XInput Game Controller APIs

07/14/2025

> **Important**
>
> GameInput is a functional superset of all legacy input APIs—XInput, DirectInput, Raw Input, Human Interface Device (HID), and WinRT—with new features that expose input devices of all kinds through a single consistent interface. It's available on all Windows platforms (including PC, Xbox, HoloLens, IoT), previous versions of Windows (all the way back to Windows 7), and is callable from GDK, Win32, and Universal Windows Platform (UWP) applications.

Purpose

XInput Game Controller API enables applications to receive input from a controller.

In this section

Expand table

Topic	Description
Programming Guide	This guide contains information on how to use the XInput API to interact with a controller when it is connected to a Windows PC.
Programming Reference	XInput functions and structures.

Developer audience

XInput Game Controller APIs is designed for use by developers who want to use a controller for their Windows applications.

Programming Guide (XInput Game Controller APIs)

This guide contains information on how to use the XInput API to interact with a controller when it is connected to a Windows PC.

- Getting Started With XInput - This section contains a description of XInput and describes how to get started using XInput in your application.
- XInput Versions - This section contains information on the versions of XInput.
- XInput and DirectInput - This section contains information on the differences between XInput and DirectInput, and how you can use both side by side.
- DirectInput and XUSB Devices - Contains information on the XUSB device mappings.
- XINPUT and Controller Subtypes - Lists the controller subtypes available in XInput.

For more information about specific API methods, see the Programming Reference.

Last updated on 07/14/2025

Getting Started With XInput in Windows applications

XInput enables Windows applications to process controller interactions (including controller rumble effects and voice input and output).

This topic provides a brief overview of the capabilities of XInput and how to set it up in an application. It includes the following:

Introduction to XInput

Applications can use the XInput API to communicate with gaming controllers when they are plugged into a Windows PC (up to four unique controllers can be plugged in at a time).

Using this API, any compatible connected controller can be queried for its state, and vibration effects can be set. Controllers that have the headset attached can also be queried for sound input and output devices that can be used with the headset for voice processing.

Controller Layout

Compatible controllers have two analog directional sticks, each with a digital button, two analog triggers, a digital directional pad with four directions, and eight digital buttons. The states of each of these inputs are returned in the **XINPUT_GAMEPAD** structure when the **XInputGetState** function is called.

The controller also has two vibration motors to supply force feedback effects to the user. The speeds of these motors are specified in the **XINPUT_VIBRATION** structure that is passed to the **XInputSetState** function to set vibration effects.

Optionally, a headset can be connected to the controller. The headset has a microphone for voice input, and a headphone for sound output. You can call the **XInputGetAudioDeviceIds** or legacy **XInputGetDSoundAudioDeviceGuids** function to obtain the device identifiers that correspond to the devices for the microphone and headphone. You can then use the Core Audio APIs to receive voice input and send sound output.

Using XInput

Using XInput is as simple as calling the XInput functions as required. Using the XInput functions, you can retrieve controller state, get headset audio IDs, and set controller rumble effects.

Multiple Controllers

The XInput API supports up to four controllers connected at any time. The XInput functions all require a *dwUserIndex* parameter that is passed in to identify the controller being set or queried. This ID will be in the range of 0-3 and is set automatically by XInput. The number corresponds to the port that the controller is plugged into, and is not modifiable.

Each controller displays which ID it is using by lighting up a quadrant on the "ring of light" in the center of the controller. A *dwUserIndex* value of 0 corresponds to the top-left quadrant; the numbering proceeds around the ring in clockwise order.

Applications should support multiple controllers.

Getting Controller State

Throughout the duration of an application, getting state from a controller will probably be done most often. From frame to frame in a game application, state should be retrieved and game information updated to reflect the controller changes.

To retrieve state, use the **XInputGetState** function:

C++

```
DWORD dwResult;
for (DWORD i=0; i< XUSER_MAX_COUNT; i++ )
{
    XINPUT_STATE state;
    ZeroMemory( &state, sizeof(XINPUT_STATE) );

    // Simply get the state of the controller from XInput.
    dwResult = XInputGetState( i, &state );
```

```
    if( dwResult == ERROR_SUCCESS )
    {
        // Controller is connected
    }
    else
    {
        // Controller is not connected
    }
}
```

Note that the return value of **XInputGetState** can be used to determine if the controller is connected. Applications should define a structure to hold internal controller information; this information should be compared against the results of **XInputGetState** to determine what changes, such as button presses or analog controller deltas, were made that frame. In the above example, *g_Controllers* represents such a structure.

Once the state has been retrieved in a **XINPUT_STATE** structure, you can check it for changes and get specific information about controller state.

The *dwPacketNumber* member of the **XINPUT_STATE** structure can be used to check if the state of the controller has changed since the last call to **XInputGetState**. If *dwPacketNumber* does not change between two sequential calls to **XInputGetState**, then there has been no change in state. If it differs, then the application should check the *Gamepad* member of the **XINPUT_STATE** structure to get more detailed state information.

For performance reasons, don't call **XInputGetState** for an 'empty' user slot every frame. We recommend that you space out checks for new controllers every few seconds instead.

Dead Zone

In order for users to have a consistent gameplay experience, your game must implement dead zone correctly. The dead zone is "movement" values reported by the controller even when the analog thumbsticks are untouched and centered. There is also a dead zone for the 2 analog triggers.

> ⓘ **Note**
>
> Games that use XInput that do not filter dead zone at all will experience poor gameplay. Please note that some controllers are more sensitive than others, thus the dead zone may vary from unit to unit. It is recommended that you test your games with several different controllers on different systems.

Applications should use "dead zones" on analog inputs (triggers, sticks) to indicate when a movement has been made sufficiently on the stick or trigger to be considered valid.

Your application should check for dead zones and respond appropriately, as in this example:

C++

```
XINPUT_STATE state = g_Controllers[i].state;

float LX = state.Gamepad.sThumbLX;
float LY = state.Gamepad.sThumbLY;

//determine how far the controller is pushed
float magnitude = sqrt(LX*LX + LY*LY);

//determine the direction the controller is pushed
float normalizedLX = LX / magnitude;
float normalizedLY = LY / magnitude;

float normalizedMagnitude = 0;

//check if the controller is outside a circular dead zone
if (magnitude > INPUT_DEADZONE)
{
    //clip the magnitude at its expected maximum value
    if (magnitude > 32767) magnitude = 32767;

    //adjust magnitude relative to the end of the dead zone
    magnitude -= INPUT_DEADZONE;

    //optionally normalize the magnitude with respect to its expected range
    //giving a magnitude value of 0.0 to 1.0
    normalizedMagnitude = magnitude / (32767 - INPUT_DEADZONE);
}
else //if the controller is in the deadzone zero out the magnitude
{
    magnitude = 0.0;
    normalizedMagnitude = 0.0;
}

//repeat for right thumb stick
```

This example calculates the controller's direction vector and how far along the vector the controller has been pushed. This allows enforcement of a circular deadzone by simply checking whether the controller's magnitude is greater than the deadzone value. In addition the code normalizes the controller's magnitude which can then be multiplied by a game specific factor to convert the controller's position to units relevant to the game.

Note that you may define your own dead zones for the sticks and triggers (anywhere from 0-65534), or you may use the provided deadzones defined as XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE, XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE, and XINPUT_GAMEPAD_TRIGGER_THRESHOLD in XInput.h:

C++

```
#define XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE  7849
#define XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE 8689
#define XINPUT_GAMEPAD_TRIGGER_THRESHOLD    30
```

Once the deadzone is enforced, you may find it useful to scale the resulting range [0.0..1.0] floating point (as in the example above), and optionally apply a non-linear transform.

