the MB device to detach from the packet domain. The MB device unregisters from the network and goes into a power save mode. Consequently, the device disappears from the VAN UI as a result of being unregistered, and can only be recovered by rebooting. In this scenario, miniport drivers should spoof the packet attach/detach operations by returning positive data without setting the MB device into such a mode.

For technologies that do not support packet-attach, miniport drivers should spoof an attach state to let the MB Service know that it can proceed with context activation. Miniport drivers should also spoof the set OID_WWAN_PACKET_SERVICE requests in the miniport driver. Miniport drivers should send **NDIS_STATUS_WWAN_PACKET_SERVICE** notifications for query operations and for unsolicited events. Miniport drivers should fail PDP activation if the device packet service state is not set to *WwanPacketServiceStateAttached*.

The MB Service shall not proceed with context activation until the packet service state has reached *WwanPacketServiceStateAttached*.

Windows 10, version 1903

A new revision 2 for this OID is supported starting in Windows 10, version 1903. The extension enables the host to query the frequency range in which the modem is currently operating in 5G.

The host can query the extended packet service state information at any time. The response is the same as revision 1, except that revision 2 has two new fields.

If the modem is registered in a 5G domain, it returns the 5G frequency range of the carrier. If multiple 5G carriers exist, then all valid ranges are returned.

For more info about 5G data class support, see MB 5G data class support.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SET_PACKET_SERVICE

NDIS_STATUS_WWAN_PACKET_SERVICE

WWAN Packet Service Attach Operations

OID_WWAN_PCO

Article • 02/18/2023

OID_WWAN_PCO reports the status and the payload of a Protocol Configuration Optiont (PCO) value that the modem has received from a mobile operator network. The PCO value that is returned from the modem corresponds to the PDN that the port number specifies in the OID request structure.

For query requests, the modem first responds with NDIS_STATUS_INDICATION_REQUIRED when it receives this OID. An NDIS_STATUS_WWAN_PCO_STATUS notification will be returned containing an NDIS_WWAN_PCO_STATUS structure when the query request is completed. **NDIS_WWAN_PCO_STATUS**, in turn, contains the PCO status and a WWAN_PCO_VALUE structure that represents the PCO value.

Set requests are not applicable.

Remarks

For modems that choose to use the Microsoft inbox miniport class driver, to receive query requests from the host, the modem must advertise that it supports the new **MBIM_CID_PCO** CID (index = 9) in the **MBB_UUID_BASIC_CONNECT_EXT_CONSTANT** service when responding to an **MBIM_CID_DEVICE_SERVICES** query. For more info about *MBIM_CID_PCO*, see MB Protocol Configuration Options (PCO) operations.

For modems that choose not use Microsoft inbox miniport class driver, to receive query requests from WWANSVC, the modem's miniport driver must advertise that it supports the *WWAN_OPTIONAL_SERVICE_CAPS_PCO* option when responding to OID OID_WWAN_DEVICE_CAPS_EX query requests.

Requirements

Version: Windows 10, version 1709 **Header**: Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_PCO_STATUS

NDIS_WWAN_PCO_STATUS

WWAN_PCO_VALUE

OID OID_WWAN_DEVICE_CAPS_EX

MB Protocol Configuration Options (PCO) operations

OID_WWAN_PIN

Article • 02/18/2023

OID_WWAN_PIN sets or returns information related to Personal Identification Numbers (PINs).

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_PIN_INFO status notification when they have completed the set or query request.

Miniport drivers should send NDIS_STATUS_WWAN_PIN_INFO status notifications containing an NDIS_WWAN_PIN_INFO structure to return PIN-type and PIN-entry state information, primarily to indicate whether a PIN is required to unlock the MB device or Subscriber Identity Module (SIM card) when completing query requests.

Callers requesting to set information related to PINs provide an NDIS_WWAN_SET_PIN structure to the miniport driver to send a PIN to the MB device, enable or disable PIN settings, or to change a PIN on the SIM.

Remarks

See WWAN Pin Operations for more information about using this OID.

Windows 7 miniport drivers should use OID_WWAN_PIN. Windows 8 miniport drivers should use OID_WWAN_PIN_EX.

Miniport drivers can access the Subscriber Identity Module (SIM card) when processing query operations, but should not access the provider network.

During the miniport driver initialization process, the MB Service does not proceed to registration until PIN1 is successfully unlocked, if enabled.

Miniport drivers provide a PIN value, entered by the end user, in the **PinAction.Pin** member of the NDIS_WWAN_SET_PIN structure when processing set requests. Only when the PIN value matches the value stored in the SIM card should the request be processed by the miniport driver. Otherwise, miniport drivers should fail the set request with status code WWAN_STATUS_FAILURE.

CDMA-based devices must report the power-on device lock as PIN1.

For all supported PIN types, miniport drivers must support the *WwanPinOperationEnter* operation. Additionally, if PIN1 is supported, miniport drivers must support the

WwanPinOperationEnable, *WwanPinOperationDisable*, and *WwanPinOperationChange* operations.

If a PIN disable operation for a PIN type is tried when that PIN type is locked, miniport drivers can either fail the request with WWAN_STATUS_PIN_REQUIRED or they can successfully complete the request. If the miniport driver completes the request successfully, the disable operation should also unlock the PIN.

If reporting multiple PINs are enabled, and only one PIN can be reported at a time, then miniport drivers are expected to report PIN1 first. For example, if reporting of SubsidyLock and SIM PIN1 are enabled, then the SubsidyLock PIN should be reported (in a subsequent query request) only after PIN1 has been successfully verified.

The MB API supports other PINs in addition to PIN1. However, a 3rd-party connection manager/GUI would need to be installed because the Windows Connection Manager/GUI supports only PIN1.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_PIN_INFO

NDIS_WWAN_SET_PIN

NDIS_STATUS_WWAN_PIN_INFO

WWAN Pin Operations

OID_WWAN_PIN_EX

Article • 02/18/2023

OID_WWAN_PIN_EX sets or returns expanded information related to Personal Identification Numbers (PINs).

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_PIN_INFO status notification when they have completed the set or query request.

Miniport drivers should send NDIS_STATUS_WWAN_PIN_INFO status notifications containing an NDIS_WWAN_PIN_INFO structure to return PIN-type and PIN-entry state information, primarily to indicate whether a PIN is required to unlock the MB device or Subscriber Identity Module (SIM card) when completing query requests.

Callers requesting to set information related to PINs provide an NDIS_WWAN_SET_PIN_EX structure to the miniport driver to send a PIN to the MB device, enable or disable PIN settings, or to change a PIN on the SIM.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

OID_WWAN_PIN_EX2

Article • 02/18/2023

OID_WWAN_PIN_EX2 sets or returns expanded information related to Personal Identification Numbers (PINs). OID_WWAN_PIN_EX2 is similar to OID_WWAN_PIN_EX, but extends it to support multi-app UICC cards.

Query payloads contain an **NDIS_WWAN_PIN_APP** structure specifying the target application ID whose PIN is being queried. Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_PIN_INFO status notification containing an **NDIS_WWAN_PIN_INFO** structure that describes the PIN for the application.

Set payloads contain an **NDIS_WWAN_SET_PIN_EX2** structure specifying the PIN action to take for the application. Miniport drivers must process Set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_PIN_INFO status notification containing an **NDIS_WWAN_PIN_INFO** structure that describes the PIN state for the application.

Remarks

Only single verification-capable UICCs are supported. Multi-verification-capable UICCs that support more than one application PIN are not supported. One application PIN (PIN1) is assigned to all ADFs/DFs and files on the UICC. However, each application can specify a local PIN (PIN2) as a level 2 user verification requirement, resulting in the need for additional validation for every access command. This scenario is what OID_WWAN_PIN_EX2 supports.

Just like OID_WWAN_PIN_EX, with OID_WWAN_PIN_EX2 the device only reports one PIN at a time. If multiple PINs are enabled and reporting multiple PINs is enabled, then miniport drivers must report PIN1 first. For example, if subsidy lock reporting is enabled and the SIM's PIN1 is enabled, then the subsidy lock PIN should be reported in a subsequent query request only after PIN1 is specified by setting the **PinSize** to zero (0). In this case, a Set request is similar to a Query and returns the state of the PIN referenced. This is fully aligned to the behavior of the VERIFY command as specified in Section 11.1.9 of the ETSI TS 102 221 technical specification .

For more information about usage of this OID, see MB UICC application and file system access.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

NDIS_STATUS_WWAN_PIN_INFO

NDIS_WWAN_PIN_APP

NDIS_WWAN_SET_PIN_EX2

NDIS_WWAN_PIN_INFO

OID_WWAN_PIN_LIST

Article • 02/18/2023

OID_WWAN_PIN_LIST returns a list of all the different types of Personal Identification Numbers (PINs) that are supported by the MB device and additional details for each PIN type, such as the length of the PIN (minimum and maximum lengths), PIN format, PIN-entry mode (enabled/disabled/not-available). This OID also specifies the current mode of each PIN supported by the device.

Set requests are not supported.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an **NDIS_STATUS_WWAN_PIN_LIST** status notification containing an **NDIS_WWAN_PIN_LIST** structure to return a list of PINs with corresponding descriptions when completing query requests.

Remarks

For more information about using this OID, see WWAN Pin Operations.

Miniport drivers can access the Subscriber Identity Module (SIM card) when processing query operations, but should not access the provider network.

Miniport drivers must report all the PINs supported by their device. If the device does not support listing PINs, the miniport driver must report this list from a static (hard-coded) list maintained in the miniport driver itself for all the devices it supports.

Any PIN that provides device power-on verification or identification functionality should be reported as PIN1 and must be compliant to PIN1 guidelines.

Miniport drivers must return this information when the device ready-state changes to *WwanReadyStateInitialized* or when the device ready-state is *WwanReadyStateDeviceLocked* (PIN locked). Miniport drivers should also return this information in other device ready-states, wherever possible.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_PIN_LIST

NDIS_STATUS_WWAN_PIN_LIST

WWAN Pin Operations

OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS

Article • 02/18/2023

OID_WWAN_PREFERRED_MULTICARRIER_PROVIDERS is used to *set* or *query* the list of preferred multi-carrier network providers. Multi-carrier providers are ones that can be *set* as home providers.

