Microsoft Quantum documentation

Learn about quantum computing with Microsoft Quantum and the Azure Quantum service. Use Python and Q#, a language for quantum programming, to create and submit quantum programs in the Azure portal, or set up your own local development environment with the Microsoft Quantum Development Kit (QDK) to write quantum programs.

About Azure Quantum

OVERVIEW

What is Azure Quantum?

Azure Quantum pricing plans

DEPLOY

Create a quantum workspace

REFERENCE

List of quantum hardware providers

TRAINING

Training path - Get started with Azure Quantum

Get started with Q# and the Microsoft Quantum Development Kit

DOWNLOAD

Set up the Microsoft Quantum Development Kit

OVERVIEW

What is Q#?

QUICKSTART

Create your first Q# program

TUTORIAL

Create a quantum random number generator

Explore quantum entanglement

Advanced development with the Microsoft Quantum Development Kit

GET STARTED

Different ways to run Q# programs

OVERVIEW

Introduction to the Microsoft Quantum resource estimator

Introduction to hybrid quantum computing

Learn about quantum computing

CONCEPT

What is quantum computing?

The qubit

Vectors and matrices

Dirac notation

Quantum circuits

Quantum error correction

What is Azure Quantum?

Azure Quantum is Microsoft Azure's cloud quantum computing service. Azure Quantum provides an open, flexible, and future-proofed path to quantum computing that adapts to your way of working.

Azure Quantum offers a range of quantum computing solutions, including quantum hardware from industry-leading providers, quantum software, and quantum services. With Azure Quantum, you can run quantum programs on real quantum hardware, simulate quantum algorithms, and estimate the resources needed to run your quantum programs on future scaled quantum machines.

To learn more about how you can use quantum computing and quantum algorithms, see What is Quantum Computing?

How to get started with Azure Quantum

To use Azure Quantum, you need to have an Azure account and an Azure Quantum workspace. To develop quantum programs and submit jobs to run your programs on Azure Quantum, use the Microsoft Quantum Development Kit (QDK).

To get an Azure account, register for free and sign up for a pay-as-you-go subscription. If you're a student, then you can take advantage of a free Azure account for students.

> **Note**
>
> You don't need to have an Azure account to use the QDK.

The Microsoft Quantum website

The Microsoft Quantum website is a central resource where you can explore quantum computing at Microsoft. You can get the latest news and information from Microsoft Quantum. You can learn from experts and enthusiasts through blogs, articles, and videos.

You don't need an Azure account to access the resources and information on the Microsoft Quantum website.

The Microsoft Quantum Development Kit

The Microsoft Quantum Development Kit (QDK) is a software development kit designed specifically for quantum development. With the QDK, you can write programs in different quantum programming languages, debug your code, visualize quantum circuits and results, and submit jobs to quantum hardware providers on Azure Quantum. The QDK supports Microsoft's Q# programming language, along with other languages like Qiskit, Cirq, and OpenQASM.

The QDK is free and open source. To get started, install the QDK extension in Visual Studio Code (VS Code) or install the QDK Python library. For more information, see Set up the Microsoft Quantum Development Kit.

> **Note**
>
> To run programs on Azure Quantum, you need to have an Azure Quantum workspace. For more information, see **Create an Azure Quantum workspace**.

The Azure portal

If you have an Azure account, use the Azure portal to create an Azure Quantum workspace. An Azure Quantum workspace is a collection of assets that are associated with running quantum programs. For more information, see Create an Azure Quantum workspace.

With the Azure portal, you can submit your quantum programs to real quantum hardware, manage your Azure Quantum workspace, view information about your quantum jobs, and monitor your quantum programs.

What is Q#?

Q# is an open-source quantum programming language created by Microsoft to develop and run your quantum programs.

You can think of a quantum program as a set of classical subroutines that interact with a quantum system to perform a computation. A Q# program doesn't directly model the quantum state, but rather describes how a classical control computer interacts with qubits. Q# is hardware-agnostic, so you don't need to consider actual qubit technologies when you write Q# programs. Your Q# code can run on any quantum hardware technology.

Q# is a standalone language that offers a high level of abstraction. There's no notion of a quantum state or circuit. Instead, Q# implements programs in terms of statements and

expressions, like classical programming languages such as Python. You can seamlessly integrate classical and quantum computing structures in your Q# code.

For more information, see Introduction to Q#. To start writing Q# code, see Create your first Q# program.

What can I do with Azure Quantum?

Azure Quantum offers a wide range of services and tools to help you develop quantum solutions.

For the latest information about Microsoft's quantum computing research, see the Microsoft Research Quantum Computing page.

Hybrid quantum computing

Hybrid quantum computing refers to the processes and architecture of a classical computer and a quantum computer working together to solve a problem. With the latest generation of hybrid quantum computer architecture available in Azure Quantum, you can get started with a classical-quantum hybrid approach to programming.

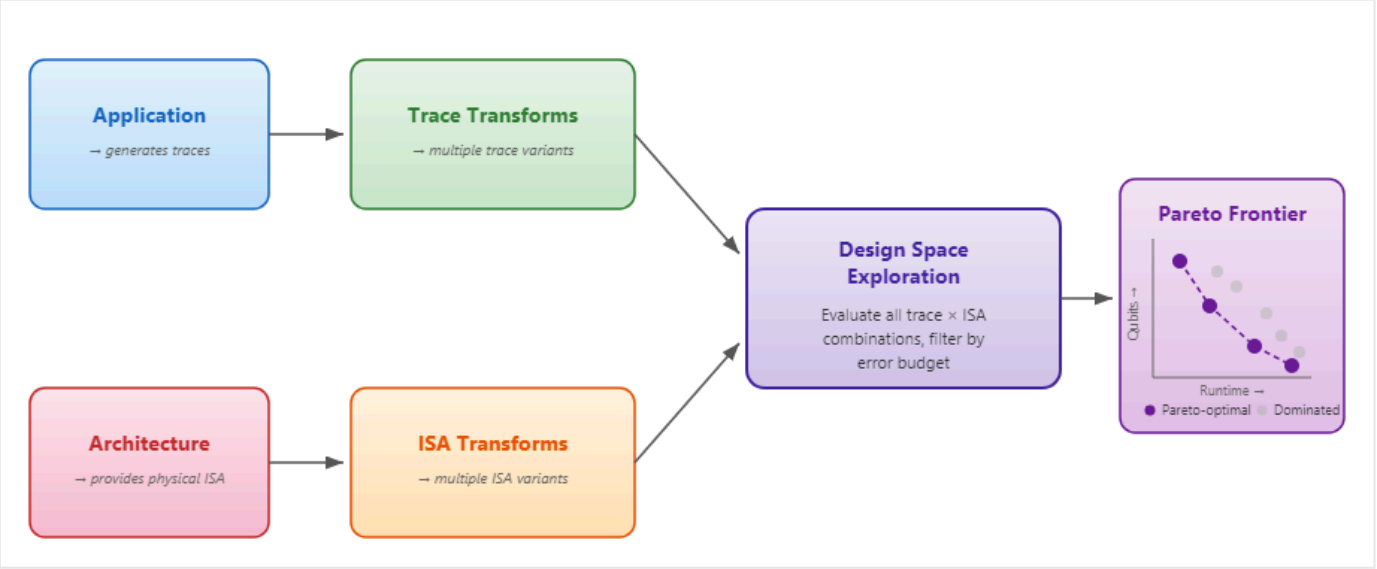
For more information, see What is hybrid quantum computing?

Resource estimation in quantum computing

In quantum computing, resource estimation is a technique to understand the resources that are required to run an algorithm on a quantum computer. When you understand the resource requirements to run your programs on different types of quantum hardware, you can prepare and refine your quantum solutions to run on future scaled quantum machines. For example, resource estimation can help you determine the feasibility of breaking a particular encryption algorithm on a specific type of quantum computer.

The Microsoft Quantum resource estimator allows you to assess architectural decisions, compare qubit technologies, and determine the resources that you need to run a specific quantum algorithm. You can choose from pre-defined fault-tolerant protocols, or design your own models. The resource estimator computes post-layout physical resource estimates from a set of inputs such as the error budget and models of the quantum application, the hardware architecture, and the quantum error correction (QEC) code.

To get started, see How to install and use the Microsoft Quantum resource estimator.



Quantum chemistry simulations with Azure Quantum and the QDK

You can think of quantum mechanics as the underlying operating system of our universe that describes how the fundamental building blocks of nature behave. Chemical reactions, cellular processes, and material properties are all quantum mechanical in nature, and often involve interactions between a huge number of quantum particles. Quantum computers have promise to simulate quantum mechanical systems, such as molecules, because qubits can be used to represent the natural quantum states in these systems. Examples of quantum systems that qubits can model include photosynthesis, superconductivity, and complex molecular formations.

The QDK and Azure Quantum are purpose-built to accelerate scientific discovery. Reinvent your research and development productivity with simulation workflows that are optimized for scaling on Azure high-performance computing (HPC) clusters, AI-accelerated computing, integration with quantum tools and quantum hardware, and future access to Microsoft's quantum supercomputer.

For more information, see Unlocking the power of Azure for Molecular Dynamics.

Quantum speedup

Quantum computers do exceptionally well with problems that require calculations of a large number of possible combinations. These types of problems can be found in many areas, such as quantum simulation, cryptography, quantum machine learning, and search problems.

One of the goals of quantum computing research is to study the kinds of problems that a quantum computer can solve faster than a classical computer, and how much faster. One well-

known example is Grover's algorithm, which yields a polynomial speedup over the classical algorithms.

Grover's algorithm speeds up the solution to unstructured data searches by running the search in fewer steps than any classical algorithm could. In general, problems that allow you to check whether a given value is a valid solution (a "yes or no problem") can be formulated in terms of the search problem.

For an implementation of Grover's algorithm, see Tutorial: Implement Grover's search algorithm in Q#.

Quantum providers available on Azure Quantum

Azure Quantum offers some of the most compelling and diverse quantum resources available today from industry leaders. Azure Quantum currently partners with the following providers to empower you to run your quantum programs on real hardware or on hardware simulators.

Choose the provider that best suits the characteristics of your problem and your needs.

- IonQ: Dynamically reconfigurable trapped-ion quantum computers for up to 36 fully connected qubits, that lets you run a two-qubit gate between any pair.
- Pasqal: Neutral atom-based quantum processors that operate at room temperature, with long coherence times and impressive qubit connectivity.
- Quantinuum: Trapped-ion systems with high-fidelity, fully connected qubits, low error rates, qubit reuse, and the ability to perform mid-circuit measurements.
- Rigetti: Powered by superconducting qubit-based quantum processors, these systems offer fast gate times, low-latency conditional logic, and fast program execution times.

For more information on the specifications of each provider, see the full Quantum computing target list.

For information about job cost, see Pricing in Azure Quantum and FAQ: Understanding Job Costs and Billing in Azure Quantum.

Related content

To start using Azure Quantum, explore the following links:

- Create an Azure Quantum workspace
- Get started with Q# and Visual Studio Code
- Install the Microsoft Quantum Development Kit

- Run a Qiskit circuit in Azure Quantum

Last updated on 06/17/2026

What is quantum computing?

09/29/2025

Quantum computing holds the promise of solving some of our planet's biggest challenges - in the areas of environment, agriculture, health, energy, climate, materials science, and more. For some of these problems, classical computing is increasingly challenged as the size of the system grows. When designed to scale, quantum systems will likely have capabilities that exceed those of today's most powerful supercomputers.

This article explains the principles of quantum computing, how it compares to classical computing, and how it uses the principles of quantum mechanics.

History of quantum computing

Quantum systems, such as atoms and molecules, can be difficult or impossible to simulate on a classical computer. In the 1980s, Richard Feynman and Yuri Manin suggested that hardware based on quantum phenomena might be more efficient for the simulation of quantum systems than conventional computers.

There are several reasons why quantum systems are difficult to simulate on regular computers. One main reason is that matter, at a quantum level, is described as a combination of multiple configurations (known as states) at the same time.

Quantum states grow exponentially

Consider a system of particles, and 40 possible locations where those particles can exist. The system could be in any of 2^{40} unique states because each location can either have or not have a particle. If these are classical particles, then the system is always in only one of the 2^{40} states, so a classical computer needs only 40 bits to describe the state of the system. But if these are quantum particles, then the system exists in a combination of all 2^{40} states. A classical computer needs to store 2^{40} numbers to describe the quantum system, which requires over 130 GB of memory. However, a quantum computer needs only 40 quantum bits to describe this quantum system.

If we add another location to the system so that the electrons can exist in 41 locations, then the number of unique configurations of the system doubles to 2^{41}. It would take more than 260 GB of memory to store that quantum state on a classical computer. We can't play this game of increasing the number of locations forever. To store a quantum state on a conventional computer, you quickly exceed the memory capacities of the world's most powerful machines. At a few hundred electrons, the memory required to store the system

exceeds the number of particles in the universe. There's no hope with our conventional computers to completely simulate quantum dynamics for larger systems!

Turning difficulty into opportunity

The observation of this exponential growth posed a powerful question: is it possible to turn this difficulty into an opportunity? If quantum systems are difficult to simulate on regular computers, what would happen if we build a machine that uses quantum effects for its fundamental operations? Could we simulate quantum systems with a machine that exploits exactly the same laws of physics? And could we use that machine to investigate other important problems outside of quantum mechanics? These are the kinds of questions that gave rise to the fields of quantum information and quantum computing.

In 1985, David Deutsch showed that a quantum computer could efficiently simulate the behavior of any physical system. This discovery was the first indication that quantum computers could be used to solve problems that are too difficult to solve on classical computers.

In 1994, Peter Shor discovered a quantum algorithm to find the prime factors of large integers. Shor's algorithm runs exponentially faster than the best known classical algorithm for this factoring problem. Such a fast algorithm could potentially break many of our modern public key cryptosystems that we use to secure transactions in e-commerce, such as Rivest–Shamir–Adleman (RSA) and Elliptic Curve Cryptography. This discovery sparked a huge interest in quantum computing and led to the development of quantum algorithms for many other problems.

Since that time, fast and efficient quantum computer algorithms were developed for other problems that are difficult to solve on classical computers. For example, we now have quantum algorithms to search an unordered database, to solve systems of linear equations, to perform machine learning, and to simulate physical systems in chemistry, physics, and materials science.

What is a qubit?

Just as bits are the fundamental object of information in classical computing, qubits (quantum bits) are the fundamental object of information in quantum computing.

Qubits play a similar role in quantum computing as bits play in classical computing, but qubits behave differently than bits. Classical bits are binary and, at any given time, can be in only one of two states, 0 or 1. But qubits can be in a superposition of both the 0 and 1 states at the same time. In fact, there are infinite possible superpositions of 0 and 1, and each of them is a valid qubit state.

In quantum computing, information is encoded in superpositions of the states 0 and 1. For example, 8 regular bits can encode up to 256 unique values, but these 8 bits can only represent one of the 256 values at a time. With 8 qubits, we could encode all 256 values at the same time, because the qubits can be in a superposition of all 256 possible states.

For more information, see The qubit in quantum computing.

What are the requirements to build a quantum computer?

A quantum computer uses quantum systems and the properties of quantum mechanics to solve computational problems. The systems in a quantum computer consist of the qubits, the interactions between qubits, and operations on the qubits to store and compute information. We can use quantum computers to program effects like quantum entanglement and quantum interference to accurately solve certain problems faster than on classical computers.

To build a quantum computer, we need to consider how to create and store the qubits. We also need to think about how to manipulate the qubits and how to measure the results of our computations.

Popular qubit technologies include trapped-ion qubits, superconducting qubits, and topological qubits. For some methods of qubit storage, the unit that houses the qubits must be kept at a temperature near absolute zero to maximize their coherence and reduce interference. Other types of qubit housing use a vacuum chamber to help minimize vibrations and stabilize the qubits. Signals can be sent to the qubits through various methods, such as microwaves, lasers, or voltages.

The five criteria for a quantum computer

A good quantum computer should have these five features:

1. **Scalable:** It can have many qubits.
2. **Initializable:** It can set the qubits to a specific state (usually the 0 state).
3. **Resilient:** It can keep the qubits in superposition state for a long time.
4. **Universal:** A quantum computer doesn't need to perform every possible operation, only a set of operations called *universal set*. A set of universal quantum operations is such that any other operation can be decomposed into a sequence of them.
5. **Reliable:** It can measure the qubits accurately.

These five criteria are often known as the Di Vincenzo criteria for quantum computation.

Building devices that meet these five criteria is one of the most demanding engineering challenges ever faced by humankind. Azure Quantum offers a variety of quantum computing solutions with different qubit technologies. For more information, see the full list of Azure Quantum providers.

Understand quantum phenomena

Quantum phenomena are the fundamental principles that differentiate quantum computing from classical computing. Understanding these phenomena is crucial for grasping how quantum computers operate and why they hold such potential. The two most important quantum phenomena are superposition and entanglement.

Superposition

Imagine that you are exercising in your living room. You turn all the way to your left and then all the way to your right. Now turn to your left and your right at the same time. You can't do it (not without splitting yourself in two, at least). Obviously, you can't be in both of those states at once – you can't be facing left and facing right at the same time.

However, if you are a quantum particle, then you can have a certain probability of *facing left* AND a certain probability of *facing right* due to a phenomenon known as *superposition* (also known as *coherence*).

Only quantum systems like ions, electrons, or superconducting circuits can exist in the superposition states that enable the power of quantum computing. For example, electrons are quantum particles that have their own "facing left or facing right" property called spin. The two spin states are called spin up and spin down, and the quantum state of an electron is a superposition of the spin up and spin down states.

If you want to learn more and practice with superposition, see Training module: Explore superposition with Q#.

Entanglement

Entanglement is a quantum correlation between two or more quantum systems. When two qubits are entangled, they're correlated and sharing the information of their states such that the quantum state of individual qubits can't be described independently. With quantum entanglement you can only know the quantum state of the global system, not the individual states.

Entangled quantum systems maintain this correlation even when separated over large distances. This means that whatever operation or process you apply to one subsystem

correlates to the other subsystem as well. Thus, measuring the state of one qubit provides information about the state of the other qubit – this particular property is very helpful in quantum computing.

If you want to learn more, see Tutorial: Explore quantum entanglement with Q# and, for a practical implementation check out Training module: Teleport a qubit using entanglement.

Related content

- Vectors and matrices in quantum computing
- The qubit
- Quantum circuit diagram conventions
- Entanglement in quantum computing
- Introduction to the quantum programming language Q#

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Microsoft's quantum machine

Microsoft is on the path to deliver a quantum machine as part of the Azure Quantum service. You can read more about this journey on this blog. This is part of a comprehensive plan to empower innovators with quantum solutions for breakthrough impact.

In 2022, Microsoft had a major scientific breakthrough that cleared a significant hurdle toward building a scaled quantum machine with topological qubits. While engineering challenges remain, this new discovery demonstrates a fundamental building block for its approach to scaled quantum computing. For more information about this scientific discovery, read the blog, watch a video, or read the preprint of the paper.

View a Jupyter Notebook of the results

The Azure Quantum team has created a set of Jupyter Notebooks on the azure-quantum-tgp GitHub site, which display all the analysis and plots that appear in the paper. From the GitHub page, you can also view the executed notebook results with nbviewer.

Next Steps

- What is Azure Quantum?
- Introduction to Q#
- Build quantum solutions with the Microsoft Quantum Development Kit

---

Last updated on 08/29/2026

Create an Azure Quantum workspace

Learn how to create an Azure Quantum workspace in the Azure portal. An Azure Quantum workspace resource, or workspace for short, is a collection of assets associated with running quantum applications.

You need a workspace to submit quantum programs to quantum hardware.

> **Tip**
>
> You can also create an Azure Quantum workspace using the Azure command-line interface (CLI). For more information, see **Manage quantum workspaces with the Azure CLI**.

Prerequisites

You need to have an Azure account with an active subscription to create an Azure Quantum workspace. If you don't have one, you can choose from one of the following subscriptions, or for a full list see Microsoft Azure Offer Details.

Expand table

Azure subscription	Description
Pay-as-you-go (recommended)	You pay for the services you use, and you can cancel anytime.
Enterprise Agreement	If your organization has an Enterprise Agreement (EA) purchasing contract with Microsoft, your organization's Account Owners can create Enterprise Dev/Test subscriptions for active Visual Studio subscribers under the EA.

If you have questions or run into issues when you use Azure Quantum, then contact AzureQuantumInfo@microsoft.com.

Create the workspace

To create an Azure Quantum workspace, follow these steps.

1. Sign in to the Azure portal, using the credentials for your Azure subscription.

2. From the Azure portal homepage, select **Create a resource** and then in **Search the Marketplace** enter **Azure Quantum**. On the results page, you should see a tile for the **Azure Quantum** service.

3. Select **Azure Quantum**, and then select **Create**. The workspace creation form opens.

4. Select a subscription to associate with the new workspace.

5. Select **Quick create** or **Advanced create**.

Quick create

This option is the easiest path to creating a workspace. It automatically creates the necessary resource group and storage account, adds the IonQ, Quantinuum, Rigetti, and Microsoft Quantum Computing providers. Your workspace can still be customized after creation, if needed.

> ⓘ **Note**
>
> To use **Quick create**, you must be an **Owner** of the subscription you selected in the previous step. To see a list of your subscriptions and access, see Check your role assignments.

1. Enter a name for the workspace.
2. Select the region for the workspace.
3. Select **Create**.

> ⓘ **Note**
>
> If you encounter issues during deployment, see Azure Quantum common issues: Creating an Azure Quantum workspace.

Deployment of your workspace may take a few minutes. The portal updates the status and deployment details. When the **Your deployment is complete** message appears, view the **Deployment details** or select **Go to resource**.

> ⓘ **Note**

If you run into any issues, see Azure Quantum common issues: Creating an Azure Quantum workspace.

Assign data access roles for users in the workspace

> ⚠ **Warning**
>
> Starting January 11, 2027, Azure Quantum requires users in a Quantum workspace to have the **Quantum Workspace Data Reader** or **Quantum Workspace Data Contributor** role to view job results. The **Quantum Workspace Data Contributor** role will be required to submit jobs. To avoid interruptions to your Azure Quantum workflows, assign these roles to the appropriate users before the deadline. For more information, see Assign data access roles for users in the workspace.

Azure Quantum workspaces have the following roles that you can assign to users in the workspace.

Expand table

Role	Description
Quantum Workspace Data Contributor	Create, read, and modify jobs and other workspace data.
Quantum Workspace Data Reader	Read jobs and other workspace data.

To submit a job to Azure Quantum, a user needs to have the **Quantum Workspace Data Contributor** role. To assign these roles, complete the following steps as a user with the **Owner** role.

1. Sign in to the Azure portal.
2. Go to your Quantum workspace.
3. In the navigation pane, select **Access control (IAM)**.
4. Select **Add**, and then select **Add role assignment**.
5. Search for the user that you want to assign a role to.
6. Select the **Quantum Workspace Data Contributor** or **Quantum WOrkspace Data Reader** role, and then select **Next**.
7. Select the user to assign the role, and then select **Review + assign**.

Last updated on 11/06/2026

Connect to your Azure Quantum workspace with the QDK or Azure CLI

If you have an Azure Quantum workspace, you can use the `qdk.azure` Python module the connect to the workspace and submit your code to run on Azure Quantum. The `qdk.azure` module provides a Workspace class that represents an Azure Quantum workspace.

> ⚠ **Warning**
>
> Starting January 11, 2027, Azure Quantum requires users in a Quantum workspace to have the **Quantum Workspace Data Reader** or **Quantum Workspace Data Contributor** role to view job results. The **Quantum Workspace Data Contributor** role will be required to submit jobs. To avoid interruptions to your Azure Quantum workflows, assign these roles to the appropriate users before the deadline. For more information, see **Assign data access roles for users in the workspace**.

Prerequisites

To connect to your workspace with the `qdk.azure` module, you need:

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. If you don't have a workspace, see Create an Azure Quantum workspace.
- The latest version of the `qdk` Python package with the `azure` extra.

  Bash

  ```
  pip install --upgrade "qdk[azure]"
  ```

If you use the Azure CLI, update to the latest version. For installation instructions, see:

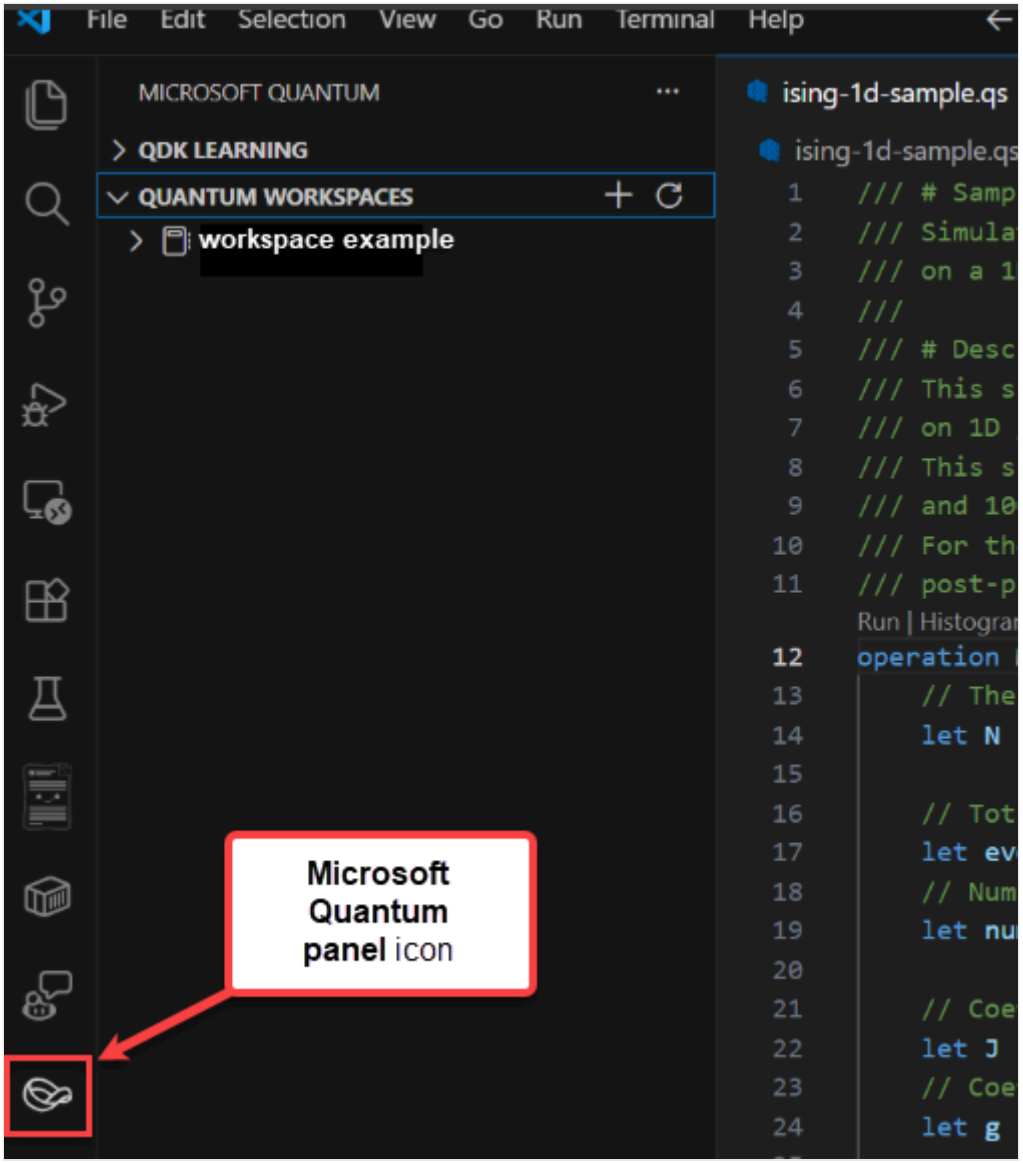
- Install Azure CLI on Windows
- Install Azure CLI on Linux
- Install Azure CLI on macOS

Connect with an Azure account

Use an Azure account to connect to your Azure Quantum workspace with the Microsoft Quantum Development Kit (QDK) extension in Visual Studio Code (VS Code).

1. Open VS Code.
2. Select **View** > **Command Palette**.
3. Enter and select **QDK: Connect to an Azure Quantum workspace**.
4. Choose **Azure account**.
5. When a VS Code message says that QDK wants to sign in, select **Allow**.
6. Select the account to use.
7. Select a tenant and subscription.

To view your Azure Quantum workspace, select the **Microsoft Quantum** icon on the Activity Bar.



Connect with a connection string

> **Note**
>
> Connection strings are less secure than an Azure account.

Use a connection string to specify the connection parameters for an Azure Quantum workspace. Connection strings are useful when:

- You want to share access to your workspace with others who don't have an Azure account.
- You want to share access to your workspace with others for a limited time.
- You can't use Microsoft Entra ID because of company policies.

> **Tip**
>
> Every Azure Quantum workspace has a primary key and a secondary key, and each key has its own connection string. To let others use your workspace, share the secondary key and keep the primary key for your own services. That way, you can replace the secondary key without causing downtime in your services. For more information about sharing access to your workspace, see Share your workspace access.