For example, with driving games, it may be helpful to cube the result to provide a better feel to driving the cars using a gamepad, as cubing the result gives you more precision in the lower ranges, which is desirable, since gamers typically either apply soft force to get subtle movement or apply hard force all the way in one direction to get rd response.

Setting Vibration Effects

In addition to getting the state of the controller, you may also send vibration data to the controller to alter the feedback provided to the user of the controller. The controller contains two rumble motors that can be independently controlled by passing values to the **XInputSetState** function.

The speed of each motor can be specified using a WORD value in the **XINPUT_VIBRATION** structure that is passed to the **XInputSetState** function as follows:

C++

```
XINPUT_VIBRATION vibration;
ZeroMemory( &vibration, sizeof(XINPUT_VIBRATION) );
vibration.wLeftMotorSpeed = 32000; // use any value between 0-65535 here
vibration.wRightMotorSpeed = 16000; // use any value between 0-65535 here
XInputSetState( i, &vibration );
```

Note that the right motor is the high-frequency motor, the left motor is the low-frequency motor. They do not always need to be set to the same amount, as they provide different effects.

Getting Audio Device Identifiers

The headset for a controller has these functions:

- Record sound using a microphone
- Play back sound using a headphone

Use this code to obtain the device identifiers for the headset:

C++

```
WCHAR renderId[ 256 ] = {0};
WCHAR captureId[ 256 ] = {0};
UINT rcount = 256;
UINT ccount = 256;

XInputGetAudioDeviceIds( i, renderId, &rcount, captureId, &ccount );
```

After you obtain the device identifiers, you can create the appropriate interfaces. For example, if you use XAudio 2.8, use this code to create a mastering voice for this device:

C++

```
IXAudio2* pXAudio2 = NULL;
HRESULT hr;
if ( FAILED(hr = XAudio2Create( &pXAudio2, 0, XAUDIO2_DEFAULT_PROCESSOR ) ) )
    return hr;

IXAudio2MasteringVoice* pMasterVoice = NULL;
if ( FAILED(hr = pXAudio2->CreateMasteringVoice( &pMasterVoice,
XAUDIO2_DEFAULT_CHANNELS, XAUDIO2_DEFAULT_SAMPLERATE, 0, renderId, NULL,
AudioCategory_Communications ) ) )
    return hr;
```

For info about how to use the captureId device identifier, see Capturing a Stream.

Getting DirectSound GUIDs (legacy DirectX SDK only)

The headset that can be connected to a controller has two functions: it can record sound using a microphone, and it can play back sound using a headphone. In the XInput API, these functions are accomplished through DirectSound, using the **IDirectSound8** and **IDirectSoundCapture8** interfaces.

To associate the headset microphone and headphone with their appropriate DirectSound interfaces, you must get the DirectSoundGUIDs for the capture and render devices by calling **XInputGetDSoundAudioDeviceGuids**.

> ⓘ **Note**
>
> Use of the legacy DirectSound is not recommended, and is not available in Windows Store apps. The info in this section only applies to the DirectX SDK version of XInput (XInput 1.3). The Windows 8 version of XInput (XInput 1.4) exclusively uses Windows Audio Session API (WASAPI) device identifiers that are obtained through XInputGetAudioDeviceIds.

C++

```
XInputGetDSoundAudioDeviceGuids( i, &dsRenderGuid, &dsCaptureGuid );
```

Once you have retrieved the GUIDs you can create the appropriate interfaces by calling DirectSoundCreate8 and DirectSoundCaptureCreate8 like this:

C++

```
// Create IDirectSound8 using the controller's render device
if( FAILED( hr = DirectSoundCreate8( &dsRenderGuid, &pDS, NULL ) ) )
   return hr;

// Set coop level to DSSCL_PRIORITY
if( FAILED( hr = pDS->SetCooperativeLevel( hWnd, DSSCL_NORMAL ) ) )
   return hr;

// Create IDirectSoundCapture using the controller's capture device
if( FAILED( hr = DirectSoundCaptureCreate8( &dsCaptureGuid, &pDSCapture, NULL ) ) )
   return hr;
```

Related topics

Programming Reference

Last updated on 07/14/2025

XInput Versions

XInput is a cross-platform API that has shipped for use on Xbox and Windows. On Xbox, XInput ships as a static library that is compiled into the main game executable. On Windows, XInput is provided as a DLL that is installed into the system folders of the operating system.

There are three current versions of the XInput DLL today. Choose the appropriate version of XInput based on the functionality of XInput you use and the versions of Windows you intend to support.

- XInput 1.4: XInput 1.4 ships as part of Windows 10. Use this version for building UWP apps.
- XInput 9.1.0: XInput 9.1.0 ships as part of Windows Vista, Windows 7, and Windows 8. Use this version if your desktop app is intended to run on these versions of Windows and you are using basic XInput functionality.
- XInput 1.3: XInput 1.3 ships as a redistributable component in the DirectX SDK with support for Windows Vista, Windows 7, and Windows 8. Use this version if your desktop app is intended to run on these versions of Windows and you need functionality that is not supported by XInput 9.1.0.

XInput 1.4

XInput 1.4 ships today as a system component in Windows 8 as XINPUT1_4.DLL. It is available "inbox" and does not require redistribution with an application. The Windows Software Development Kit (SDK) contains the header and import library for statically linking against XINPUT1_4.DLL. To download the Windows 8 SDK, see Downloads for developing desktop apps.

XInput 1.4 has these primary advantages over other versions of XInput:

- This is the only version that can be used in C++/DirectX Windows Store apps.
- The new **XInputGetAudioDeviceIds** function provides an audio device ID string that you can use to open an XAudio2 mastering voice or audio device for a headset attached to a controller. The **XInputGetDSoundAudioDeviceGuids** function is not available in this version.
- Provides improved device capabilities reporting including XINPUT_CAPS_WIRELESS, XINPUT_CAPS_FFB_SUPPORTED, XINPUT_CAPS_PMD_SUPPORTED, and XINPUT_CAPS_NO_NAVIGATION flags and more accurate reporting of XINPUT_CAPS_VOICE_SUPPORTED. These flags are combined in the **Flags** member of the **XINPUT_CAPABILITIES** structure. The **XInputGetCapabilities** function returns **XINPUT_CAPABILITIES**.

XInput 9.1.0

Like XInput 1.4, XInput 9.1.0 ships today as a system component in Windows 10, Windows 8.x, Windows 7, and Windows Vista as XINPUT9_1_0.DLL. It is maintained primarily for backward compatibility with existing applications. It has a reduced function set so we recommend that you use XInput 1.4, if possible. But it is convenient to use for applications that must run on down-level versions of Windows but don't need the additional audio functionality provided by XInput 1.4 or XInput 1.3.

The Windows SDK contains the header and import library for statically linking against XINPUT9_1_0.DLL.

XInput 9.1.0 has these disadvantages over other versions of XInput:

- For backward compatibility reasons, **XInputGetCapabilities** in this version of XInput returns fixed capability information. Regardless of the controller device attached, **XInputGetCapabilities** in XInput 9.1.0 will always report a device subtype of GAMEPAD. It will not return the XINPUT_CAPS_WIRELESS capability bit even if a wireless device is connected.
- You can't determine the headset for a given user ID. The **XInputGetAudioDeviceIds** function is not available and **XInputGetDSoundAudioDeviceGuids** function will return no results on Windows 8.x or Windows 10.
- The **XInputEnable**, **XInputGetBatteryInformation**, and **XInputGetKeystroke** functions are not available.

XInput 1.3

Some previous versions of XInput have been provided as redistributable DLLs in the DirectX SDK. The first redistributable version of XInput, XInput 1.1, shipped in the April 2006 release of the DirectX SDK. The last version to ship in the DirectX SDK was XInput 1.3, available in the June 2010 release of the legacy DirectX SDK. *The DirectX SDK is no longer available on Microsoft Downloads*.