Both *set* and *query* requests are supported. Miniport drivers must process *set* and *query* requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS status notification containing an NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS structure.

Miniport drivers should set the **PreferredListHeader.ElementType** member to **WwanStructProvider2** and the **PreferredListHeader.ElementCount** member to the number of providers in the list when responding to OID_WWAN_PREFERRED_PROVIDERS *query* requests. The multi-carrier providers returned in a *query* must be able to be set as the home provider at the time the preferred multi-carrier list is returned to the service.

Miniport drivers should set the **PreferredListHeader.ElementType** member to **WwanStructProvider2** and the **PreferredListHeader.ElementCount** member to 0 when responding to OID_WWAN_PREFERRED_PROVIDERS *set* requests.

On error miniports should set the **uStatus** member of NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS structure with the failure status and **PreferredListHeader.ElementCount** to 0 and **PreferredLIstHeader.ElementType** to **WwanStructProvider2**.

The **Rssi** and **ErrorRate** members of WWAN_PROVIDER2 structure should be set if available.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS

NDIS_STATUS_WWAN_PREFERRED_MULTICARRIER_PROVIDERS

MB Provider Operations

OID_WWAN_PREFERRED_PROVIDERS

Article • 05/20/2024

OID_WWAN_PREFERRED_PROVIDERS returns information about the list of preferred providers for GSM-based devices. Miniport drivers of CDMA-based devices do not need to support this OID.

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_PREFERRED_PROVIDERS status notification containing an NDIS_WWAN_PREFERRED_PROVIDERS structure to provide information about the Preferred Provider List (PPL) regardless of completing set or query requests.

Remarks

For more information about using this OID, see WWAN Provider Operations.

Miniport drivers can access the Subscriber Identity Module (SIM card) when processing query requests, but should not access the provider network.

Miniport drivers can access the Subscriber Identity Module (SIM card) or the provider network, when processing set requests.

When processing OID_WWAN_PREFERRED_PROVIDERS, miniport drivers may set only the WWAN_PROVIDER_STATE_PREFERRED or WWAN_PROVIDER_STATE_FORBIDDEN flags to tag the list entries. Be aware that forbidden providers might not appear in the list for GSM-based devices.

Miniport drivers should set the **PreferredListHeader.ElementType** member to *WwanStructProvider*. The miniport driver should set the **PreferredListHeader.ElementCount** member to 0 when responding to OID_WWAN_PREFERRED_PROVIDERS set requests.

Whether the PPL on the device can be overwritten or not when processing set requests depends on the device capability, the cellular technology, and/or the network provider's policy.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support returning or setting the PPL.

Requirements

Expand table

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_PREFERRED_PROVIDERS

NDIS_STATUS_WWAN_PREFERRED_PROVIDERS

WWAN Provider Operations

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

OID_WWAN_PRESHUTDOWN

Article • 02/18/2023

OID_WWAN_PRESHUTDOWN is sent to notify the modem that the system is entering the shutdown phase and the modem should finish its operations so it can be shut down properly. It is only sent down with the port number corresponding to the physical MBB adapters. Virtual adapters that support multiple PDP contexts should not receive this OID.

Query requests are not supported.

Miniport drivers must process set requests asynchronously, initially returning **NDIS_STATUS_INDICATION_REQUIRED** to the original request, and later sending a NDIS_STATUS_WWAN_PRESHUTDOWN_STATE status notification when the MBB driver has finished all necessary modem operations prior to shutting down. The set request has a NDIS_WWAN_SET_PRESHUTDOWN_STATE structure.

Miniport drivers should return **NDIS_STATUS_NOT_SUPPORTED** if they do not support this operation.

Requirements

Version	Available starting with Windows 10, version 1511.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_PRESHUTDOWN_STATE

NDIS_WWAN_SET_PRESHUTDOWN_STATE

OID_WWAN_PROVISIONED_CONTEXTS

Article • 02/18/2023

OID_WWAN_PROVISIONED_CONTEXTS reads or updates the provisioned context entries stored on the MB device or the Subscriber Identity Module (SIM).

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an **NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS** status notification containing an **NDIS_WWAN_PROVISIONED_CONTEXTS** structure to provide information about provisioned context entries stored on the MB device or the Subscriber Identity Module (SIM) regardless of completing set or query requests.

Remarks

For more information about using this OID, see WWAN Packet Context Management.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if the MB device they support does not support retrieval of provisioned contexts.

GSM-based devices can optionally support query and set operations. CDMA-based devices can optionally support query operations reporting Simple IP (WWAN_CTRL_CAPS_CDMA_SIMPLE_IP).

The provisioned context entries stored on the MB device or the SIM are local to the device. Miniport drivers should not connect to the network to read in these fields.

The input structure for a set request is NDIS_WWAN_SET_PROVISIONED_CONTEXT and status indication of this object is NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS.

Provisioned contexts are not same as that of the GPRS context definitions in 3GPP that caches the list of APNs. Provisioned contexts are the connectivity parameters (AccessString, UserName, and Password) that are either pre-provisioned by the Operators or OTA provisioned by the device and can be stored either in the device memory or SIM. The connectivity parameters returned by the Provisioned contexts will be used by the MB Service for PDP activation.

Both query and set form of this object is used.

Processing of this request does not require network access, but requires access to the SIM or auxiliary memory on the MB device.

The miniport driver sends NDIS_STATUS_WWAN_PROVISIONED_CONTEXTS notification to the operating system. The **ContextListHeader.ElementType** member shall be set to *WwanStructContext*. Miniport driver should set the **ContextListHeader.ElementCount** member to 0 when notification is sent in response to a set request.

The MB Service should retrieve the list of provisioned contexts from the device before conducting any individual context activation or deactivation. The list of provisioned contexts must be restricted only to the home provider network even though the device may have the capability to store multiple network provider contexts. The context list must always be the home provider network specific even in case of roaming.

SET OID_WWAN_PROVISIONED_CONTEXT operation should associate the context with the network provider that is specified in the set request in **ProviderId** member of the WWAN_SET_CONTEXT structure. Provisioned context stored through set OID_WWAN_PROVISIONED_CONTEXT requests must persist across system restarts and device power recycles.

All the empty contexts need to be reported on a query along with the provisioned contexts applicable to the home provider network.

CDMA devices that are configured for SimpleIP, reporting in WWAN_CTRL_CAPS_CDMA_SIMPLE_IP in WwanControlCaps can optionally return at least one provisioned context filled with the correct **AccessString**, **UserName**, and **Password** members for the query request from MB Service.

Provisioned context list should be pre-provisioned in the device, updated by set OID_WWAN_PROVISIONED_CONTEXT operations, or updated by device/operator using SMS or OTA. It must not be updated dynamically based on the context information provided in the OID_WWAN_CONNECT operation by MB Service.

For more information about how to access AccessString, UserName, and Password from the MB device for each provisioned context in the list, see WWAN_CONTEXT.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

WWAN Packet Context Management

OID_WWAN_RADIO_STATE

Article • 02/18/2023

OID_WWAN_RADIO_STATE sets or returns information about a MB device's radio power state.

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_RADIO_STATE status notification containing an NDIS_WWAN_RADIO_STATE structure that indicates the MB device's current radio power state regardless of completing set or query requests.

Callers requesting to set the MB device's radio power state provide an NDIS_WWAN_SET_RADIO_STATE structure to the miniport driver with the appropriate information.

Remarks

For more information about using this OID, see MB Radio State.

Miniport drivers should not access the provider network, or the Subscriber Identity Module (SIM card), when processing query or set operations.

Miniport drivers must retain software radio power states across system restart or device removal and reinsertion. Miniport drivers should store the device's software radio information and use it for setting the device software radio power state immediately on each restart or reinsertion of device. The effective radio power state of the device is decided based on combination of software and hardware radio power state as per the table in WWAN_RADIO_STATE.

If the value is *WwanRadioOn*, miniport drivers must turn on the radio power and set the **RadioState.SwRadioState** member of the WWAN_RADIO_STATE structure to *WwanRadioOn*. If the **RadioState.HwRadioState** member was *WwanRadioOff*, miniport drivers should cache this power state information and ensure to physically turn on the radio power state when **RadioState.HwRadioState** changes to *WwanRadioOn*.

If the value is *WwanRadioOff*, miniport drivers must turn off the radio power state and set the **RadioState.SwRadioState** member to *WwanRadioOff*.

Refer to the following table for the expected radio state programming by miniport drivers.

Valid Combinations for PIN Mode and PIN State

HwRadioState value	SwRadioState value	Overall radio power state
WwanRadioOff	WwanRadioOff	WwanRadioOff
WwanRadioOff	WwanRadioOn	WwanRadioOff
WwanRadioOn	WwanRadioOff	WwanRadioOff
WwanRadioOn	WwanRadioOn	WwanRadioOn

For devices that do not provide a hardware radio power switch, the **RadioState.HwRadioState** member of the NDIS_WWAN_RADIO_STATE structure must always be set to *WwanRadioOn*.

Starting in Windows 10, version 1703, OID_WWAN_RADIO_STATE has additional specifications for how a multi-executor supported modem should handle radio state configuration from the OS.

With a multi-executor supported modem, there are power benefits to configuring radio power state per executor. When an executor’s radio is turned off, the OS expects the modem to de-register from the network and does not attempt any scanning or location updates from it. The modem should support a radio state for each executor that it advertises to the OS so it can determine the hardware power state in which it should be.

As an example, if the modem has two executors and one of the executors' radio is off while the other is on, then the modem may keep the RF front end powered on to maintain registration on the executor whose radio is on but does not need to do scanning/pinging/location updates or other cellular services for the executor that is turned off. If both radios are turned off, the modem can turn off its RF front end and bring the overall hardware to a lower power state. The implementation specifics are left to each IHV.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_RADIO_STATE

NDIS_WWAN_SET_RADIO_STATE

NDIS_STATUS_WWAN_RADIO_STATE

MB Radio State

WWAN_RADIO_STATE

OID_WWAN_READY_INFO

Article • 02/18/2023

OID_WWAN_READY_INFO returns the device ready-state, which includes its Subscriber Identity Module (SIM card). This typically occurs at the beginning of any session.