Copy the connection string

1. Sign in to the Azure portal.
2. Go to your Azure Quantum workspace.
3. In the workspace panel, expand the **Operations** dropdown and select **Access Keys**.
4. If the **Access Keys** slider is set to **Disabled**, set it to **Enabled**.
5. Select the **Copy** icon for the connection string that you want to copy. You can choose either the primary or secondary connection string.

> **Warning**
>
> Storing your account access keys or connection strings in clear text is a security risk. Store your account keys in an encrypted format, or migrate your apps to use Microsoft Entra authorization for your Azure Quantum workspace.

Use the connection string to access your Azure Quantum workspace

Use the connection string you copied to connect to your Azure Quantum workspace with the `qdk.azure` module or the QDK extension in VS Code.

Python

To connect to your Azure Quantum workspace, choose one of the following methods.

- Use the `from_connection_string` function to create a `Workspace` object.

  Python

  ```python
  from qdk.azure import Workspace

  connection_string = "<ConnectionString>" # Replace with your connection string
  workspace = Workspace.from_connection_string(connection_string)

  print(workspace.get_targets())
  ```

- Store your connection string in an environment variable.

  Python

  ```python
  import os
  from qdk.azure import Workspace

  connection_string = "<ConnectionString>" # Replace with your connection string
  os.environ["AZURE_QUANTUM_CONNECTION_STRING"] = connection_string

  workspace = Workspace()
  print(workspace.get_targets())
  ```

For more information about how to work with keys, see Authenticate in your workspace using an access key.

> **ⓘ Important**
>
> If you disable access keys for your workspace, you can't connect with connection strings. Use workspace parameters instead.

Connect to your workspace with workspace parameters

Every Azure Quantum workspace has a unique set of parameters that you use to connect to the workspace.

Expand table

Parameter	Description	Use when you connect with
`subscription_id`	The Azure subscription ID.	Azure CLI
`resource_group`	The Azure resource group name.	Azure CLI
`name`	The name of your Azure Quantum workspace.	Azure CLI
`resource_id`	The Azure resource ID of the Azure Quantum workspace.	Python

To find your workspace parameters, follow these steps:

1. Sign in to the Azure portal.
2. Go to your Azure Quantum workspace.
3. From your workspace panel, select **Overview**.
4. Expand the **Essentials** dropdown.
5. Copy the parameters from their corresponding fields.

> **Note**
>
> Ensure that you sign in to the correct tenant before you connect to your workspace. For more information about tenants, see How to manage Azure subscriptions with the Azure CLI.

Use workspace parameters to connect to your Azure Quantum workspace

To stay signed in when you run your Python scripts, open a terminal and run the following Azure CLI command to set the subscription for your workspace. Replace `<subscriptionId>` with your subscription ID.

Azure CLI

```
az account set --subscription <subscriptionId>
```

Use your workspace parameters to connect to your Azure Quantum workspace through the `qdk.azure` module or the Azure CLI.

Python

To connect to your Azure Quantum workspace, choose one of the following options.

- Specify the resource ID (recommended):

  Python

  ```python
  from qdk.azure import Workspace

  workspace = Workspace(resource_id="") # Add the resource ID of your
  workspace
  ```

- Specify the subscription ID, resource group, and workspace name:

  Python

  ```python
  from qdk.azure import Workspace

  workspace = Workspace(
      subscription_id="", # Add the subscription ID of your workspace
      resource_group="", # Add the resource group of your workspace
      name="" # Add the name of your workspace
      )
  ```

- Specify only the workspace name. This option might fail if you have multiple workspaces with the same name in the same tenant.

  Python

  ```python
  from qdk.azure import Workspace
  workspace = Workspace(name="") # Add the name of your workspace
  ```

Related content

- Manage quantum workspaces with Azure CLI
- Share access to your Azure Quantum workspace
- Manage quantum workspaces with the Azure Resource Manager
- Authenticate using a service principal
- Authenticate using a managed identity

Last updated on 11/06/2026

Understanding role-based access to your Azure Quantum workspace

Learn about the different security principals and roles you can use to manage access to your Azure Quantum workspace.

Azure role-based access control (RBAC)

Azure role-based access control (Azure RBAC) is the authorization system you use to manage access to Azure resources, such as a workspace. To grant access, you assign roles to a security principal.

Security principal

A security principal is an object that represents a user, group, service principal, or managed identity.

Expand table

Security principal	Definition
User	A user account that signs in to Azure to create, manage, and use resources.
Group	A group of users. Used to manage users that need the same access and permissions to resources.
Service principal	A user identity for an application, service, or platform that needs to access resources.
Managed identity	An automatically managed identity in Azure Active Directory (Azure AD) for applications to use when connecting to resources that support Azure AD authentication.

Role

When you grant access to a security principal, you assign a built-in role or create a custom role. The most commonly used built-in roles are **Owner**, **Contributor**, **Quantum Workspace Data Contributor**, and **Reader**.

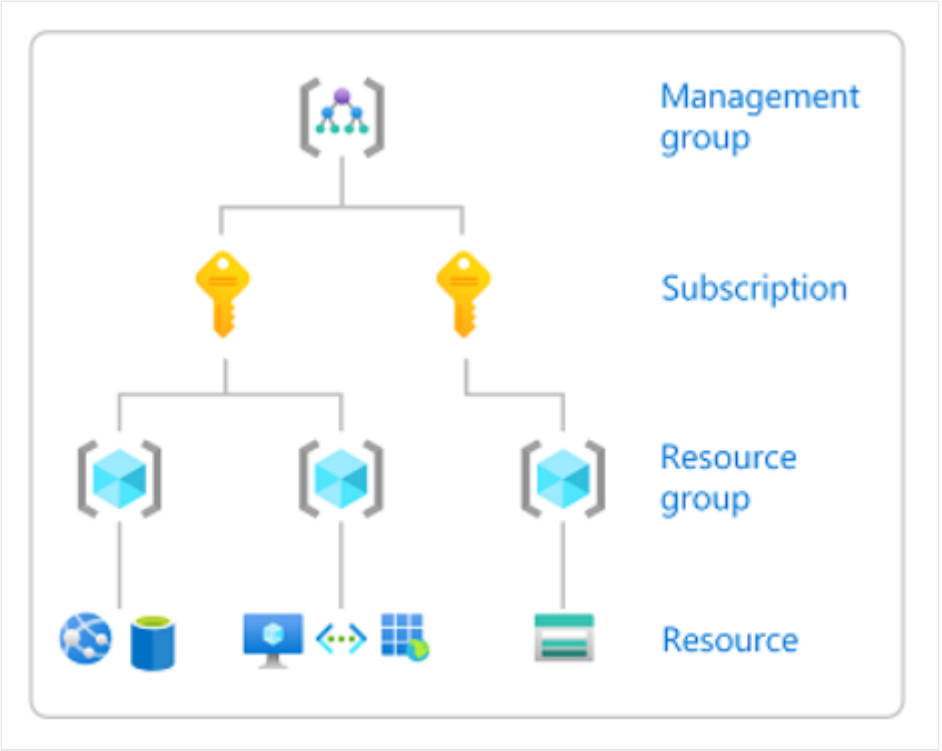
Expand table

Role	Access level
Owner	Grants full access to manage all resources, including the ability to assign roles in Azure RBAC.
Contributor	Grants full access to manage all resources, but doesn't allow you to assign roles in Azure RBAC.
Quantum Workspace Data Contributor	Grants access to submit and view jobs in the workspace, but doesn't allow you to create, delete, or modify a workspace.
Reader	View all resources, but doesn't allow you to make any changes.

> **Important**
>
> A role that grants workspace data access also grants access to every job in the workspace and the data in the associated storage account. Azure Quantum doesn't support job-specific access control lists (ACLs). For more information, see Understand Azure Quantum workspace and storage security boundaries.

Scope

Roles are assigned at a particular scope. Scope is the set of resources that the access applies to. Scopes are structured in a parent-child relationship. Each level of hierarchy makes the scope more specific. The level you select determines how widely the role is applied. Lower levels inherit role permissions from higher levels. You can assign roles at four levels of scope: management group, subscription, resource group, or resource.



Expand table

Scope	Description
Management group	Helps you manage access, policy, and compliance for multiple subscriptions. All subscriptions in a management group automatically inherit the conditions that are applied to the management group. You may need a management group if your organization has multiple subscriptions.
Subscription	Logically associates user accounts with the resources that they create. A user account is a user identity and one or more subscriptions. A subscription represents a grouping of Azure resources. An invoice is generated at the subscription scope. You must have an account with an active subscription to create Azure resources. For subscription options, see Create an Azure Quantum workspace.
Resource group	A container that holds related resources for an Azure solution. The resource group includes those resources that you want to manage as a group. For example, the following resources are required to run applications in Azure Quantum: • Azure storage account: stores input and output data for quantum jobs. • Azure Quantum workspace: a collection of assets associated with running quantum applications. These resources live in a single resource group.
Resource	An instance of a service that you can create, such as a workspace or storage account.

Note

Because access can be scoped to multiple levels in Azure, a user may have different roles at each level. For example, someone with owner access to a workspace may not have owner access to the resource group that contains the workspace.

Role requirements for creating a workspace

When you create a new workspace, you first select a subscription, resource group, and storage account to associate with the workspace. Your ability to create a workspace depends on the levels of access you have, starting at the subscription scope. To view your authorization for various resources, see Check your role assignments.

Subscription Owner

Subscription owners can create workspaces using either the **Quick create** or **Advanced create** options. You can either choose a resource group and storage account that already exists under the subscription or create new ones. You also have the ability to assign roles to other users.

Subscription Contributor

Subscription contributors can create workspaces using the **Advanced create** option.

- To create a new storage account, you must select an existing resource group that you're an owner of.
- To select an existing storage account, you must be an owner of the storage account. You must also select the existing resource group that the storage account belongs to.

Subscription contributors can't assign roles to others.

Subscription Reader

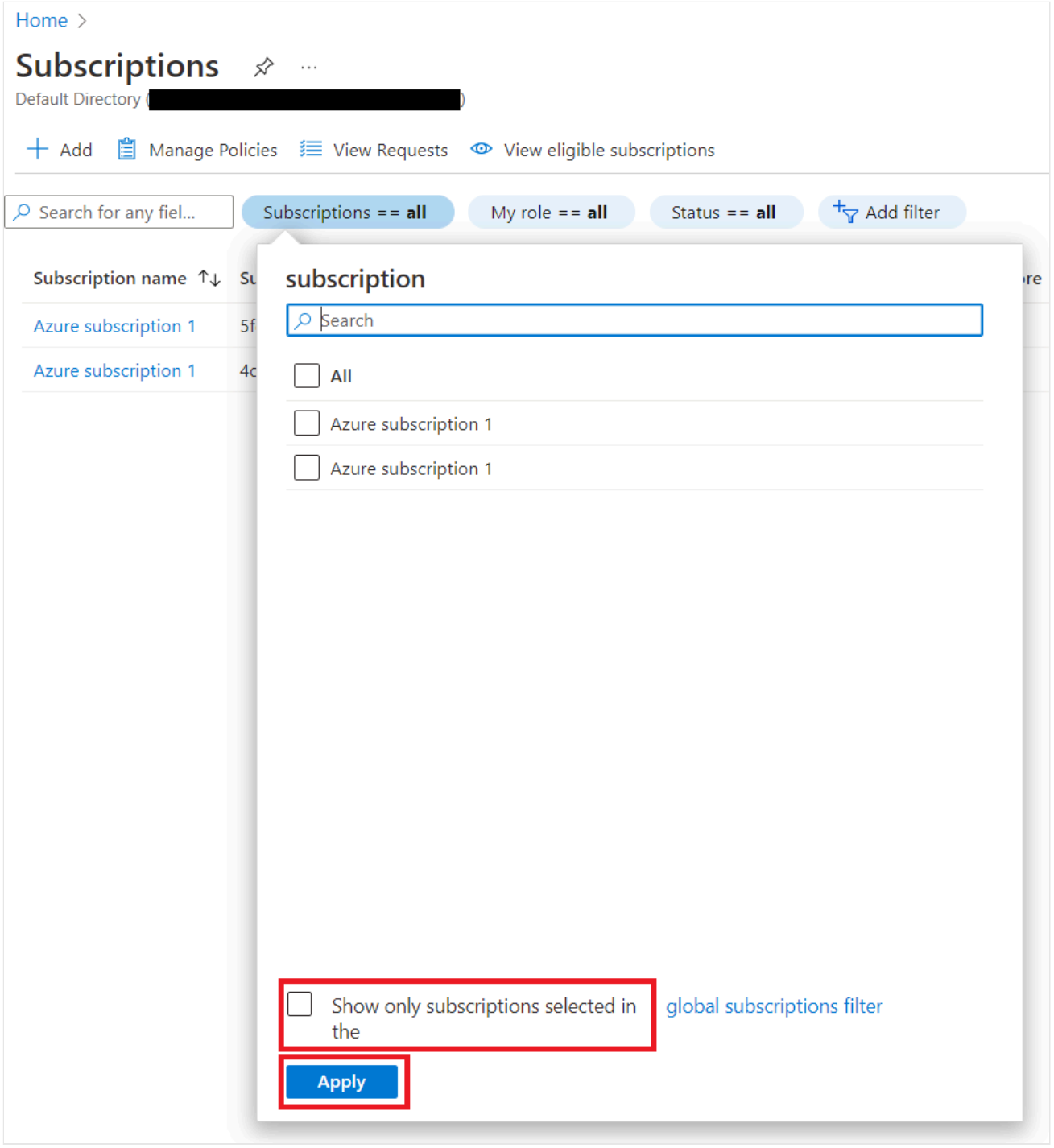
Subscription readers can't create workspaces. You can view all resources created under the subscription, but can't make any changes or assign roles.

Check your role assignments

Check your subscriptions

To see a list of your subscriptions and associated roles:

1. Sign in to the Azure portal.
2. Under the Azure services heading, select **Subscriptions**. If you don't see **Subscriptions** here, use the search box to find it.
3. The Subscriptions filter beside the search box may default to Subscriptions == **global filter**. To see a list of all your subscriptions, select the Subscriptions filter and **deselect** the "Select only subscriptions selected in the..." box. Then select **Apply**. The filter should then show Subscriptions == **all**.



Check your resources

To check the role assignment you or another user has for a particular resource, see Check access for a user to Azure resources.

Assign roles

To add new users to a workspace, you must be an owner of the workspace. To grant access to 10 or less users to your workspace, see Share access to your Azure Quantum workspace. To grant access to more than 10 users, see Add a group to your Azure Quantum workspace.

To assign roles for any resource at any scope, including the subscription level, see Assign Azure roles using the Azure portal.

Troubleshooting

For solutions to common issues, see Troubleshoot Azure Quantum: Creating an Azure Quantum workspace.

- When you create a resource in Azure, such as a workspace, you're not directly the owner of the resource. Your role is inherited from the highest scope role that you're authorized against in that subscription.
- When you create new role assignments, they can sometimes take up to one hour to take effect over cached permissions across the stack.

Last updated on 10/23/2026

Bulk add users to your Azure Quantum workspace

Summarize this article for me

Learn how to grant a group of users access to your Azure Quantum workspace. For example, you may need to grant your team members or students access to your workspace.

We recommend using the instructions in this article if you need to grant access to more than 10 users. For a smaller number of users, see Share access to your Azure Quantum workspace.

In this article you'll:

1. Create a group in the Microsoft Entra ID portal.
2. Add the group as a contributor to your Quantum workspace.
3. Bulk invite users to the Microsoft Entra ID.
4. Bulk import members to your group.

Prerequisites

You need the following prerequisites to bulk add users to an Azure Quantum workspace:

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. See Create an Azure Quantum workspace.
- A link for your Azure Quantum Workspace. To get the link for your Quantum workspace, go to your Azure Quantum workspace in the Azure portal and copy the URL from the address bar of your browser.

Create a group in the Microsoft Entra ID portal

1. Sign in to the Azure portal. You must be an **Owner** of the workspace, or have role assignment privileges, in order to add the group in the next section.

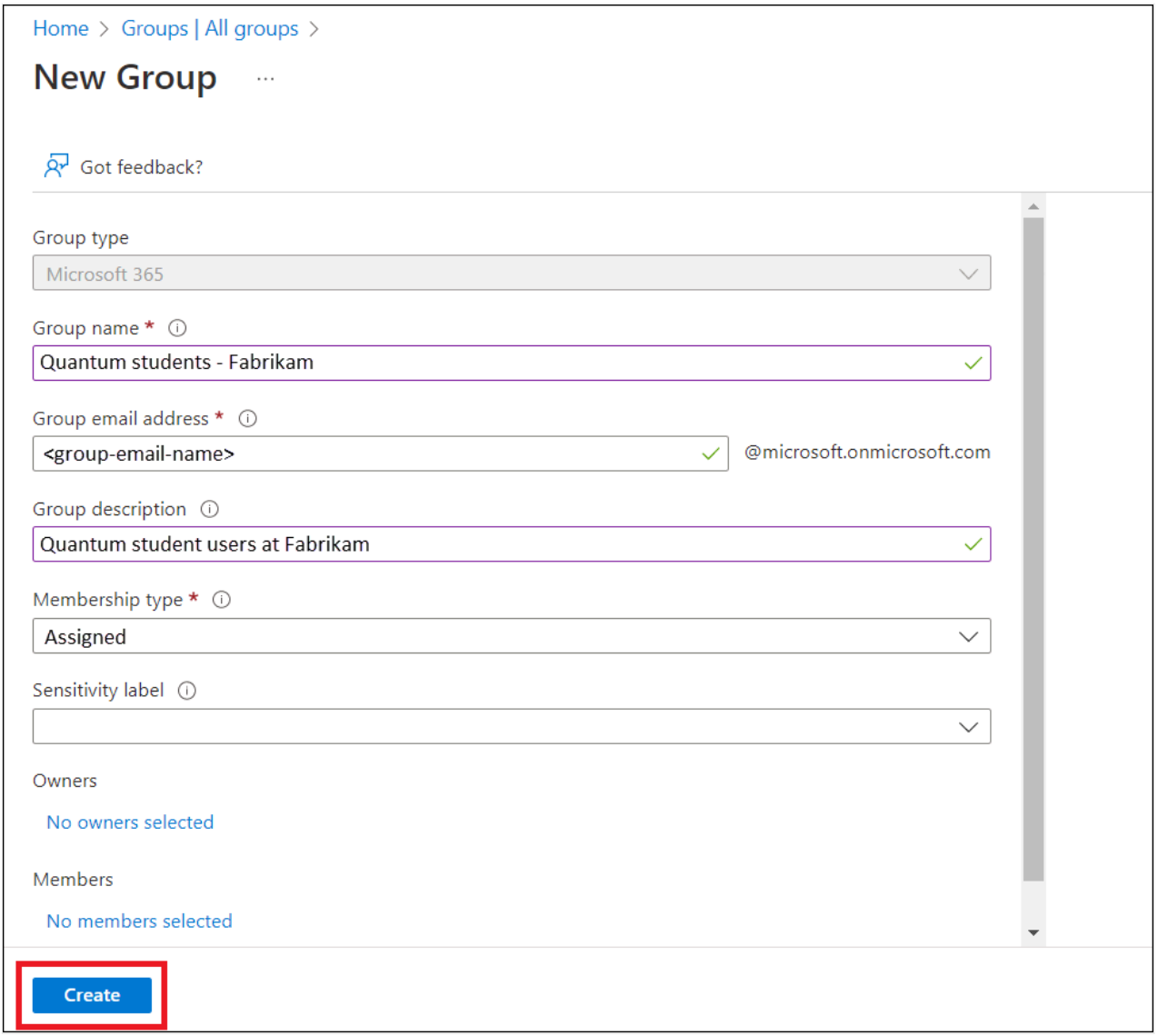
2. Search for and select **Microsoft Entra ID**.

3. On the **Microsoft Entra ID** page, select **Groups** from the left menu and then select **New group**.

4. Fill out the required information on the **New Group** page.

    - Select **Microsoft 365** as the **Group type**.

- Create and add a **Group name.**
- Add a **Group email address** for the group, or keep the email address that is filled in automatically.
- **Group description**. Add an optional description to your group.
- Select **Assigned** as the **Membership type**.



5. Select **Create**. You should get a notification that you've successfully created your group.

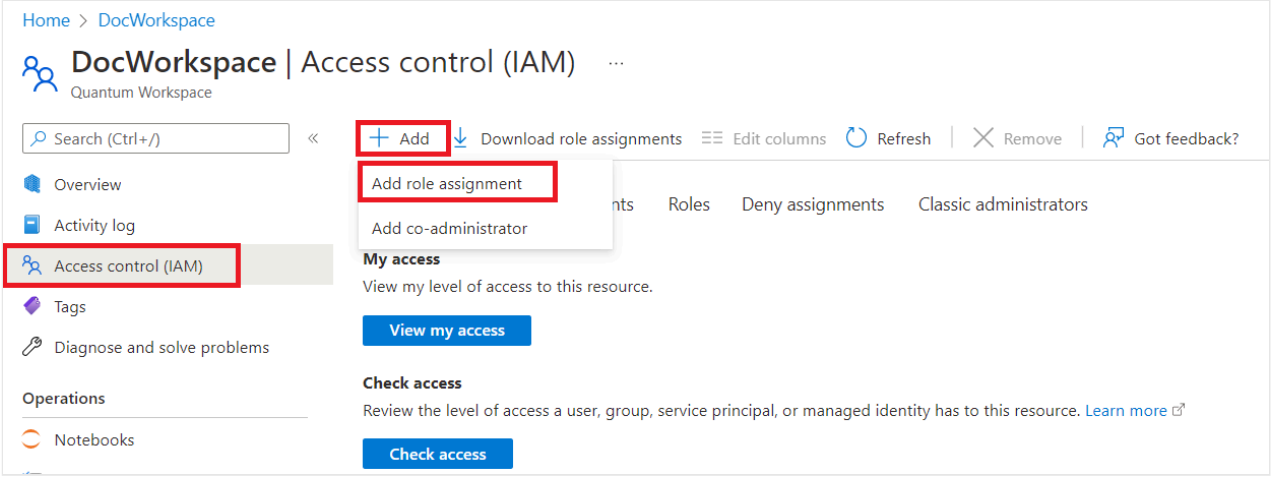
Add the group as a contributor to your Quantum workspace

> **Note**
>
> You can add your group as a **Contributor** role or as a **Quantum Workspace Data Contributor** role. The Contributor role allows users to manage the workspace properties, whereas the Quantum Workspace Data Contributor role only allows users to submit and

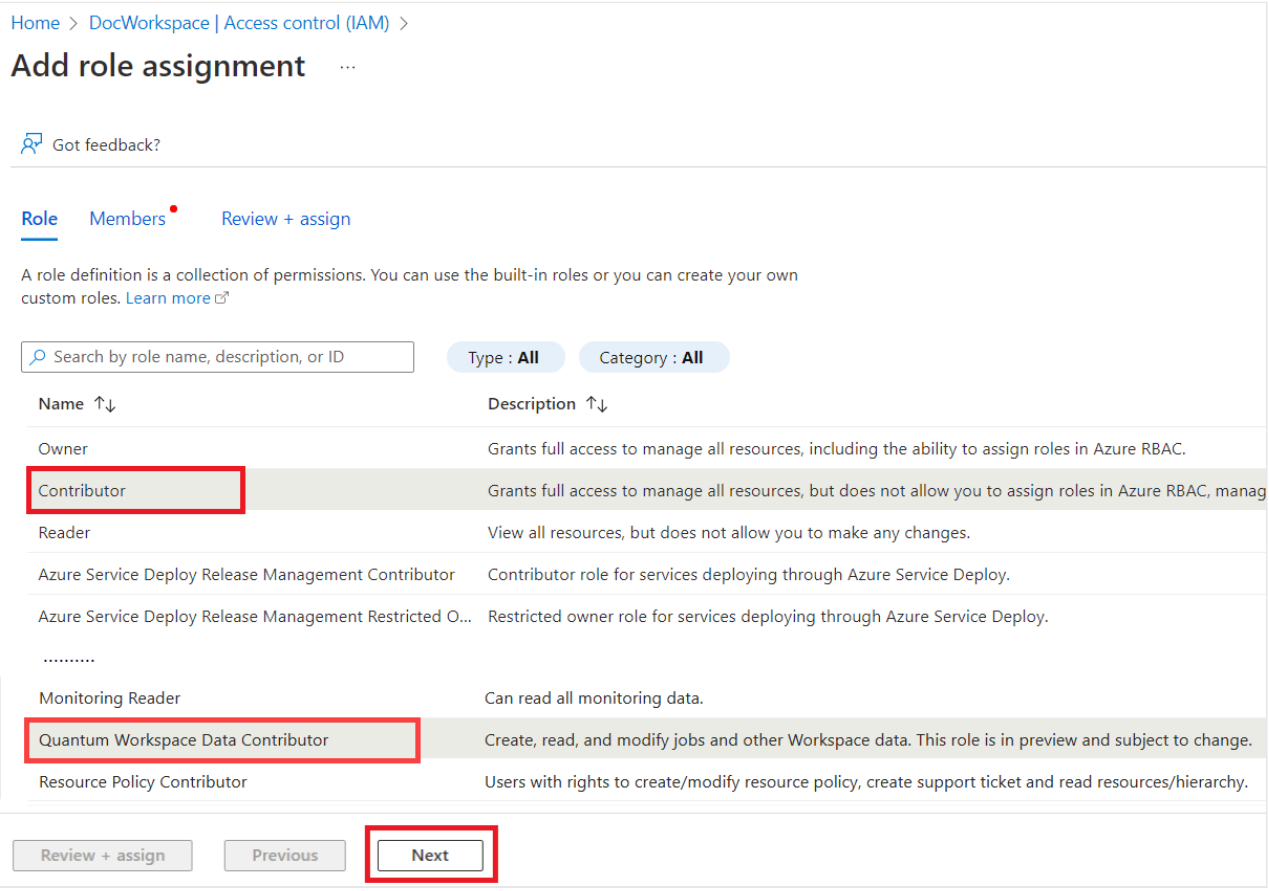
view jobs in the workspace. For more information, see Manage access to your Azure Quantum workspace.

1. In the Azure portal, navigate to your Azure Quantum workspace.

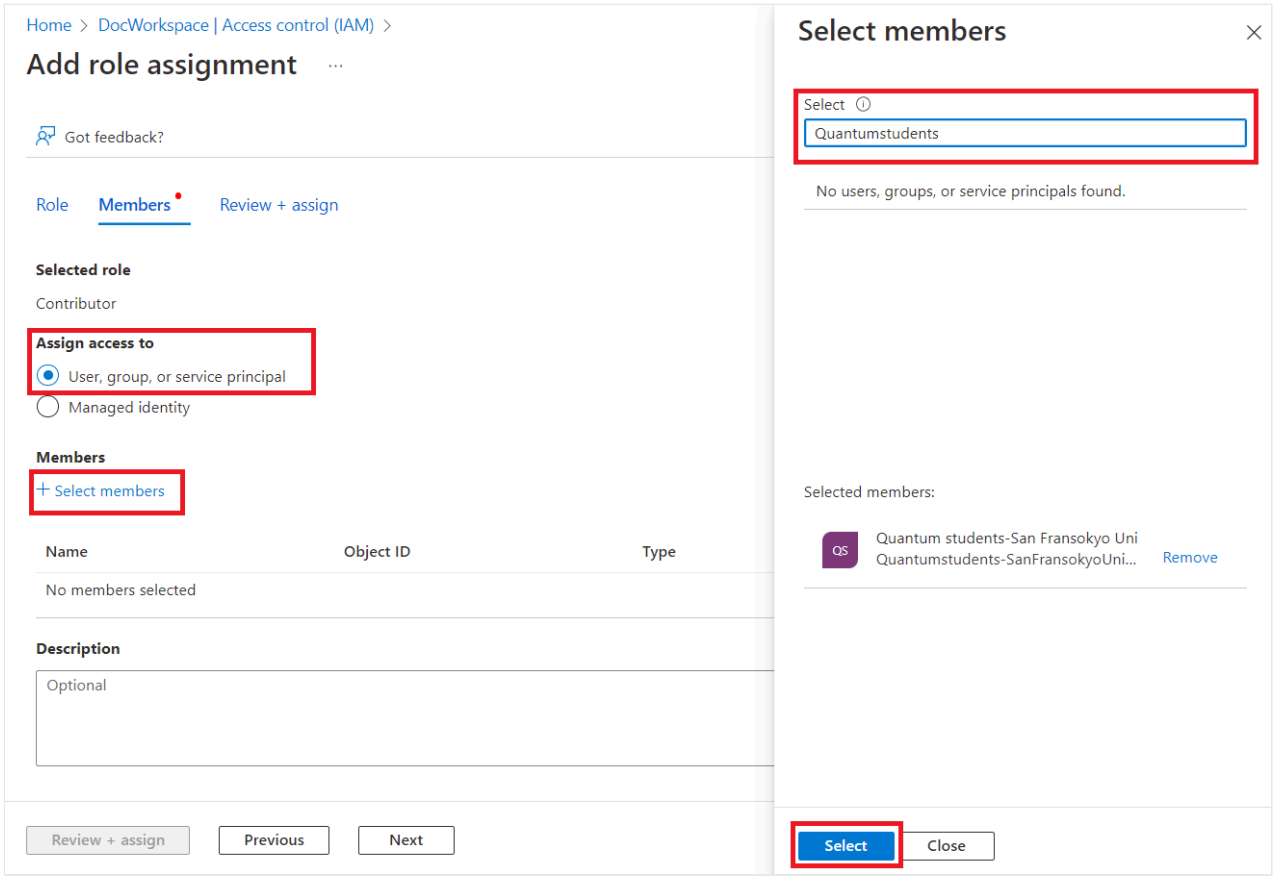
2. Allow your group to access your workspace. Select **Access control (IAM)** from the left-side menu. Select **Add**, and then **Add role assignment**.