You can use XInput 1.3 for applications that support down-level versions of Windows and require functionality not provided by XInput 9.1.0 (that is, correct subtype reporting, audio support, explicit battery reporting support, and so on).

Last updated on 07/14/2025

Comparison of XInput and DirectInput features

> **ⓘ Important**
>
> See GameInput API for details on the next-generation input API supported on PC and Xbox through the Microsoft Game Development Kit (GDK).

This document compares XInput and DirectInput implementations of controller input and how to support both XInput devices and legacy DirectInput devices.

Windows Store apps do not support DirectInput.

Overview

XInput enables applications to receive input from the XUSB controllers. The APIs are available through the DirectX SDK, and the driver is available through Windows Update.

There are several advantages to using XInput over DirectInput:

- XInput is easier to use and requires less setup than DirectInput
- Both Xbox and Windows programming will use the same sets of core APIs, allowing programming to translate cross-platform much easier
- There will be a large installed base of controllers
- XInput device will have vibration functionality only when using XInput APIs

Using XUSB controllers with DirectInput

The XUSB controllers are properly enumerated on DirectInput, and can be used with the DirectInputAPIs. However, some functionality provided by XInput will be missing from the DirectInput implementation:

- The left and right trigger buttons will act as a single button, not independently
- The vibration effects will not be available
- Querying for headset devices will not be available

The combination of the left and right triggers in DirectInput is by design. Games have always assumed that DirectInput device axes are centered when there is no user interaction with the device. However, the newer controllers were designed to register minimum value, not center, when the triggers are not being held. Older games would therefore assume user interaction.

The solution was to combine the triggers, setting one trigger to a positive direction and the other to a negative direction, so no user interaction is indicative to DirectInput of the "control" being at center.

In order to test the trigger values separately, you must use XInput.

XInput and DirectInput Side by Side

By supporting XInput only, your game will not work with legacy DirectInput devices. XInput will not recognize these devices.

If you want your game to support legacy DirectInput devices, you may use DirectInput and XInput side by side. When enumerating your DirectInput devices, all DirectInput devices will enumerate correctly. All XInput devices will show up as both XInput and DirectInput devices, but they should not be handled through DirectInput. You will need to determine which of your DirectInput devices are legacy devices, and which are XInput devices, and remove them from the enumeration of DirectInput devices.

To do this, insert this code into your DirectInput enumeration callback:

C++

```
#include <wbemidl.h>
#include <oleauto.h>

#ifndef SAFE_RELEASE
#define SAFE_RELEASE(p) { if (p) { (p)->Release(); (p) = nullptr; } }
#endif

//-----------------------------------------------------------------------------
// Enum each PNP device using WMI and check each device ID to see if it contains
// "IG_" (ex. "VID_0000&PID_0000&IG_00"). If it does, then it's an XInput device
// Unfortunately this information cannot be found by just using DirectInput
//-----------------------------------------------------------------------------
BOOL IsXInputDevice( const GUID* pGuidProductFromDirectInput )
{
    IWbemLocator*           pIWbemLocator = nullptr;
    IEnumWbemClassObject*   pEnumDevices = nullptr;
    IWbemClassObject*       pDevices[20] = {};
    IWbemServices*          pIWbemServices = nullptr;
    BSTR                    bstrNamespace = nullptr;
    BSTR                    bstrDeviceID = nullptr;
    BSTR                    bstrClassName = nullptr;
    bool                    bIsXinputDevice = false;

    // CoInit if needed
    HRESULT hr = CoInitialize(nullptr);
    bool bCleanupCOM = SUCCEEDED(hr);
```

```
    // So we can call VariantClear() later, even if we never had a successful
IWbemClassObject::Get().
    VARIANT var = {};
    VariantInit(&var);

    // Create WMI
    hr = CoCreateInstance(__uuidof(WbemLocator),
        nullptr,
        CLSCTX_INPROC_SERVER,
        __uuidof(IWbemLocator),
        (LPVOID*)&pIWbemLocator);
    if (FAILED(hr) || pIWbemLocator == nullptr)
        goto LCleanup;

    bstrNamespace = SysAllocString(L"\\\\.\\root\\cimv2");  if (bstrNamespace ==
nullptr) goto LCleanup;
    bstrClassName = SysAllocString(L"Win32_PNPEntity");     if (bstrClassName ==
nullptr) goto LCleanup;
    bstrDeviceID = SysAllocString(L"DeviceID");             if (bstrDeviceID ==
nullptr)  goto LCleanup;

    // Connect to WMI
    hr = pIWbemLocator->ConnectServer(bstrNamespace, nullptr, nullptr, 0L,
        0L, nullptr, nullptr, &pIWbemServices);
    if (FAILED(hr) || pIWbemServices == nullptr)
        goto LCleanup;

    // Switch security level to IMPERSONATE.
    hr = CoSetProxyBlanket(pIWbemServices,
        RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE, nullptr,
        RPC_C_AUTHN_LEVEL_CALL, RPC_C_IMP_LEVEL_IMPERSONATE,
        nullptr, EOAC_NONE);
    if ( FAILED(hr) )
        goto LCleanup;

    hr = pIWbemServices->CreateInstanceEnum(bstrClassName, 0, nullptr,
&pEnumDevices);
    if (FAILED(hr) || pEnumDevices == nullptr)
        goto LCleanup;

    // Loop over all devices
    for (;;)
    {
        ULONG uReturned = 0;
        hr = pEnumDevices->Next(10000, _countof(pDevices), pDevices, &uReturned);
        if (FAILED(hr))
            goto LCleanup;
        if (uReturned == 0)
            break;

        for (size_t iDevice = 0; iDevice < uReturned; ++iDevice)
        {
            // For each device, get its device ID
            hr = pDevices[iDevice]->Get(bstrDeviceID, 0L, &var, nullptr, nullptr);
            if (SUCCEEDED(hr) && var.vt == VT_BSTR && var.bstrVal != nullptr)
```

```
            {
                // Check if the device ID contains "IG_".  If it does, then it's an
XInput device
                // This information cannot be found from DirectInput
                if (wcsstr(var.bstrVal, L"IG_"))
                {
                    // If it does, then get the VID/PID from var.bstrVal
                    DWORD dwPid = 0, dwVid = 0;
                    WCHAR* strVid = wcsstr(var.bstrVal, L"VID_");
                    if (strVid && swscanf_s(strVid, L"VID_%4X", &dwVid) != 1)
                        dwVid = 0;
                    WCHAR* strPid = wcsstr(var.bstrVal, L"PID_");
                    if (strPid && swscanf_s(strPid, L"PID_%4X", &dwPid) != 1)
                        dwPid = 0;

                    // Compare the VID/PID to the DInput device
                    DWORD dwVidPid = MAKELONG(dwVid, dwPid);
                    if (dwVidPid == pGuidProductFromDirectInput->Data1)
                    {
                        bIsXinputDevice = true;
                        goto LCleanup;
                    }
                }
            }
            VariantClear(&var);
            SAFE_RELEASE(pDevices[iDevice]);
        }
    }

LCleanup:
    VariantClear(&var);

    if(bstrNamespace)
        SysFreeString(bstrNamespace);
    if(bstrDeviceID)
        SysFreeString(bstrDeviceID);
    if(bstrClassName)
        SysFreeString(bstrClassName);

    for (size_t iDevice = 0; iDevice < _countof(pDevices); ++iDevice)
        SAFE_RELEASE(pDevices[iDevice]);

    SAFE_RELEASE(pEnumDevices);
    SAFE_RELEASE(pIWbemLocator);
    SAFE_RELEASE(pIWbemServices);

    if(bCleanupCOM)
        CoUninitialize();

    return bIsXinputDevice;
}


//-----------------------------------------------------------------------------
// Name: EnumJoysticksCallback()
```

```
// Desc: Called once for each enumerated joystick. If we find one, create a
//       device interface on it so we can play with it.
//-----------------------------------------------------------------------------
BOOL CALLBACK EnumJoysticksCallback( const DIDEVICEINSTANCE* pdidInstance,
                                     VOID* pContext )
{
    if( IsXInputDevice( &pdidInstance->guidProduct ) )
        return DIENUM_CONTINUE;

     // Device is verified not XInput, so add it to the list of DInput devices

     return DIENUM_CONTINUE;
}
```

> A slightly improved version of this code is in the legacy DirectInput Joystick sample.