Set requests are not supported.

The host can query the ready-state from either the active SIM slot or inactive SIM slot in the device if the device supports dual SIM slots. This OID's payload contains an **NDIS_WWAN_QUERY_READY_INFO** structure, which in turn contains a **WWAN_QUERY_READY_INFO** structure that specifies the UICC slot ID.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an **NDIS_STATUS_WWAN_READY_INFO** status notification containing an **NDIS_WWAN_READY_INFO** structure that indicates the MB device's ready-state when completing query requests.

Remarks

For more information about using this OID, see MB device Readiness.

Miniport drivers can access device memory or the SIM card when processing query operations, but should not access the provider network.

Miniport drivers should wait until the PIN is cleared (if required) and then read the subscriber's identity and telephone number(s) (TNs), and then set the ReadyInfo.ReadyState member of the NDIS_WWAN_READY_INFO structure to WwanReadyStateInitialized.

Miniport drivers must never fail OID_WWAN_READY_INFO and must always return the correct device ready-state.

Miniport drivers must always notify the MB Service whenever the device ready-state changes.

Miniport drivers should follow these steps to provide a good user experience:

- If PIN1 is locked, miniport drivers must first send a ready-state event notification with **ReadyInfo.ReadyState** set to *WwanReadyStateDeviceLocked*. The MB Service then sends the miniport driver an OID set request of OID_WWAN_PIN. After the device unlocks then the miniport driver must send another ready-state event notification with **ReadyInfo.ReadyState** set to *WwanReadyStateInitialized*. Until PIN1 is successfully unlocked, miniport drivers must not change the device ready-state to *WwanReadyStateInitialized*.

- Miniport drivers must first send an event notification with **ReadyInfo.ReadyState** set to *WwanReadyStateSimNotInserted* when the MB Service loads the miniport driver if no SIM card is present, as may be the case with devices that allow SIM cards to be inserted or removed. If the device has the capability to detect a hot insertion of a SIM card, the miniport driver must send another event notification with **ReadyInfo.ReadyState** set to *WwanReadyStateInitialized* when the user inserts a SIM.

- Devices that have the capability to detect service activation state must set **ReadyInfo.ReadyState** to *WwanReadyStateNotActivated*. Furthermore, if the miniport driver supports service activation, the miniport driver will receive an OID set request of OID_WWAN_SERVICE_ACTIVATION. On successful completion of service activation, miniport drivers must send another event notification with **ReadyInfo.ReadyState** set to *WwanReadyStateInitialized*.

- Miniport drivers that require a specific firmware revision must ensure that the correct firmware revision is available. If the firmware revision is not available, the miniport driver should complete the event notification transaction by setting **ReadyInfo.ReadyState** to *WwanReadyStateFailure*.

Requirements

Version: Available in Windows 7 and later versions of Windows.

Header: Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_READY_INFO

NDIS_STATUS_WWAN_READY_INFO

NDIS_WWAN_QUERY_READY_INFO

WWAN_QUERY_READY_INFO

MB device Readiness

OID_WWAN_REGISTER_PARAMS

Article • 02/18/2023

OID_WWAN_REGISTER_PARAMS sets or returns the parameters that an MB device uses during 5G registration requests.

Before turning on the device radio, the host typically sends an OID_WWAN_REGISTER_PARAMS set request to configure the device with the desired registration parameters. This OID's payload contains an **NDIS_WWAN_SET_REGISTER_PARAMS** structure, which in turn contains a **WWAN_REGISTRATION_PARAMS_INFO** structure that specifies the registration parameters such as a default PDU session hint. If the device accepts these parameters, it will use them during 5G registration requests.

The host may send an OID_WWAN_REGISTER_PARAMS set request at any time. When the device receives this request, it must compare the new parameters to any parameters it previously used for 5G registration. If there are differences, the device should use the newly received parameters for the next 5G registration. The host can also use the **WWAN_REGISTRATION_PARAMS_INFO** structure's **ReRegisterIfNeeded** parameter to force immediate 5G re-registration.

The host may use this OID to query the registration parameters that an MB device is currently using for 5G registration.

Remarks

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_REGISTER_PARAMS_STATE status notification containing an **NDIS_WWAN_REGISTER_PARAMS_INFO** structure, which contains a **WWAN_REGISTRATION_PARAMS_INFO** structure.

For a failured set or query response, the information shall be null and the **InformationBufferLength** shall be zero.

In a successful set response, the **WWAN_REGISTRATION_PARAMS_INFO** structure shall contain the parameters set by the host and accepted by the device.

In a successful query response:

- If the parameters have been set by the host and accepted by the device since the device was rebooted or a different SIM was inserted, the structure shall contain the parameters set by the host and accepted by the device.

- If the parameters have not been set by the host and accepted by the device since the device was rebooted or a different SIM was inserted, the structure shall contain the parameters that the device will most likely use for 5G registration.

For more information about usage of this OID, see MBIM_CID_MS_REGISTRATION_PARAMS in the MBIMEx 3.0 specification .

Requirements

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022. NDIS 6.84 and later.
Header	Ntddndis.h (include Ndis.h)

See also

WWAN_REGISTRATION_PARAMS_INFO

NDIS_STATUS_WWAN_REGISTER_PARAMS_STATE

OID_WWAN_REGISTER_STATE

Article • 02/18/2023

OID_WWAN_REGISTER_STATE selects a network provider to register with.

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_REGISTER_STATE status notification containing an NDIS_WWAN_REGISTRATION_STATE structure to provide information about the registered network provider regardless of completing set or query requests.

Callers requesting to set the network provider to register with provide an NDIS_WWAN_SET_REGISTER_STATE structure to the miniport driver with the appropriate information.

Remarks

For more information about using this OID, see WWAN Registration Operations.

Miniport drivers can access the provider network when processing query or set operations, but should not access the Subscriber Identity Module (SIM card).

The MB driver model supports two registration methods--automatic and manual. For CDMA-based networks, the MB driver model supports only automatic registration.

Devices supporting manual registration must set the **WwanControlCaps** member in WWAN_DEVICE_CAPS structure to WWAN_CTRL_CAPS_REG_MANUAL. Be aware that GSM-based devices must support manual registration.

If the registration state is automatic, miniport drivers must instruct their device to select a network provider based on the selection algorithm specific to the cellular technology and proceed with registration.

The semantics of RegisterAction values are defined as follows:

- The *WwanRegisterActionAutomatic* flag is used by the MB Service to instruct the miniport driver to set the device to automatic register mode and let the device select the best provider network. The miniport driver must ignore **ProviderId** parameter. This setting is persistent across radio states (ON/OFF), and device power cycles, until it is explicitly change by the MB Service.
- The *WwanRegisterActionManual* flag is used by the MB Service to instruct the miniport driver to register with the provider network identified by the **ProviderId**

parameter. The **ProviderId** value shall come from the **ProviderId** member of WWAN_PROVIDER data structure of one of the visible providers. This setting is persistent across radio states (ON/OFF), and device power cycles, until it is explicitly changed by the MB Service.

- Changing between the different RegisterAction values are allowed even if the device is currently registered to a provider. If the device need to deregister before switching between the Automatic and Manual registration modes, the miniport driver must ensure that the device is set to deregistration before setting to the new registration mode.
- The *Manual* and *Automatic* registration mode only affects the network selection mode. The MB device should try to register to selected network whenever the radio is turned on.

Windows 10, version 1903

A new revision 3 for this OID is supported starting in Windows 10, version 1903. This extension enables the host to query the preferred radio access technologies (RATs) from the miniport driver.

To control the preferred RAT, the host sets a bitmask representing WWAN_DATA_CLASS values in the **WwanDataClass** member of the WWAN_SET_REGISTER_STATE structure. This member represents the data access technologies that are preferred for a connection. If this field is set to **WWAN_DATA_CLASS_NONE**, then the modem should take no action for this parameter.

The host can also query the currently preferred data classes from the miniport driver. The miniport driver uses the **PreferredDataClasses** field of the WWAN_REGISTRATION_STATE structure to report the preferred data access technologies that are currently set in the modem.

For more info about 5G data class support, see MB 5G data class support.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SET_REGISTER_STATE

NDIS_STATUS_WWAN_REGISTER_STATE

WWAN Registration Operations

OID_WWAN_SAR_CONFIG

Article • 02/18/2023

OID_WWAN_SAR_CONFIG gets or sets information about a mobile broadband (MB) device's Specific Absorption Rate (SAR) back off mode and level.

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_SAR_CONFIG status notification containing an **NDIS_WWAN_SAR_CONFIG_INFO** structure that describes the current SAR configuration.

For Set requests, this OID's payload contains an **NDIS_WWAN_SET_SAR_CONFIG** structure that specifies the new SAR configuration for the modem.

Remarks

After each Query or Set request, the miniport driver should return an **NDIS_WWAN_SAR_CONFIG_INFO** structure that contains information for all antennas on the device associated with Mobile Broadband.

For more information about usage of this OID, see MBIM_CID_MS_SAR_CONFIG.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB SAR Platform Support

NDIS_STATUS_WWAN_SAR_CONFIG

NDIS_WWAN_SAR_CONFIG_INFO

NDIS_WWAN_SET_SAR_CONFIG

OID_WWAN_SAR_TRANSMISSION_STATUS

Article • 02/18/2023

OID_WWAN_SAR_TRANSMISSION_STATUS enables or disables notifications from a mobile broadband (MB) modem on Specific Absorption Rate (SAR) transmission state.

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_SAR_TRANSMISSION_STATUS status notification containing an **NDIS_WWAN_SAR_TRANSMISSION_STATUS_INFO** structure that describes whether notifications on SAR transmit state are enabled in the modem.