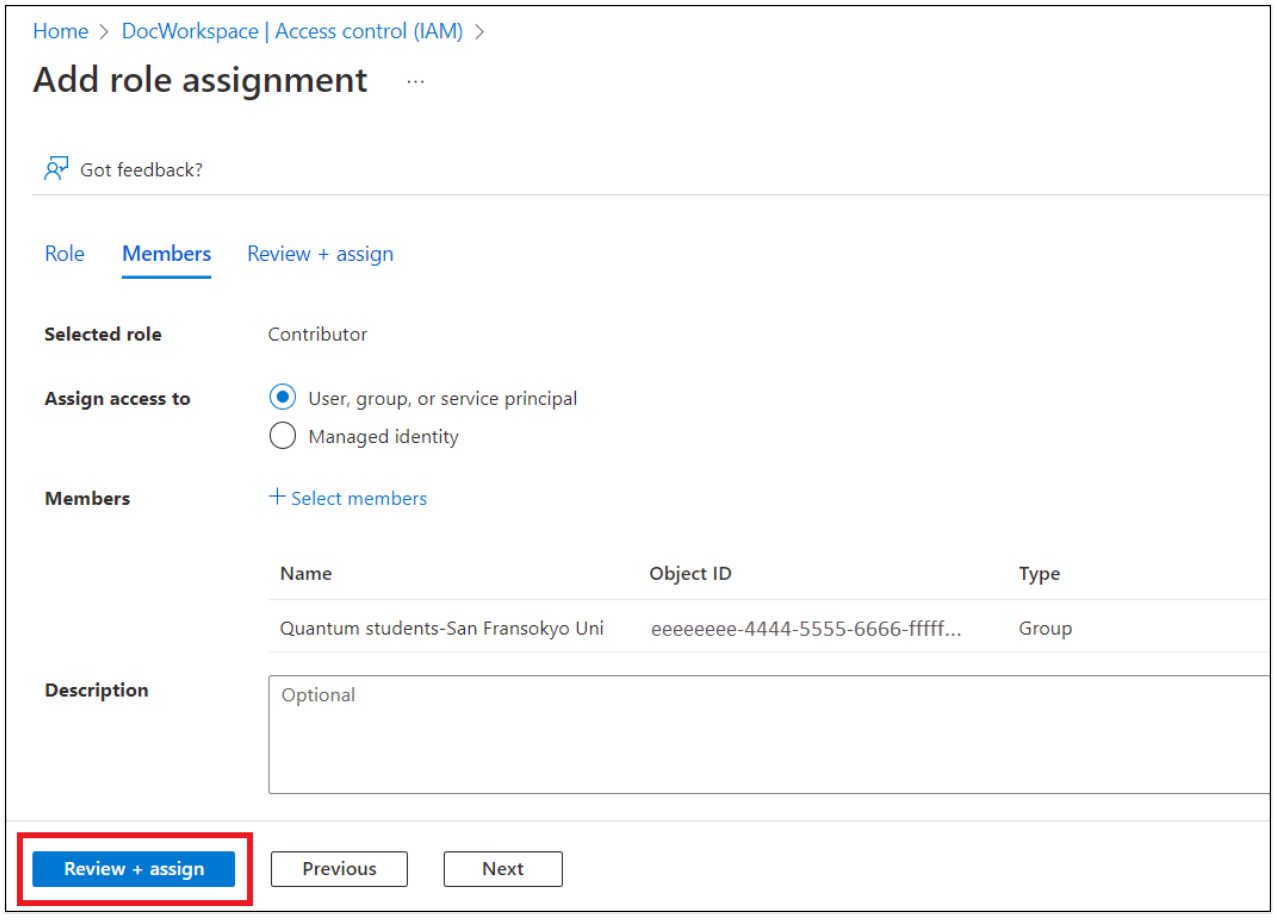
3. The **Add role assignment** page opens. On the **Role** pane, select **Contributor** or **Quantum Workspace Data Contributor** and then select **Next**.



4. On the **Members** pane, select Assign access to **User, group, or service principal**. Then select **+Select members**. The **Select members** blade opens. Search for your group name and select your group. Then select **Select**.



5. Your group name will appear under **Members**. Select **Review + assign**. On the **Review + assign** pane, Select **Review + assign** again. You should get a notification that your group was added as Contributor for your workspace.

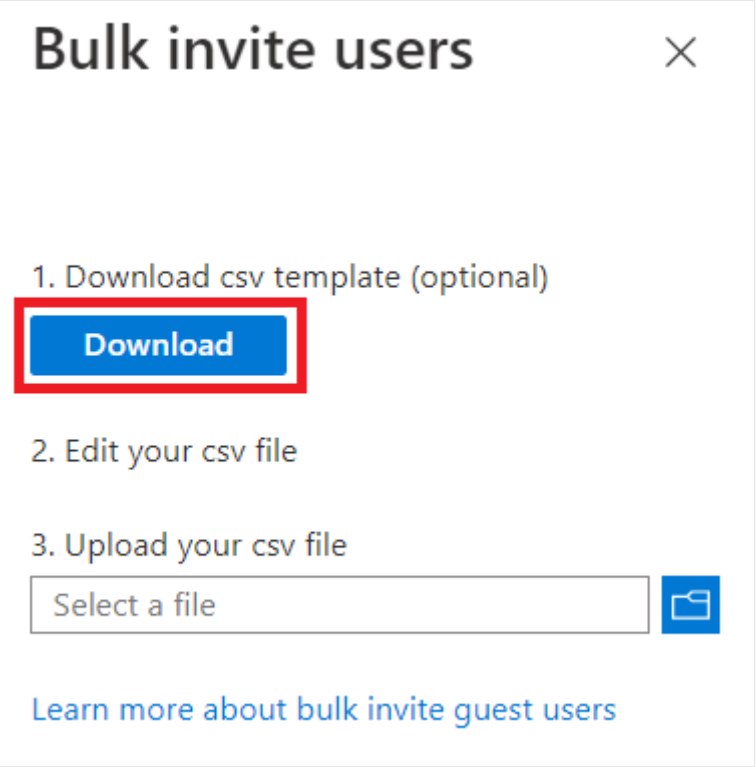


Bulk invite users to the Microsoft Entra ID

1. Go to Users - Microsoft Azure. On the left-side menu, go to **All users**.

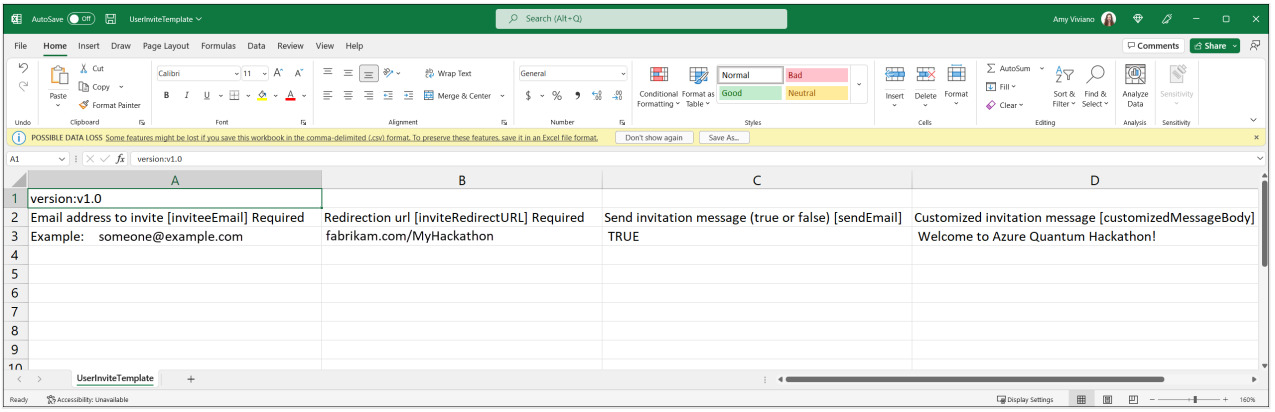
2. Select **Bulk Operations**, and then **Bulk invite**.

3. On the **Bulk invite users** pane, select **Download** to get a valid CSV template with invitation properties.



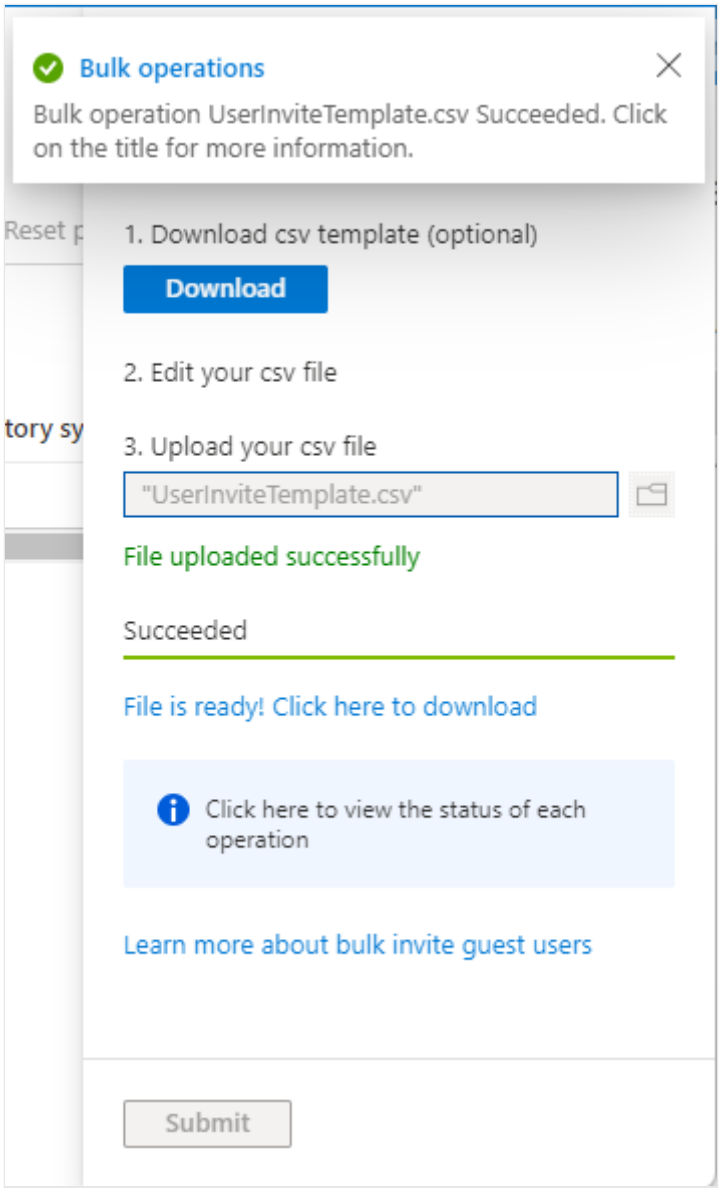
4. Open the CSV template and add a line for each user. The first two rows of the upload template must not be removed or modified, or the upload can't be processed. The third row provides examples of values for each column. You must remove the examples row and replace it with your own entries. Required values are:

   - **Email address to invite** - the user who will receive an invitation
   - **Redirection url** - the link to your Azure Quantum workspace. The invited user is forwarded to this link when they accept the invitation



5. Save the file.

6. On the **Bulk invite users** pane, under **Upload your csv file**, browse to the file. When you select the file, validation of the CSV file starts. Once the **File uploaded successfully** message appears, select **Submit** to start the bulk invite operation.



7. After the bulk invite users operation completes, your users will receive an invitation email. They want to accept the invitation.

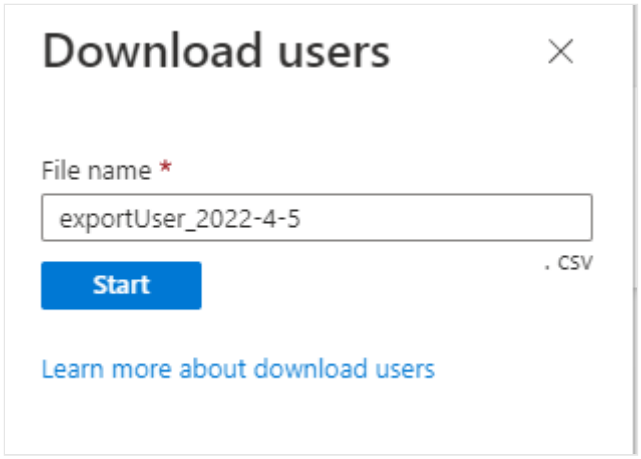
8. On the **Review permissions** page, users must select **Accept** before they can continue.

9. After permissions are accepted, your users will be added to the Microsoft Entra ID.

Bulk import members to your group

1. After the bulk invite is complete, download all Microsoft Entra ID users into a CSV file. Go to **All users**, and select **Download users**.

2. In the **Download users** pane, select **Start**.

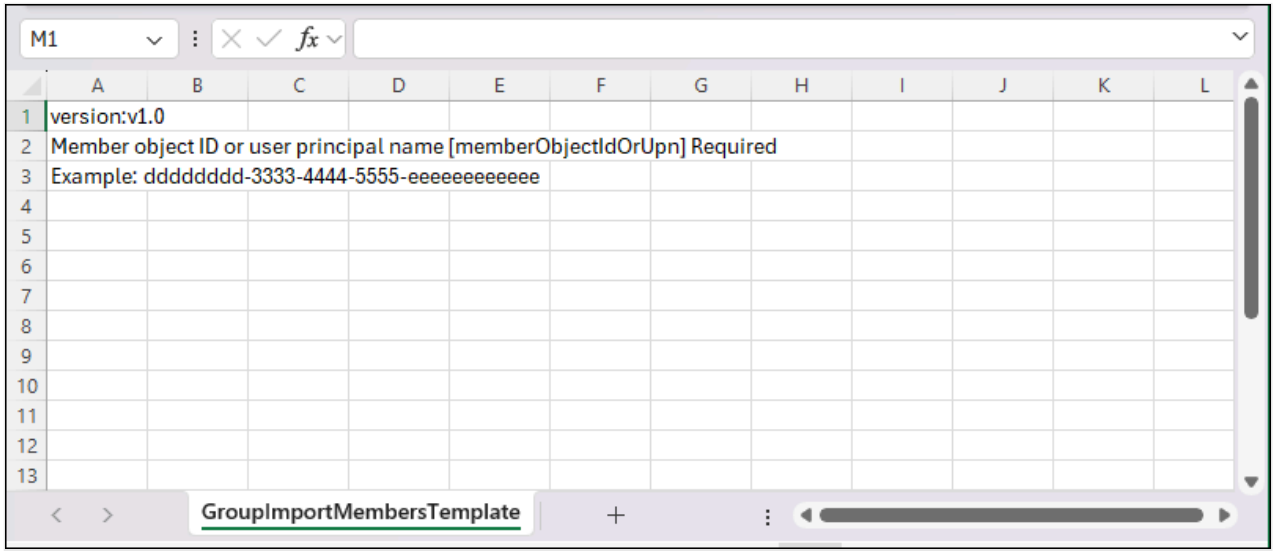


3. After the bulk users export is complete, click on **Download results** to download the CSV file with all your Microsoft Entra ID users.

4. After the download is complete, import your group members. Go to Groups - Microsoft Azure. Select your group, on the left-side menu, go to **Members**. In this blade, select **Bulk operations**, and then **Import members**.

5. On the **Bulk import group** pane, select **Download** to get a valid CSV template.

6. Open the CSV template and add a line for each user you want to invite to the group. Copy and paste your users' User Principal Names from the CSV that you downloaded in step 1. The third row provides an example value. You must remove the example row and replace it with your own entry.



7. Save the file.

8. On the **Bulk import group** pane, under **Upload your csv file**, browse to the file and upload it.

9. Once the **File uploaded successfully** message appears, select **Submit** to start the bulk import operation.
10. After the bulk import group operation completes, your group members will be added to the group successfully.

Last updated on 03/13/2026

Share access to your Azure Quantum workspace

Article • 01/07/2025

Learn how to share access to your Azure Quantum workspace. For example, you may need to grant your team members or students access to your workspace. You can assign roles to users to control their access to your workspace. For example, a **Contributor** role can create, delete, or modify a workspace, whereas a **Quantum Workspace Data Contributor** role has limited permissions and can primarily just submit and view jobs.

We recommend using the instructions in this article if you need to grant access to 10 or less users. For a larger number of users, it may be easier for you or your IT department to create a group of users and then grant it access to your workspace. For instructions, see Add a group to your Azure Quantum workspace.

Prerequisites

You need the following prerequisites to share access to your Azure Quantum workspace:

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. See Create an Azure Quantum workspace.

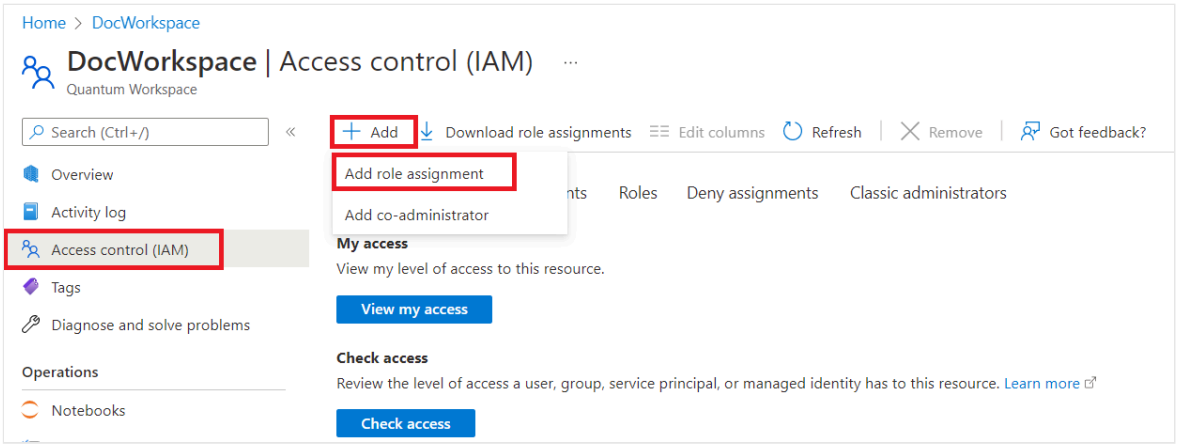
Azure Active Directory

Each user must have an account in your organization's Azure Active Directory before you can grant them access to your workspace. To add new users, you must be at least a **User Administrator**. For instructions, see Add a new user.

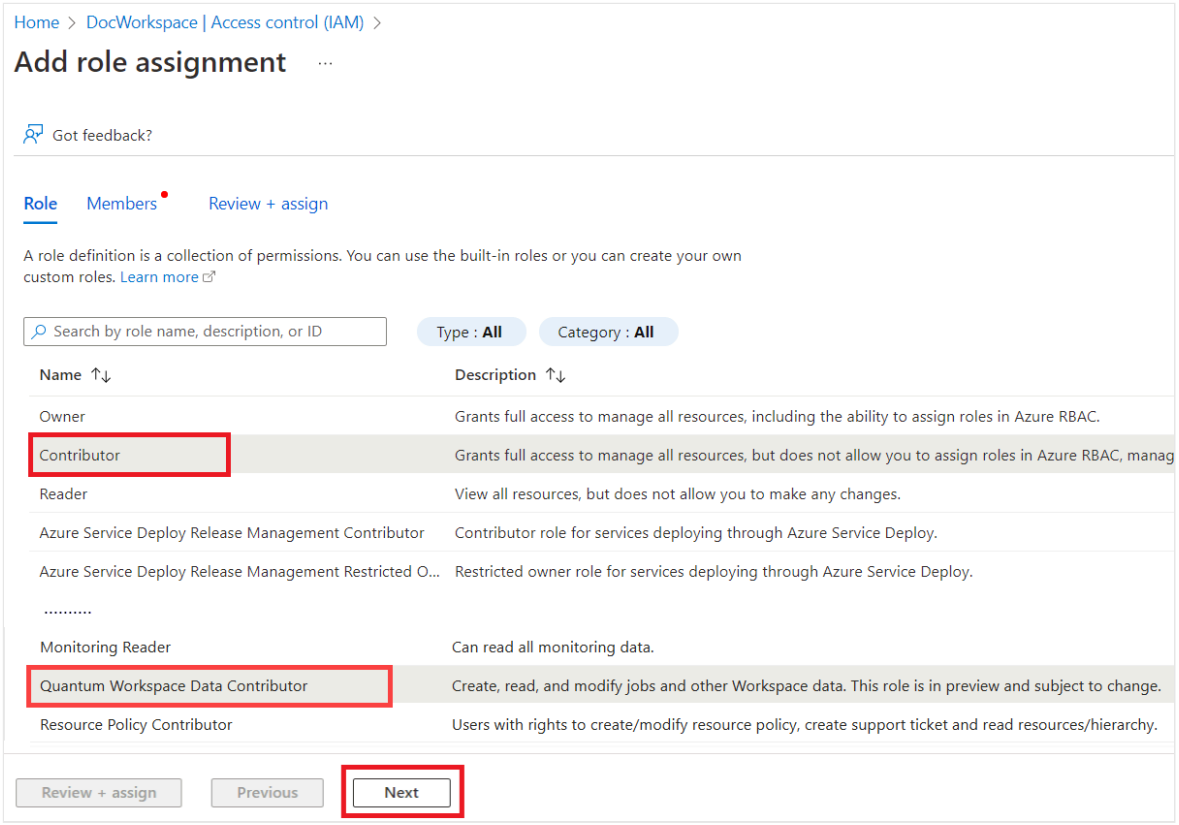
Add users to your Azure Quantum workspace

1. Sign in to the Azure portal and navigate to your Azure Quantum workspace. You must be an **Owner** of the workspace in order to add users.

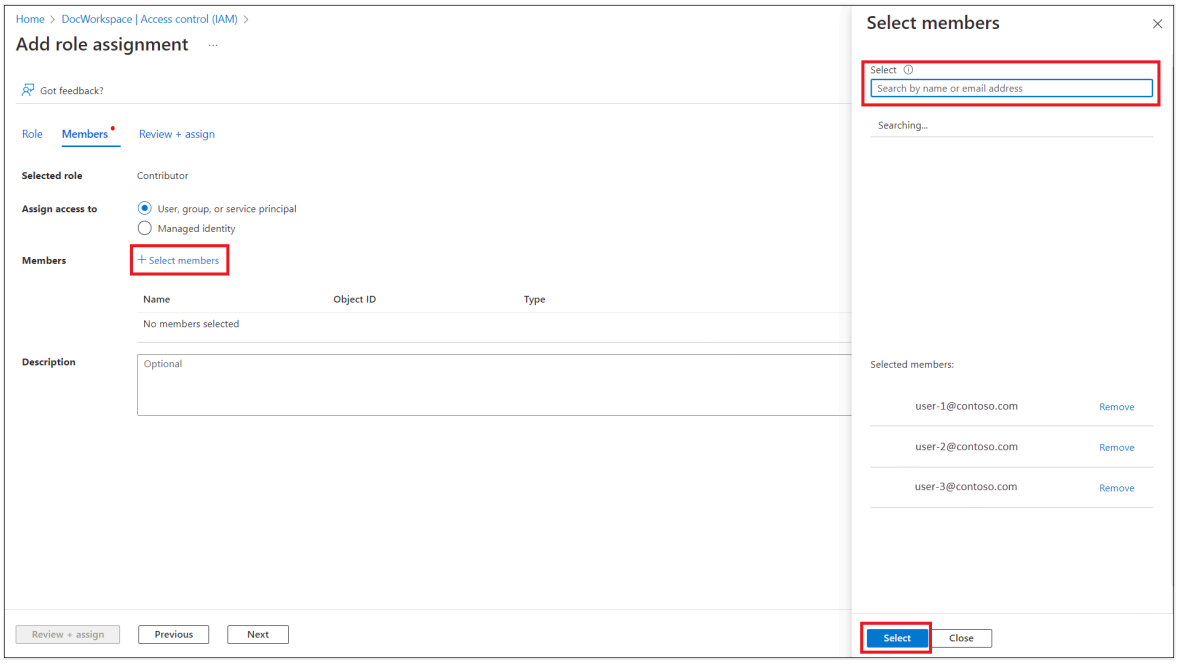
2. Select **Access control (IAM)** from the left-side menu. Select **Add**, and then **Add role assignment**.



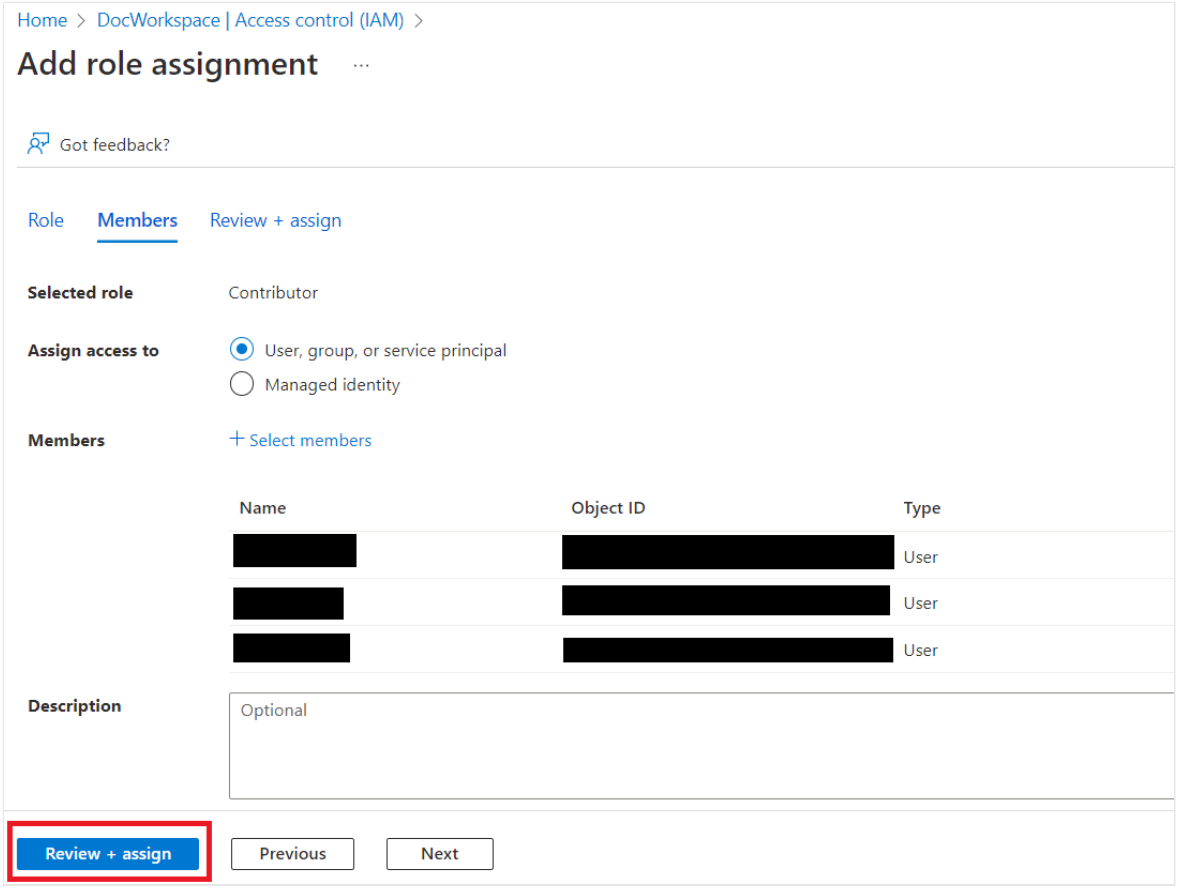
3. The **Add role assignment** page opens. On the **Role** pane, select **Contributor** or **Quantum Workspace Data Contributor** and then select **Next**.



4. On the **Members** pane, select **Assign access to User, group, or service principal**. Then select **+Select members**. The **Select members** blade opens. Search for and select each of your users. Then select the blue **Select** button.



5. A list of your users will appear under **Members**. Select **Review + assign**. On the **Review + assign** pane, Select **Review + assign** again. You should get a notification that your users were added to your workspace.



Share access using a connection string

You can share the access to your Azure Quantum workspace using a connection string. The connection string contains the information needed to connect to your workspace,

including the subscription ID, resource group, workspace name, and location.

Every Azure Quantum workspace has **primary and secondary** keys, and their corresponding connection strings. If you want to allow access to your workspace to others, you can share your secondary key and you use your primary for your own services. This way, you can replace the secondary key as needed without having downtime in your own services.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Migrate your Azure Quantum job data

Summarize this article for me

If you have an existing Azure Quantum workspace and you create new workspace, then you can migrate your job data from the old workspace to the new workspace. When you do the migration, your job input and output data are available in your new workspace. But you can't see the job history for those jobs anymore in the Azure portal, Azure Quantum SDK, or the Azure CLI.