Related topics

Getting Started With XInput

Programming Reference

Last updated on 07/14/2025

DirectInput and XUSB Devices

The driver for XUSB on Windows implements the kernel-mode interface for the XINPUT DLL. To provide a good experience for legacy titles that use the DirectInput API with the common controller device, the driver also exports a Human Interface Device (HID) class interface, which is picked up by DirectInput. We chose the mapping of XUSB to HID based on typical behavior in a set of gaming applications for the original XINPUT version, and we updated the mapping for newer subtypes. This topic describes the mapping.

Human Interface Device (HID)

HID standard is a standard from the Universal Serial Bus (USB) committee originally proposed by Microsoft to generalize protocols for input devices. It consists of a byte-code description language and can express gamepads, mice, joysticks, throttle and rudder controls, and multi-axis controllers. Because this standard is so generalized, you might have difficulty writing software that consumes input from arbitrary devices. Therefore, for the game-centric DirectInput API, we developed a specific sub-mapping of types to encourage hardware manufactures to support through their drivers.

- USB Device Class Definition for HID v1.11

> **Important**
>
> You can also access HID input devices via **RawInput API** and process input reports via low level **HID API** but vibration feedback will not work as with **DirectInput**.

Mappings

The XUSB driver implements both an XUSB class interface and a HID class interface for devices in order to support both XINPUT and DirectInput usage. This mapping is based on the XUSB subtype information. The driver implements four distinct groups of mappings.

Expand table

XUSB Subtype	Mapping
XINPUT_DEVSUBTYPE_GAMEPAD (Subtype 1)	Gamepad
XINPUT_DEVSUBTYPE_WHEEL (Subtype 2)	Wheel
XINPUT_DEVSUBTYPE_ARCADE_STICK (Subtype 3)	Arcade Stick/Arcade Pad

XUSB Subtype	Mapping
XINPUT_DEVSUBTYPE_FLIGHT_STICK (Subtype 4)	Flight Stick
XINPUT_DEVSUBTYPE_DANCE_PAD (Subtype 5)	Default for any new subtype
XINPUT_DEVSUBTYPE_GUITAR (Subtype 6)	Guitar
XINPUT_DEVSUBTYPE_GUITAR_ALTERNATE (Subtype 7)	
XINPUT_DEVSUBTYPE_DRUM_KIT (Subtype 8)	
XINPUT_DEVSUBTYPE_GUITAR_BASS (Subtype 11)	
XINPUT_DEVSUBTYPE_ARCADE_PAD (Subtype 19)	

> **Note**
>
> The following HID mappings are static. This means that even if the device capabilities report indicates that a particular button or axis is not supported, the mapping will still include it but will always report an off state or center value.

Gamepad

This is the default mapping and is designed around a standard gamepad, and is exposed as a *Gamepad* HID usage type.

Expand table

Control	HID Usage Name	Usage Page	Usage ID
Left Stick	X, Y	0x01	0x30, 0x31
Right Stick	Rx, Ry	0x01	0x33, 0x34
Left Trigger + Right Trigger	Z*	0x01	0x32
D-Pad Up, Down, Left, Right	Hat Switch	0x01	0x39
A	Button 1	0x09	0x01
B	Button 2	0x09	0x02
X	Button 3	0x09	0x03
Y	Button 4	0x09	0x04

Control	HID Usage Name	Usage Page	Usage ID
LB (left bumper)	Button 5	0x09	0x05
RB (right bumper)	Button 6	0x09	0x06
BACK	Button 7	0x09	0x07
START	Button 8	0x09	0x08
LSB (left stick button)	Button 9	0x09	0x09
RSB (right stick button)	Button 10	0x09	0x0A

> **Note**
>
> (*): This is combined so that Z exhibits the centering behavior expected by most titles for rotation; this does mean it is not possible to see all possible trigger combination values through DirectInput and HID.

Arcade Stick/Arcade Pad

This is the mapping designed around the Arcade Stick controller, and is exposed as a *Gamepad* HID usage type. The Arcade Pad is very much like an Arcade Stick, but in a smaller form-factor. These designs replace the analog Left Trigger and Right Trigger with digital buttons that report the minimum and maximum axis value.

Expand table

Control	HID Usage Name	Usage Page	Usage ID
D-Pad Up, Down, Left, Right	Hat Switch	0x01	0x39
A	Button 1	0x09	0x01
B	Button 2	0x09	0x02
X	Button 3	0x09	0x03
Y	Button 4	0x09	0x04
LB (left bumper)	Button 5	0x09	0x05
RB (right bumper)	Button 6	0x09	0x06
BACK	Button 7	0x09	0x07

Control	HID Usage Name	Usage Page	Usage ID
START	Button 8	0x09	0x08
Left Trigger	Button 9	0x09	0x09
Right Trigger	Button 10	0x09	0x0A

These devices may or may not support additional controls, but these are not exposed by the HID mapping: Left Stick, Right Stick, LSB (left stick button), and RSB (right stick button).

Wheel

This mapping is designed around a racing wheel, and is exposed as a *Gamepad* HID usage type.

Expand table

Control	HID Usage Name	Usage Page	Usage ID
Wheel (Left Stick X)	X	0x01	0x30
Accelerator Pedal (Right Trigger) + Brake Pedal (Left Trigger)	Z*	0x01	0x32
D-Pad Up, Down, Left, Right	Hat Switch	0x01	0x39
A	Button 1	0x09	0x01
B	Button 2	0x09	0x02
X	Button 3	0x09	0x03
Y	Button 4	0x09	0x04
LB (left bumper)	Button 5	0x09	0x05
RB (right bumper)	Button 6	0x09	0x06
LSB (left stick button)	Button 7	0x09	0x07
RSB (right stick button)	Button 8	0x09	0x08
BACK	Button 9	0x09	0x09
START	Button 10	0x09	0x0A

> **Note**
>
> (*): This is combined so that Z exhibits the centering behavior expected by most titles for the brake and accelerator controls; this does mean it is not possible to see all possible pedal combination values through DirectInput.

Flight Stick

This mapping is designed around a flight stick, and is exposed as a *Joystick* HID usage type.

Expand table

Control	Usage Name	Usage Page	Usage ID
Flight Stick (Left Stick)	X, Y	0x01	0x30, 0x31
POV Hat (Right Stick)	Rx, Ry	0x01	0x33, 0x34
Throttle (Right Trigger)	Z	0x01	0x32
Rudder (Left Trigger)	Rz	0x01	0x35
D-Pad Up, Down, Left, Right	Hat Switch	0x01	0x39
Primary Weapon (A)	Button 1	0x09	0x01
Secondary Weapon (B)	Button 2	0x09	0x02
X	Button 3	0x09	0x03
Y	Button 4	0x09	0x04
LB (left bumper)	Button 5	0x09	0x05
RB (right bumper)	Button 6	0x09	0x06
BACK	Button 7	0x09	0x07
START	Button 8	0x09	0x08
LSB (left stick button)	Button 9	0x09	0x09
RSB (right stick button)	Button 10	0x09	0x0A

> **Note**

This is based on the final Flight Stick design. Because this differs from early Flight Stick definitions, many devices have a mode switch that supports the old versus new model. This mapping assumes the new model.