For Set requests, this OID's payload contains an **NDIS_WWAN_SET_SAR_TRANSMISSION_STATUS** structure that specifies if SAR transmission status notifications should be enabled or disabled.

Remarks

After each Query or Set request, the miniport driver should return an **NDIS_WWAN_SAR_TRANSMISSION_STATUS_INFO** structure that describes whether SAR notifications on transmit state are enabled in the modem.

For more information about usage of this OID, see MBIM_CID_MS_TRANSMISSION_STATUS.

Requirements

Version: Windows 10, version 1703 **Header**: Ntddndis.h (include Ndis.h)

See also

MB SAR Platform Support

NDIS_STATUS_WWAN_SAR_TRANSMISSION_STATUS

NDIS_WWAN_SAR_TRANSMISSION_STATUS_INFO

NDIS_WWAN_SET_SAR_TRANSMISSION_STATUS

OID_WWAN_SERVICE_ACTIVATION

Article • 02/18/2023

OID_WWAN_SERVICE_ACTIVATION instructs miniport drivers to initiate service activation in order to gain access to the provider's network.

Query requests are not supported.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_SERVICE_ACTIVATION status notification containing an NDIS_WWAN_SERVICE_ACTIVATION structure to initiate service activation in order to gain access to the provider's network when they have completed the transaction.

Remarks

For more information about using this OID, see MB Service Detection and Activation.

Miniport drivers can access the Subscriber Identity Module (SIM card) or the provider network when processing query or set requests.

The MB Service uses OID_WWAN_SERVICE_ACTIVATION in the case where the service activation process requires miniport driver and user interactions.

This is not needed for miniport driver initiated or out-of-band manual service activations such as calling into the service provider's helpdesk. After the device is activated as in the above scenarios, if the current miniport driver **ReadyState** is *WwanReadyStateNotActivated*, the miniport driver shall proceed with MB initialization and notify the MB Service of ready-state change using NDIS_STATUS_WWAN_READY_INFO INDICATION .

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support miniport driver-based service activation.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

OID_WWAN_READY_INFO

NDIS_WWAN_SERVICE_ACTIVATION

NDIS_STATUS_WWAN_SERVICE_ACTIVATION

MB Service Detection and Activation

OID_WWAN_SIGNAL_STATE

Article • 02/18/2023

OID_WWAN_SIGNAL_STATE returns or sets the current signal state.

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_SIGNAL_STATE status notification containing an NDIS_WWAN_SIGNAL_STATE structure to provide information about the current signal state indication shown to the end-user regardless of completing set or query requests.

Callers requesting to set the current signal state indication to the end user provide an NDIS_WWAN_SET_SIGNAL_INDICATION structure to the miniport driver with the appropriate information.

Remarks

For more information about using this OID, see WWAN Signal Strength Operations.

Miniport drivers should not access the provider network, or the Subscriber Identity Module (SIM card), when processing query or set operations.

Generally, signal state should be indicated rather than polled. However, this OID is made available in case the current signal state needs to be determined by the MB Service.

For response to query requests, miniport drivers should send an NDIS_STATUS_WWAN_SIGNAL_STATE notification.

On a set request from the MB Service, miniport drivers should:

- Return the current values for **Rssi** and **ErrorRate** in the NDIS_WWAN_SIGNAL_STATE structure in addition to reporting the absolute values for **RssiInterval** and **RssiThreshold** that has been set in the miniport driver.

- Internally cache the **RssiInterval** and/or **RssiThreshold** values even if the device is not currently registered with any operator and that any restriction imposed by device in setting parameters can only be possible post-registration state. The miniport driver should try to apply these settings in the next immediate available situation.

- Complete the request successfully, if the hardware and/or software radio switch state is currently OFF. Miniport driver cache the request data and start reporting the signal strength after the switch is turned ON.

- Can fail this request with the appropriate **uStatus** error code set.

Miniport drivers can do the following when processing query requests from the MB Service:

- Return the current values for **Rssi** and **ErrorRate** in the NDIS_WWAN_SIGNAL_STATE structure in addition to reporting the absolute values for **RssiInterval** and **RssiThreshold** that has been set in the miniport driver.

- Fail this request with the appropriate **uStatus** error code set.

Return Values:

NDIS_STATUS_NOT_SUPPORTED

Miniport drivers can return this for specific devices that are aware of device capabilities not supporting the signal strength can fail the request with this error code.

Recommended Implementation

1. Devices must support signal strength indications.

2. Drivers must report signal strength indications of at least 50% of the **RssiInterval** setting over a time period of five minutes.

3. Devices must avoid reporting the signal strength in the following states:

   a. Device not registered or deregistered and is applicable only for GSM devices.

   b. Effective state of radio is OFF.

   c. In the above states, a query to the signal strength must be returned with the following data by the miniport driver:

      Rssi = WWAN_RSSI_UNKNOWN

      ErrorRate = WWAN_ERROR_RATE_UNKNOWN;

      RssiInterval = < WWAN_RSSI_DISABLE, WWAN_RSSI_DEFAULT or last set value>

      RssiThreshold = < WWAN_RSSI_DISABLE, WWAN_RSSI_DEFAULT or the last set value>

Windows 10, version 1903

Starting in Windows 10, version 1903, OID_WWAN_SIGNAL_STATE has been upgraded to revision 3. This revision enables the host to query new reference signal received power

(RSRP) and Signal-to-Noise (SNR) values from the miniport driver. A miniport driver must use revision 3 of this OID and its data structures if the driver supports 5G.

For more info about 5G data class support, see MB 5G data class support.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SET_SIGNAL_INDICATION

WWAN Signal Strength Operations

OID_WWAN_SLOT_INFO

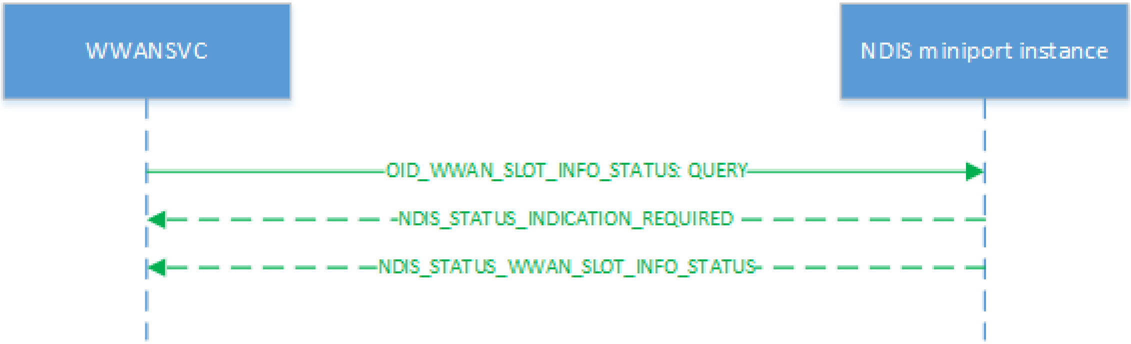
Article • 02/18/2023

OID_WWAN_SLOT_INFO retrieves a high-level aggregated status of a specified UICC slot and the card within it (if any). It may also be used to deliver an unsolicited notification when the status of one of the slots changes.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_SLOT_INFO status notification containing an NDIS_WWAN_SLOT_INFO structure, which in turn contains a WWAN_SLOT_INFO structure, to provide information about the overall modem system capability.

Query requests specify NDIS_WWAN_GET_SLOT_INFO structure as input. The miniport driver should return the slot status according to the slot ID specified in the **SlotIndex** member of the WWAN_GET_SLOT_INFO structure.

The following diagram illustrates a query request.


WWANSVC
NDIS miniport instance
OID_WWAN_SLOT_INFO_STATUS: QUERY
NDIS_STATUS_INDICATION_REQUIRED
NDIS_STATUS_WWAN_SLOT_INFO_STATUS

Set requests are not applicable.

An NDIS_STATUS_WWAN_SLOT_INFO notification with a NDIS_WWAN_SLOT_INFO structure is sent to host when the slot/card state changes.

Remarks

The host uses OID_WWAN_SLOT_INFO to query the status of a specific slot on the modem. This OID is not executor-specific and may be sent to any NDIS instance belonging to one modem. The slot state represents a summary of both the slot and card state.

The set of reported states is constrained by the capability of the slot hardware. In the most restrictive case, the slot hardware may only be able to determine that a card is present when it is powered on and active—in such a case the **OffEmpty** and **Off** states will not be reported.

OID_WWAN_READY_INFO and OID_WWAN_SLOT_INFO perform the same core function of retrieving the device ready-state of a SIM card slot; however, OID_WWAN_READY_INFO is a per-executor command whereas OID_WWAN_SLOT_INFO could be used on any physical instance (executor) and is expected to return the appropriate slot state even if it is not mapped to any executors at the moment.

Requirements

Version	Windows 10, version 1703
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_SLOT_INFO

NDIS_WWAN_SLOT_INFO

WWAN_SLOT_INFO

NDIS_WWAN_GET_SLOT_INFO

WWAN_GET_SLOT_INFO

WWAN_UICCSLOT_STATE

OID_WWAN_READY_INFO

OID_WWAN_SMS_CONFIGURATION

Article • 02/18/2023

OID_WWAN_SMS_CONFIGURATION sets or returns a MB device's SMS text message configuration.

Miniport drivers must process set and query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_SMS_CONFIGURATION status notification regardless of completing set or query requests.

Query requests return the MB device's current SMS text message configuration stored in the device or Subscriber Identity Module (SIM) card.

Set requests use the NDIS_WWAN_SET_SMS_CONFIGURATION structure to change the SMS text message configuration of the MB device.

Remarks

For more information about using this OID, see WWAN SMS Operations.

When processing this OID, miniport drivers can access the SIM card, but should not access the provider network.

Miniport drivers of GSM-based devices should support both query and set operations. Miniport drivers of CDMA-based devices should support only query operations. Miniport drivers of CDMA-based devices should return a valid value in the **ulMaxMessageIndex** member of the WWAN_SMS_CONFIGURATION structure for query requests and can ignore the other members.