Migration instructions depend on whether the existing Azure Quantum workspace uses **managed** or **unmanaged** storage:

- An **unmanaged** workspace uses a user-specified Azure Storage account that already exists and is explicitly selected during workspace creation.
- A **managed workspace** uses a storage account that Azure automatically creates and manages as part of the workspace.

> **Important**
>
> Don't delete your old workspace until after you migrate your data to your new workspace. Before you migrate your job data, make sure that you don't have open jobs in the queue. To remove an open job from the queue, either wait for the job to finish running or cancel the job.

Unmanaged storage

If your old workspace uses an unmanaged storage account, then link that storage account to your new workspace. The linked storage account contains your old job data along with data for jobs that you submit in your new workspace.

> **Note**
>
> To link a storage account to a Quantum workspace, you must have a role assignment that allows you to perform role assignments on that storage account, such as Owner or User Access Administrator.

To create a new Quantum workspace and link your storage account, follow these steps:

1. Sign in to the Azure portal, go to **Quantum Workspaces**, and then choose **Create**.

2. Choose your subscription in the **Subscription** dropdown list, and then select **Advanced create**.
3. In the **Project details** section, open the **Resource group** dropdown list and choose the resource group that contains the storage account for your old workspace.
4. In the **Instance details** section, enter a name for your new workspace in the **Workspace name** field, and then choose one of the supported regions from the **Region** dropdown list (East US, West US, North Europe, or West Europe).
5. For **Data Storage**, choose **Customize storage account settings (advanced)**. A **Storage Account** dropdown list appears that contains all the storage accounts in your selected resource group. Choose the storage account ID for your old workspace's storage account.
6. To review your workspace settings, choose **Review + create**. Or, choose **Next: Providers >** to customize your provider and plan options, and then choose **Next: Tags >** to add tags to your workspace.
7. Verify that all your workspace settings are correct, and then choose **Create**.

The storage account for your old workspace is now associated with your new workspace, and all job history is retained in that storage account.

> **Note**
>
> For questions about migration issues, email azurequantuminfo@microsoft.com.

Last updated on 03/13/2026

Work with Azure Quantum jobs

When you run a quantum program in Azure Quantum, you create and run a **job**. The steps to create and run a job depend on the job type and the provider and target that you configure for the workspace.

Job properties

All jobs have the following properties in common:

Expand table

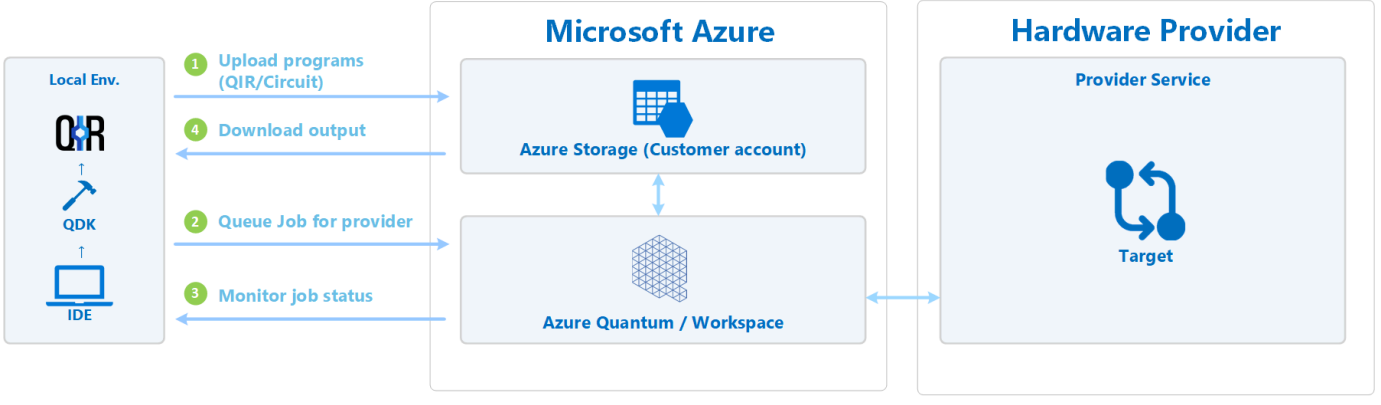
Property	Description
ID	A unique identifier for the job. It must be unique within the workspace.
Provider	*Who* you want to run your job.
Target	*What* you want to run your job on. For example, the exact quantum hardware or quantum simulator offered by the provider.
Name	A user-defined name to help organize your jobs.
Parameters	Optional input parameters for targets. See the documentation for the selected target for a definition of available parameters.

Once you create a job, various metadata is available about its state and run history.

Job lifecycle

Once you've written your quantum program, you can select a target and submit your job.

This diagram shows the basic workflow after you submit your job:



First, Azure Quantum uploads the job to the Azure Storage account that you configured in the workspace. Then, the job is added to the job queue for the provider that you specified in the job. Azure Quantum then downloads your program and translates it for the provider. The provider processes the job and returns the output to Azure Storage, where it is available for download.

> **Important**
>
> All principals with workspace data access can access every job and its stored inputs and outputs. Azure Quantum doesn't support job-specific access control lists. For more information, see Understand Azure Quantum workspace and storage security boundaries.

Monitoring jobs

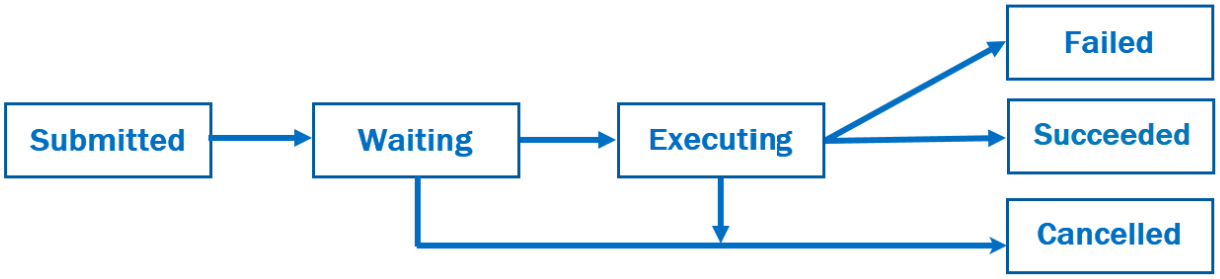
Once you submit a job, you can monitor the status of the job. Jobs have the following possible states:

Expand table

Status	Description
waiting	The job is waiting to run. Some jobs will perform pre-processing tasks in the waiting state. `waiting` is always the first state, however, a job may move to the `executing` state before you can observe it in `waiting`.
executing	The target is currently running the job.
succeeded	The job has succeeded, and output is available. This is a *final* state.
failed	The job has failed, and error information is available. This is a *final* state.
cancelled	The user requested to cancel the job run. This is a *final* state. For more information, see Job Cancellation in this article.

The `succeeded`, `failed`, and `cancelled` states are considered **final states**. Once a job is in one of these states, no more updates occur, and the corresponding job output data does not change.

This diagram shows the possible job state transitions:



After a job completes successfully, it displays a link to the output data in your Azure Storage account. How you access this data depends on the SDK or tool you used to submit the job.

How to monitor jobs

You can monitor jobs through Python, the Azure portal, and Azure CLI.

Python

All the properties of the job are accessible in `job.details`. For example, you can access the job name, status, and ID as follows:

Python

```
print(job.details)
print("\nJob name:", job.details.name)
print("Job status:", job.details.status)
print("Job ID:", job.details.id)
```

Output

```
{'additional_properties': {'isCancelling': False}, 'id': '0fc396d2-97dd-11ee-
9958-6ca1004ff31f', 'name': 'MyPythonJob', 'provider_id': 'rigetti'...}
Job name: MyPythonJob
Job status: Succeeded
Job ID: fc396d2-97dd-11ee-9958-6ca1004ff31f
```

Job counts

To get counts that result from a large number of jobs, you may want to perform a local install of the Quantum Development Kit tools. With a local install, you can store the job IDs locally.

You can copy the following code to get the list of jobs and their results:

Python

```
for job in workspace.list_jobs():
    print(job.id, job.details.name, job.details.output_data_uri)
```

Job cancellation

When a job is not yet in a final state (for example, `succeeded`, `failed`, or `cancelled`), you can request to cancel the job. All providers will cancel your job if it is in the `waiting` state. However, not all providers support cancellation if your job is in the `executing` state.

> **Note**
>
> If you cancel a job after it has started running, your account may still be billed a partial or full amount for that job. See the billing documentation for your selected provider

---

Last updated on 10/23/2026

Submit jobs to Azure Quantum with Azure CLI

Azure CLI is for advanced Azure Quantum users who already compile their programs to QIR and submit jobs from a terminal, shell script, or CI/CD workflow. To submit jobs with Azure CLI, you need detailed knowledge of job input and output formats, QIR entry points, and provider-specific parameters.

For most scenarios, submit jobs with the QDK extension for VS Code or the QDK Python package. These tools compile your programs and handle these details for you.

Azure CLI manages the Azure Quantum workspace, target, job metadata, and input file, but doesn't compile programs into QIR for you. The exact method to submit a job depends on the Azure Quantum target. The following steps are a general example of how to submit a QIR job with Azure CLI.

Install Azure CLI support

1. Install Azure CLI.

2. Install or update the Azure Quantum extension.

Azure CLI

```
az extension add --upgrade --name quantum
```

Connect to your workspace

1. Sign in to Azure.

Azure CLI

```
az login
```

2. Set the subscription that contains your Azure Quantum workspace.

Azure CLI

```
az account set --subscription <subscription-id>
```

3. Set the default workspace for subsequent commands.

Azure CLI

```
az quantum workspace set \
    --resource-group <resource-group-name> \
    --workspace-name <workspace-name>
```

4. List the targets that are available in the workspace.

Azure CLI

```
az quantum target list --output table
```

Submit a compiled input file

For a QIR job, specify the target, job name, QIR input format, input file, and QIR entry point.

Azure CLI

```
az quantum job submit \
    --target-id <target-id> \
    --job-name <job-name> \
    --job-input-format qir.v1 \
    --job-input-file <path-to-qir-file> \
    --entry-point ENTRYPOINT__main
```

The command returns the job ID. Save the ID so that you can monitor the job and retrieve its output.

> **Note**
>
> This example shows common QIR parameters. A provider-native job can require different input and output formats or additional parameters. See the documentation for the selected target and the az quantum job command reference.

Monitor the job and get the output

To check the job status, run the following command.

Azure CLI

```
az quantum job show --job-id <job-id> --output table
```

To get the output from a successful job, run the following command:

Azure CLI

```
az quantum job output --job-id <job-id> --output table
```

For workspace setup and other Azure Quantum commands, see Manage quantum workspaces with Azure CLI.

Related content

- How to submit jobs to Azure Quantum
- Submit jobs with the QDK extension for VS Code
- Submit jobs with the QDK Python package
- Work with Azure Quantum jobs

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/08/2026

Manage quantum workspaces with the Azure CLI

In this guide, you learn to use the Azure Command-Line Interface (Azure CLI) to create Azure Quantum workspaces and the required resource groups and storage accounts, and start running your quantum applications in Azure Quantum.

Prerequisites

To use the Azure Quantum service, you need:

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure resource group where the quantum workspace lives.
- A storage account in the resource group to be associated with the quantum workspace. Multiple workspaces can be associated with the same account.
- The Azure CLI.
- The Microsoft Quantum Development Kit.

> **Important**
>
> Although multiple workspaces can use the same storage account, don't share a storage account across workspaces that require different access. Each workspace can provide another path to the account's data. For more information, see **Understand Azure Quantum workspace and storage security boundaries**.

Environment setup

1. Install the Azure CLI `quantum` extension. Open a command prompt and run the following command, which also upgrades the extension if a previous version is already installed:

   Azure CLI

   ```
   az extension add --upgrade -n quantum
   ```

2. Log in to Azure using your credentials. You see a list of subscriptions associated with your account.

   Azure CLI

   ```
   az login
   ```

3. Specify the subscription you want to use.

   Azure CLI

   ```
   az account set -s <Your subscription ID>
   ```

4. If this is the first time creating quantum workspaces in your subscription, register the resource provider with this command:

   Azure CLI

   ```
   az provider register --namespace Microsoft.Quantum
   ```

Create an Azure Quantum workspace

To create a new Azure Quantum workspace, you need to know:

- The location or Azure region name to create the resource. You can use the list of regions and their resource manager codes supported by the Azure CLI tool, for example, **westus**.
- The resource group associated with the new workspace, for example, **MyResourceGroup**.
- A storage account in the same resource group and subscription as the quantum workspace. It's possible to create a new storage account from the Az CLI tool, for example, **MyStorageAccount**.
- The name of the quantum workspace to create, for example, **MyQuantumWorkspace**.
- The list of Azure Quantum providers to use in the workspace. A provider offers a set of plans, each of them representing a plan with associated terms and conditions, costs, and quotas. To create workspaces, you need to specify the corresponding plan along with the providers.

If you already know the provider and plan names to use in your workspace, you can skip to step four, below.

If you need to determine which providers and plans to use, proceed as follows:

1. To retrieve the list of available quantum providers, use the `list` command (this example uses **westus** as the location):

   Azure CLI

   ```
   az quantum offerings list \
        -l westus \
        -o table
   ```

   > **Tip**
   >
   > If want to see which providers give free credit, use the `--autoadd-only` parameter, for example:
   >
   > `az quantum offerings list --autoadd-only -l westus -o table`

2. Once you determine the provider and plan to include in your workspace, you can review terms using the `show-terms` command (adding your **MyProviderID** and **MyPlan** as example values):

   Azure CLI

   ```
   az quantum offerings show-terms \
        -l westus \
        -p MyProviderId \
        -k MyPlan
   ```

3. The output of the `show-terms` command includes a Boolean field `accepted` that shows whether the terms for this provider have been accepted already or not, as well as a link to the license terms to review. If you decide to accept those terms, use the `accept-terms` command to record your acceptance.

   Azure CLI

   ```
   az quantum offerings accept-terms \
        -l westus \
        -p MyProviderId \
        -k MyPlan
   ```

4. Once you have reviewed and accepted all required terms and conditions, you can create your workspace using the `create` command, specifying a list of provider and plan combinations separated by commas, as in the following example:

Azure CLI

```
az quantum workspace create \
     -l westus \
     -g MyResourceGroup \
     -w MyQuantumWorkspace \
     -a MyStorageAccount \
     -r "MyProvider1/MyPlan1, MyProvider2/MyPlan2"
```

Once you create a workspace, you can still add or remove providers using the Azure portal.

Change the default storage account for a quantum workspace

If you need to change the default storage account for an existing workspace, you can use the `create` command, specifying all the current properties along with the new storage account. The following example uses the same settings as the workspace created in the previous example:

Azure CLI

```
az quantum workspace create \
   -l westus \
   -g MyResourceGroup \
   -w MyQuantumWorkspace \
   -a MyNEWStorageAccount \
   -r "MyProvider1/MyPlan1, MyProvider2/MyPlan2"
```

> **Important**
>
> This procedure actually re-creates the workspace with the new storage account. Ensure that all properties other than the storage account are exactly the same as the original, otherwise a second workspace is created.

Delete a quantum workspace

If you know the name and resource group of a quantum workspace you want to delete, you can do it with the `delete` command (using the same names as the previous example):

Azure CLI

```
az quantum workspace delete \
   -g MyResourceGroup \
   -w MyQuantumWorkspace
```

> **Tip**
>
> If you don't remember the exact name, you can view the entire list of quantum workspaces in your subscription using `az quantum workspace list -o table`.

After you delete a workspace, it is still listed while it's being deleted in the cloud. However, the `provisioningState` property of the workspace changes immediately to indicate that it's being deleted. You can see this information by using the `show` command:

Azure CLI

```
az quantum workspace show \
   -g MyResourceGroup \
   -w MyQuantumWorkspace
```

> **Note**
>
> In case you used the `az quantum workspace set` command previously to specify a default quantum workspace, then you can call the `delete` command without parameters to delete (and clear) the default workspace.

Azure CLI

```
az quantum workspace delete
```

Next steps

Now that you can create and delete workspaces, learn about the different targets to run quantum algorithms in Azure Quantum.

Last updated on 10/23/2026

az quantum

Preview

> **Note**
>
> This reference is part of the **quantum** extension for the Azure CLI (version 2.73.0 or higher). The extension will automatically install the first time you run an **az quantum** command. Learn more about extensions.

This command group is in preview and under development. Reference and support levels: https://aka.ms/CLI_refstatus

Manage Azure Quantum Workspaces and submit jobs to Azure Quantum Providers.

Commands

Expand table

Name	Description	Type	Status
az quantum execute	Submit a job to run on Azure Quantum, and wait for the result. Equivalent to `az quantum run`.	Extension	Preview
az quantum job	Manage jobs for Azure Quantum.	Extension	Preview
az quantum job cancel	Request to cancel a job on Azure Quantum if it hasn't completed.	Extension	Preview
az quantum job delete	Delete a job from an Azure Quantum workspace.	Extension	Preview
az quantum job file	Manage a quantum job's associated files.	Extension	Preview
az quantum job file download	Download a file from a job's output storage container.	Extension	Preview
az quantum job file list	List the files stored in a job's output storage container.	Extension	Preview
az quantum job list	Get the list of jobs in a Quantum Workspace.	Extension	Preview
az quantum job output	Get the results of running a job.	Extension	Preview

Name	Description	Type	Status
az quantum job show	Get the job's status and details.	Extension	Preview
az quantum job submit	Submit a program or circuit to run on Azure Quantum.	Extension	Preview
az quantum job update	Update a submitted job's name, priority, and/or tags.	Extension	Preview
az quantum job wait	Place the CLI in a waiting state until the job finishes running.	Extension	Preview
az quantum offerings	Manage provider offerings for Azure Quantum.	Extension	Preview
az quantum offerings accept-terms	Accept the terms of a provider and SKU combination to enable it for workspace creation.	Extension	Preview
az quantum offerings list	Get the list of all provider offerings available on the given location.	Extension	Preview
az quantum offerings show-terms	Show the terms of a provider and SKU combination including license URL and acceptance status.	Extension	Preview
az quantum run	Submit a job to run on Azure Quantum, and wait for the result. Equivalent to `az quantum execute`.	Extension	Preview
az quantum suite-offer	View Azure Quantum suite offers for V2 providers available to the subscription.	Extension	Preview
az quantum suite-offer list	List V2 provider suite offers available to the current subscription, including provider ID, name, company, and location.	Extension	Preview
az quantum suite-offer quotas	View quota allocations and their consumed usages for a V2 provider suite offer in the current subscription.	Extension	Preview
az quantum suite-offer target	List targets available through an Azure Quantum suite offer for a V2 provider.	Extension	Preview
az quantum suite-offer target list	List targets and their status available through a V2 provider suite offer, without requiring a workspace.	Extension	Preview
az quantum target	Manage targets for Azure Quantum workspaces.	Extension	Preview
az quantum target clear	Clear the default target-id.	Extension	Preview
az quantum target list	Get the list of providers and their targets in an Azure Quantum workspace.	Extension	Preview
az quantum target set	Select the default target to use when submitting jobs to Azure Quantum.	Extension	Preview

Name	Description	Type	Status
az quantum target show	Get the Target ID of the current default target to use when submitting jobs to Azure Quantum.	Extension	Preview
az quantum workspace	Manage Azure Quantum workspaces.	Extension	Preview
az quantum workspace clear	Clear the default Azure Quantum workspace.	Extension	Preview
az quantum workspace create	Create a new Azure Quantum workspace.	Extension	Preview
az quantum workspace delete	Delete the given (or current) Azure Quantum workspace.	Extension	Preview
az quantum workspace keys	Manage Azure Quantum Workspace api keys.	Extension	Preview
az quantum workspace keys list	List api keys for the given (or current) Azure Quantum workspace.	Extension	Preview
az quantum workspace keys regenerate	Regenerate api key for the given (or current) Azure Quantum workspace.	Extension	Preview
az quantum workspace list	Get the list of Azure Quantum workspaces available.	Extension	Preview
az quantum workspace quotas	List quota allocations and consumed usages for an Azure Quantum workspace.	Extension	Preview
az quantum workspace set	Select a default Azure Quantum workspace for future commands.	Extension	Preview
az quantum workspace show	Get the details of the given (or current) Azure Quantum workspace.	Extension	Preview
az quantum workspace update	Update the given (or current) Azure Quantum workspace.	Extension	Preview
az quantum workspace user	Manage users of an Azure Quantum workspace.	Extension	Preview
az quantum workspace user add	Grant a user access to an Azure Quantum workspace.	Extension	Preview
az quantum workspace user list	List the users with access to an Azure Quantum workspace.	Extension	Preview
az quantum workspace user remove	Remove a user's access to an Azure Quantum workspace.	Extension	Preview

az quantum execute

Preview

> Command group 'quantum' is in preview and under development. Reference and support levels: https://aka.ms/CLI_refstatus

Submit a job to run on Azure Quantum, and wait for the result. Equivalent to `az quantum run`.

Azure CLI

```
az quantum execute --job-input-file
                   --job-input-format
                   --resource-group
                   --target-id
                   --workspace-name
                   [--acquire-policy-token]
                   [--change-reference]
                   [--entry-point]
                   [--job-name]
                   [--job-output-format]
                   [--job-params]
                   [--shots]
                   [--storage]
                   [--target-capability]
```

Examples

Run QIR bitcode from a file in the current folder and wait for the result.

Azure CLI

```
az quantum execute -g MyResourceGroup -w MyWorkspace -t MyTarget \
    --job-name MyJob --job-input-format qir.v1 --job-input-file MyQirBitcode.bc \
    --entry-point MyQirEntryPoint
```

Run a Quil pass-through job on the Rigetti simulator and wait for the result.

Azure CLI

```
az quantum execute -g MyResourceGroup -w MyWorkspace \
   -t rigetti.sim.qvm --job-name MyJob --job-input-file MyProgram.quil \
   --job-input-format rigetti.quil.v1 --job-output-format rigetti.quil-results.v1
```

Submit a Qiskit circuit to the IonQ simulator with job params and wait for the results.

Azure CLI

```
az quantum execute -g MyResourceGroup -w MyWorkspace \
   -t ionq.simulator --job-name MyJobName --job-input-file MyCircuit.json \
   --job-input-format ionq.circuit.v1 --job-output-format ionq.quantum-results.v1 \
   --job-params count=100 content-type=application/json
```

Required Parameters

`--job-input-file`

The location of the input file to submit.

`--job-input-format`

The format of the file to submit.

`--resource-group -g`

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

`--target-id -t`

Execution engine for quantum computing jobs. When a workspace is configured with a set of providers, they each enable one or more targets. You can configure the default target using `az quantum target set`.

`--workspace-name -w`

Name of the Quantum Workspace. You can configure the default workspace using `az quantum workspace set`.

Optional Parameters

The following parameters are optional, but depending on the context, one or more might become required for the command to execute successfully.

`--acquire-policy-token`

Acquiring an Azure Policy token automatically for this resource operation.

Expand table

Property	Value
Parameter group:	Global Policy Arguments

`--change-reference`

The related change reference ID for this resource operation.

Expand table

Property	Value
Parameter group:	Global Policy Arguments

`--entry-point`

The entry point for the QIR program or circuit. Required for some provider QIR jobs.

`--job-name`

A friendly name to give to this run of the program.

`--job-output-format`

The expected job output format.

`--job-params`

Job parameters passed to the target as a list of key=value pairs, json string, or `@{file}` with json content.

`--shots`

The number of times to run the program on the given target.

`--storage`

If specified, the ConnectionString of an Azure Storage is used to store job data and results.

`--target-capability`

Target-capability parameter passed to the compiler.

Global Parameters

`--debug`

Increase logging verbosity to show all debug logs.

Expand table

Property	Value
Default value:	False

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

Expand table

Property	Value
Default value:	False

`--output -o`

Output format.

Expand table

Property	Value
Default value:	json
Accepted values:	json, jsonc, none, table, tsv, yaml, yamlc

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

Expand table

Property	Value
Default value:	False

az quantum run

Preview

Command group 'quantum' is in preview and under development. Reference and support levels: https://aka.ms/CLI_refstatus

Submit a job to run on Azure Quantum, and wait for the result. Equivalent to `az quantum execute`.

Azure CLI

```
az quantum run --job-input-file
               --job-input-format
               --resource-group
               --target-id
               --workspace-name
               [--acquire-policy-token]
               [--change-reference]
               [--entry-point]
               [--job-name]
               [--job-output-format]
               [--job-params]
               [--shots]
               [--storage]
               [--target-capability]
```

Examples

Run QIR bitcode from a file in the current folder and wait for the result.

Azure CLI

```
az quantum run -g MyResourceGroup -w MyWorkspace -t MyTarget \
    --job-name MyJob --job-input-format qir.v1 --job-input-file MyQirBitcode.bc \
    --entry-point MyQirEntryPoint
```

Run a Quil pass-through job on the Rigetti simulator and wait for the result.

Azure CLI

```
az quantum run -g MyResourceGroup -w MyWorkspace \
   -t rigetti.sim.qvm --job-name MyJob --job-input-file MyProgram.quil \
   --job-input-format rigetti.quil.v1 --job-output-format rigetti.quil-results.v1
```

Submit a Qiskit circuit to the IonQ simulator with job params and wait for the results.

Azure CLI

```
az quantum run -g MyResourceGroup -w MyWorkspace \
   -t ionq.simulator --job-name MyJobName --job-input-file MyCircuit.json \
   --job-input-format ionq.circuit.v1 --job-output-format ionq.quantum-results.v1 \
   --job-params count=100 content-type=application/json
```

Required Parameters

`--job-input-file`

The location of the input file to submit.

`--job-input-format`

The format of the file to submit.

`--resource-group -g`

Name of resource group. You can configure the default group using `az configure --defaults group=<name>`.

`--target-id -t`

Execution engine for quantum computing jobs. When a workspace is configured with a set of providers, they each enable one or more targets. You can configure the default target using `az quantum target set`.

`--workspace-name -w`

Name of the Quantum Workspace. You can configure the default workspace using `az quantum workspace set`.

Optional Parameters

The following parameters are optional, but depending on the context, one or more might become required for the command to execute successfully.

`--acquire-policy-token`

Acquiring an Azure Policy token automatically for this resource operation.

Expand table

Property	Value
Parameter group:	Global Policy Arguments

`--change-reference`

The related change reference ID for this resource operation.

Expand table

Property	Value
Parameter group:	Global Policy Arguments

`--entry-point`

The entry point for the QIR program or circuit. Required for some provider QIR jobs.

`--job-name`

A friendly name to give to this run of the program.

`--job-output-format`

The expected job output format.

`--job-params`

Job parameters passed to the target as a list of key=value pairs, json string, or `@{file}` with json content.

`--shots`

The number of times to run the program on the given target.

`--storage`

If specified, the ConnectionString of an Azure Storage is used to store job data and results.

`--target-capability`

Target-capability parameter passed to the compiler.

Global Parameters ^

`--debug`

Increase logging verbosity to show all debug logs.

Expand table

Property	Value
Default value:	False

`--help -h`

Show this help message and exit.

`--only-show-errors`

Only show errors, suppressing warnings.

Expand table

Property	Value
Default value:	False

`--output -o`

Output format.

Expand table

Property	Value
Default value:	json
Accepted values:	json, jsonc, none, table, tsv, yaml, yamlc

`--query`

JMESPath query string. See http://jmespath.org/ for more information and examples.

`--subscription`

Name or ID of subscription. You can configure the default subscription using `az account set -s NAME_OR_ID`.

`--verbose`

Increase logging verbosity. Use --debug for full debug logs.

Expand table

Property	Value
Default value:	False

Manage quantum workspaces using Bicep

In this guide, learn to use a Bicep template to create Azure Quantum workspaces and the required resource groups and storage accounts. After template deployment, you can start running your quantum applications in Azure Quantum. Treating your infrastructure as code enables you to track changes to your infrastructure requirements and makes your deployments more consistent and repeatable.

Bicep uses a declarative syntax that you treat like application code. If you're familiar with the JSON syntax for writing Azure Resource Manager (ARM) templates, you'll find that Bicep provides a more concise syntax and improved type safety.

Prerequisites

Azure account

Before you begin, you must have an Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.

Editor

To create Bicep templates, you need a good editor. We recommend the latest version of VS Code with the Bicep extension.

Command-line deployment

You also need either Azure PowerShell or Azure CLI to deploy the template. If you use Azure CLI, you must have the latest version. For the installation instructions, see:

- Install Azure PowerShell
- Install Azure CLI on Windows
- Install Azure CLI on Linux
- Install Azure CLI on macOS

Sign in to Azure

After installing either Azure PowerShell or Azure CLI, make sure you sign in for the first time. Choose one of the following tabs and run the corresponding command line commands to sign in

to Azure:

Azure CLI

Azure CLI

```
az login
```

If you have multiple Azure subscriptions, select the subscription you want to use. Replace `SubscriptionName` with your subscription name. You can also use the subscription ID instead of the subscription name.

Azure CLI

```
az account set --subscription SubscriptionName
```

Create an empty resource group

When you deploy a template, specify a resource group that contains the quantum workspace with its associated resources. Before you run the deployment command, create the resource group with either Azure CLI or Azure PowerShell.