Last updated on 07/14/2025

XINPUT and Controller Subtypes

A table of controller subtypes available in XInput.

Expand table

Subtype	Value	Meaning
XINPUT_DEVSUBTYPE_UNKNOWN	0x00	Unknown. The controller type is unknown.
XINPUT_DEVSUBTYPE_GAMEPAD	0x01	Gamepad controller. Includes Left and Right Sticks, Left and Right Triggers, Directional Pad, and all standard buttons (A, B, X, Y, START, BACK, LB, RB, LSB, RSB).
XINPUT_DEVSUBTYPE_WHEEL	0x02	Racing wheel controller. Left Stick X reports the wheel rotation, Right Trigger is the acceleration pedal, and Left Trigger is the brake pedal. Includes Directional Pad and most standard buttons (A, B, X, Y, START, BACK, LB, RB). LSB and RSB are optional.
XINPUT_DEVSUBTYPE_ARCADE_STICK	0x03	Arcade stick controller. Includes a Digital Stick that reports as a DPAD (up, down, left, right), and most standard buttons (A, B, X, Y, START, BACK). The Left and Right Triggers are implemented as digital buttons and report either 0 or 0xFF. LB, LSB, RB, and RSB are optional.
XINPUT_DEVSUBTYPE_FLIGHT_STICK	0x04	Flight stick controller. Includes a pitch and roll stick that reports as the Left Stick, a POV Hat which reports as the Right Stick, a rudder (handle twist or rocker) that reports as Left Trigger, and a throttle control as the Right Trigger. Includes support for a primary weapon (A), secondary weapon (B), and other standard buttons (X, Y, START, BACK). LB, LSB, RB, and RSB are optional.
XINPUT_DEVSUBTYPE_DANCE_PAD	0x05	Dance pad controller. Includes the Directional Pad and standard buttons (A, B, X, Y) on the pad, plus BACK and START.
XINPUT_DEVSUBTYPE_GUITAR	0x06	Guitar controller. The strum bar maps to DPAD (up and down), and the frets are assigned to A (green), B (red), Y (yellow), X (blue), and LB (orange). Right Stick Y is associated with a vertical orientation sensor; Right Stick X is the whammy bar. Includes support for

Subtype	Value	Meaning
		BACK, START, DPAD (left, right). Left Trigger (pickup selector), Right Trigger, RB, LSB (fret modifier), RSB are optional.
XINPUT_DEVSUBTYPE_GUITAR_ALTERNATE	0x07	Alternate guitar controller. Supports a larger range of movement for the vertical orientation sensor.
XINPUT_DEVSUBTYPE_DRUM_KIT	0x08	Drum controller. The drum pads are assigned to buttons: A for green (Floor Tom), B for red (Snare Drum), X for blue (Low Tom), Y for yellow (High Tom), and LB for the pedal (Bass Drum). Includes Directional-Pad, BACK, and START. RB, LSB, and RSB are optional.
XINPUT_DEVSUBTYPE_GUITAR_BASS	0x0B	Bass guitar controller. Identical to Guitar, with the distinct subtype to simplify setup.
XINPUT_DEVSUBTYPE_ARCADE_PAD	0x13	Arcade pad controller. Includes Directional Pad and most standard buttons (A, B, X, Y, START, BACK, LB, RB). The Left and Right Triggers are implemented as digital buttons and report either 0 or 0xFF. Left Stick, Right Stick, LSB, and RSB are optional.

> **Note**
>
> The legacy version of XINPUT on Windows Vista (XInput 9.1.0) always returns a fixed subtype of **XINPUT_DEVSUBTYPE_GAMEPAD**, regardless of attached device.

Last updated on 07/14/2025

Programming Reference

XInput functions and structures.

- XInput Functions
- XInput Structures

Last updated on 07/14/2025

Functions (Programming Reference)

Available XInput functions.

In this section

Expand table

Topic	Description
XInputEnable	Sets the reporting state of XInput.
XInputGetAudioDeviceIds	Retrieves the sound rendering and sound capture audio device IDs that are associated with the headset connected to the specified controller.
XInputGetBatteryInformation	Retrieves the battery type and charge status of a wireless controller.
XInputGetCapabilities	Retrieves the capabilities and features of a connected controller.
XInputGetDSoundAudioDeviceGuids	Gets the sound rendering and sound capture device GUIDs that are associated with the headset connected to the specified controller.
XInputGetKeystroke	Retrieves a gamepad input event.
XInputGetState	Retrieves the current state of the specified controller.
XInputSetState	Sends data to a connected controller. This function is used to activate the vibration function of a controller.

Last updated on 07/14/2025

XInputEnable function (xinput.h)

Article02/22/2024

Sets the reporting state of XInput.

Syntax

```
C++
```

```
void XInputEnable(
  [in] BOOL enable
);
```

Parameters

`[in] enable`

If enable is **FALSE**, XInput will only send neutral data in response to XInputGetState (all buttons up, axes centered, and triggers at 0). XInputSetState calls will be registered but not sent to the device. Sending any value other than **FALSE** will restore reading and writing functionality to normal.

Return value

None

Remarks

This function is meant to be called when an application gains or loses focus (such as via WM_ACTIVATEAPP). Using this function, you will not have to change the XInput query loop in your application as neutral data will always be reported if XInput is disabled.

In a controller that supports vibration effects:

- Passing **FALSE** will stop any vibration effects currently playing. In this state, calls to XInputSetState will be registered, but not passed to the device.
- Passing **TRUE** will pass the last vibration request (even if it is 0) sent to XInputSetState to the device.

Windows 10 or later: *Deprecated*, as game controller input is automatically enabled/disabled by the system based on the application window focus.

Platform Requirements

Windows 8 (XInput 1.4), DirectX SDK (XInput 1.3)

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib
DLL	Xinput1_4.dll

See also

XINPUT_GAMEPAD

XINPUT_STATE

XInput Functions

XInputGetState

XInputSetState

XInputGetAudioDeviceIds function (xinput.h)

Article02/22/2024

Retrieves the sound rendering and sound capture audio device IDs that are associated with the headset connected to the specified controller.

Syntax

```
C++
```

```
DWORD XInputGetAudioDeviceIds(
  [in]                DWORD  dwUserIndex,
  [out, optional]     LPWSTR pRenderDeviceId,
  [in, out, optional] UINT   *pRenderCount,
  [out, optional]     LPWSTR pCaptureDeviceId,
  [in, out, optional] UINT   *pCaptureCount
);
```

Parameters

`[in] dwUserIndex`

Index of the gamer associated with the device.

`[out, optional] pRenderDeviceId`

Windows Core Audio device ID string for render (speakers).

`[in, out, optional] pRenderCount`

Size, in wide-chars, of the render device ID string buffer.

`[out, optional] pCaptureDeviceId`

Windows Core Audio device ID string for capture (microphone).

`[in, out, optional] pCaptureCount`

Size, in wide-chars, of capture device ID string buffer.

Return value

If the function successfully retrieves the device IDs for render and capture, the return code is **ERROR_SUCCESS**.

If there is no headset connected to the controller, the function will also retrieve **ERROR_SUCCESS** with **NULL** as the values for *pRenderDeviceId* and *pCaptureDeviceId*.

If the controller port device is not physically connected, the function will return **ERROR_DEVICE_NOT_CONNECTED**.

If the function fails, it will return a valid Win32 error code.

Remarks

Callers must allocate the memory for the buffers passed to **XInputGetAudioDeviceIds**. The resulting strings can be of arbitrary length.