Miniport drivers must send an unsolicited NDIS_STATUS_WWAN_SMS_CONFIGURATION indication when the MB device's SMS subsystem is ready for SMS operation. Thereafter, when responding to OID_WWAN_SMS_CONFIGURATION query requests, miniport drivers must return valid values for all members of the WWAN_SMS_CONFIGURATION structure.

Miniport drivers should return NDIS_STATUS_NOT_INITIALIZED if the device is initialized but the SMS subsystem is not yet initialized.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support configuring SMS text messages.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SET_SMS_CONFIGURATION

NDIS_STATUS_WWAN_SMS_CONFIGURATION

WWAN SMS Operations

OID_WWAN_SMS_DELETE

Article • 02/18/2023

OID_WWAN_SMS_DELETE deletes SMS text messages stored in the MB device, or Subscriber Identity Module (SIM card), or any other auxiliary non-volatile memory or memories.

Query requests are not supported.

Set requests use the NDIS_WWAN_SMS_DELETE structure.

Miniport drivers process this OID asynchronously, and should return an NDIS_STATUS_INDICATION_REQUIRED provisional response to any set requests. Miniport drivers should send an NDIS_STATUS_WWAN_SMS_DELETE indication when they have completed the transaction.

Remarks

For more information about using this OID, see WWAN SMS Operations.

When processing this OID, miniport drivers can access the Subscriber Identity Module (SIM card), but should not access the provider's network.

Miniport drivers may receive requests to delete SMS text messages based on an index, or to delete all SMS text messages. Delete requests may consist of any one of the basic filters such as new (unread) messages, old (read) messages, draft messages, or sent messages.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support SMS text messages, or the ability to delete SMS text messages.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SMS_DELETE

WWAN SMS Operations

OID_WWAN_SMS_READ

Article • 02/18/2023

OID_WWAN_SMS_READ reads SMS text messages stored in the MB device, or Subscriber Identity Module (SIM card), or any other auxiliary non-volatile memory or memories.

Set requests are not supported.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an **NDIS_STATUS_WWAN_SMS_RECEIVE** status notification containing an **NDIS_WWAN_SMS_READ** structure to provide the SMS messages requested that was initially provided by the caller when completing query requests.

Callers requesting to read SMS text messages provide an NDIS_WWAN_SMS_READ structure to indicate which SMS messages the caller wants the miniport to return.

Remarks

For more information about using this OID, see WWAN SMS Operations.

When processing this OID, miniport drivers can access the Subscriber Identity Module (SIM card), but should not access the provider network.

OID_WWAN_SMS_READ supports reading both PDU-mode and CDMA-mode SMS text messages, depending on the capabilities of the device.

Miniport drivers may receive requests to read SMS text messages based on an index, or to read all SMS text messages. Read requests may consist of any one of the basic filters such as new (unread) messages, old (read) messages, draft messages, or sent messages.

Miniport drivers that implement SMS text message functionality must support the reading of new messages using the basic filter for *WwanSmsFlagNew*. All other filter types are optional to support.

Miniport drivers must logically project a single SMS text message store across all available physically different SMS text message stores.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support SMS text messages.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SMS_READ

WWAN SMS Operations

OID_WWAN_SMS_SEND

Article • 02/18/2023

OID_WWAN_SMS_SEND sends SMS text messages to another MB device.

Query requests are not supported.

Set requests use the NDIS_WWAN_SMS_SEND structure.

Miniport drivers process this OID asynchronously, and should return an NDIS_STATUS_INDICATION_REQUIRED provisional response to any set requests. Miniport drivers should send an NDIS_STATUS_WWAN_SMS_SEND indication when they have completed the transaction.

Remarks

For more information about using this OID, see WWAN SMS Operations.

When processing this OID, miniport drivers can access the provider network, but should not access the Subscriber Identity Module (SIM card).

OID_WWAN_SMS_SEND supports sending both PDU-mode and CDMA-mode SMS text messages, depending on the capabilities of the device.

GSM-based devices are expected to support only PDU-mode SMS text messages. CDMA-based devices are expected to support only CDMA-mode SMS text messages. Miniport drivers must be able to complete set requests irrespective of SMS text message mode.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support SMS text messages, or the ability to send SMS text messages.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_SMS_SEND

WWAN SMS Operations

OID_WWAN_SMS_STATUS

Article • 02/18/2023

OID_WWAN_SMS_STATUS reports the status of the MB device's message store.

Set requests are not supported.

Query requests do not use a structure.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an **NDIS_STATUS_WWAN_SMS_STATUS** status notification that indicates the status of the MB device's message store when completing query requests.

Remarks

For more information about using this OID, see WWAN SMS Operations.

When processing this OID, miniport drivers can access the Subscriber Identity Module (SIM card), but should not access the provider network.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support SMS text messages.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

WWAN SMS Operations

NDIS_STATUS_WWAN_SMS_STATUS

OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS

Article • 02/18/2023

OID_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS sets information about the list of device services for which the MB device must send NDIS_STATUS_WWAN_DEVICE_SERVICE_EVENT notifications. The MB device should not indicate events for any device service which is not in this list.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a NDIS_STATUS_WWAN_DEVICE_SERVICE_SUBSCRIPTION status notification that contains the current list of event subscriptions on the MB device.

Callers requesting to set the MB device service event subscription list provide a NDIS_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS structure to the miniport driver with the appropriate information.

Requirements

Version	Versions: Supported in Windows 8 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_DEVICE_SERVICE_EVENT

NDIS_STATUS_WWAN_DEVICE_SERVICE_SUBSCRIPTION

NDIS_WWAN_SUBSCRIBE_DEVICE_SERVICE_EVENTS

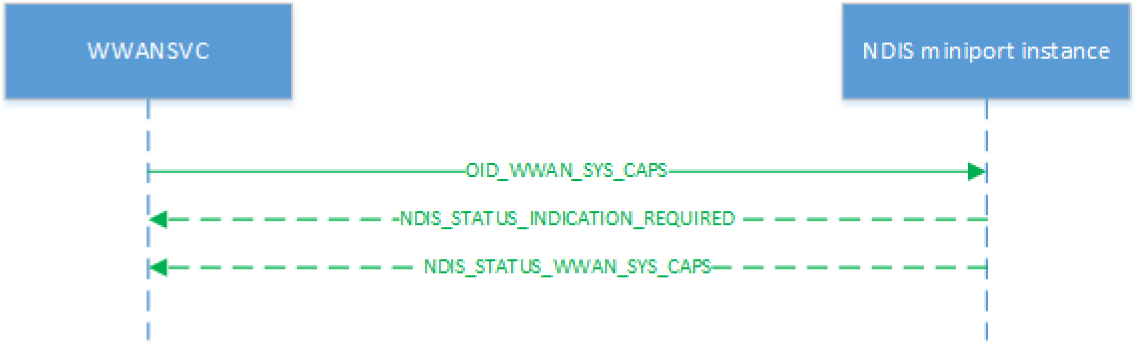
OID_WWAN_SYS_CAPS_INFO

Article • 02/18/2023

OID_WWAN_SYS_CAPS_INFO retrieves information about the modem. It can be sent on any of the NDIS instances exposed by the modem.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_SYS_CAPS_INFO status notification containing an NDIS_WWAN_SYS_CAPS_INFO structure, which in turn contains a WWAN_SYS_CAPS_INFO structure, to provide information about the overall modem system capability.

The following diagram illustrates a query request.



Set requests are not applicable.

Remarks

The host uses OID_WWAN_SYS_CAPS_INFO to query the number of devices (executors) and slots in the modem as well as the number of executors that may be active concurrently. A dual-standby modem would have a concurrency of 1; a dual-active modem would have a concurrency of 2. This OID is not executor-specific and may be sent to any NDIS instance.

The modem may expose multiple configurations with differing numbers of executors and slots. Regardless of which configuration is selected, this query will return the maximum number of devices and slots that the modem can support as currently configured.

A modem supporting OID_WWAN_SYS_CAPS_INFO is expected to also support OID_WWAN_DEVICE_CAPS_EX. Versions of Windows that support multi-executor modems will not use the legacy OID_WWAN_DEVICE_CAPS if the underlying modem supports OID_WWAN_SYS_CAPS_INFO. For legacy versions of the OS (any version before Windows 10 Version 1703 for the purposes of this OID), a multi-executor modem would be represented as multiple independent modems and the existing OID_WWAN_DEVICE_CAPS, available starting in Windows 8, will be used.

Requirements

Version	Windows 10, version 1703
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_SYS_CAPS_INFO

NDIS_WWAN_SYS_CAPS_INFO

WWAN_SYS_CAPS_INFO

OID_WWAN_DEVICE_CAPS_EX

OID_WWAN_DEVICE_CAPS

OID_WWAN_UE_POLICY

Article • 02/18/2023

OID_WWAN_UE_POLICY returns the UE policies from an MB device.

Set requests are not supported.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_UE_POLICY_STATE status notification containing an **NDIS_WWAN_UE_POLICY_INFO** structure that indicates the MB device's UE policies when completing query requests.

Miniport drivers can also send unsolicited events with this notification.

Requirements

Version: Windows 11, version 21H2

Header: Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_UE_POLICY_STATE

NDIS_WWAN_UE_POLICY_INFO

OID_WWAN_UICC_ACCESS_BINARY

Article • 02/18/2023

OID_WWAN_UICC_ACCESS_BINARY accesses a UICC binary file, the structure type of which is **WwanUiccFileStructureTransparent** or **WwanUiccFileStructureBerTLV**.

Query requests read a binary file. Query payloads contain an **NDIS_WWAN_UICC_ACCESS_BINARY** structure specifying information about the file to read. Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_UICC_BINARY_RESPONSE status notification containing an **NDIS_WWAN_UICC_RESPONSE** structure that describes the UICC's response.