Azure CLI

Azure CLI

```
az group create --name myResourceGroup --location "East US"
```

Review the Azure Bicep template

Azure CLI

```
@description('Application name used as prefix for the Azure Quantum workspace and its associated Storage account.')
param appName string

@description('Location of the Azure Quantum workspace and its associated Storage account.')
@allowed([
  'westus'
```

```
  'eastus'
  'northeurope'
  'westeurope'
  'canary'
])
param location string

var quantumWorkspaceName = '${appName}-ws'
var storageAccountName = '${appName}${substring(uniqueString(resourceGroup().id), 0,
5)}'


resource storageAccount 'Microsoft.Storage/storageAccounts@2021-06-01' = {
  name: storageAccountName
  location: location
  sku: {
    name: 'Standard_LRS'
  }
  kind: 'StorageV2'
}

resource quantumWorkspace 'Microsoft.Quantum/Workspaces@2019-11-04-preview' = {
  name: quantumWorkspaceName
  location: location
  identity: {
    type: 'SystemAssigned'
  }
  properties: {
    providers: [
      {
        providerId: 'Microsoft'
        providerSku: 'DZH3178M639F'
        applicationName: '${quantumWorkspaceName}-Microsoft'
      }
    ]
    storageAccount: storageAccount.id
  }
}

resource roleAssignment 'Microsoft.Authorization/roleAssignments@2020-04-01-preview' =
{
  scope: storageAccount
  name: guid(quantumWorkspace.id,
'/subscriptions/${subscription().subscriptionId}/providers/Microsoft.Authorization/rol
eDefinitions/b24988ac-6180-42a0-ab88-20f7382dd24c', storageAccount.id)
  properties: {
    roleDefinitionId:
'/subscriptions/${subscription().subscriptionId}/providers/Microsoft.Authorization/rol
eDefinitions/b24988ac-6180-42a0-ab88-20f7382dd24c'
    principalId: reference(quantumWorkspace.id, '2019-11-04-preview',
'full').identity.principalId
  }
}

output subscription_id string = subscription().subscriptionId
```

```
output resource_group string = resourceGroup().name
output name string = quantumWorkspace.name
output location string = quantumWorkspace.location
output tenant_id string = subscription().tenantId
```

The following Azure resources are created by the template:

- Azure Storage Account: storage account for storing input and output data for quantum jobs.
- Azure Quantum workspace: a collection of assets associated with running quantum applications.

The template also grants the quantum workspace **Contributor** permission to the storage account. This step is needed so that the workspace can read and write job data.

> ⓘ **Important**
>
> The storage account becomes part of the workspace security boundary. Use a dedicated storage account when another workspace, user group, or application requires different access to the data. For more information, see Understand Azure Quantum workspace and storage security boundaries.

The template generates the following output. You can use these values later to identify the generated quantum workspace and authenticate to it:

- **Subscription ID** hosting all the deployed resources.
- **Resource Group** containing all deployed resources.
- **Name** of the quantum workspace.
- **Location** of the datacenter that hosts the workspace.
- **Tenant ID** holding credentials used at the deployment.

Deploy the template

To deploy the template, use either Azure CLI or Azure PowerShell. Use the resource group you created. Give a name to the deployment so you can easily identify it in the deployment history. Replace `{provide-the-path-to-the-template-file}` and the curly braces `{}` with the path to your template file. Furthermore, replace `{provide-app-name}` and `{provide-location}` with values for the overall application name and the location where the workspace should reside. The app name should only contain letters.

Azure CLI

To run this deployment command, you must have the latest version of Azure CLI.

Azure CLI

```
templateFile="{provide-the-path-to-the-template-file}"
az deployment group create \
  --name myDeployment \
  --resource-group myResourceGroup \
  --template-file $templateFile \
  --parameters appName="{provide-app-name}" location="{provide-location}"
```

The deployment command returns results. Look for `ProvisioningState` to see whether the deployment succeeded.

> **Important**
>
> In some cases you might get a deployment error (Code: PrincipalNotFound). Reason for this is that the workspace principal was not created yet when the resource manager tried to configure the role assignment. If this is the case, just repeat the deployment. It should succeed in the second run.

Validate the deployment

You can verify the deployment by exploring the resource group from the Azure portal.

1. Sign in to the Azure portal.
2. From the left menu, select **Resource groups**.
3. Select the resource group deploy in the last procedure. The default name is **myResourceGroup**. You should see two resources deployed within the resource group - the storage account and the quantum workspace.
4. Verify that the quantum workspace has necessary access rights for the storage account. Select the storage account. In the left menu pane, select **Access Control (IAM)** and verify that under **Role assignments** the quantum workspace resource is listed under **Contributor**.

Clean up resources

If you no longer need the quantum workspace, you might want to delete the resource group.

Azure CLI

Azure CLI

```
az group delete --name myResourceGroup
```

Next steps

Now that you can create and delete workspaces, learn about the different targets to run quantum algorithms in Azure Quantum. You now also have the tools to do workspace deployments from within Azure Pipelines or GitHub Actions.

Last updated on 10/23/2026

Pricing plans for Azure Quantum providers

In Azure Quantum, hardware and software providers define and control the pricing of their offerings. The information in this article is subject to change by providers and some delays in reflecting latest pricing information may exist. Be sure to verify the latest pricing information from the Azure Quantum workspace that you're using.

IonQ

IonQ charges based on a token pricing model which billing unit is the *Azure Quantum Token (AQT)*. This model is specific to Azure Quantum. The number of Azure Quantum Tokens is calculated by the following formula:

$$AQT \quad m \qquad N_q \; C \qquad N_q \; C$$

where:

- AQT is the number of Azure Quantum Tokens consumed by the program
- m is the minimum price per program execution, which is USD97.50 if error mitigation is on and USD12.4166 if error mitigation is off
- The units $N_q \; C$ and $N_q \; C$ are called *gate-shots*, where N_q and N_q are the number of one- and two-qubit gates submitted, respectively, and C is the number of execution shots

Multi-controlled two-qubit gates are billed as N two-qubit gates, where N is the number of qubits involved in the gate. For example, a NOT gate with three controls would be billed as or 12 two-qubit gates. One-qubit gates are billed as 0.225 of a two-qubit gate (rounded down). To learn more about IonQ, visit IonQ provider page.

IonQ offers a **pay-as-you-go** plan and a **monthly subscription** plan with access to the quantum simulator, the Aria 1 25-qubit quantum computers, and IonQ Forte 1 and Forte Enterprise 1 36-qubit quantum computers.

Pay As You Go

The Pay-as-you-go plan consists of *a la carte* access to the IonQ Aria 1 25-qubit, IonQ Forte 1 and Forte Enterprise 1 36-qubit quantum computers, and the IonQ simulator. The use of the quantum computers is charged based on the number of AQTs + Azure infrastructure costs.

Expand table

Includes access to	Pricing
IonQ simulator	Free of charge

IonQ Aria 1	• USD0.000220 / 1-qubit-gate shot
	• USD0.000975 / 2-qubit-gate shot
	• Minimum price per program execution:
	  ◦ USD97.50 - default setting, error mitigation is on
	  ◦ USD12.4166 if error mitigation is off

IonQ Forte 1	• USD0.0001645 / 1-qubit-gate shot
	• USD0.001121 / 2-qubit-gate shot
	• Minimum price per program execution:
	  ◦ USD168.195 - default setting, error mitigation is on
	  ◦ USD25.7899 if error mitigation is off

IonQ Forte Enterprise 1	• USD0.0001645 / 1-qubit-gate shot
	• USD0.001121 / 2-qubit-gate shot
	• Minimum price per program execution:
	  ◦ USD168.195 - default setting, error mitigation is on
	  ◦ USD25.7899 if error mitigation is off

For more information about Azure infrastructure costs, see Azure Blob Storage pricing.

> **Note**
>
> If you are an existing research grant or a Pay-as-you-go customer, you may see another billing unit in addition to AQT, called *Quantum Gate-Shots (QGS)*. The QGS units are equivalent to AQT units. The number of QGS is calculated by the following formula:
>
> $$QGS \quad N \quad C$$
>
> where:
>
> - QGS is the number of quantum gate-shots
> - N is the number of one- or two-qubit gates submitted
> - C is the number of execution shots

Pasqal

Pasqal charges for job execution time on its quantum processor - the 100-qubit Orion Beta generation called Fresnel - and its state-of-the-art emulators - EMU-MPS and EMU-SV.

Pasqal offers one billing plan: **Pay As You Go**.

Pay As You Go

In the Pay-as-you-go plan the usage is charged based on the job execution time only.

Expand table

Pricing	Includes access to
• EUR 3,000 per QPU-hour + Azure infrastructure costs • EUR 15 per emulator-hour + Azure infrastructure costs	• Pasqal FRESNEL, FRESNEL_CAN1 QPUs • Pasqal EMU-MPS, EMU-SV emulators

For more information about Azure infrastructure costs, see Azure Blob Storage pricing.

Quantinuum

Quantinuum uses a system that charges each job depending on the number of operations in the job, and the number of shots you run. The usage units defined by Quantinuum are *Hardware Quantum Credits (HQCs)* for jobs submitted to quantum computers and emulator HQCs (eHQCs) for jobs submitted to emulators.

HQCs and eHQCs are used to calculate the cost of running a job, and they are calculated based on the following formula:

$$HQC \qquad C\ N_q \qquad N_q \qquad N_m$$

where:

- N_q is the number of single-qubit operations in a circuit.
- N_q is the number of native two-qubit operations in a circuit. Native gate is equivalent to CNOT up to several single-qubit gates.
- N_m is the number of state preparation and measurement (SPAM) operations in a circuit including initial implicit state preparation and any intermediate and final measurements and state resets.
- C is the shot count.

To learn more about Quantinuum, visit the Quantinuum provider page.

Quantinuum provides two subscription plans: **Standard** and **Premium**. Quantinuum also offers a **Pay as You Go** offering.

Subscriptions

Subscriptions are monthly subscription plans available through queued access.

Expand table

Plan	Pricing	Includes access to
Standard Plan	USD125,000/Month + Azure infrastructure costs	• Monthly standard subscription plan with 10k HQCs of standard priority access / month with unlimited credits at a basic tier 2 priority on Quantinuum System Model H2. • 100k eHQCs of premium priority access / month on and unlimited credits at basic tier 2 priority for use on the Quantinuum System Model H2 emulator
Premium Plan	USD175,000/Month + Azure infrastructure costs	• Monthly premium subscription plan with 17k HQCs of premium priority access / month with unlimited credits at a basic tier 1 priority on Quantinuum System Model H2. • 170k eHQCs of premium priority access / month and unlimited credits at basic tier 1 priority for use on the Quantinuum System Model H2 emulators

For more information about Azure infrastructure costs, see Azure Blob Storage pricing.

Rigetti

Rigetti charges for job execution time on their quantum processors. There's no added charge per job, per shot, or per gate. The Quantum Virtual Machine (QVM) simulator is free for all users under all plans.

Rigetti offers one billing plan: **Pay As You Go**.

Pay As You Go

The Pay-as-you-go plan consists of *a la carte* access to Rigetti QPUs. The usage is charged based on the job execution time only.

Expand table

Pricing	Includes access to
USD 0.02 per 10-millisecond increment of job execution time	Rigetti Cepheus-1-108Q

> **Note**
>
> If you have questions or run into any issue using Azure Quantum, you can contact AzureQuantumInfo@microsoft.com.

Related content

- Azure Quantum quotas
- Quantum computing target list

Last updated on 11/02/2026

FAQ: Limits and quotas in Azure Quantum

In this article, you find answers to questions about limits and quotas for usage of Azure Quantum providers.

What are quotas in Azure Quantum?

Quotas are limits on the usage of quantum processing unit (QPU) targets. Each Azure Quantum provider sets their own quotas. Quotas help prevent accidental cost overages for the user and preserve the integrity of the provider's systems.

Quotas are based on your provider plan selection. If you want to increase your quotas, then submit a support ticket. The usage tracked by quotas isn't necessarily tied to a cost or credit, but might be correlated.

How are quotas defined in Azure Quantum?

In Azure Quantum, hardware and software providers define and control the quotas of their offerings. For detailed quota information, see each provider reference page.

- IonQ quota
- PASQAL quota
- Quantinuum quota

> **Note**
>
> Providers that aren't in this list don't define quotas for their targets.

How can I view my remaining quota?

Azure Quantum usage and quotas are measured in terms of each provider's unit of usage. Some providers don't define quotas and don't display usage information.

Track quotas in the Azure portal

1. Sign in to the Azure portal with the credentials for your Azure subscription.
2. Go to your Azure Quantum workspace.

3. In the workspace navigation pane, expand the **Operations** dropdown and choose **Quotas**.
4. See the consumed and the remaining quotas for each of your workspace's providers.

Quota information is displayed in three columns:

- **Workspace usage**: The usage limit for the current workspace. Each Azure Quantum workspace has a usage limit.
- **Azure subscription usage**: The usage for all workspaces within the current region and subscription. Not all quotas are tracked at this level.
- **Cadence**: The period when your quota is renewed. If monthly, the usage is reset on the 1st of each month. If one-time, usage is never reset.

Track quotas with Azure CLI

You can see your quotas by using the Azure Command-Line Interface (Azure CLI). For more information, see Manage quantum workspaces with the Azure CLI.

1. Install the Azure CLI `quantum` extension. Open a command prompt and run the following command, which also upgrades the extension when you already have a previous version installed.

   Azure CLI

   ```
   az extension add --upgrade -n quantum
   ```

2. Sign in to Azure by using your credentials. You see a list of subscriptions associated with your account.

   Azure CLI

   ```
   az login
   ```

3. If you're prompted to select a subscription, then enter the number that corresponds to your subscription. Or, press **Enter** and then set the subscription that you want to use. Replace `MySubscription` with your subscription ID.

   Azure CLI

   ```
   az account set -s MySubscription
   ```

4. Set the workspace that you want to use. Replace `MyResourceGroup` and `MyWorkspace` with your resource group and workspace name.

   Azure CLI

   ```azurecli
   az quantum workspace set -g MyResourceGroup -w MyWorkspace -o table
   ```

5. Display your quota information in a table for the selected workspace.

   Azure CLI

   ```azurecli
   az quantum workspace quotas -o table
   ```

Your output looks like the following example:

Output

```
| Dimension | Holds | Limit      | Period   | ProviderId | Scope        |
Utilization |
| --------- | ----- | ---------  | -------- | ---------- | ------------ | ---------
-- |
| qgs       |  0.0  |  8333334.0 | Infinite | ionq       | Subscription | 33334.0
|
| hqc       |  0.0  |  800.0     | Infinite | quantinuum | Subscription | 0.0
|
```

In this example, the `qgs` row shows that the account has a limit of `8333334 qgs` with IonQ, of which `33334 qgs` have already been used. The account also has a limit of `800` HQCs with Quantinuum, of which `0` have been used.

The `Scope` column indicates whether the quota refers to the current workspace or to the subscription.

- `Workspace`: Quota is tracked for an individual workspace.
- `Subscription`: Quota is tracked together for all workspaces within the same subscription/region.

The `Period` column indicates the period when your quota is renewed.

- `Monthly`: The usage is reset on the 1st of every month.
- `Infinite`: The usage is never reset (also referred as **One-Time** in the Azure portal view).

Track quotas with the QDK Python library

1. Install the latest version of the qdk Python library.
2. Open a new Python file.
3. Create a `Workspace` object to connect to the workspace that you previously deployed in Azure.

   Python

   ```python
   from qdk.azure import Workspace

   # Add your resource_id
   workspace = Workspace (resource_id = "")
   ```

4. Use the `get_quotas` method to display the quotas information for the selected workspace.

   Python

   ```python
   quotas = workspace.get_quotas()

   for quota in quotas:
       print(quota)
   ```

How do I request a quota increase?

To request a quota increase, submit a support ticket.

Create a support request

1. Sign in to the Azure portal with the credentials for your Azure subscription.
2. Go to your Azure Quantum workspace.
3. In the workspace navigation pane, under the **Operations** dropdown, go to the **Quotas** blade.
4. Choose the **Increase** button on the quota page.

   Or, under the **Help** dropdown, got to the **Support + Troubleshooting** blade.
5. In the issue description box, enter **Azure Quantum quota override request**.
6. For the service issue, choose **None of the above**.
7. In the **Select a service** menu, choose **Service and subscription limits (quotas)**.

8. Choose the **Next** button.
9. Choose the **Create a support request** button. A **New support request** ticket opens on the **1. Problem description** tab.

Fill out the support request

To fill out the support request, follow these steps:

1. For **Issue type**, choose **Service and subscription limits (quotas)**.
2. For **Subscription**, choose the subscription that contains the workspace that you want a quota increase for.
3. For **Quota type**, choose **Azure Quantum**.
4. Choose the **Next** button to go to the **3. Additional details** tab.
5. In the **Description** box of the **Problem details** section, enter the following information:
    - The name of the provider that you want to increase quotas for
    - Whether you want to increase quotas at the subscription level or workspace level
    - The quotas that you want to increase, and what you want the increased quota to be
    - Reasons why you want to increase the quotas
6. Choose your preferred options for **Advanced diagnostic information** and **Support method** sections.
7. Fill out the **Contact info** section.
8. Choose the **Next** button to go to the **4. Review + create** tab.
9. If all the information is correct, then choose the **Create** button.

> ⓘ **Note**
>
> To submit a support ticket for a quota increase in a workspace, you must be the owner of the workspace.

Related content

- FAQ: Understanding Job Costs and Billing in Azure Quantum

Last updated on 04/13/2026

FAQ: Understanding Job Costs and Billing in Azure Quantum

This article explains the guidelines to understand the cost of running quantum programs in Azure Quantum and how to manage your invoices.

How many usage plans are available in Azure Quantum?

Azure Quantum provides services from Microsoft and from our partner companies, so billing details depend on the provider and the pricing plan that you select.

Most providers have a pay-as-you-go billing plan, which is based on the resources that you consume when you run a job. Some providers also offer subscription plans. For more information about how each provider charges, see Azure Quantum pricing.

How do I modify the usage plan for my Azure Quantum workspace?

To modify your current usage plans for your providers and see the different billing plans available in your local currency, follow these steps:

1. Sign in to the Azure portal with the credentials for your Azure subscription.
2. Go to your Azure Quantum workspace.
3. In the workspace navigation pane, expand the **Operations** dropdown and choose **Providers**.
4. View the current usage plan for each provider in the **Plan** column.
5. Choose **Modify** to see and modify the different billing plans that are available for that provider in your local currency.

The billing plans are set by the quantum hardware providers. For more information about each provider's pricing, see Azure Quantum pricing.

How do I view the job cost report after I run a job?

After you run a job, you can check the detailed cost estimates and use this information to understand the cost of individual jobs. This cost is the amount that the provider bills. Refer to

your final bill for the exact charges including relevant taxes. For more information, see How do I receive my invoices?

To review job costs, follow these steps:

1. Sign in to the Azure portal, using the credentials for your Azure subscription.
2. Go to your Azure Quantum workspace.
3. In the workspace navigation pane, expand the **Operations** dropdown and choose **Job Management**.
4. View the estimated costs for each job in the **Cost estimate** column.
5. To open the **Job details** pane for a job, choose the job in the **Name** column.
6. To view detailed cost estimate information, choose the **Cost Estimation** tab. The table displays the billing dimensions and associated costs for the job run.

The cost estimation table contains the following columns:

- **Dimension**: The billing item that the provider charges you for.
- **Unit Price**: The cost per one unit of the dimension.
- **Consumed Units**: The number of units of the dimension that the job consumed.
- **Billed Units**: The number of units that you're billed for. This number might be less than the consumed units when, for example, the provider's billing plan offers an amount of free usage or is credits-based.
- **Estimated Cost**: The estimated cost for this dimension, calculated as $\text{Billed Units} \times \text{Unit Price}$.

> **Note**
>
> IonQ has a USD 1.00 minimum cost to run a job on the IonQ QPU, so the `Consumed Units` reported on the job cost estimation table might be less than the `Billed Units` for small jobs.

Why don't I see the cost of my job?

Some Azure Quantum providers don't support reports for per-job costs, but your invoice is still available in the **Cost Management** page in the Azure portal. For more information, see How can I view my invoices?

How do I receive my invoices?

Invoices are sent to the email address that you used to sign up for Azure. If you need to change the email address, then contact Azure Support.

If you're using a custom subscription, then invoices might be sent to the email address of the person who set up the subscription.

The invoices from Azure Quantum third party providers have the **Azure Marketplace and Reservations** billing type. You receive a single invoice from Azure Marketplace and Reservations, which provides a breakdown of the charges from each quantum provider.

What does my Azure Quantum invoice include?

The Azure Marketplace and Reservations invoices include spending against Azure 3rd party providers. Azure Quantum currently offers quantum hardware from IonQ, Quantinuum, and Rigetti. For more information, see Azure Quantum provider list.

The Azure Marketplace and Reservations invoices don't include spending against first party Azure services, such as storage accounts that are attached to the Azure Quantum workspace. These first party Azure charges are typically very small amounts for QPU jobs.

For more information about Azure infrastructure costs, see Azure Blob Storage pricing.

How are invoices billed?

The invoices are billed in arrears. For example, the invoice you receive in February corresponds to the usage you incurred in January.

You pay your invoices with the same payment method that you use for the Azure subscription of your Azure Quantum workspace. You might have different payment methods for different subscriptions.

How can I view my invoices?

To view your past invoices in the Azure portal, follow these steps:

1. Sign in to the Azure portal with the credentials for your Azure subscription.
2. In the top search bar, enter **Cost Management + Billing**.
3. If you have more than one billing scope, then select the scope that you want to view in the **Billing scopes** pane.
4. In the navigation pane, choose **Cost Management**.
5. In the navigation pane, expand the **Billing** dropdown and choose **Invoices**.

6. To view invoice details from Azure Quantum third party providers, choose the **Invoice ID** in rows that have **Azure Marketplace and Reservations** in the **Type** column.

You can filter invoices by date, subscription, and status. You can also download the invoices as PDF files. You can see the invoice date and the billing period. The billing period isn't the same as the month that you receive the invoice in.

Related content

- Azure Quantum quotas

Last updated on 04/13/2026

Understand Azure Quantum workspace and storage security boundaries

An Azure Quantum workspace stores job inputs and outputs in an associated Azure Storage account that's either managed or customer provided. Either way, treat the storage account as part of the workspace security boundary.

> **Important**
>
> Anyone with permission to access workspace data can access all jobs in the workspace and all data in its associated storage account. Azure Quantum doesn't support access control lists for individual jobs.

Understand the effective security boundary

Azure Quantum uses the workspace's system-assigned identity to read and write data in the associated storage account. Users and apps with the required workspace data permissions can access that data through the workspace.

Azure roles can grant resource management permissions, workspace data permissions, or both. A role that lets a user or app view or manage the workspace resource doesn't always grant access to job data. Review the role definition before you assign it.

For more information, see Understanding role-based access to your Azure Quantum workspace.

A principal with workspace data access can:

- View all jobs in the workspace.
- Access job inputs and outputs.
- Access the contents of the associated storage account through the workspace authorization model.

Direct Azure Storage authorization is a separate access path. Storage role assignments, account keys, and shared access signatures can grant more access outside the Azure Quantum workspace.

Compare storage options

Azure Quantum supports two storage options, managed storage and customer-provided storage.

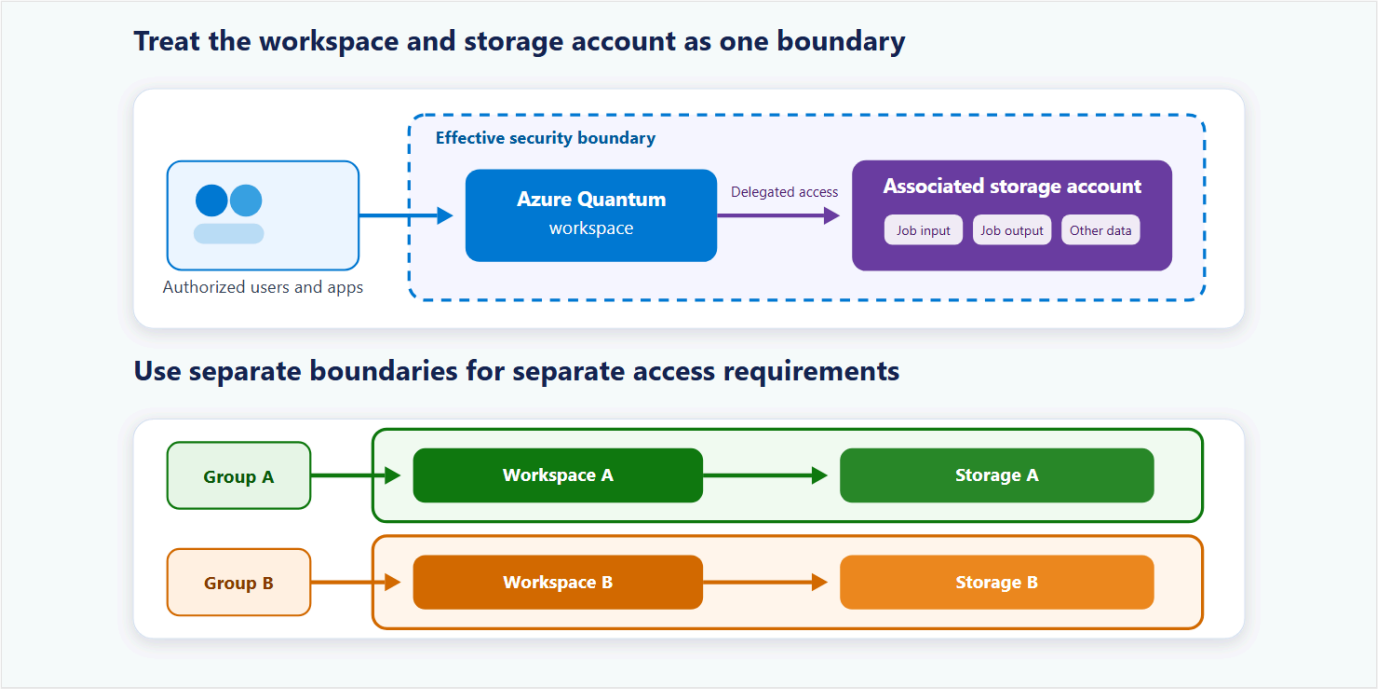
Expand table

Consideration	Managed storage account	Customer-provided storage account
Creation	Azure Quantum creates the account when you create the workspace.	You select an existing account when you create the workspace.
Location	Azure creates the account in a managed resource group in your subscription.	The account remains in the resource group that you manage.
Intended use	By default, Azure Quantum dedicates the account to one workspace.	The account can have other data or support more than one workspace.
Direct access	Azure restricts your direct use of the account.	You control direct use through Azure Storage authorization.
Workspace deletion	Azure deletes the account when you delete the workspace.	Azure doesn't delete the account when you delete the workspace.
Recommended use	Use this option for the simplest dedicated boundary.	Use a dedicated account that doesn't have data with different access requirements.

The workspace security boundary applies to both options. A customer-provided account doesn't separate workspace users from data that's already in that account.

Create separate security boundaries

Use separate workspaces and storage accounts when users, teams, applications, or data require different access permissions.



For example:

- Grant Group A access to Workspace A, and associate Workspace A with Storage A.
- Grant Group B access to Workspace B, and associate Workspace B with Storage B.
- Don't grant either group access to the other group's workspace or storage account.

Don't share a storage account between Workspace A and Workspace B if the workspaces have different authorized users. A user who can access either workspace's data can access all data in the shared account.

Don't use logical separation as a security boundary

Azure Quantum controls access to the whole workspace, not individual jobs. Folders, blob paths, job IDs, and similar naming conventions help organize data, but they don't restrict access through Azure Quantum.

Don't depend on logical separation within one storage account when the data requires different authorization. If your organization needs to share an account, set up an independent Azure Storage authorization model for the required resources. Validate the complete access path for every user, application, and workspace identity before you treat that configuration as a security boundary.

Plan the resource lifecycle

Managed and customer-provided storage accounts have different lifecycle behavior:

- When you delete a workspace that uses managed storage, Azure also deletes the managed storage account and its data.
- When you delete a workspace that uses customer-provided storage, the storage account and its data remain.
- When you link a customer-provided storage account to multiple workspaces, each workspace can give access to the account's data.

Before you delete or migrate a workspace, review your data retention requirements and the access assignments on the linked storage account. To prevent accidental deletion, see Protect Azure Quantum resources with ARM locks.

Related content

- Create an Azure Quantum workspace
- Understand workspace access
- Share access to your Azure Quantum workspace
- Work with Azure Quantum jobs

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/23/2026

Authenticate in your workspace using a service principal

Article • 02/26/2025

Sometimes it's unsuitable to use interactive authentication or to authenticate as a user account. These cases may arise when you want to submit jobs from a web service, another worker role, or an automated system. One option is to configure a managed identity, another option is to use a Service Principal, which this article will explain.

Prerequisite: Create a service principal and application secret

To authenticate as a service principal, you must first create a service principal.

To create a service principal, assign access, and generate a credential:

1. Create an Azure AD application:

   > ⓘ **Note**
   >
   > You don't need to set a redirect URI.

   a. Once created, write down the *Application (client) ID* and the *Directory (tenant) ID*.