Platform Requirements

Windows 8 (XInput 1.4)

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib
DLL	Xinput1_4.dll

See also

Core Audio APIs

XInput Functions

XInputGetDSoundAudioDeviceGuids

XInputGetBatteryInformation function (xinput.h)

Article02/22/2024

Retrieves the battery type and charge status of a wireless controller.

Syntax

```
C++
```

```
DWORD XInputGetBatteryInformation(
  [in]  DWORD                       dwUserIndex,
  [in]  BYTE                        devType,
  [out] XINPUT_BATTERY_INFORMATION *pBatteryInformation
);
```

Parameters

`[in] dwUserIndex`

Index of the signed-in gamer associated with the device. Can be a value in the range 0–XUSER_MAX_COUNT − 1.

`[in] devType`

Specifies which device associated with this user index should be queried. Must be **BATTERY_DEVTYPE_GAMEPAD** or **BATTERY_DEVTYPE_HEADSET**.

`[out] pBatteryInformation`

Pointer to an XINPUT_BATTERY_INFORMATION structure that receives the battery information.

Return value

If the function succeeds, the return value is **ERROR_SUCCESS**.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib
DLL	Xinput1_4.dll

See also

XInput Functions

XInputGetCapabilities function (xinput.h)

Article02/22/2024

Retrieves the capabilities and features of a connected controller.

Syntax

```
C++
```

```
DWORD XInputGetCapabilities(
  [in]  DWORD               dwUserIndex,
  [in]  DWORD               dwFlags,
  [out] XINPUT_CAPABILITIES *pCapabilities
);
```

Parameters

`[in] dwUserIndex`

Index of the user's controller. Can be a value in the range 0–3. For information about how this value is determined and how the value maps to indicators on the controller, see Multiple Controllers.

`[in] dwFlags`

Input flags that identify the controller type. If this value is 0, then the capabilities of all controllers connected to the system are returned. Currently, only one value is supported:

Expand table

Value	Description
XINPUT_FLAG_GAMEPAD	Limit query to devices of controller type.

Any value of *dwflags* other than the above or 0 is illegal and will result in an error break when debugging.

`[out] pCapabilities`

Pointer to an XINPUT_CAPABILITIES structure that receives the controller capabilities.

Return value

If the function succeeds, the return value is **ERROR_SUCCESS**.

If the controller is not connected, the return value is **ERROR_DEVICE_NOT_CONNECTED**.

If the function fails, the return value is an error code defined in WinError.h. The function does not use *SetLastError* to set the calling thread's last-error code.

Remarks

> **Note** The legacy XINPUT 9.1.0 version (included in Windows Vista and later) always returned a fixed set of capabilities regardless of attached device.

Platform Requirements

Windows 8 (XInput 1.4), DirectX SDK (XInput 1.3), Windows Vista (XInput 9.1.0)

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib; Xinput9_1_0.lib
DLL	Xinput1_4.dll; Xinput9_1_0.dll

See also

XInput Functions

XInputGetState

XInputSetState

XInputGetDSoundAudioDeviceGuids function (xinput.h)

Article02/22/2024

Gets the sound rendering and sound capture device GUIDs that are associated with the headset connected to the specified controller.

Syntax

```
C++

DWORD XInputGetDSoundAudioDeviceGuids(
  DWORD dwUserIndex,
  GUID  *pDSoundRenderGuid,
  GUID  *pDSoundCaptureGuid
);
```

Parameters

`dwUserIndex`

[in] Index of the user's controller. It can be a value in the range 0–3. For information about how this value is determined and how the value maps to indicators on the controller, see Multiple Controllers.

`pDSoundRenderGuid`

[out] Pointer that receives the GUID of the headset sound rendering device.

`pDSoundCaptureGuid`

[out] Pointer that receives the GUID of the headset sound capture device.

Return value

If the function successfully retrieves the device IDs for render and capture, the return code is **ERROR_SUCCESS**.

If there is no headset connected to the controller, the function also retrieves **ERROR_SUCCESS** with **GUID_NULL** as the values for *pDSoundRenderGuid* and *pDSoundCaptureGuid*.

If the controller port device is not physically connected, the function returns **ERROR_DEVICE_NOT_CONNECTED**.

If the function fails, it returns a valid Win32 error code.

Remarks

Use of legacy DirectSound is not recommended, and DirectSound is not available for Windows Store apps.

> **Note** **XInputGetDSoundAudioDeviceGuids** is deprecated because it isn't supported by Windows 8 (XInput 1.4).

Platform Requirements

DirectX SDK (XInput 1.3), Windows Vista (XInput 9.1.0)

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib; Xinput9_1_0.lib
DLL	Xinput9_1_0.dll

See also

XInput Functions

XInputGetState

XInputGetKeystroke function (xinput.h)

Article02/22/2024

Retrieves a gamepad input event.

Syntax

```
C++
```

```
DWORD XInputGetKeystroke(
  DWORD             dwUserIndex,
  DWORD             dwReserved,
  PXINPUT_KEYSTROKE pKeystroke
);
```

Parameters

`dwUserIndex`

[in] Index of the signed-in gamer associated with the device. Can be a value in the range 0–XUSER_MAX_COUNT − 1, or XUSER_INDEX_ANY to fetch the next available input event from any user.

`dwReserved`

[in] Reserved

`pKeystroke`

[out] Pointer to an XINPUT_KEYSTROKE structure that receives an input event.

Return value

If the function succeeds, the return value is **ERROR_SUCCESS**.

If no new keys have been pressed, the return value is **ERROR_EMPTY**.

If the controller is not connected or the user has not activated it, the return value is **ERROR_DEVICE_NOT_CONNECTED**. See the Remarks section below.

If the function fails, the return value is an error code defined in Winerror.h. The function does not use SetLastError to set the calling thread's last-error code.

Remarks

Wireless controllers are not considered active upon system startup, and calls to any of the *XInput* functions before a wireless controller is made active return **ERROR_DEVICE_NOT_CONNECTED**. Game titles must examine the return code and be prepared to handle this condition. Wired controllers are automatically activated when they are inserted. Wireless controllers are activated when the user power on the controller.

Platform Requirements

Windows 8 (XInput 1.4), DirectX SDK (XInput 1.3)

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib
DLL	Xinput1_4.dll

See also

XINPUT_KEYSTROKE

XInput Functions

XInputGetState function (xinput.h)

Article02/22/2024

Retrieves the current state of the specified controller.

Syntax

```
C++

DWORD XInputGetState(
  [in]  DWORD        dwUserIndex,
  [out] XINPUT_STATE *pState
);
```

Parameters

`[in] dwUserIndex`

Index of the user's controller. Can be a value from 0 to 3. For information about how this value is determined and how the value maps to indicators on the controller, see Multiple Controllers.

`[out] pState`

Pointer to an XINPUT_STATE structure that receives the current state of the controller.

Return value

If the function succeeds, the return value is **ERROR_SUCCESS**.

If the controller is not connected, the return value is **ERROR_DEVICE_NOT_CONNECTED**.

If the function fails, the return value is an error code defined in Winerror.h. The function does not use **SetLastError** to set the calling thread's last-error code.

Remarks

When **XInputGetState** is used to retrieve controller data, the left and right triggers are each reported separately. For legacy reasons, when DirectInput retrieves controller data, the two triggers share the same axis. The legacy behavior is noticeable in the current Game Device Control Panel, which uses DirectInput for controller state.

Platform Requirements

Windows 8 (XInput 1.4), DirectX SDK (XInput 1.3), Windows Vista (XInput 9.1.0)

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib; Xinput9_1_0.lib
DLL	Xinput1_4.dll; Xinput9_1_0.dll; Xinputuap.dll

See also

XINPUT_GAMEPAD

XINPUT_STATE

XInput Functions

XInputSetState

XInputSetState function (xinput.h)

Article02/22/2024

Sends data to a connected controller. This function is used to activate the vibration function of a controller.