Remarks

For more information about usage of this OID, see MB UICC application and file system access.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

NDIS_STATUS_WWAN_UICC_BINARY_RESPONSE

NDIS_WWAN_UICC_ACCESS_BINARY

NDIS_WWAN_UICC_RESPONSE

OID_WWAN_UICC_ACCESS_RECORD

Article • 02/18/2023

OID_WWAN_UICC_ACCESS_RECORD accesses a UICC linear fixed or cyclic file, the structure type of which is **WwanUiccFileStructureCyclic** or **WwanUiccFileStructureLinear**.

Query requests read the contents of a record. Query payloads contain an **NDIS_WWAN_UICC_ACCESS_RECORD** structure specifying information about the file to read. Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_UICC_RECORD_RESPONSE status notification containing an **NDIS_WWAN_UICC_RESPONSE** structure that describes the UICC's response.

Remarks

For more information about usage of this OID, see MB UICC application and file system access.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

NDIS_STATUS_WWAN_UICC_RECORD_RESPONSE

NDIS_WWAN_UICC_ACCESS_RECORD

NDIS_WWAN_UICC_RESPONSE

OID_WWAN_UICC_APP_LIST

Article • 02/18/2023

OID_WWAN_UICC_APP_LIST retrieves a list of applications in a UICC and information about them.

Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_UICC_UICC_APP_LIST status notification containing an **NDIS_WWAN_UICC_APP_LIST** structure that describes the app list for the UICC.

Set requests are not applicable.

Remarks

When the UICC in the modem is fully initialized and ready to register with the mobile operator, a UICC application must be selected for registration and a Query request with this OID should return the selected application in the **ActiveAppIndex** field of the **WWAN_UICC_APP_LIST** structure used in response.

For more information about usage of this OID, see MB UICC application and file system access.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

NDIS_STATUS_WWAN_UICC_UICC_APP_LIST

NDIS_WWAN_UICC_APP_LIST

WWAN_UICC_APP_LIST

OID_WWAN_UICC_ATR

Article • 02/18/2023

An OID_WWAN_UICC_ATR query request is sent by the mobile broadband host to a modem miniport adapter to get the UICC smart card's Answer to Reset (ATR) information. The ATR is the first string of bytes sent by the UICC after a reset has been performed. It describes the capabilities of the card, such as the number of logical channels that it supports.

The host can query the ATR information from either the active SIM slot or the inactive SIM slot in the device if the device supports dual SIM slots. This OID's payload contains an **NDIS_WWAN_QUERY_ATR_INFO** structure, which in turn contains a **WWAN_QUERY_ATR_INFO** structure that specifies the UICC slot ID.

Modem miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_ATR_INFO notification containing a **NDIS_WWAN_ATR_INFO** structure, which in turn contains a **WWAN_QUERY_ATR_INFO** structure that represents the passthrough status of the adapter.

Unsolicited events are not applicable.

For more info, see MB low level UICC access.

Requirements

Version: Windows 10, version 1607

Header: Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_ATR_INFO

NDIS_WWAN_QUERY_ATR_INFO

WWAN_QUERY_ATR_INFO

NDIS_WWAN_ATR_INFO

WWAN_ATR_INFO

MB low level UICC access

OID_WWAN_UICC_FILE_STATUS

Article • 02/18/2023

OID_WWAN_UICC_FILE_STATUS retrieves information about a specified UICC file.

Query payloads contain an **NDIS_WWAN_UICC_FILE_PATH** structure containing information about the target file. Miniport drivers must process Query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_UICC_FILE_STATUS status notification containing an **NDIS_WWAN_UICC_FILE_STATUS** structure that describes the specified file.

Set requests are not applicable.

Remarks

For more information about usage of this OID, see MB UICC application and file system access.

Requirements

Version: Windows 10, version 1903 **Header**: Ntddndis.h (include Ndis.h)

See also

MB UICC application and file system access

NDIS_STATUS_WWAN_UICC_UICC_FILE_STATUS

NDIS_WWAN_UICC_FILE_PATH

NDIS_WWAN_UICC_FILE_STATUS

OID_WWAN_UICC_RESET

Article • 02/18/2023

OID_WWAN_UICC_RESET is sent by the mobile broadband host to a modem miniport adapter to reset a UICC smart card and specify the UICC's passthrough status after reset, or query the passthrough state of the adapter.

The host can query the passthrough state from either the active SIM slot or inactive SIM slot in the device if the device supports dual SIM slots. This OID's payload for query requests contains an **NDIS_WWAN_QUERY_UICC_RESET** structure, which in turn contains a **WWAN_QUERY_UICC_RESET** structure that specifies the UICC slot ID.

Modem miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_UICC_RESET_INFO notification containing a **NDIS_WWAN_UICC_RESET_INFO** structure, which in turn contains a **WWAN_UICC_RESET_INFO** structure that represents the passthrough status of the adapter.

The host can reset either the active SIM slot or inactive SIM slot in the device if the device supports dual SIM slots. For set requests, OID_WWAN_UICC_RESET uses the **NDIS_WWAN_SET_UICC_RESET** structure, which in turn contains a **WWAN_SET_UICC_RESET** structure that represents the passthrough action the host specifies for the miniport adapter after it resets the UICC. After reset is complete, the miniport adapter responds with the NDIS_STATUS_WWAN_UICC_RESET_INFO notification, which in turn contains a **NDIS_WWAN_UICC_RESET_INFO** structure, to indicate its passthrough status.

Unsolicited events are not applicable.

For more info about passthrough actions and passthrough status, see MB low level UICC access.

Requirements

Version: Windows 10, version 1709

Header: Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_QUERY_UICC_RESET

WWAN_QUERY_UICC_RESET

NDIS_WWAN_UICC_RESET_INFO

WWAN_UICC_RESET_INFO

NDIS_WWAN_SET_UICC_RESET

WWAN_SET_UICC_RESET

NDIS_STATUS_WWAN_UICC_RESET_INFO

MB low level UICC access

OID_WWAN_UICC_TERMINAL_CAPABILITY

Article • 02/18/2023

The mobile broadband host sends OID_WWAN_UICC_TERMINAL_CAPABILITY to a modem miniport adapter to inform the modem about the terminal capabilities of the host.

If the device supports dual SIM slots, the host can set terminal capability on either the active or inactive SIM slot to inform the modem of the host OS's capabilities. The host can also query the terminal capability that persisted in the modem from a previous SIM insertion/reset.

For query requests, this OID's payload contains an **NDIS_WWAN_QUERY_UICC_TERMINAL_CAPABILITY** structure, which in turn contains a **WWAN_QUERY_UICC_TERMINAL_CAPABILITY** structure that specifies the slot ID. Modem miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request before later sending an NDIS_STATUS_WWAN_UICC_TERMINAL_CAPABILITY_INFO notification containing an **NDIS_WWAN_UICC_TERMINAL_CAPABILITY_INFO** structure, which in turn contains a **WWAN_UICC_TERMINAL_CAPABILITY_INFO** structure that represents the terminal capabilities.

For set requests, OID_WWAN_UICC_TERMINAL_CAPABILITY uses the **NDIS_WWAN_SET_UICC_TERMINAL_CAPABILITY** structure, which in turn contains a **WWAN_SET_UICC_TERMINAL_CAPABILITY** structure that represents the terminal capabilities and specifies the slot ID. The miniport driver responds with the NDIS_STATUS_WWAN_UICC_TERMINAL_CAPABILITY_INFO notification, which in turn contains an **NDIS_WWAN_UICC_TERMINAL_CAPABILITY_INFO** structure.

Unsolicited events are not applicable.

For more information, see MB low level UICC access.

Requirements

Version: Windows 10, version 1607

Header: Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_UICC_TERMINAL_CAPABILITY_INFO

NDIS_WWAN_UICC_TERMINAL_CAPABILITY_INFO

WWAN_UICC_TERMINAL_CAPABILITY_INFO

NDIS_WWAN_SET_UICC_TERMINAL_CAPABILITY

WWAN_SET_UICC_TERMINAL_CAPABILITY

MB low level UICC access

OID_WWAN_USSD

Article • 02/18/2023

OID_WWAN_USSD sends Unstructured Supplementary Service Data (USSD) requests to the underlying MB device.

Query requests are not supported.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_USSD status notification containing the status of the initial USSD request when they have completed the transaction.

Windows does not send an OID_WWAN_USSD request to a miniport driver if a previous request is still in progress, with the exception of a request to cancel a pending operation by setting the WWAN_USSD_REQUEST **RequestType** member of the request to *WwanUssdRequestCancel*.

When a request is canceled, the miniport driver must respond to both the canceled request and the cancel request.

Requirements

Version	Supported starting with Windows 8.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_STATUS_WWAN_USSD

WWAN_USSD_REQUEST

OID_WWAN_VENDOR_SPECIFIC

Article • 02/18/2023

OID_WWAN_VENDOR_SPECIFIC allows miniport drivers to implement vendor specific objects.

Query requests are not supported.

Miniport drivers must process set requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending a **NDIS_STATUS_WWAN_VENDOR_SPECIFIC** status notification containing a vendor-defined structure to implement private objects when they have completed the transaction.

Remarks

For more information about using this OID, see WWAN Vendor Specific Operations.

Miniport drivers should return NDIS_STATUS_NOT_SUPPORTED if they do not support vendor-specific operations.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

WWAN Vendor Specific Operations

NDIS_STATUS_WWAN_VENDOR_SPECIFIC

OID_WWAN_VISIBLE_PROVIDERS

Article • 02/18/2023

OID_WWAN_VISIBLE_PROVIDERS returns a list of network providers currently visible within the MB device's range.

Set requests are not supported.

Miniport drivers must process query requests asynchronously, initially returning NDIS_STATUS_INDICATION_REQUIRED to the original request, and later sending an NDIS_STATUS_WWAN_VISIBLE_PROVIDERS status notification containing an NDIS_WWAN_VISIBLE_PROVIDERS structure to provide information about visible network providers when completing query requests.