2. Create a credential to sign in as the application:
   a. In the settings for your application, select **Certificates & secrets**.
   b. Under **Client Secrets**, select **Create New Secret**.
   c. Provide a description and duration, then select **Add**.
   d. Copy the value of the secret to a safe place immediately - you won't be able to see it again!

3. Give your service principal permissions to access your workspace:
   a. Open the Azure portal.
   b. In the search bar, enter the name of the resource group you created your workspace in. Select the resource group when it comes up in the results.
   c. On the resource group overview, select **Access control (IAM)**.
   d. Select **Add Role Assignment**.
   e. Search for and select the service principal.
   f. Assign either the **Contributor** or **Owner** role.

> ⓘ Note
>
> In order to create a role assignment on the resource group or workspace, you need to be an *owner* or *user access administrator* at the scope of the role assignment. If you don't have permissions to create the Service Principal in your subscription, you'll need to request permission from the *owner* or *administrator* of the Azure subscription.
>
> If you have permissions only at the resource group or workspace level, you can to create the service principal under the Contributor role using:
>
> `az ad sp create-for-rbac --role Contributor --scopes /subscriptions/<SUBSCRIPTION-ID>`

Authenticate as the service principal

Option 1: Using environment variables: The default credential used in the `Workspace` object creation is the DefaultAzureCredential, which will attempt several types of authentication. The first one is the EnvironmentCredential, and with that you pass the Service Principal credentials via the following environment variables:

- **AZURE_TENANT_ID**: ID of the service principal's tenant. Also called its 'directory' ID.
- **AZURE_CLIENT_ID**: the service principal's client ID.
- **AZURE_CLIENT_SECRET**: one of the service principal's client secrets.

Option 2: Using the ClientSecretCredential: Pass a ClientSecretCredential during the instantiation of the `Workspace` object or set the `credentials` property.

Python

```
from azure.identity import ClientSecretCredential

tenant_id = os.environ["AZURE_TENANT_ID"]
client_id = os.environ["AZURE_CLIENT_ID"]
client_secret = os.environ["AZURE_CLIENT_SECRET"]
credential = ClientSecretCredential(tenant_id=tenant_id,
client_id=client_id, client_secret=client_secret)

workspace.credentials = credential
```

> ⓘ Note

The `workspace.login()` method has been deprecated and is no longer necessary. The first time there's a call to the service, an authentication will be attempted using the credentials passed in the `Workspace` constructor or its `credentials` property. If no credentials were passed, several authentication methods will be attempted by the DefaultAzureCredential.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

How to use a managed identity to authenticate in your Azure Quantum workspace

Article • 06/18/2024

There are several scenarios where it is unsuitable to use interactive authentication or to authenticate as a user account. For example, you may want to submit jobs from a virtual machine (VM), or from a Function App. One option is to authenticate using a service principal, another option is to configure a managed identity, which this article will explain.

Configure a managed identity

A managed identity allows an application to access other Azure resources (such as your Azure Quantum workspace) and authenticate with these resources.

To configure a managed identity:

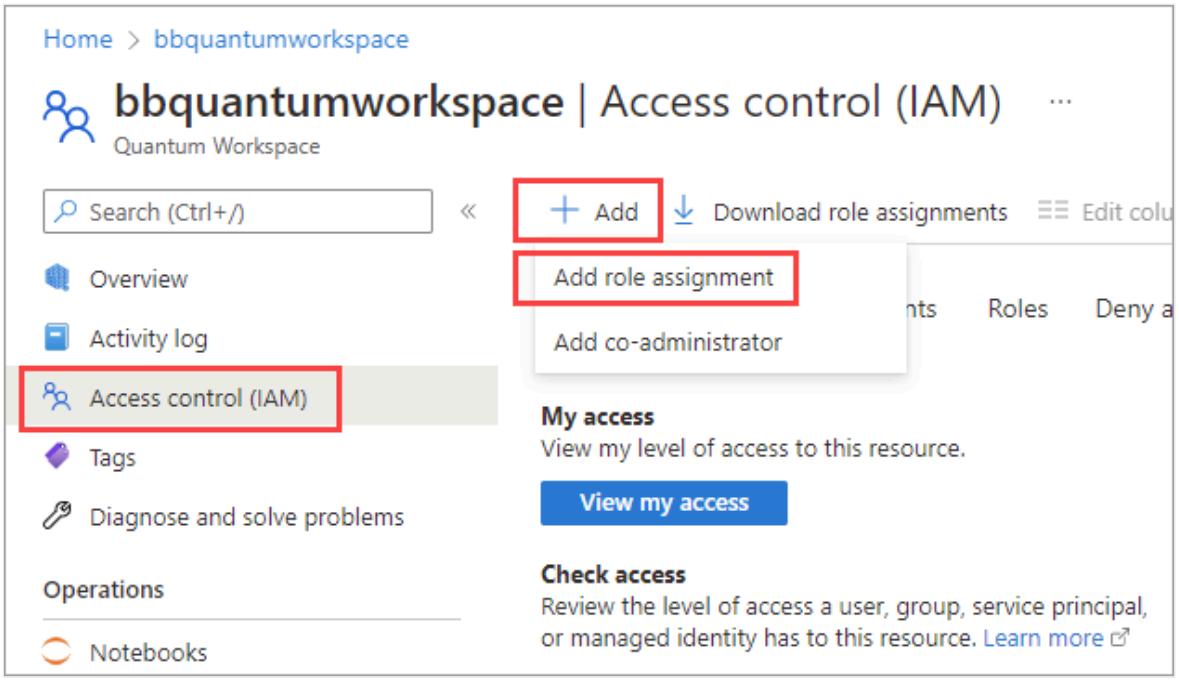
1. Via the Azure portal, locate the resource that you wish to give access to. This resource may be a VM, a Function App, or other application.
2. Select the resource, and view the overview page.
3. Under **Settings**, select **Identity**.
4. Configure the **Status** setting to **On**.
5. Select **Save** to persist your configuration and confirm the opening dialog with **Yes**.

Grant access to your Azure Quantum workspace

In order to allow the resource to access your Azure Quantum workspace:

1. Navigate to your Azure Quantum workspace and select **Access control (IAM)** from the left-side menu.

2. Select **Add** and **Add role assignment**.

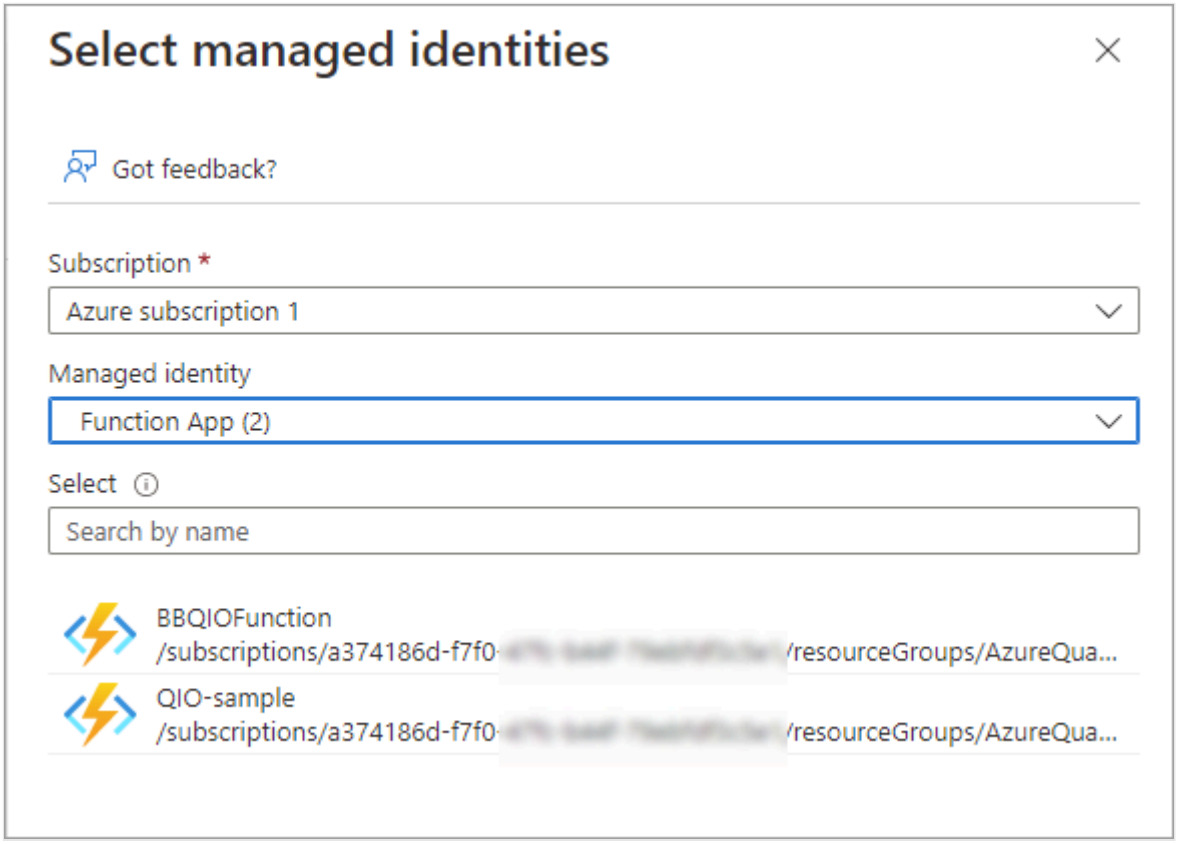


3. On the **Add role assignment** page, select **Contributor** and select **Next**.

4. On the **Members** tab, in **Assign access to**, select **Managed Identity**, and then select **+ Select members**.

5. In the **Select managed identities** popup, select a category from the **Managed identity** dropdown.

6. Select the desired resource from the list and click **Select**.



7. Select **Next** and then select **Review and assign**.

Logging in to your Azure Quantum workspace

You should now be able to use your quantum workspace from your chosen resource. For example when using your workspace from within a VM, you will no longer need to authenticate each time.

In some cases, you may also want to specify explicitly within the code to use a Managed Identity Credential:

Python

```
from azure.identity import ManagedIdentityCredential

from azure.quantum import Workspace
workspace = Workspace (
    resource_id = "",
    location = "" ,
    credential=ManagedIdentityCredential()
)
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Authenticate in your workspace using an access key

Access keys are used to authenticate and authorize access to your Azure Quantum workspace. You can use access keys to connect and grant access to your workspace using connection strings.

In this article, you learn how to enable or disable the access keys for your Azure Quantum workspace. You can also regenerate new keys to ensure the security of your workspace.

> ⚠ **Warning**
>
> Storing your account access keys or connection string in clear text presents a security risk and is not recommended. Store your account keys in an encrypted format, or migrate your applications to use Microsoft Entra authorization for access to your Azure Quantum workspace.

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. See Create an Azure Quantum workspace.
- The latest version of the `qdk` Python library with the `azure` extra.

  Bash

  ```
  pip install --upgrade "qdk[azure]"
  ```

If you use Azure CLI, then you must have the latest version. For installation instructions, see:

- Install Azure CLI on Windows
- Install Azure CLI on Linux
- Install Azure CLI on macOS

Connect to your Azure Quantum workspace with a connection string

The `qdk.azure` module provides a Workspace class that represents an Azure Quantum workspace. To connect to your Azure Quantum workspace, create `Workspace` object using the

connection string as the authenticator. For more information, see how to copy a connection string.

When you create a `Workspace` object, you have two options to identify your Azure Quantum workspace.

- Create a `Workspace` object by calling `from_connection_string`.

  Python

  ```python
  # Creating a new Workspace object from a connection string
  from qdk.azure import Workspace

  connection_string = "[Copy connection string]"
  workspace = Workspace.from_connection_string(connection_string)

  print(workspace.get_targets())
  ```

- If you don't want to copy your connection string in the code, then store your connection string in an environment variable and use `Workspace`.

  Python

  ```python
  # Using environment variable to connect with  connection string

  connection_string = "[Copy connection string]"

  import os

  os.environ["AZURE_QUANTUM_CONNECTION_STRING"] = connection_string

  from qdk.azure import Workspace

  workspace = Workspace()
  print(workspace.get_targets())
  ```

Manage your access keys and connection strings

> **Tip**
>
> Every Azure Quantum workspace has **primary and secondary** keys, and their corresponding connection strings. If you want to allow access to your workspace to others, you can share your secondary key and you use your primary for your own services. This way, you can replace the secondary key as needed without having downtime in your

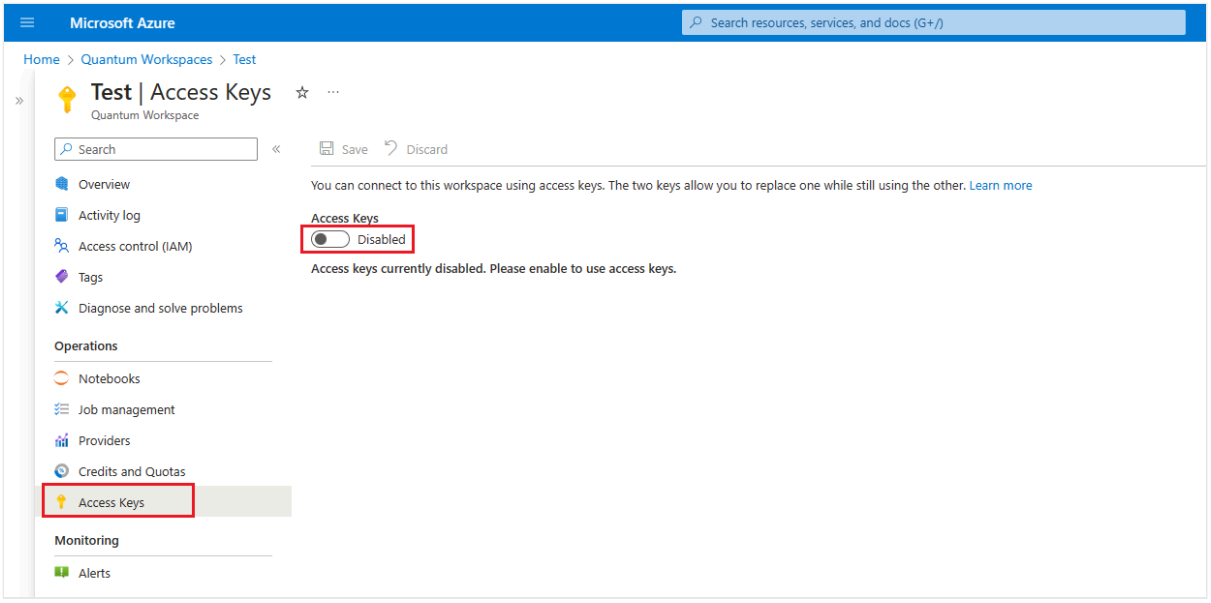
own services. For more information about sharing your workspace access, see Share your workspace access.

Azure portal

You can manage the access keys and connection strings for your Azure Quantum workspace in the Azure portal.

Enable and disable access keys

1. Log in to the Azure portal and select your Azure Quantum workspace.
2. On the left panel, navigate to **Operations** > **Access keys**.
3. Switch the toggle under Access Keys to **Enabled** or **Disabled**.
4. Click on **Save** to save the changes.



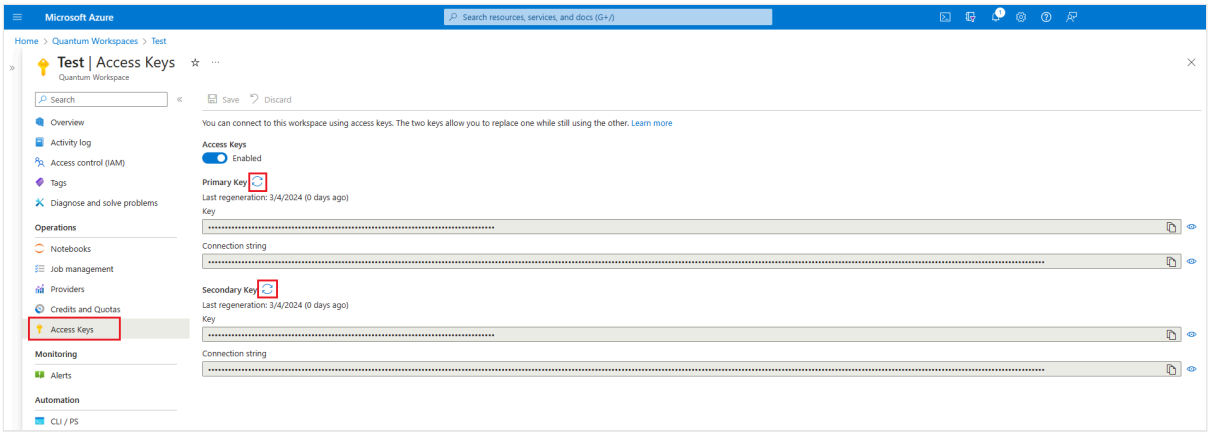
> **Important**
>
> When Access Keys are **disabled**, all request using connection strings or access keys are unauthorized. You can still use the workspace parameters to connect to your workspace.

Regenerate new access keys

If you suspect that your Access Keys have been compromised, or you want to stop sharing your workspace access with others, you can regenerate either the primary or secondary

access keys, or both, to ensure the security of your workspace.

1. Log in to the Azure portal and select your Azure Quantum workspace.
2. On the left panel, navigate to **Operations** > **Access keys**.
3. **Access Keys** have to be enabled to regenerate new keys. If Access Keys are disabled, you need to enable them first.
4. Click on the **circular arrow icon** to regenerate the primary or secondary key.



Last updated on 12/24/2025

Protect Azure Quantum resources with Azure Resource Manager (ARM) locks

Microsoft recommends locking all of your Azure Quantum workspaces and linked storage accounts with an Azure Resource Manager (ARM) resource lock to prevent accidental or malicious deletion. For example, professors might want to restrict students from modifying provider SKUs, but still allow them to submit jobs.

There are two types of ARM resource locks:

- A **CannotDelete** lock prevents users from deleting a resource, but permits reading and modifying its configuration.
- A **ReadOnly** lock prevents users from modifying a resource's configuration (including deleting it), but permits reading its configuration. For more information about resource locks, see Lock resources to prevent unexpected changes.

> **Note**
>
> If you already use an ARM or Bicep template to manage your Azure Quantum workspaces, you can add the procedures in this article to your existing templates.

Recommended lock configurations

The following table shows the recommended resource lock configurations to deploy for an Azure Quantum workspace.

Expand table

Resource	Lock type	Notes
Workspace	Delete	Prevents the workspace from being deleted.
Workspace	Read-only	Prevents any modifications to the workspace, including additions or deletions of providers, while still allowing users to submit jobs. To modify providers when this lock is set, you need to remove the resource lock, make your changes, then redeploy the lock.
Storage account	Delete	Prevents the storage account from being deleted.

The following configurations should be avoided:

> **Important**
>
> Setting the following ARM locks may cause your workspace to function incorrectly.

Expand table

Resource	Lock type	Notes
Storage account	Read-only	Setting a Read-only resource lock on the storage account can cause failures with workspace creation, job submission, and fetching jobs.
Parent subscription of the workspace or the parent resource group of the workspace or storage account	Read-only	When a resource lock is applied to a parent resource, all resources under that parent inherit the same lock, including resources created at a later date. For more granular control, resource locks should be applied directly at the resource level.

Prerequisites

You must be an **Owner** or **User Access Administrator** of a resource to apply ARM resource locks. For more information, see Azure built-in roles.

Command-line deployment

You will need either Azure PowerShell or Azure CLI to deploy the lock. If you use Azure CLI, you must have the latest version. For the installation instructions, see:

- Install Azure PowerShell
- Install Azure CLI

> **Important**
>
> If you haven't used Azure CLI with Azure Quantum before, follow the steps in the Environment setup section to add the `quantum` extension and register the Azure Quantum namespace.

Sign in to Azure

After installing either Azure CLI or Azure PowerShell, make sure you sign in for the first time. Choose one of the following tabs and run the corresponding command line commands to sign in to Azure:

Azure CLI

Azure CLI

```
az login
```

If you have multiple Azure subscriptions, select the subscription with the resources that you want to lock. Replace `SubscriptionName` with your subscription name or subscription ID. For example,

Azure CLI

```
az account set --subscription "Azure subscription 1"
```

Create an ARM resource lock

When you deploy a resource lock, you specify a name for the lock, the type of lock, and additional information about the resource. This information can be copied and pasted from the resource's home page in the Azure Quantum portal.

Azure CLI

Azure CLI

```
az lock create \
    --name <lock> \
    --resource-group <resource-group> \
    --resource <workspace> \
    --lock-type CanNotDelete \
    --resource-type Microsoft.Quantum/workspaces
```

- name: A descriptive name for the lock
- resource-group: The name of the parent resource group.
- resource: The name of the resource to apply the lock to.
- lock-type: The type of lock to apply, either **CanNotDelete** or **ReadOnly**.
- resource-type: The type of the target resource.

For example, to create a **CanNotDelete** lock on a workspace:

Azure CLI

```
az lock create \
    --name ArmLockWkspDelete \
    --resource-group armlocks-resgrp \
    --resource armlocks-wksp \
    --lock-type CanNotDelete \
    --resource-type Microsoft.Quantum/workspaces
```

If successful, Azure returns the lock configuration in JSON format:

JSON

```
{
  "id": "/subscriptions/<ID>/resourcegroups/armlocks-
resgrp/providers/Microsoft.Quantum/workspaces/armlocks-
wksp/providers/Microsoft.Authorization/locks/ArmLockWkspDelete",
  "level": "CanNotDelete",
  "name": "ArmLockWkspDelete",
  "notes": null,
  "owners": null,
  "resourceGroup": "armlocks-resgrp",
  "type": "Microsoft.Authorization/locks"
}
```

To create a **ReadOnly** lock on a workspace:

Azure CLI

```
az lock create \
    --name ArmLockWkspRead \
    --resource-group armlocks-resgrp \
    --resource armlocks-wksp \
    --lock-type ReadOnly \
    --resource-type Microsoft.Quantum/workspaces
```

JSON

```
{
  "id": "/subscriptions/<ID>/resourcegroups/armlocks-
resgrp/providers/Microsoft.Quantum/workspaces/armlocks-
wksp/providers/Microsoft.Authorization/locks/ArmLockWkspRead",
  "level": "ReadOnly",
  "name": "ArmLockWkspRead",
  "notes": null,
  "owners": null,
  "resourceGroup": "armlocks-resgrp",
```

```
  "type": "Microsoft.Authorization/locks"
}
```

To create a **CanNotDelete** lock on a storage account:

Azure CLI

```
az lock create \
    --name ArmLockStoreDelete \
    --resource-group armlocks-resgrp \
    --resource armlocksstorage \--lock-type CanNotDelete \
    --resource-type Microsoft.Storage/storageAccounts
```

JSON

```
{
  "id": "/subscriptions/<ID>/resourcegroups/armlocks-
resgrp/providers/Microsoft.Storage/storageAccounts/armlocksstorage/providers/Mic
rosoft.Authorization/locks/ArmLockStoreDelete",
  "level": "CanNotDelete",
  "name": "ArmLockStoreDelete",
  "notes": null,
  "owners": null,
  "resourceGroup": "armlocks-resgrp",
  "type": "Microsoft.Authorization/locks"
}
```

Viewing and deleting locks

To view or delete locks:

Azure CLI

For more information, see the az lock reference.

View all locks in a subscription

Azure CLI

```
az lock list
```

View all locks in a workspace

Azure CLI

```
az lock list \
    --resource-group armlocks-resgrp \
    --resource-name armlocks-wksp  \
    --resource-type Microsoft.Quantum/workspaces
```

View all locks for all resources in a resource group

Azure CLI

```
az lock list --resource-group armlocks-resgrp
```

View the properties of a single lock

azcli

```
az lock show \
    --name ArmLockStoreDelete \
    --resource-group armlocks-resgrp \
    --resource-name armlocksstorage \
    --resource-type  Microsoft.Storage/storageAccounts
```

Delete a lock

azcli

```
az lock delete \
    --name ArmLockStoreDelete \
    --resource-group armlocks-resgrp \
    --resource-name armlocksstorage \
    --resource-type  Microsoft.Storage/storageAccounts
```

If the deletion is successful, Azure does not return a message. To verify the deletion, you can run `az lock list`.

Next steps

- Manage quantum workspaces with the Azure CLI
- Manage quantum workspaces with the Azure Resource Manager

Last updated on 10/16/2025

Update the Microsoft Quantum Development Kit to the latest version

Summarize this article for me

Learn how to update the Microsoft Quantum Development Kit (QDK) to the latest version.

Prerequisites

- This article assumes that you already have the QDK extension installed in Visual Studio Code (VS Code). If you're installing for the first time, then refer to Install the QDK extension.

Update the VS Code extension

By default, VS Code automatically updates extensions. After an extension is updated, you're prompted to reload VS Code. To disable auto-updates and update your extensions manually, see Extension auto-update in the VS Code documentation.

Update the QDK Python libraries

> **Important**
>
> If you're updating from a previous Qiskit environment, then see **Update the qdk.azure module with Qiskit support in a virtual Python environment (recommended)**.

1. Update to the latest `qdk` Python library with the `azure` extra:

   ```bash
   pip install --upgrade "qdk[azure]"
   ```

2. To add support for the analysis, transformation, code generation, and simulation of Qiskit circuits, install the `qiskit` and `jupyter` extras.

   ```bash
   pip install --upgrade "qdk[qiskit,jupyter]"
   ```

> **Note**
>
> The QDK supports versions 1 and 2 of Qiskit.

Update the `qdk.azure` module with Qiskit support in a virtual Python environment (recommended)

The `qdk.azure` and `qdk.qiskit` Python modules includes optional support to create and submit Qiskit circuits to Azure Quantum. When you install the `qdk.azure` and `qdk.qiskit` modules, the latest version of Qiskit is installed by default, which might cause issues with an existing Qiskit environment. To ensure a stable development environment, create a virtual Python environment and install `qdk.azure` and `qdk.qiskit` in the virtual environment.

> **Note**
>
> If you're using Qiskit version 1, then see Qiskit 1.0 packaging changes for more information on package compatibility.

To create a virtual Python environment and install `qdk.azure` and `qdk.qiskit`, follow these steps:

1. Create a local folder, for example `~/qiskit10-env`.
2. Run `venv` with the path to the folder.

   Bash
   ```
   python -m venv ~/qiskit10-env
   ```

3. Activate the environment.

   Bash
   ```
   ~/qiskit10-env/bin/activate
   ```

4. Run `pip list` to see that only the core packages are installed in the new environment.
5. To install the `qdk.azure` and `qdk.qiskit` modules, run the following command:

   Bash

```
pip install "qdk[azure,qiskit]"
```

6. Install other packages that you used in your previous environment as needed. Run the `pip list` command in each environment to compare packages and versions.

> **Note**
>
> You can also open your virtual environment in VS Code. From the **View** menu, choose **Command Palette**, and then enter **Python: Create Environment** and choose **venv**. Then, choose **Open Folder...** and select the environment folder that you created earlier. For more information on using environments in VS Code, see Python environments in VS Code.

Update the `qdk.azure` and `qdk.qiskit` modules in the current environment

You can also update the `qdk.azure` and `qdk.qiskit` modules without using a virtual environment. However, updates to the `qiskit` packages in an existing environment might cause dependency conflicts with other packages. See Qiskit 1.0 packaging changes for more information on package compatibility.

To update the `qdk.azure` and `qdk.qiskit` modules in your current environment, complete the following steps:

1. Uninstall the existing `qdk` modules and other Qiskit dependencies:

   Bash
   ```
   pip uninstall -y "qdk[azure,qiskit]" qiskit qiskit-terra qiskit-qir
   ```

2. Reinstall the `qdk` library with the `qiskit` extra:

   Bash
   ```
   pip install "qdk[qiskit]"
   ```

Update the Azure CLI `quantum` extension

Update or install the latest Azure CLI `quantum` extension.

1. Open a Windows command prompt.

2. From the command prompt, run the following command:

Azure CLI