Syntax

```
C++

DWORD XInputSetState(
  [in]      DWORD             dwUserIndex,
  [in, out] XINPUT_VIBRATION *pVibration
);
```

Parameters

`[in] dwUserIndex`

Index of the user's controller. Can be a value from 0 to 3. For information about how this value is determined and how the value maps to indicators on the controller, see Multiple Controllers.

`[in, out] pVibration`

Pointer to an XINPUT_VIBRATION structure containing the vibration information to send to the controller.

Return value

If the function succeeds, the return value is **ERROR_SUCCESS**.

If the controller is not connected, the return value is **ERROR_DEVICE_NOT_CONNECTED**.

If the function fails, the return value is an error code defined in WinError.h. The function does not use *SetLastError* to set the calling thread's last-error code.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	xinput.h
Library	Xinput.lib; Xinput9_1_0.lib
DLL	Xinput1_4.dll; Xinput9_1_0.dll

See also

XINPUT_VIBRATION

XInput Functions

XInputGetState

Structures (XInput Game Controller APIs)

Available XInput Structures

In this section

Expand table

Topic	Description
XINPUT_BATTERY_INFORMATION	Contains information on battery type and charge state.
XINPUT_CAPABILITIES	Describes the capabilities of a connected controller. The XInputGetCapabilities function returns XINPUT_CAPABILITIES.
XINPUT_GAMEPAD	Describes the current state of a controller.
XINPUT_KEYSTROKE	Specifies keystroke data returned by XInputGetKeystroke.
XINPUT_STATE	Represents the state of a controller.
XINPUT_VIBRATION	Specifies motor speed levels for the vibration function of a controller.

Last updated on 07/14/2025

XINPUT_BATTERY_INFORMATION structure (xinput.h)

Article02/22/2024

Contains information on battery type and charge state.

Syntax

C++

```
typedef struct _XINPUT_BATTERY_INFORMATION {
  BYTE BatteryType;
  BYTE BatteryLevel;
} XINPUT_BATTERY_INFORMATION, *PXINPUT_BATTERY_INFORMATION;
```

Members

`BatteryType`

The type of battery. *BatteryType* will be one of the following values.

Expand table

Value	Description
BATTERY_TYPE_DISCONNECTED	The device is not connected.
BATTERY_TYPE_WIRED	The device is a wired device and does not have a battery.
BATTERY_TYPE_ALKALINE	The device has an alkaline battery.
BATTERY_TYPE_NIMH	The device has a nickel metal hydride battery.
BATTERY_TYPE_UNKNOWN	The device has an unknown battery type.

`BatteryLevel`

The charge state of the battery. This value is only valid for wireless devices with a known battery type. *BatteryLevel* will be one of the following values.

Expand table

Value
BATTERY_LEVEL_EMPTY
BATTERY_LEVEL_LOW
BATTERY_LEVEL_MEDIUM
BATTERY_LEVEL_FULL

Requirements

Expand table

Requirement	Value
Header	xinput.h

See also

XINPUT_GAMEPAD

XInput Structures

XInputGetCapabilities

XInputSetState

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

XINPUT_CAPABILITIES structure (xinput.h)

Article10/05/2021

Describes the capabilities of a connected controller. The XInputGetCapabilities function returns **XINPUT_CAPABILITIES**.

Syntax

C++

```
typedef struct _XINPUT_CAPABILITIES {
  BYTE             Type;
  BYTE             SubType;
  WORD             Flags;
  XINPUT_GAMEPAD   Gamepad;
  XINPUT_VIBRATION Vibration;
} XINPUT_CAPABILITIES, *PXINPUT_CAPABILITIES;
```

Members

`Type`

Controller type. It must be one of the following values.

Value	Description
XINPUT_DEVTYPE_GAMEPAD	The device is a game controller.

`SubType`

Subtype of the game controller. See XINPUT and Controller Subtypes for a list of allowed subtypes.

> **Note** For restrictions on the use of this subtype value, see Remarks. More subtypes may be added in the future.

`Flags`

Features of the controller.

Value	Description
XINPUT_CAPS_VOICE_SUPPORTED	Device has an integrated voice device.
XINPUT_CAPS_FFB_SUPPORTED	Device supports force feedback functionality. Note that these force-feedback features beyond rumble are not currently supported through XINPUT on Windows.
XINPUT_CAPS_WIRELESS	Device is wireless.
XINPUT_CAPS_PMD_SUPPORTED	Device supports plug-in modules. Note that plug-in modules like the text input device (TID) are not supported currently through XINPUT on Windows.
XINPUT_CAPS_NO_NAVIGATION	Device lacks menu navigation buttons (START, BACK, DPAD).

`Gamepad`

XINPUT_GAMEPAD structure that describes available controller features and control resolutions.

`Vibration`

XINPUT_VIBRATION structure that describes available vibration functionality and resolutions.

Remarks

XInputGetCapabilities returns **XINPUT_CAPABILITIES** to indicate the characteristics and available functionality of a specified controller.

XInputGetCapabilities sets the structure members to indicate which inputs the device supports. For binary state controls, such as digital buttons, the corresponding bit reflects whether or not the control is supported by the device. For proportional controls, such as thumbsticks, the value indicates the resolution for that control. Some number of the least significant bits may not be set, indicating that the control does not provide resolution to that level.

The *SubType* member indicates the specific subtype of controller present. Games may detect the controller subtype and tune their handling of controller input or output based on subtypes that are well suited to their game genre. For example, a car racing game might check for the presence of a wheel controller to provide finer control of the car being driven. However, titles must not disable or ignore a device based on its

subtype. Subtypes not recognized by the game or for which the game is not specifically tuned should be treated as a standard Xbox 360 Controller (XINPUT_DEVSUBTYPE_GAMEPAD).

Older XUSB Windows drivers report incomplete capabilities information, particularly for wireless devices. The latest XUSB Windows driver provides full support for wired and wireless devices, and more complete and accurate capabilities flags.

Requirements

Header	xinput.h

See also

XINPUT_GAMEPAD

XINPUT_VIBRATION

XInput Structures

XInputGetCapabilities

Feedback

XINPUT_GAMEPAD structure (xinput.h)

Article10/27/2023

Describes the current state of the controller.

Syntax

C++

```
typedef struct _XINPUT_GAMEPAD {
  WORD  wButtons;
  BYTE  bLeftTrigger;
  BYTE  bRightTrigger;
  SHORT sThumbLX;
  SHORT sThumbLY;
  SHORT sThumbRX;
  SHORT sThumbRY;
} XINPUT_GAMEPAD, *PXINPUT_GAMEPAD;
```

Members

`wButtons`

Bitmask of the device digital buttons, as follows. A set bit indicates that the corresponding button is pressed.

Device button	Bitmask
XINPUT_GAMEPAD_DPAD_UP	0x0001
XINPUT_GAMEPAD_DPAD_DOWN	0x0002
XINPUT_GAMEPAD_DPAD_LEFT	0x0004
XINPUT_GAMEPAD_DPAD_RIGHT	0x0008
XINPUT_GAMEPAD_START	0x0010
XINPUT_GAMEPAD_BACK	0x0020
XINPUT_GAMEPAD_LEFT_THUMB	0x0040
XINPUT_GAMEPAD_RIGHT_THUMB	0x0080
XINPUT_GAMEPAD_LEFT_SHOULDER	0x0100

XINPUT_GAMEPAD_RIGHT_SHOULDER	0x0200
XINPUT_GAMEPAD_A	0x1000
XINPUT_GAMEPAD_B	0x2000
XINPUT_GAMEPAD_X	0x4000
XINPUT_GAMEPAD_Y	0x8000

Bits that are set but not defined above are reserved, and their state is undefined.