Query requests specify NDIS_WWAN_GET_VISIBLE_PROVIDERS structure as input. When the **Action** member in WWAN_GET_VISIBLE_PROVIDERS is set to WWAN_GET_VISIBLE_PROVIDERS_ALL the miniport should return all visible providers. When the **Action** member in WWAN_GET_VISIBLE_PROVIDERS is set to WWAN_GET_VISIBLE_PROVIDERS_MULTICARRIER the miniport should only return visible multi-carrier providers that can be set as the home provider.

The visible provider list returned by the device should have the provider state set correctly for each of the providers. For example, the multicarrier preferred providers should be tagged as WWAN_PROVIDER_STATE_PREFERRED_MULTICARRIER, the current home provider if any should be tagged as WWAN_PROVIDER_STATE_HOME, The current registered provider if any should be tagged as WWAN_PROVIDER_STATE_REGISTERED.

The **Rssi** and **ErrorRate** members of WWAN_PROVIDER2 structure should be set if available.

Remarks

For more information about using this OID, see WWAN Provider Operations.

Miniport drivers can access the Subscriber Identity Module (SIM card) when processing query operations, but should not access the provider network.

Miniport drivers should set the **VisibleListHeader.ElementType** member to *WwanStructProvider*.

For CDMA-based networks, miniport driver should return only the home provider, if any of the networks in the Preferred Roaming List (PRL) is currently visible. For GSM-based networks, more than one provider may be present in the visible provider list.

Devices that do not support scanning for visible providers while connected should return the WWAN_STATUS_BUSY error value in the **uStatus** member of the NDIS_WWAN_VISIBLE_PROVIDERS structure.

Both GSM-based and CDMA-based devices must support scanning for visible providers while in registered mode. However, miniport drivers are not required to support scanning for visible provider while a Packet Data Protocol (PDP) context is active (for example, the device is connected to the provider's network).

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Ntddndis.h (include Ndis.h)

See also

NDIS_WWAN_VISIBLE_PROVIDERS

NDIS_STATUS_WWAN_VISIBLE_PROVIDERS

WWAN Provider Operations

Ws2def.h

Article • 12/15/2021

This section contains kernel mode network driver topics for the Ws2def.h header. This header is included in the Windows SDK as it is also shared with user mode networking applications.

The Ws2def.h header contains definitions for the Winsock2 specification. It is included in Winsock2.h. User mode applications should include Winsock2.h rather than including Ws2def.h directly. Ws2def.h cannot be included by a module that also includes Winsock.h.

> ⓘ **Important**
>
> This section's topics contains pages for definitions, macros, OIDs, status indications, and other data structures that are not part of network driver reference (structures, enumerations, functions, and callbacks).
>
> For more information about network driver reference for this header, see **Ws2def.h (reference)**.

In this section

- AF_INET
- AF_INET6
- SIO_ADDRESS_LIST_CHANGE
- SIO_ADDRESS_LIST_QUERY
- SO_BROADCAST
- SO_CONDITIONAL_ACCEPT
- SO_EXCLUSIVEADDRUSE
- SO_KEEPALIVE
- SO_RCVBUF
- SO_REUSEADDR

AF_INET

Article • 03/03/2023

The AF_INET address family is the address family for IPv4.

Socket Address Structure

An IPv4 transport address is specified with the SOCKADDR_IN structure.

Socket Types

IPv4 supports the following socket types:

SOCK_STREAM
Supports reliable connection-oriented byte stream communication.

SOCK_DGRAM
Supports unreliable connectionless datagram communication.

SOCK_RAW
Supports raw access to the transport protocol.

A WSK application specifies a socket type when it calls the WskSocket function or the WskSocketConnect function to create a new socket.

Protocols

The following IPv4 IPPROTO_*XXX* protocol values of the IPPROTO enumeration are defined in the WSK header files:

IPPROTO_IP
Internet protocol options

IPPROTO_ICMP
Internet control message protocol

IPPROTO_IGMP
Internet group management protocol

IPPROTO_GGP
Gateway to gateway protocol

IPPROTO_IPV4

IPv4 encapsulation

IPPROTO_ST

Stream protocol

IPPROTO_TCP

Transmission control protocol

IPPROTO_CBT

Core based trees protocol

IPPROTO_EGP

Exterior gateway protocol

IPPROTO_IGP

Private interior gateway protocol

IPPROTO_PUP

PARC universal packet protocol

IPPROTO_UDP

User datagram protocol

IPPROTO_IDP

Internet datagram protocol

IPPROTO_RDP

Reliable data protocol

IPPROTO_ND

Net disk protocol

IPPROTO_ICLFXBM

Wideband monitoring

IPPROTO_PIM

Protocol independent multicast

IPPROTO_PGM

Pragmatic general multicast

IPPROTO_L2TP

Level 2 tunneling protocol

IPPROTO_SCTP
Stream control transmission protocol

IPPROTO_RAW
Raw IP packets

Additional protocols are supported through the use of raw sockets.

A WSK application specifies a protocol when it calls the **WskSocket** function or the **WskSocketConnect** function to create a new socket.

A WSK application also specifies a protocol (as the *Level* parameter) when it calls the **WskControlSocket** function to set or retrieve transport protocol level or network protocol level socket options.

Combinations

IPv4 supports the following combinations of socket types and protocols for each WSK socket category:

Basic Sockets SOCK_STREAM + IPPROTO_TCP SOCK_DGRAM + IPPROTO_UDP SOCK_RAW + IPPROTO_*Xxx* Listening Sockets SOCK_STREAM + IPPROTO_TCP

Datagram Sockets SOCK_DGRAM + IPPROTO_UDP SOCK_RAW + IPPROTO_*Xxx* Connection-Oriented Sockets SOCK_STREAM + IPPROTO_TCP

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

AF_INET6

Article • 03/03/2023

The AF_INET6 address family is the address family for IPv6.

Socket Address Structure

An IPv6 transport address is specified with the SOCKADDR_IN6 structure.

Socket Types

IPv6 supports the following socket types:

SOCK_STREAM
Supports reliable connection-oriented byte stream communication.

SOCK_DGRAM
Supports unreliable connectionless datagram communication.

SOCK_RAW
Supports raw access to the transport protocol.

A WSK application specifies a socket type when it calls the WskSocket function or the WskSocketConnect function to create a new socket.

Protocols

The following IPv6 IPPROTO_*XXX* protocol values of the IPPROTO enumeration are defined in the WSK header files:

IPPROTO_HOPOPTS
IPv6 hop-by-hop options

IPPROTO_ICMP
Internet control message protocol

IPPROTO_IGMP
Internet group management protocol

IPPROTO_GGP
Gateway to gateway protocol

IPPROTO_IPV4

IPv4 encapsulation

IPPROTO_ST

Stream protocol

IPPROTO_TCP

Transmission control protocol

IPPROTO_CBT

Core based trees protocol

IPPROTO_EGP

Exterior gateway protocol

IPPROTO_IGP

Private interior gateway protocol

IPPROTO_PUP

PARC universal packet protocol

IPPROTO_UDP

User datagram protocol

IPPROTO_IDP

Internet datagram protocol

IPPROTO_RDP

Reliable data protocol

IPPROTO_IPV6

IPv6 header

IPPROTO_ROUTING

IPv6 routing header

IPPROTO_FRAGMENT

IPv6 fragmentation header

IPPROTO_ESP

Encapsulating security payload

IPPROTO_AH

Authentication header

IPPROTO_ICMPV6

IPv6 Internet control message protocol

IPPROTO_NONE

IPv6 no next header

IPPROTO_DSTOPTS

IPv6 destination options

IPPROTO_ND

Net disk protocol

IPPROTO_ICLFXBM

Wideband monitoring

IPPROTO_PIM

Protocol independent multicast

IPPROTO_PGM

Pragmatic general multicast

IPPROTO_L2TP

Level 2 tunneling protocol

IPPROTO_SCTP

Stream control transmission protocol

IPPROTO_RAW

Raw IP packets

Additional protocols are supported through the use of raw sockets.

A WSK application specifies a protocol when it calls the **WskSocket** function or the **WskSocketConnect** function to create a new socket.

A WSK application also specifies a protocol (as the *Level* parameter) when it calls the **WskControlSocket** function to set or retrieve transport protocol level or network protocol level socket options.

Combinations

IPv6 supports the following combinations of socket types and protocols for each WSK socket category:

Basic Sockets SOCK_STREAM + IPPROTO_TCP SOCK_DGRAM + IPPROTO_UDP SOCK_RAW + IPPROTO_*Xxx* Listening Sockets SOCK_STREAM + IPPROTO_TCP

Datagram Sockets SOCK_DGRAM + IPPROTO_UDP SOCK_RAW + IPPROTO_*Xxx* Connection-Oriented Sockets SOCK_STREAM + IPPROTO_TCP

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SIO_ADDRESS_LIST_CHANGE

Article • 03/03/2023

The SIO_ADDRESS_LIST_CHANGE socket I/O control operation notifies a WSK application when there has been a change to the list of local transport addresses for a socket's address family. This socket I/O control operation applies to all socket types.

To be notified when there has been a change to the list of local transport addresses for a socket's address family, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_ADDRESS_LIST_CHANGE
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to be notified of a change to the list of local transport addresses for a socket's address family. The WSK subsystem queues the IRP and returns STATUS_PENDING. If a change is made to the list of local transport addresses for the socket's address family, the WSK subsystem completes the IRP. When the IRP's completion routine is called, the WSK application can use the SIO_ADDRESS_LIST_QUERY socket I/O control operation to query the new list of local transport addresses for the socket's address family.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SIO_ADDRESS_LIST_QUERY

Article • 03/03/2023

The SIO_ADDRESS_LIST_QUERY socket I/O control operation allows a WSK application to query the current list of local transport addresses for a socket's address family. This socket I/O control operation applies to all socket types.

To query the current list of local transport addresses for a socket's address family, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskIoctl**
ControlCode	SIO_ADDRESS_LIST_QUERY
Level	0
InputSize	0
InputBuffer	NULL
OutputSize	The size, in bytes, of the buffer that is pointed to by the *OutputBuffer* parameter.
OutputBuffer	A pointer to the buffer that receives the current list of local transport addresses. The size of the buffer is specified in the *OutputSize* parameter.
OutputSizeReturned	A pointer to a ULONG-typed variable that receives the number of bytes of data that is copied into the buffer that is pointed to by the *OutputBuffer* parameter.