```
az extension add --upgrade -n quantum
```

Last updated on 02/07/2026

Add or remove a provider in an existing workspace

When you create an Azure Quantum workspace, you choose the providers and plans that are available in the workspace. However, you can add new providers or remove existing providers at any time. If you submit a quantum job to a provider that isn't in your workspace, then you receive an error message that prompts you to add that provider to your workspace.

Add a provider or plan

To add a provider to your workspace, follow these steps:

1. Sign in to the Azure portal with the credentials for your Azure subscription.
2. Go to your Azure Quantum workspace.
3. In the workspace navigation pane, expand the **Operations** dropdown and choose **Providers**.
4. Above the list of your existing providers, choose **+ Add a Provider**. The **Add additional providers** pane opens and shows all the providers that you can add to your workspace.

   > **Note**
   >
   > If the provider that you want to add isn't in the list of available providers, then your billing account might be located in a country or region that the provider doesn't support. For more information, see Global availability of Azure Quantum providers.

5. Choose **+ Add** next to the provider that you want to add, and then choose a pricing plan.
6. Select the **I accept this provider's terms and conditions** box, and then choose the **Add** button to begin deployment.

When the deployment finishes, the provider is the list on the **Providers** page. If the deployment succeeds, then you see a green checkmark for the provider **Status**. If the deployment fails, then review the notification or check the **Activity log**.

Remove a provider or plan

To remove a provider from your workspace, follow these steps:

1. Sign in to the Azure portal with the credentials for your Azure subscription.
2. Go to your Azure Quantum workspace.
3. In the workspace navigation pane, expand the **Operations** dropdown and choose **Providers**.
4. For the provider that you want to remove, choose **Remove** in the last column of the providers list.
5. Choose the **Yes** button to remove the provider.

Last updated on 04/13/2026

Get started with DevOps architecture design

DevOps integrates development, quality assurance (QA), and IT operations into a unified culture and set of processes for delivering software. By adopting DevOps practices, teams can deliver higher-quality solutions faster and respond to changing business requirements with greater agility.

Azure services for DevOps

Azure provides a range of operations and services for DevOps.

DevOps operations

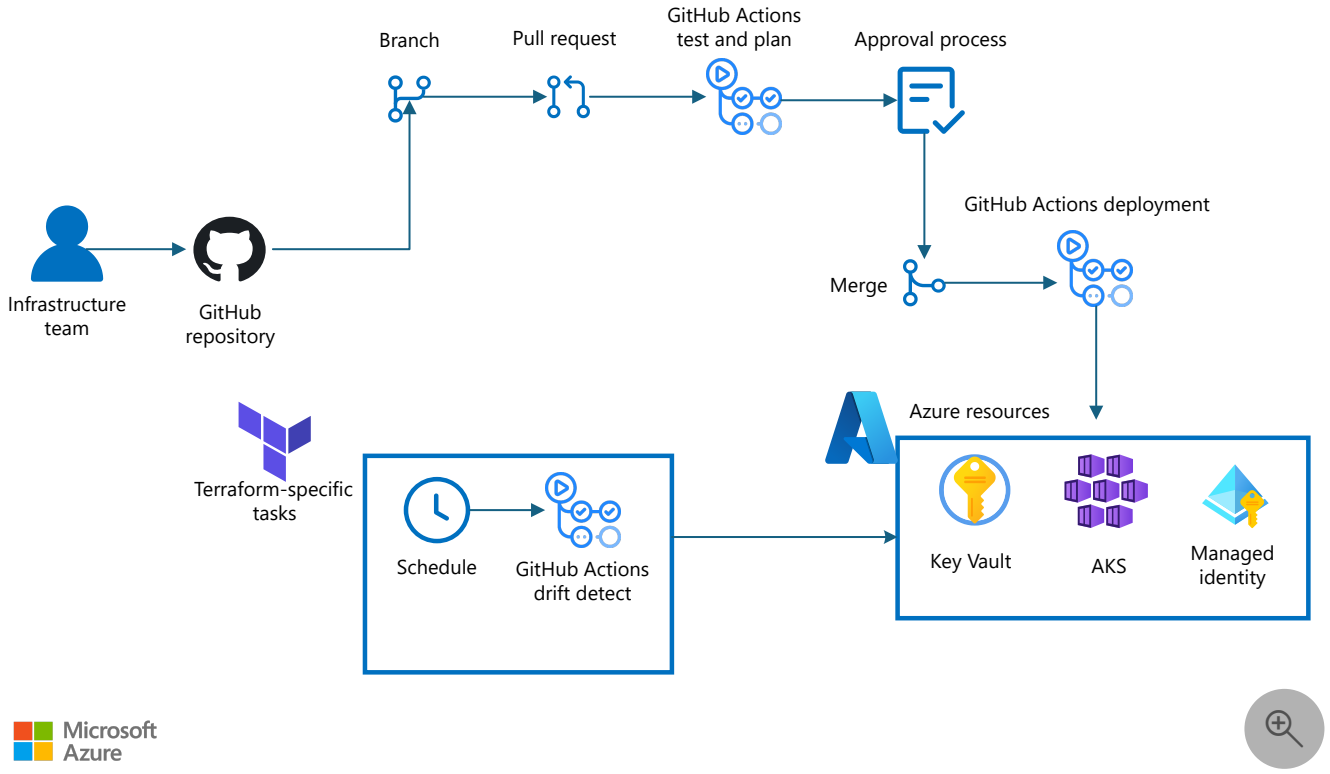
- Continuous integration (CI): The practice of frequently merging all developer code into a central codebase and performing automated build and test processes. The objectives are to quickly discover and correct code problems, streamline deployment, and ensure code quality.
- Continuous delivery (CD): The practice of automatically building, testing, and deploying code to production-like environments. The objective is to ensure that code is always ready to deploy. Adding CD to create a full CI/CD pipeline helps you detect code defects as soon as possible. It also ensures that properly tested updates can be released in a short time.
- Continuous deployment: A process that automatically takes any updates that pass through the CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning. It might not be appropriate for all teams.
- Continuous monitoring: The process and technology required to incorporate monitoring across each phase of DevOps and IT operations lifecycles. Monitoring helps ensure the health, performance, and reliability of your application and infrastructure as the application moves from development to production. Continuous monitoring builds on the concepts of CI and CD.

DevOps services

- Azure DevOps: Plan, develop, and deliver software faster with integrated DevOps services, including version control, CI/CD pipelines, and package management.

- Azure Pipelines: Automate build, test, and deployment processes with cloud-hosted CI/CD pipelines for any platform or language.
- GitHub Actions: Automate software workflows directly from your GitHub repository with native CI/CD, testing, and deployment capabilities.
- Azure Monitor: Monitor, analyze, and optimize application and infrastructure performance with comprehensive observability and diagnostics.

Architecture



Download a Visio file of this architecture.

The previous diagram demonstrates a typical basic or baseline DevOps implementation. For real-world solutions that you can build in Azure, see DevOps architectures.

Explore DevOps guides, architectures, and solution ideas

The articles in this section include guides and fully developed architectures that you can deploy in Azure and expand to production-grade solutions. Solution ideas demonstrate implementation patterns and possibilities to consider as you plan your DevOps proof-of-concept (POC) development. These articles can help you decide how to use DevOps technologies in Azure.

DevOps solution ideas

The following DevOps solution ideas demonstrate implementation patterns and possibilities to explore:

- DevSecOps for infrastructure as code (IaC): Implement a DevSecOps pipeline by using GitHub for IaC with governance for operational excellence, security, and cost optimization.

DevOps architectures

The following production-ready architectures demonstrate end-to-end DevOps solutions that you can deploy and customize.

- CI/CD baseline architecture with Azure Pipelines: Build a CI/CD pipeline by using Azure Pipelines to deploy application changes to staging and production environments.
- Automate API deployments with APIOps: Apply GitOps and DevOps techniques to API deployment by using Azure API Management and Azure DevOps.
- Scalable cloud applications and site reliability engineering (SRE): Learn how to model application performance and apply SRE practices to scale cloud applications reliably.
- Manage Microsoft 365 tenant configuration by using Microsoft365DSC and Azure DevOps: Track and automate changes to Microsoft 365 tenant configurations by using Azure DevOps and PowerShell DSC.

DevOps guides

- Azure Sandbox: Learn how to use Azure Sandbox to provide isolated, self-service Azure environments for development and testing.
- DevSecOps on Azure Kubernetes Service (AKS): Embed security controls and best practices across DevOps lifecycle stages for Azure Kubernetes Service (AKS)-hosted workloads.
- Use deployment scripts to check resource properties: Learn how to use deployment scripts to validate Azure resource properties as part of your DevOps pipeline.

Organizational readiness

Organizations at the beginning of the cloud adoption process can use the Cloud Adoption Framework for Azure to access proven guidance that accelerates cloud adoption.

- DevOps considerations for Azure landing zones: Define your DevOps framework, establish metrics, plan your implementation journey, and select your DevOps toolchain.

To help ensure the quality of your DevOps solution on Azure, follow the guidance in the Azure Well-Architected Framework. The Well-Architected Framework provides prescriptive guidance for organizations that seek architectural excellence and describes how to design, provision, and monitor cost-optimized Azure solutions.

Best practices

Follow these best practices to improve the security, reliability, performance, and operational quality of your DevOps workloads on Azure.

DevOps

- Make your Azure DevOps secure: Comprehensive security guidance for Azure DevOps, including network protection, Zero Trust implementation, access control, and service-level security measures.
- Shift left to make testing fast and reliable: Guidance for shifting testing earlier in the development cycle, including test pyramids, fast feedback loops, and reliable test practices.
- Shift right to test in production: Guidance for production testing practices, including canary releases, feature flags, and fault injection.
- Migrate from Team Foundation Version Control (TFVC) to Git: Guidance for migrating source control from Team Foundation Version Control to Git repositories in Azure DevOps.
- Reliability guides by service: Service-specific guidance for building reliable solutions on Azure.
- Monitor your Azure cloud estate: Plan, configure, and optimize monitoring across Azure environments by using Azure Monitor, Microsoft Defender for Cloud, Microsoft Sentinel, and other tools.
- Reliability best practices in Azure Monitor: Guidance for monitoring application reliability by using Azure Monitor.
- Overview of the Microsoft cloud security benchmark: Recommendations for securing cloud solutions on Azure, aligned with industry frameworks such as NIST and PCI.

- DevOps security: Security controls for DevOps processes, including static application security testing, vulnerability management, threat modeling, and software supply chain security.
- Azure identity management and access control security best practices: Best practices for implementing identity management and access control by using Microsoft Entra ID.
- Azure security best practices and patterns: A collection of security best practices for designing, deploying, and managing cloud solutions on Azure.
- Azure operational security checklist: A checklist that covers operational security practices for Azure deployments.
- Secure development best practices on Azure: Best practices for building secure applications on Azure, including the security development lifecycle.

Azure Artifacts

- Best practices for Azure Artifacts: Guidelines for creating, publishing, and consuming packages in Azure Artifacts, including feed management and retention policies.

Azure Resource Manager

- Azure Resource Manager template (ARM template) best practices: Recommended practices for constructing Azure Resource Manager templates (ARM templates), including parameter design, resource dependencies, and API versioning.
- Best practices for Bicep: Recommendations for developing Bicep files, including naming conventions, resource definitions, and output security.

Stay current with DevOps

Azure DevOps services evolve to address modern development and operations challenges. Stay informed about the latest updates and features.

To stay current with key DevOps services, see the following articles:

- Azure DevOps roadmap
- Azure DevOps documentation - what's new?
- What's new in Azure Monitor documentation

Other resources

DevOps is a broad category and covers a range of solutions. The following resources can help you discover more about Azure services that support DevOps implementations.

Microsoft Fabric CI/CD

In Microsoft Fabric, you achieve CI/CD by connecting your Fabric workspace to a Git repository (Azure DevOps or GitHub) for version control and branch-based workflows. You manage continuous deployment by using deployment pipelines.

- Microsoft Fabric Git integration: Connect your Fabric workspace to Azure DevOps or GitHub for version control and CI.
- Introduction to deployment pipelines: Promote content across development, test, and production environments by using deployment pipelines.

Amazon Web Services (AWS) or Google Cloud professionals

To help you get started quickly, the following articles compare Azure DevOps options to other cloud services and provide migration guidance.

Service comparison

- Azure for AWS professionals - DevOps and application monitoring
- Google Cloud to Azure services comparison - DevOps and application monitoring

Migration guidance

If you're migrating from another cloud platform, see the following article:

- Migrate a workload from Amazon Web Services (AWS) to Azure

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 08/26/2026

Quantum computing integration with classical apps

Azure Quantum

Some computational problems are impractical or intractable to solve on classical computers, even on large supercomputers. For some of these problems, a quantum algorithm can reach a solution using far fewer resources than the best known classical approach. A quantum computer uses quantum-mechanical effects, such as superposition and entanglement, to represent and process information in ways that a classical computer can't.

Quantum programs run on quantum providers that you reach by submitting jobs. Quantum targets expose different target profiles. Some profiles allow only quantum operations, and the classical logic you can run on them is limited. Other profiles allow both quantum and classical operations to run together on the provider.

Regardless of the target profile, classical compute components handle the surrounding application integration. Even when a quantum target accepts classical operations, you reach the target by submitting a job and waiting for the results. This article describes and compares two orchestration models for integrating quantum work with classical applications.

In practice, running a quantum program is a service call. Your classical application or client code submits a job to a target, waits for it to run, and retrieves the results. One or more classical compute components orchestrate each quantum job by doing the following activities:

- Preparing input data
- Submitting quantum computing jobs to a target quantum environment
- Monitoring job execution
- Postprocessing job results

Quantum integration models

You integrate quantum work with a classical application by using one of two orchestration patterns:

- Direct quantum integration. A client application or lightweight classical harness interacts directly with the Azure Quantum workspace. The client owns input preparation, job

submission, monitoring, result handling, and the classical logic that surrounds the quantum execution.

- Workflow-orchestrated quantum integration. A workflow orchestrator owns the overall state and transitions. Quantum execution occurs in workflow steps that run alongside steps running on high-performance computing (HPC) or graphics processing unit (GPU) compute.

For a given application integration boundary, these patterns are alternatives. Either the client integrates directly with the workspace, or a broader workflow owns the quantum step. This article describes the implementation of each pattern.

> **Note**
>
> The architectures in this article run part of a compute task on a quantum target. For some compute challenges, existing services built to perform high-performance computing or provide AI functionality might be alternatives.

The integration choice is independent of where the classical computation runs. Depending on the target profile and workload, classical logic can run within the quantum program, in the client harness between quantum executions, in the workflow orchestrator, or in a classical workflow step on HPC or GPU compute.

Decision matrix

Use the following decision matrix to guide your choice of integration patterns:

Expand table

If your workload has these characteristics	Use this approach
One application or lightweight harness owns the full quantum job lifecycle, and the input preparation and result processing fit inside that client.	Direct quantum integration
You explore the properties of quantum hardware with handwritten quantum code, and one application or a small set of related applications consumes the results.	Direct quantum integration
The classical side is heavyweight and consumes quantum as one step among many, in a multistage pipeline whose stages each run on the back	Workflow-orchestrated quantum

If your workload has these characteristics	Use this approach
end that fits them, such as HPC, GPU, or a quantum target.	integration
Quantum executions repeat, and the result of one execution generates the next quantum program or determines its parameters.	Workflow-orchestrated quantum integration

Scenario details

Both workflows implement the Asynchronous Request-Reply pattern and the steps defined for the Azure Quantum job lifecycle.

Quantum targets, especially quantum hardware, are limited resources. Azure Quantum allocates these resources through a job queue. When you submit a job, the job enters the queue for the target that you select and runs after that target finishes the earlier entries. To see the expected wait time, list the available targets. Calculate your full response time as the queue wait plus the job execution time.

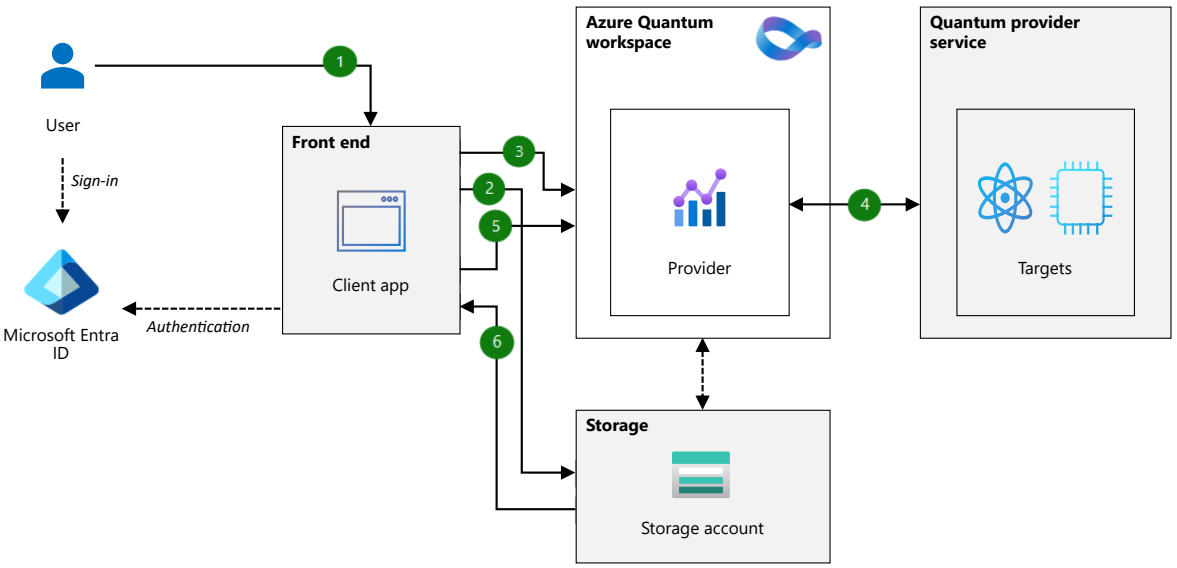
The capabilities of a quantum target also vary. Some targets accept quantum operations only, while others run classical logic with quantum operations in a single job. This model supports algorithms that adapt during execution. Before you commit to a target, confirm that it supports the operations that your algorithm needs. For more information about how classical and quantum instructions run together, see Introduction to hybrid quantum computing.

> **Note**
>
> These patterns describe how you integrate quantum work into your application architecture. They're distinct from Azure Quantum *hybrid computing models*, which describe how classical and quantum computation interact within quantum jobs. Target profiles constrain what can run within these jobs. For more information, see **Introduction to hybrid quantum computing**. Application integration patterns are independent of within-job behavior and control whether a client or a workflow runs the quantum jobs.

Direct quantum integration

The following sections describe the direct integration model for integrating quantum work with a classical application.

Architecture



Download a PowerPoint file of this architecture.

Data flow

The following data flow corresponds to the preceding diagram:

1. A signed-in user triggers quantum job execution via a classical client application.
2. The client application inputs data into Azure Storage.
3. The client application submits the job to an Azure Quantum workspace, specifying the execution target or targets.

   The client identifies the workspace from its configuration and authenticates to the workspace by using a Microsoft Entra identity. A client that runs on an Azure-hosted resource can use a managed identity. A local client application authenticates with another Microsoft Entra identity, like a service principal or an interactive user sign-in.
4. A quantum provider runs the job on a target environment.
5. The client application monitors job execution by polling job status.
6. As soon as the quantum job finishes, the client application gets the compute result from Storage.

Components

- Azure Quantum provides a workspace, accessible from the Azure portal, for assets associated with running quantum jobs on various targets. Jobs run on quantum simulators or quantum hardware, depending on the provider you choose.
- Microsoft Entra ID coordinates user authentication and helps protect access to the Azure Quantum workspace.
- Storage provides storage for input data and results from the quantum provider.

Direct integration potential use cases

The direct quantum integration pattern fits the following use cases:

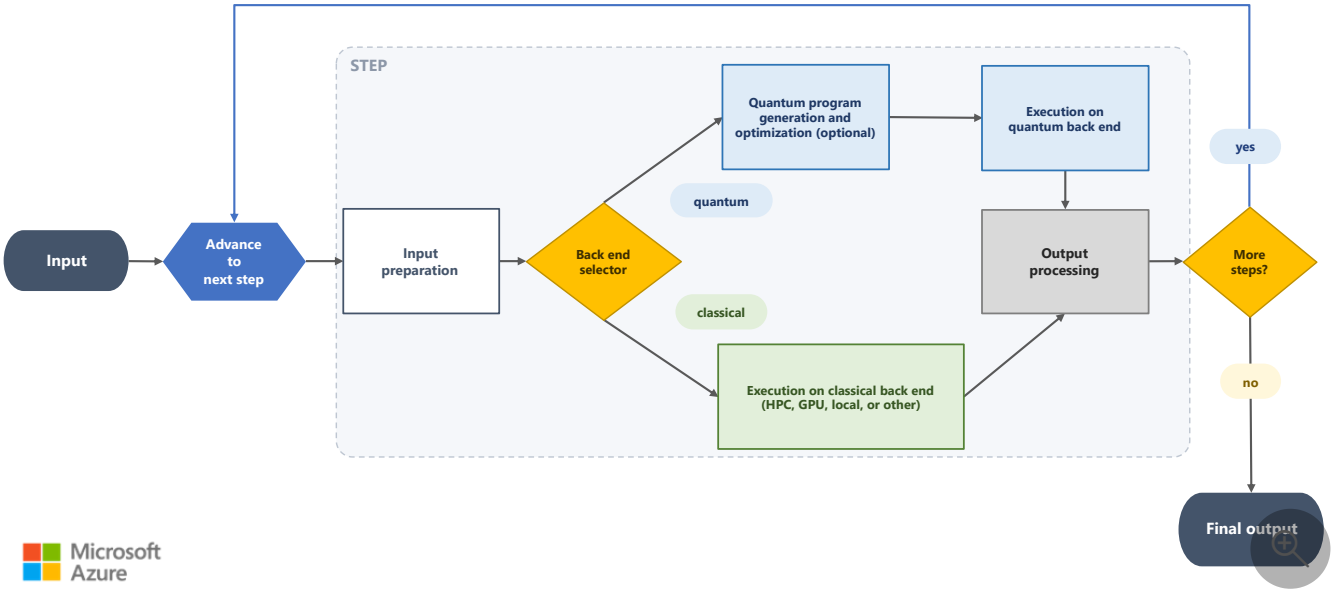
- One client application or lightweight classical harness owns the complete quantum job lifecycle without a broader workflow.
- The client can run surrounding classical work, such as input preparation and result processing.
- You explore the properties of quantum hardware, so you typically handwrite the quantum code rather than generate it dynamically.
- Use of the quantum components is limited to a single application or a small set of related applications.
- The quantum job represents a specialized solution, such as a molecular simulation, that only one specialized classical application uses.

Workflow-orchestrated quantum integration

The following sections describe the workflow-orchestrated model for integrating quantum work with a classical application.

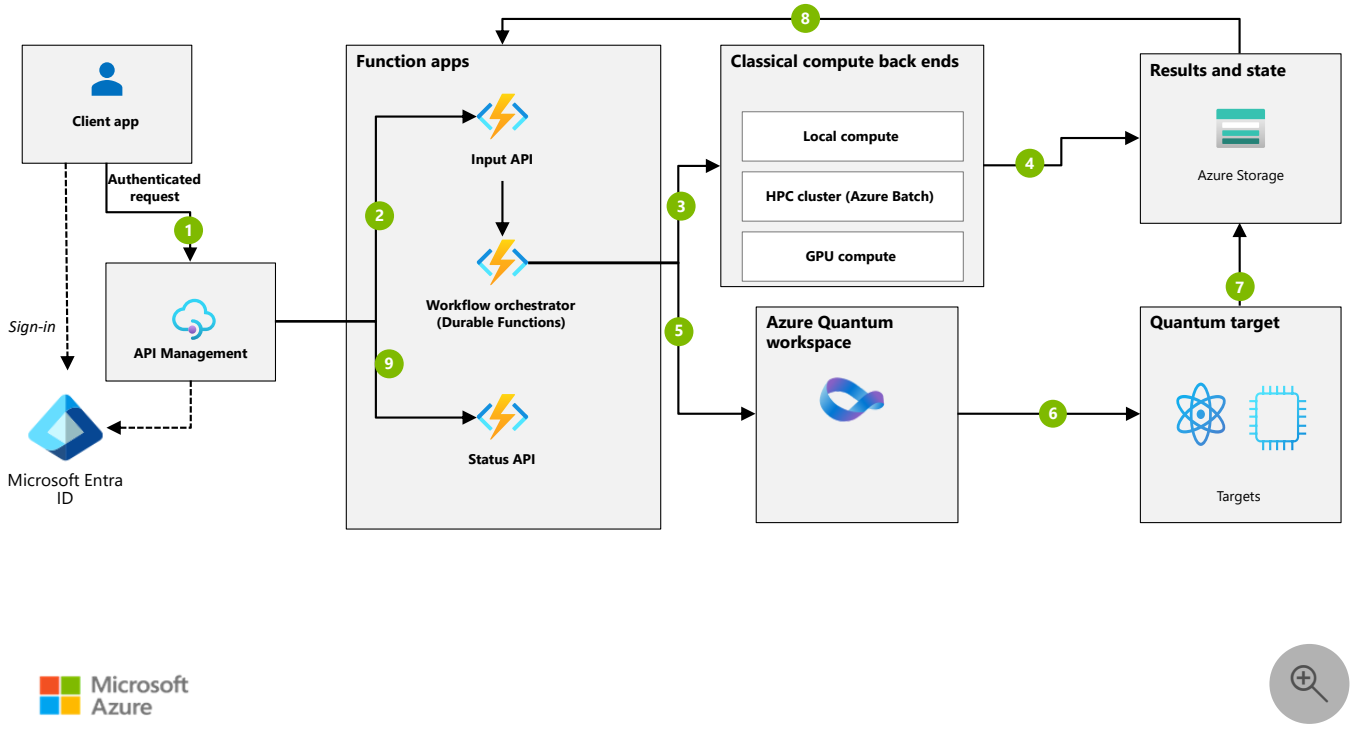
Architecture

The workflow logic doesn't depend on any specific host or provider. The following flowchart shows the state machine: a single input, a loop over the steps, and the per-step phases. The quantum branch can include an optional program generation and optimization phase that classical steps don't use.



The back-end selector in the preceding flowchart is a logical phase rather than a required component, and can be a configuration lookup that routes each step to a fixed back-end type.

You can host this workflow on Azure with a set of classical services that submit and monitor the quantum work. The following diagram shows an example topology:



Download a PowerPoint file of this architecture.

Data flow

The following data flow corresponds to the deployment topology:

1. A client application submits a request through API Management, which authenticates the caller with Microsoft Entra ID and applies throttling before the request reaches the compute tier.
2. API Management forwards the request to the Input API, an HTTP-triggered function that validates it and starts the workflow orchestrator.
3. For a step that runs classically, the orchestrator routes the execution to a classical back end, such as an HPC cluster or GPU compute.
4. The classical back end writes its results and the updated pipeline state to Storage.
5. For a step that runs on quantum, the orchestrator routes the execution through the quantum path. The quantum path prepares the step's program, generating and optimizing it from the step's input or using a fixed implementation, and submits the job to the Azure Quantum workspace. The submission authenticates via managed identity.
6. The workspace runs the job on the selected quantum target.
7. The quantum target writes its results to Storage.
8. The orchestrator reads the updated state Storage and evaluates the pipeline's transition logic. If a convergence or iteration condition requires another quantum execution, the orchestrator uses the result to determine parameters for the next quantum program or to provide inputs that generate the next program. The pipeline then loops back to the appropriate step.
9. The client polls the Status API through API Management to track progress and retrieve the final results when the pipeline reaches a terminal state.

Components

- Durable Functions acts as the workflow orchestrator to run the pipeline as a state machine, coordinate the steps, and select the quantum or other compute type back end for each step's execution. You can implement the orchestrator with Durable Functions or another workflow engine.
- Azure HPC and GPU classical compute back ends run the non-quantum steps.
- Azure Functions hosts the HTTP APIs that start and monitor the workflow, along with the orchestrator that runs it.
- API Management is the entry point for client requests. It authenticates callers and applies throttling. To prevent the function app from being called directly, use Azure Functions networking options, such as inbound access restrictions or a private endpoint, so that it accepts traffic only from API Management.

- Azure Quantum provides a workspace for assets associated with running quantum jobs. Jobs run on quantum simulators or quantum hardware, depending on the target you choose.
- Microsoft Entra ID coordinates authentication and helps protect access to the Azure Quantum workspace.
- Storage stores input data, intermediate pipeline state, and results.

Alternatives

The architectures in this article run part of a compute task on a quantum target. For some compute challenges, existing services built to perform high-performance computing or provide AI functionality might be alternatives.

For scientific R&D workloads, Microsoft Discovery uses AI to orchestrate complex tasks across models, tools, and compute resources. Microsoft Discovery can coordinate various tools across multistep pipelines and is extensible, so you can connect your own tools and agents instead of building and operating the orchestration yourself. Integration with quantum capabilities, including running quantum steps, could use AI to take on the orchestration that this pattern builds by hand.

Workflow-orchestrated scenario details

In this pattern, a workflow orchestrator runs the workload as a pipeline of steps. The orchestrator behaves like a state machine in which steps repeat and loop until a termination or convergence condition is met. Each step follows the same shape: It prepares its input, executes, and processes the output.

The orchestrator selects the back end that runs each step. This selection is part of orchestration logic, not a separate service, and can be as simple as reading the target back end for each step from configuration. A step runs on a quantum back end or on classical compute, such as an HPC cluster or GPU compute. A lightweight classical step can run in the orchestrator itself. When quantum steps repeat, the workflow processes the result of one quantum execution to determine parameters for the next quantum program or to provide inputs that generate the next program.

A step that runs on a quantum back end can add one phase that classical steps don't have. That phase generates and optimizes the quantum program before execution. This phase is optional. The program can generate dynamically from the step's input, or it can come from a fixed implementation that skips this phase. A step that's eligible for quantum might still run on a classical back end when that fits the input better.

Quantum chemistry is a representative example. A typical pipeline uses classical steps to prepare a molecular system: Geometry optimization, a self-consistent field calculation, and active-space selection. The pipeline then computes a target property, such as the ground-state energy of a molecule. The energy-computing step keeps the same intent whether it runs on a classical approximation or on a quantum algorithm. You choose the back end based on the accuracy you need and the size of the problem.

The QDK/Chemistry library supports pipelines like these. The library provides modular components for the classical preparation steps and for generating a state-preparation circuit from the classically computed wavefunction, which a quantum back end then uses to estimate the energy with an algorithm such as quantum phase estimation. This circuit-generation capability is a concrete example of producing the quantum program dynamically from a step's input.

Potential use cases

The workflow-orchestrated quantum integration pattern fits these use cases:

- The classical side is heavyweight and problem-state-logic oriented, and it consumes the quantum capability as one step or several steps among many.
- The workload is a multistage pipeline that is often iterative, like a state machine. Each stage runs on the compute back end that best fits it: HPC, GPU, or a quantum target. Quantum computing is one option for one or more stages.
- A quantum step represents a well-defined building block, such as a scientific calculation that computes a molecular property. The precise quantum code might even be generated and optimized dynamically to fit the actual inputs.
- The workflow orchestrator owns multiple self-contained quantum executions and the classical processing between them. The result of one execution determines parameters for the next quantum program or provides inputs that generate the next program. Iterative phase estimation fits this structure, as do variational algorithms such as the Variational Quantum Eigensolver (VQE) and the Quantum Approximate Optimization Algorithm (QAOA).

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that you can use to improve the quality of a workload. For more information, see Well-Architected Framework.

Reliability

Reliability helps ensure that your application can meet the commitments that you make to your customers. For more information, see Design review checklist for Reliability.

Quantum jobs run on remote, shared targets, so job execution can fail from transient errors, such as a target timeout. Regardless of the integration pattern you choose, monitor job execution so that you can surface job status to the user. When a job fails because of a transient error, apply the Retry pattern. Submit jobs through asynchronous calls and poll for the result so that you don't block the calling client.

Availability of the quantum compute functionality depends highly on the availability and capacity characteristics of the quantum computing provider. Depending on the compute target, the classical client application might experience long delays or unavailability of the target.

For the surrounding Azure services, the usual availability considerations apply. If necessary, consider using the replication options in Azure Storage redundancy.

Reliability for workflow-orchestrated integration

- For high availability in workflow-orchestrated integration, deploy API Management across availability zones or multiple regions. Zone redundancy requires the Premium or Premium v2 tier, and multiregion deployment requires the Premium tier.

- If you implement the orchestrator with Durable Functions, plan its disaster recovery as a unit rather than treating the function app and its state as independent services. Durable functions persist all orchestration state in a task hub in a storage back end, which is Azure Storage by default. Because the runtime state and the compute are coupled through that task hub, provisioning the function app in a second region and replicating Storage separately doesn't define a safe failover. Orchestrations can pause, lose recent transactions, or read a cross-region task hub, depending on the topology.

  For a safe failover, use an active-passive configuration that fails over to a secondary region, fronted by a global load-balancing service such as Azure Front Door or Azure Traffic Manager. Confirm that the service health probes can reach the function app under your networking restrictions, because the same controls that limit the app to API Management traffic can also block those probes. Choose the topology that matches your tolerance for data loss and cross-region latency. For the coordinated options and their tradeoffs, see Disaster recovery and geo-distribution in Durable Functions.

Security

Security provides assurances against deliberate attacks and the misuse of your valuable data and systems. For more information, see Design review checklist for Security.

Apply the following hardening practices to the classical services that surround the quantum job, regardless of the integration pattern you choose:

- Authenticate to Azure Quantum and the surrounding services by using Microsoft Entra identities, and turn off local authentication where the service supports it. Use managed identities wherever the hosting environment supports them. For a client that can't use a managed identity, authenticate with another Microsoft Entra identity, such as a service principal or an interactive user sign-in.
- Grant each component that accesses Storage the data-plane access it needs through Azure role-based access control (Azure RBAC) with Microsoft Entra ID, scoped to that component. Don't embed storage account keys in application code.

In general, apply the Well-Architected Framework security recommendations when appropriate.

Security for direct quantum integration

Unlike workflow-orchestrated quantum integration, this pattern assumes that a single client accesses the Azure Quantum workspace. The client is typically a lightweight classical harness that focuses on job submission and operation rather than broader workflow state. This scenario leads to the following configurations:

- Because the client is known, you can give it a fixed identity. When the client runs on an Azure-hosted resource, associate a managed identity with it. When the client runs outside Azure, use a service principal or an interactive user sign-in.
- You can implement request throttling and result caching in the client itself.

Security for workflow-orchestrated integration

Unlike direct quantum integration, this pattern puts a classical service tier in front of the quantum work. API Management is the front door for that tier, so the security configurations emphasize protecting the entry point and the path to the quantum workspace.

- Clients must authenticate to the API. Implement this authentication by using authentication policies.

- You can implement authentication of the Azure functions via managed identities associated with the functions. You use those identities to authenticate outbound calls to the Azure Quantum workspace.
- API Management can apply request throttling to protect the quantum back end and limit the use of quantum resources. For more information, see API Management request throttling.
- Depending on the request pattern, you might be able to implement the caching of quantum computing results by using API Management caching policies.

Cost Optimization

Cost Optimization focuses on ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see Design review checklist for Cost Optimization.

The overall cost of this solution depends on the quantum computing target that you select to run the quantum job. The classical components are straightforward to estimate. For a representative deployment of the workflow-orchestrated pattern, see this example cost estimate, which covers the classical components, including API Management, Azure Functions, and Storage. The direct integration pattern is lighter, but its classical costs still include Storage and any hosting, networking, and monitoring that the client application uses.

You can consume quantum computing providers for Azure Quantum via a Microsoft Marketplace offering. Pricing depends on the type of resource (simulator or hardware), the SKU, and your usage. For more information, access the reference page for your scenario's provider from Quantum computing providers on Azure Quantum.

Contributors

Microsoft maintains this article. The following contributor wrote this article.

Principal author:

- Zander Chocron | Principal Software Engineer, Microsoft Quantum

To see nonpublic LinkedIn profiles, sign in to LinkedIn.

Next steps

- For an overview of the Microsoft quantum computing ecosystem, see the Microsoft Quantum website and complete the Quantum computing foundations learning path.
- For more information about the Azure Quantum service, see What is Azure Quantum?.
- For general information about Azure Quantum job management, see Work with Azure Quantum jobs.
- For information about combining classical and quantum instructions in a job, see Introduction to hybrid quantum computing.

Related resources

- Operational Excellence design principles
- Asynchronous Request-Reply pattern

Get started with sessions

Sessions are a key feature of hybrid quantum computing. A session is a logical grouping of one or more jobs that you submit to a single target. Each session has a unique ID attached to each job in that session. Sessions are useful when you want to run multiple quantum computing jobs in sequence, and run classical code between quantum jobs.

> **Note**
>
> Sessions group jobs for submission from Azure Quantum to a provider, but jobs in a session aren't batched on the provider's side.

Prerequisites

To create a session, you need the following prerequisites:

- An Azure account with an active subscription. If you don't have an Azure account, then register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. For more information, see Create an Azure Quantum workspace.
- A Python environment with Python and Pip installed.
- The latest version of Visual Studio Code (VS Code), with the QDK extension, Python extension, and Jupyter extension installed.
- The `qdk` Python package. To submit Qiskit and Cirq programs, install the `azure`, `qiskit`, and `cirq` extras.

  Bash