`bLeftTrigger`

The current value of the left trigger analog control. The value is between 0 and 255.

`bRightTrigger`

The current value of the right trigger analog control. The value is between 0 and 255.

`sThumbLX`

Left thumbstick x-axis value. Each of the thumbstick axis members is a signed value between -32768 and 32767 describing the position of the thumbstick. A value of 0 is centered. Negative values signify down or to the left. Positive values signify up or to the right. The constants XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE or XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE can be used as a positive and negative value to filter a thumbstick input.

`sThumbLY`

Left thumbstick y-axis value. The value is between -32768 and 32767.

`sThumbRX`

Right thumbstick x-axis value. The value is between -32768 and 32767.

`sThumbRY`

Right thumbstick y-axis value. The value is between -32768 and 32767.

Remarks

This structure is used by the XINPUT_STATE structure when polling for changes in the state of the controller.

The specific mapping of button to game function varies depending on the game type.

The constant XINPUT_GAMEPAD_TRIGGER_THRESHOLD may be used as the value which *bLeftTrigger* and *bRightTrigger* must be greater than to register as pressed. This is optional, but often desirable. Controller buttons do not manifest crosstalk.

Requirements

Header	xinput.h

See also

XINPUT_STATE

XInput Structures

XInputGetState

Feedback

XINPUT_KEYSTROKE structure (xinput.h)

Article02/22/2024

Specifies keystroke data returned by XInputGetKeystroke.

Syntax

C++

```
typedef struct _XINPUT_KEYSTROKE {
  WORD  VirtualKey;
  WCHAR Unicode;
  WORD  Flags;
  BYTE  UserIndex;
  BYTE  HidCode;
} XINPUT_KEYSTROKE, *PXINPUT_KEYSTROKE;
```

Members

`VirtualKey`

Virtual-key code of the key, button, or stick movement. See XInput.h for a list of valid virtual-key (VK_xxx) codes. Also, see Remarks.

`Unicode`

This member is unused and the value is zero.

`Flags`

Flags that indicate the keyboard state at the time of the input event. This member can be any combination of the following flags:

Expand table

Value	Description
XINPUT_KEYSTROKE_KEYDOWN	The key was pressed.
XINPUT_KEYSTROKE_KEYUP	The key was released.
XINPUT_KEYSTROKE_REPEAT	A repeat of a held key.

`UserIndex`

Index of the signed-in gamer associated with the device. Can be a value in the range 0–3.

`HidCode`

HID code corresponding to the input. If there is no corresponding HID code, this value is zero.

Remarks

Future devices may return HID codes and virtual key values that are not supported on current devices, and are currently undefined. Applications should ignore these unexpected values.

A *virtual-key* code is a byte value that represents a particular physical key on the keyboard, not the character or characters (possibly none) that the key can be mapped to based on keyboard state. The keyboard state at the time a virtual key is pressed modifies the character reported. For example, VK_4 might represent a "4" or a "$", depending on the state of the SHIFT key.

A reported keyboard event includes the virtual key that caused the event, whether the key was pressed or released (or is repeating), and the state of the keyboard at the time of the event. The keyboard state includes information about whether any CTRL, ALT, or SHIFT keys are down.

If the keyboard event represents an Unicode character (for example, pressing the "A" key), the *Unicode* member will contain that character. Otherwise, *Unicode* will contain the value zero.

The valid virtual-key (VK_xxx) codes are defined in XInput.h. In addition to codes that indicate key presses, the following codes indicate controller input.

Expand table

Value	Description
VK_PAD_A	**A** button
VK_PAD_B	**B** button
VK_PAD_X	**X** button
VK_PAD_Y	**Y** button
VK_PAD_RSHOULDER	Right shoulder button

VK_PAD_LSHOULDER	Left shoulder button
VK_PAD_LTRIGGER	Left trigger
VK_PAD_RTRIGGER	Right trigger
VK_PAD_DPAD_UP	Directional pad up
VK_PAD_DPAD_DOWN	Directional pad down
VK_PAD_DPAD_LEFT	Directional pad left
VK_PAD_DPAD_RIGHT	Directional pad right
VK_PAD_START	**START** button
VK_PAD_BACK	**BACK** button
VK_PAD_LTHUMB_PRESS	Left thumbstick click
VK_PAD_RTHUMB_PRESS	Right thumbstick click
VK_PAD_LTHUMB_UP	Left thumbstick up
VK_PAD_LTHUMB_DOWN	Left thumbstick down
VK_PAD_LTHUMB_RIGHT	Left thumbstick right
VK_PAD_LTHUMB_LEFT	Left thumbstick left
VK_PAD_LTHUMB_UPLEFT	Left thumbstick up and left
VK_PAD_LTHUMB_UPRIGHT	Left thumbstick up and right
VK_PAD_LTHUMB_DOWNRIGHT	Left thumbstick down and right
VK_PAD_LTHUMB_DOWNLEFT	Left thumbstick down and left
VK_PAD_RTHUMB_UP	Right thumbstick up
VK_PAD_RTHUMB_DOWN	Right thumbstick down
VK_PAD_RTHUMB_RIGHT	Right thumbstick right
VK_PAD_RTHUMB_LEFT	Right thumbstick left
VK_PAD_RTHUMB_UPLEFT	Right thumbstick up and left
VK_PAD_RTHUMB_UPRIGHT	Right thumbstick up and right
VK_PAD_RTHUMB_DOWNRIGHT	Right thumbstick down and right
VK_PAD_RTHUMB_DOWNLEFT	Right thumbstick down and left

Requirements

Expand table

Requirement	Value
Header	xinput.h

See also

XInputGetKeystroke

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

XINPUT_STATE structure (xinput.h)

Article02/22/2024

Represents the state of a controller.

Syntax

```
C++
```

```
typedef struct _XINPUT_STATE {
  DWORD          dwPacketNumber;
  XINPUT_GAMEPAD Gamepad;
} XINPUT_STATE, *PXINPUT_STATE;
```

Members

`dwPacketNumber`

State packet number. The packet number indicates whether there have been any changes in the state of the controller. If the *dwPacketNumber* member is the same in sequentially returned **XINPUT_STATE** structures, the controller state has not changed.

`Gamepad`

XINPUT_GAMEPAD structure containing the current state of a controller.

Remarks

The *dwPacketNumber* member is incremented only if the status of the controller has changed since the controller was last polled.

Requirements

Expand table

Requirement	Value
Header	xinput.h

See also

XINPUT_GAMEPAD

XInput Structures

XInputGetState

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

XINPUT_VIBRATION structure (xinput.h)

Article02/22/2024

Specifies motor speed levels for the vibration function of a controller.

Syntax

```
C++
```

```
typedef struct _XINPUT_VIBRATION {
  WORD wLeftMotorSpeed;
  WORD wRightMotorSpeed;
} XINPUT_VIBRATION, *PXINPUT_VIBRATION;
```

Members

`wLeftMotorSpeed`

Speed of the left motor. Valid values are in the range 0 to 65,535. Zero signifies no motor use; 65,535 signifies 100 percent motor use.

`wRightMotorSpeed`

Speed of the right motor. Valid values are in the range 0 to 65,535. Zero signifies no motor use; 65,535 signifies 100 percent motor use.

Remarks

The left motor is the low-frequency rumble motor. The right motor is the high-frequency rumble motor. The two motors are not the same, and they create different vibration effects.

Requirements

Expand table

Requirement	Value
Header	xinput.h

See also

XINPUT_GAMEPAD

XInput Structures

XInputGetCapabilities

XInputSetState

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A