A WSK application does not specify a pointer to an IRP when calling the **WskControlSocket** function to query the current list of local transport addresses for a socket's address family.

If the call to the **WskControlSocket** function succeeds, the output buffer contains a SOCKET_ADDRESS_LIST structure followed by the SOCKADDR structures for each of the local transport addresses for the socket's address family.

If the **WskControlSocket** function returns STATUS_BUFFER_OVERFLOW, the variable that is pointed to by the *OutputSizeReturned* parameter contains the output buffer size, in bytes, that is required to contain the complete list of local transport addresses for the socket's address family.

The SIO_ADDRESS_LIST_CHANGE socket I/O control operation allows a WSK application to be notified when there has been a change to the list of local transport addresses for a socket's address family.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SO_BROADCAST

Article • 12/15/2021

The state of the SO_BROADCAST socket option determines whether broadcast messages can be transmitted over a datagram socket. This socket option applies only to datagram sockets.

To set the state of this socket option, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_BROADCAST
Level	SOL_SOCKET
InputSize	sizeof(ULONG)
InputBuffer	A pointer to a ULONG-typed variable that contains the value for the new state of the socket option: 0: Do not allow broadcast messages 1: Allow broadcast messages
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

To retrieve the state of this socket option, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskGetOption**
ControlCode	SO_BROADCAST
Level	SOL_SOCKET
InputSize	0
InputBuffer	NULL

Parameter	Value
OutputSize	sizeof(ULONG)
OutputBuffer	A pointer to a ULONG-typed variable that receives the value of the state of the socket option: 0: Broadcast messages are not allowed 1: Broadcast messages are allowed
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or retrieve the state of the SO_BROADCAST socket option.

The default state of this socket option is that broadcast messages are not allowed.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SO_CONDITIONAL_ACCEPT

Article • 12/15/2021

The state of the SO_CONDITIONAL_ACCEPT socket option determines whether conditional acceptance mode is enabled on a listening socket. This socket option applies only to listening sockets.

If a WSK application sets this socket option, it must do so before the listening socket is bound to a local transport address.

To set the state of this socket option, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_CONDITIONAL_ACCEPT
Level	SOL_SOCKET
InputSize	sizeof(ULONG)
InputBuffer	A pointer to a ULONG-typed variable that contains the value for the new state of the socket option: 0: Disable conditional accept mode 1: Enable conditional accept mode
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

To retrieve the state of this socket option, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskGetOption**
ControlCode	SO_CONDITIONAL_ACCEPT
Level	SOL_SOCKET

Parameter	Value
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(ULONG)
OutputBuffer	A pointer to a ULONG-typed variable that receives the value of the state of the socket option: 0: Conditional accept mode is disabled 1: Conditional accept mode is enabled
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or retrieve the state of the SO_CONDITIONAL_ACCEPT socket option.

The default state of this socket option is that conditional accept mode is disabled.

Some transport protocols might not support conditional accept mode on listening sockets.

For more information about conditionally accepting incoming connections, see Listening for and Accepting Incoming Connections.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SO_EXCLUSIVEADDRUSE

Article • 12/15/2021

The state of the SO_EXCLUSIVEADDRUSE socket option determines whether the local transport address to which a socket will be bound is exclusively reserved for use by that socket. This socket option applies only to listening sockets, datagram sockets, and connection-oriented sockets.

If a WSK application sets this socket option, it must do so before the socket is bound to a local transport address.

To set the state of this socket option, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_EXCLUSIVEADDRUSE
Level	SOL_SOCKET
InputSize	sizeof(ULONG)
InputBuffer	A pointer to a ULONG-typed variable that contains the value for the new state of the socket option: 0: Disable exclusive use of the local transport address 1: Enable exclusive use of the local transport address
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

To retrieve the state of this socket option, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskGetOption**

Parameter	Value
ControlCode	SO_EXCLUSIVEADDRUSE
Level	SOL_SOCKET
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(ULONG)
OutputBuffer	A pointer to a ULONG-typed variable that receives the value of the state of the socket option: 0: Exclusive use of the local transport address is disabled 1: Exclusive use of the local transport address is enabled
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or retrieve the state of the SO_EXCLUSIVEADDRUSE socket option.

The default state of this socket option is that exclusive use of the local transport address is disabled.

For more information about using the SO_EXCLUSIVEADDRUSE socket option and its impact on the sharing of local transport addresses between sockets, see Sharing Transport Addresses.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SO_KEEPALIVE

Article • 12/15/2021

The state of the SO_KEEPALIVE socket option determines whether keep-alive packets are sent on a connection-oriented socket. This socket option applies only to listening sockets and connection-oriented sockets.

To set the state of this socket option, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_KEEPALIVE
Level	SOL_SOCKET
InputSize	sizeof(ULONG)
InputBuffer	A pointer to a ULONG-typed variable that contains the value for the new state of the socket option: • 0: Disable sending keep-alive packets • 1: Enable sending keep-alive packets
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

To retrieve the state of this socket option, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskGetOption**
ControlCode	SO_KEEPALIVE
Level	SOL_SOCKET
InputSize	0
InputBuffer	NULL

Parameter	Value
OutputSize	sizeof(ULONG)
OutputBuffer	A pointer to a ULONG-typed variable that receives the value of the state of the socket option: - 0: Sending keep-alive packets is disabled - 1: Sending keep-alive packets is enabled
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or retrieve the state of the SO_KEEPALIVE socket option.

The default state of this socket option is that sending keep-alive packets is disabled.

If this socket option is enabled on a listening socket, all incoming connections that are accepted on that listening socket have this socket option enabled by default. A WSK application can call the **WskControlSocket** function on an accepted socket to override the state of this socket option that was inherited from the listening socket.

Keep-alive packets are sent by the underlying network transport. Not all network transports support sending keep-alive packets.

For more information about using keep-alive packets, see *RFC 1122*, section 4.2.3.6, "TCP Keep-Alives".

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SO_RCVBUF

Article • 12/15/2021

The SO_RCVBUF socket option determines the size of a socket's receive buffer that is used by the underlying transport. This socket option applies only to listening sockets, datagram sockets, and connection-oriented sockets.

To set the value of this socket option, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_RCVBUF
Level	SOL_SOCKET
InputSize	sizeof(ULONG)
InputBuffer	A pointer to a ULONG-typed variable that contains the new size of the socket's receive buffer
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

To retrieve the value of the SO_RCVBUF socket option, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskGetOption**
ControlCode	SO_RCVBUF
Level	SOL_SOCKET
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(ULONG)

Parameter	Value
OutputBuffer	A pointer to a ULONG-typed variable that receives the current size of the socket's receive buffer
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or retrieve the value of the SO_RCVBUF socket option.

The default size of a socket's receive buffer is transport-specific. Some transports might not support this socket option.

If this socket option is set on a listening socket, all incoming connections that are accepted on that listening socket have their receive buffer set to the same size that is specified for the listening socket. A WSK application can call the **WskControlSocket** function on an accepted socket to override the size of the receive buffer that was inherited from the listening socket.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

SO_REUSEADDR

Article • 12/15/2021

The state of the SO_REUSEADDR socket option determines whether the local transport address to which a socket will be bound is always shared with other sockets. This socket option applies only to listening sockets, datagram sockets, and connection-oriented sockets.

If a WSK application sets this socket option, it must do so before the socket is bound to a local transport address.

To set the state of this socket option, a WSK application calls the WskControlSocket function with the following parameters.

Parameter	Value
RequestType	**WskSetOption**
ControlCode	SO_REUSEADDR
Level	SOL_SOCKET
InputSize	sizeof(ULONG)
InputBuffer	A pointer to a ULONG-typed variable that contains the value for the new state of the socket option: • 0: Disable always sharing the local transport address • 1: Enable always sharing the local transport address
OutputSize	0
OutputBuffer	NULL
OutputSizeReturned	NULL

To retrieve the state of this socket option, a WSK application calls the **WskControlSocket** function with the following parameters.

Parameter	Value
RequestType	**WskGetOption**

Parameter	Value
ControlCode	SO_REUSEADDR
Level	SOL_SOCKET
InputSize	0
InputBuffer	NULL
OutputSize	sizeof(ULONG)
OutputBuffer	A pointer to a ULONG-typed variable that receives the value of the state of the socket option: • 0: Always sharing the local transport address is disabled • 1: Always sharing the local transport address is enabled
OutputSizeReturned	NULL

A WSK application must specify a pointer to an IRP when calling the **WskControlSocket** function to set or retrieve the state of the SO_REUSEADDR socket option.

The default state of this socket option is that always sharing the local transport address is disabled.

For more information about using the SO_REUSEADDR socket option and its impact on the sharing of local transport addresses between sockets, see Sharing Transport Addresses.

Requirements

Version	Available in Windows Vista and later versions of the Windows operating systems.
Header	Ws2def.h (include Wsk.h)

Network Drivers Prior to Windows Vista

Article • 12/15/2021

To access the design guide and reference topics for Windows 2000 and Windows XP network drivers, see Network Drivers Prior to Windows Vista.

Network drivers prior to Windows Vista used NDIS 5.1, which was superceded in Windows Vista and later by NDIS 6.X.

Related topics

Previous Versions of Network Drivers

Native 802.11 Wireless LAN Drivers

Article • 02/17/2023

To access the design guide and reference topics for Native 802.11 Wireless LAN drivers, see Native 802.11 Wireless LAN.

The Native 802.11 Wireless LAN interface was superseded in Windows 10 and later by the WLAN Universal Driver Model (WDI).

Related topics

WLAN Universal Driver Model

Previous Versions of Network Drivers

Cellular COM API Reference

Article • 12/15/2021

To access reference topics related to the Cellular COM API, see Cellular COM API reference.

The Cellular COM API was used in Windows Phone 8.1 and is deprecated in Windows 10 and later.

Related topics

Previous Versions of Network Drivers