  ```
  pip install --upgrade "qdk[azure,qiskit,cirq]"
  ```

What is a session?

In sessions, you can move the client compute resource to the cloud for lower-latency and the ability to run your quantum program multiple times with different parameters. You can logically

group jobs into one session, and you can prioritize jobs in that session over non-session jobs. Qubit states don't persist between jobs, but jobs in a session have shorter queue times. Shorter queue times allow you to run complex algorithms to better organize and track your individual quantum computing jobs.

Sessions are useful for parameterized quantum algorithms, where the output of one quantum computing job is used to define input parameters for the next quantum computing job. The most common examples of this type of algorithm are Variational Quantum Eigensolvers (VQE) and Quantum Approximate Optimization Algorithms (QAOA).

Supported hardware

Sessions are supported on all quantum computing hardware providers. In some cases, jobs that are submitted within a session are prioritized in the queue of that target. For more information, see Target behavior.

How to create a session

To create a session, follow these steps:

Q# + Python

This example shows how to create a session with Q# inline code in a Jupyter notebook in VS Code.

> **Note**
>
> Sessions are managed with Python, even when running Q# inline code.

1. In VS Code, open the **View** menu and choose **Command Palette**.
2. Enter and select **Create: New Jupyter Notebook**.
3. In the top-right, VS Code will detect and display the version of Python and the virtual Python environment that was selected for the notebook. If you have multiple Python environments, you may need to select a kernel using the kernel picker in the top right. If no environment was detected, see Jupyter Notebooks in VS Code for setup information.
4. In the first cell of the notebook, run the following code:

Python

```
from qdk.azure import Workspace

workspace = Workspace(resource_id="") # add your resource ID
```

5. Add a new cell in the notebook and import the `qsharp` Python package:

Python

```
from qdk import qsharp
```

6. Choose your quantum target. In this example, the target is the IonQ simulator.

Python

```
target = workspace.get_targets("ionq.simulator")
```

7. Select the configurations of the target profile, either `Base`, `Adaptive_RI`, or `Unrestricted`.

Python

```
qsharp.init(target_profile=qsharp.TargetProfile.Base)
```

> **Note**
>
> `Adaptive_RI` target profile jobs are currently supported on Quantinuum targets. For more information, see Integrated hybrid quantum computing.

8. Write your Q# program. For example, the following Q# program generates a random bit. To illustrate the use of input arguments, this program takes an integer, `n`, and an array of angles, `angle`, as input.

Q#

```
%%qsharp
import Std.Measurement.*;
import Std.Arrays.*;

operation GenerateRandomBits(n: Int, angle: Double[]) : Result[] {
   use qubits = Qubit[n]; // n parameter as the size of the qubit array
   for q in qubits {
       H(q);
```

```
    }
    R(PauliZ, angle[0], qubits[0]); // arrays as entry-points parameters
    R(PauliZ, angle[1], qubits[1]);
    let results = MeasureEachZ(qubits);
    ResetAll(qubits);
    return results;
}
```

9. Next, you create a session. Let's say you want to run `GenerateRandomBit` operation three times, so you use `target.submit` to submit the Q# operation with the `target` data and you repeat the code three times - in a real world scenario, you may want to submit different programs instead of the same code.

Python

```
angle = [0.0, 0.0]
with target.open_session(name="Q# session of three jobs") as session:
    target.submit(input_data=qsharp.compile(f"GenerateRandomBits(2,
{angle})"), name="Job 1", shots=100) # First job submission
    angle[0] += 1
    target.submit(input_data=qsharp.compile(f"GenerateRandomBits(2,
{angle})"), name="Job 2", shots=100) # Second job submission
    angle[1] += 1
    target.submit(input_data=qsharp.compile(f"GenerateRandomBits(2,
{angle})"), name="Job 3", shots=100) # Third job submission

session_jobs = session.list_jobs()
[session_job.details.name for session_job in session_jobs]
```

> **ⓘ Important**
>
> When you pass arguments as parameters to the job, the arguments are formatted into the Q# expression when `qsharp.compile` is called. This means that you need to format your arguments as Q# objects. In this example, because arrays in Python are already printed as `[item0, item1, ...]`, the input arguments match the Q# formatting. For other Python data structures, you might need more handling to get the string values inserted into the Q# in a compatible way.

10. Once you create a session, you can use `workspace.list_session_jobs` to retrieve a list of all jobs in the session. For more information, see How to manage sessions.

Target behavior

Each quantum hardware provider defines their own heuristics to best manage the prioritization of jobs within a session.

Quantinuum

If you choose to submit jobs within a session to a Quantinuum target, then your session has exclusive access to the hardware as long as you queue jobs within one minute of each other. After that, your jobs are accepted and handled with the standard queueing and prioritization logic.

Related content

- How to manage your sessions
- Run hybrid quantum computing

Last updated on 08/13/2026

How to manage your sessions

In Azure Quantum, you can group multiple jobs against a single target to effectively manage your jobs. This is called a session. For more information, see Get started with sessions.

In this article, you learn how use sessions to manually manage your jobs. You also learn about the job failure policies and how to avoid session timeouts.

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. For more information, see Create an Azure Quantum workspace.
- A Python environment with Python and Pip installed.
- The latest version of the `qdk` Python package with the `azure` extra.

  Bash

  ```
  pip install --upgrade "qdk[azure]"
  ```

  If you want to use Qiskit and Cirq, then you need to install the `qiskit` and `cirq` extras.

  Bash

  ```
  pip install --upgrade "qdk[azure,qiskit,cirq]"
  ```

> **Note**
>
> Sessions are managed with Python, even when you run Q# inline code.

Monitor your sessions

To view and manage your submitted sessions, follow these steps.

In the Azure portal, use the **Job management** blade in your Quantum workspace to view all top-level submitted items, including sessions and individual jobs that aren't associated with any session.

1. Sign in to the Azure portal.
2. Go to your Quantum workspace.
3. Expand the **Operations** dropdown and select **Job management** to view a table of your submitted jobs and sessions. Sessions have a value of **Session** in the **Type** column.
4. To view details about a session, including a list of **All jobs** in the session, select the session **Name**.

Each session has a unique ID and status. Sessions can have the following statuses.

- **Waiting**: Jobs within the session are currently running.
- **Succeeded**: The session ended successfully.
- **TimeOut**: If no new job is submitted within the session for 10 minutes, then that session times out. For more information, see Session timeouts.
- **Failed**: If a job within a session fails, then that session ends and reports a status of **Failed**. For more information, see Job failure policy within sessions.

Retrieve and list sessions

The following table shows the Python commands to get the list of all sessions and all jobs for a given session. To connect to your workspace through the `qdk.azure` module, see Connect to your Azure Quantum workspace with the QDK or Azure CLI.

Expand table

Command	Description
workspace.list_sessions() or `session.list_sessions()`	Retrieve a list of all sessions in a Quantum Workspace.
workspace.get_session(sessionId) or `session.get_session(sessionId)`	Retrieve the session with ID `sessionId`. Each session has a unique ID.
workspace.list_session_jobs(sessionId) or `session.list_session_jobs(sessionId)`	Retrieve a list of all jobs in the session with ID `sessionId`. Each session has a unique ID.

For example, the following code defines a function that gets a session with a minimum number of jobs. Then, for that session, it lists all the jobs, the total number of jobs, and the first 10 jobs.

Python

```
def get_a_session_with_jobs(min_jobs):
    all_sessions = workspace.list_sessions() # list of all sessions
    for session in all_sessions:
        if len(workspace.list_session_jobs(session.id)) >= min_jobs:
            return session

session = get_a_session_with_jobs(min_jobs=3) # Get a Session with at least 3 jobs

session_jobs = workspace.list_session_jobs(session.id) # List of all jobs within
Session ID

print(f"Job count: {len(session_jobs)} \n")
print(f"First 10 jobs for session {session.id}:")
for job in session_jobs[0:10]:
    print(f"Id: {job.id}, Name={job.details.name}")
```

Manually open or close sessions

To create a new session, it's a best practice to follow the steps in Get started with sessions. To manually create sessions instead, choose the tab for your development environment.

Q# + Python

1. Create a **Session object**.

   Python

   ```
   from qdk.azure.job import Session, SessionDetails, SessionJobFailurePolicy
   import uuid

   session = Session(
       workspace=workspace, # required
       id=f"{uuid.uuid1()}", # optional, if not passed will use uuid.uuid1()
       name="", # optional, will be blank if not passed
       provider_id="ionq", # optional, if not passed will try to parse from the
   target
       target="ionq.simulator", # required
       job_failure_policy=SessionJobFailurePolicy.ABORT # optional, defaults to
   abort
       )

   print(f"Session status: {session.details.status}")
   ```

   ⓘ Note

At this point, the session only exists on the client, and you can see that the status is **None**. To view the status of the session, you also need to create the session in the service.

2. To **create** a session in the service, you can use workspace.open_session(session) or session.open().

3. You can refresh the **status** and the session details with session.refresh(), or by getting a new session object from a session ID.

Python

```
same_session = workspace.get_session(session.id)
print(f"Session: {session.details} \n")
print(f"Session: {same_session.details} \n")
```

4. You can **close** a session with session.close() or workspace.close_session(session).

5. To **attach the session** to a target, you can use `target.latest_session`.

6. You can **wait** for a session to be completed:

Python

```
session_jobs = session.list_jobs()
[session_job.id for session_job in session_jobs]

import time
while (session.details.status != "Succeeded" and session.details.status !=
"Failed" and session.details.status != "TimedOut"):
  session.refresh()
  time.sleep(5)
```

Pass arguments in Q#

If your Q# operation takes input arguments, pass those arguments as Q# objects when you submit jobs through Python.

When you pass arguments as parameters to the job, the arguments are formatted as Q# code when `qsharp.compile` is called, so the values from Python need to be formatted into a string as valid Q# syntax.

Consider the following Q# program, which takes an integer, `n`, and an array of angles, `angle`, as input.

Q#

```
import Std.Measurement.*;
import Std.Arrays.*;

operation GenerateRandomBits(n: Int, angle: Double[]) : Result[] {
   use qubits = Qubit[n]; // n parameter as the size of the qubit array
   for q in qubits {
       H(q);
   }
   R(PauliZ, angle[0], qubits[0]); // arrays as entry-points parameters
   R(PauliZ, angle[1], qubits[1]);
   let results = MeasureEachZ(qubits);
   ResetAll(qubits);
   return results;
}
```

You want to run `GenerateRandomBits` operation three times with `n=2` and different angles. You can use the following Python code to submit three jobs with different angles.

Python

```
angle = [0.0, 0.0]
with target.open_session(name="Q# session of three jobs") as session:
    target.submit(input_data=qsharp.compile(f"GenerateRandomBits(2, {angle})"),
name="Job 1", shots=100) # First job submission
    angle[0] += 1
    target.submit(input_data=qsharp.compile(f"GenerateRandomBits(2, {angle})"),
name="Job 2", shots=100) # Second job submission
    angle[1] += 1
    target.submit(input_data=qsharp.compile(f"GenerateRandomBits(2, {angle})"),
name="Job 3", shots=100) # Third job submission

session_jobs = session.list_jobs()
[session_job.details.name for session_job in session_jobs]
```

In this example, because arrays in Python are already printed as [item0, item1, ...], the input arguments match the Q# formatting. For other Python data structures you might need more handling to get the string values inserted into the Q# in a compatible way. For example, a Q# tuple has to be in parentheses with comma separated values.

Session timeouts

A session times out if no new job is submitted within the session for 10 minutes. The session reports a status of **TimedOut**. To avoid this situation, add a `with` block using `backend.open_session(name="Name")`, so the session `close()` is invoked by the service at the end of the code block.

> **Note**
>
> If there are errors or bugs in your program, it might take more than 10 minutes to submit a new job after the previous jobs in the session are complete.

The following code snippets show an example of a session that times out after 10 minutes because no new jobs are submitted. To avoid that, the next code snippet shows how to use a `with` block to create a session.

Python

```
# Example of a session that times out

session = backend.open_session(name="Qiskit circuit session") # Session times out
because only contains one job
backend.run(circuit=circuit, shots=100, job_name="Job 1")
```

Python

```
# Example of a session that includes a with block to avoid timeout

with backend.open_session(name="Qiskit circuit session") as session: # Use a with
block to submit multiple jobs within a session
    job1 = backend.run(circuit=circuit, shots=100, job_name="Job 1") # First job
submission
    job1.wait_for_final_state()
    job2 = backend.run(circuit=circuit, shots=100, job_name="Job 2") # Second job
submission
    job2.wait_for_final_state()
    job3 = backend.run(circuit=circuit, shots=100, job_name="Job 3") # Third job
submission
    job3.wait_for_final_state()
```

Job failure policy within sessions

The default policy for a session when a job fails is to end that session. If you submit an additional job within the same session, the service rejects it and the session reports a status of **Failed**. Any in progress jobs are canceled.

However, this behavior can be changed by specifying a job failure policy of `job_failure_policy=SessionJobFailurePolicy.CONTINUE`, instead of the default `SessionJobFailurePolicy.ABORT`, when creating the session. When the job failure policy is `CONTINUE`, the service continues to accept jobs. The session reports a status of **Failure(s)** in this case, which will change to **Failed** once the session is closed.

If the session is never closed and times out, the status is **TimedOut** even if jobs have failed.

For example, the following program creates a session with three jobs. The first job fails because it specifies `"garbage"` as input data. To avoid the end of the session at this point, the program shows how to add `job_failure_policy=SessionJobFailurePolicy.CONTINUE` when you create the session.

Python

```
#Example of a session that does not close but reports Failure(s) when a jobs fails

with target.open_session(name="JobFailurePolicy Continue",
job_failure_policy=SessionJobFailurePolicy.CONTINUE) as session:
    target.submit(input_data="garbage", name="Job 1") #Input data is missing, this job
fails
    target.submit(input_data=quil_program, name="Job 2") #Subsequent jobs are accepted
because of CONTINUE policy
    target.submit(input_data=quil_program, name="Job 3")
```

Related content

- Running hybrid quantum computing
- Get started with sessions

Last updated on 08/14/2026

Quantum computing providers on Azure Quantum

Azure Quantum offers various quantum solutions, such as different quantum hardware devices and quantum simulators that you can use to run your quantum computing programs. This article lists the providers that you can access with Azure Quantum, and provides a description of what each provider offers.

Expand table

Provider	Description
IONQ	IonQ's trapped-ion gate-based quantum computers are universal and dynamically reconfigurable in software, providing up to 25 qubits in the Ionq Aria QPUs, and 36 qubits in the IonQ Forte 1 QPU and Forte Enterprise 1 QPU. All qubits are fully connected, meaning you can run a two-qubit gate between any pair. The implementation of quantum gate operations is done by manipulating Ytterbium ions with laser pulses. IonQ provides a GPU-accelerated quantum simulator supporting up to 36 qubits, using the same set of gates IonQ provides on its quantum hardware. For more information, see the IonQ provider page.
PASQAL	Pasqal's neutral atom-based quantum processors operating at room temperature have long coherence times and impressive qubit connectivity. The operations are performed with optical tweezers, using laser light to manipulate 1D and 2D quantum registers with up to a hundred qubits. For more information, see the Pasqal provider page.
QUANTINUUM	Quantinuum's trapped-ion quantum computers have high-fidelity, fully connected qubits, and qubit reuse. Quantum operations are laser-based gates with low error rates, and have the ability to perform mid-circuit measurements. The System Model H2 generation of hardware, Powered by Honeywell, uses a Quantum Charge-Coupled Device (QCCD) architecture. Quantinuum provides emulation tools, two System Model H2 Emulators, which contain detailed physical models and noise models of the actual quantum hardware. For more information, see the Quantinuum provider page.
rigetti	Rigetti's systems are powered by superconducting qubit-based quantum processors. They offer fast gate times, low-latency conditional logic, and fast program execution times. At the chip level, each superconducting qubit consists of a non-linear Josephson inductance in parallel with an ultra-low-loss capacitor to create a resonant structure in the 3-6GHz range. Qubits are coupled to a linear superconducting resonator for readout. The combination of the qubit, the linear readout resonator, and the associated wiring provides a general-purpose quantum circuit element capable of reliably encoding, manipulating, and reading out quantum information. Rigetti's processors use arrays of qubits coupled to one another with on-chip capacitances. Single and multi-qubit logic operations are implemented through the application of microwave or DC pulses. For more information, see the Rigetti provider page.

> **Important**
>
> Quantum hardware devices are still an emerging technology. These devices have some limitations and requirements for quantum programs that run on them. For more information, see the target profile types in Azure Quantum.

For information on which quantum computing providers are available in your region, see Global availability of Azure Quantum providers.

Qubit availability for quantum computing providers

Microsoft's provider partners offer a wide-range of qubit availability for their hardware processors and simulators.

Expand table

Target name	Number of qubits
IonQ Quantum simulator	29 qubits
IonQ Aria 1	25 qubits
IonQ Forte 1	36 qubits
IonQ Forte Enterprise 1	36 qubits
Pasqal EMU_SV	25 qubits
Pasqal EMU_MPS	80 qubits
Pasqal EMU_FREE	12 qubits
Pasqal FRESNEL	100 qubits
Pasqal FRESNEL_CAN1	100 qubits
Quantinuum H2-1 Syntax Checker	32 qubits
Quantinuum H2-2 Syntax Checker	32 qubits
Quantinuum H2-1 Emulator	20 qubits
Quantinuum H2-2 Emulator	20 qubits
Quantinuum H2-1	20 qubits
Quantinuum H2-2	32 qubits

Target name	Number of qubits
Rigetti Quantum Virtual Machine (QVM)	30 qubits
Rigetti Cepheus-1-108Q	108 qubits

Coming soon to Azure Quantum

Azure Quantum is a platform for innovation. As the quantum hardware partners across the Azure Quantum ecosystem keep growing, you can explore these upcoming quantum hardware solutions.

Expand table

Provider	Description
qci	Quantum Circuits' full-stack superconducting circuits have real-time feedback that enables error-correcting, encoding-agnostic entangling gates. You can pre-register today for Azure Quantum's private preview of QCI.

Last updated on 04/23/2026

Global availability of Azure Quantum providers

10/15/2025

The availability of the quantum computing providers IonQ, Pasqal, Quantinuum, and Rigetti varies based on the country/region where your billing account is registered. Select the tab for each provider to view its availability by country/region.

IonQ

IonQ offers a Pay-As-You-Go plan through Azure Quantum. For more information about IonQ resources on Azure Quantum, see IonQ provider and targets.

IonQ is available in the following countries/regions:

- Armenia
- Australia
- Austria
- Belarus
- Belgium
- Bulgaria
- Canada
- Chile
- Colombia
- Croatia
- Cyprus
- Czechia
- Denmark
- Estonia
- Finland
- France
- Germany
- Greece
- Hungary
- Iceland
- India
- Indonesia
- Ireland
- Israel

- Italy
- Kenya
- Korea
- Latvia
- Liechtenstein
- Lithuania
- Luxembourg
- Malaysia
- Malta
- Monaco
- Netherlands
- New Zealand
- Nigeria
- Norway
- Poland
- Portugal
- Puerto Rico
- Romania
- Saudi Arabia
- Serbia
- Singapore
- Slovakia
- Slovenia
- South Africa
- Spain
- Sweden
- Switzerland
- Taiwan
- Thailand
- Türkiye
- United Arab Emirates
- United Kingdom
- United States
- Vietnam

IonQ is unavailable in the following countries/regions:

- Afghanistan
- Albania
- Algeria

- Andorra
- Angola
- Argentina
- Azerbaijan
- Bahrain
- Bangladesh
- Barbados
- Belize
- Bermuda
- Bolivia
- Bosnia and Herzegovina
- Botswana
- Brazil
- Brunei
- Cabo Verde
- Cameroon
- Cayman Islands
- Costa Rica
- Côte d'Ivoire
- Curaçao
- Dominican Republic
- Ecuador
- Egypt
- El Salvador
- Ethiopia
- Faroe Islands
- Fiji
- Georgia
- Ghana
- Guatemala
- Honduras
- Hong Kong SAR
- Iraq
- Jamaica
- Japan
- Jordan
- Kazakhstan
- Kuwait
- Kyrgyzstan
- Lebanon
- Libya

- Macao SAR
- Mauritius
- Mexico
- Moldova
- Mongolia
- Montenegro
- Morocco
- Namibia
- Nepal
- Nicaragua
- North Macedonia
- Oman
- Pakistan
- Palestinian Authority
- Panama
- Paraguay
- Peru
- Philippines
- Qatar
- Russia
- Rwanda
- Saint Kitts and Nevis
- Senegal
- Sri Lanka
- Tajikistan
- Tanzania
- Trinidad and Tobago
- Tunisia
- Turkmenistan
- U.S. Virgin Islands
- Uganda
- Ukraine
- Uruguay
- Uzbekistan
- Vatican City
- Venezuela
- Yemen
- Zambia
- Zimbabwe

Backend quantum simulators from quantum providers

This article describes the backend simulators available from quantum providers. These simulators are available to all Azure Quantum users, and are a great way to test your Q# programs before running them on a real quantum computer.

IonQ

IonQ provides a GPU-accelerated idealized simulator supporting up to 29 qubits, using the same set of gates that IonQ provides on its quantum hardware. The simulator is a great place to preflight jobs before running them on an actual quantum computer.

- Job type: `Simulation`
- Data Format: `ionq.circuit.v1`
- Target ID: `ionq.simulator`
- Target Execution Profile: QIR Base (Quantum Intermediate Representation)

For more information, see the IonQ provider page.

Pasqal

Pasqal's EMU_MPS is a Pulser backend emulating this dynamic with matrix product states (MPS). Matrix Product States (MPS) or tensor train (TT) are a specific class of tensor networks that provide a tractable parametrization of quantum states.

EMU_MPS emulator runs on a cluster of NVIDIA DGX nodes, each equipped with NVIDIA A100 GPUs, enabling the emulation of Pasqal's quantum processors. It's a key tool to prototype and validate quantum programs before running them on the QPU (quantum processing unit). Up to 80 qubits in 2D arrays can be emulated to develop industrial applications and to advance scientific discovery.

- Job Type: `Simulation`
- Data Format: `application/json`
- Target ID: `pasqal.sim.emu-mps`
- Target Execution Profile: N/A

For more information, see the Pasqal provider page.

Quantinuum

Quantinuum provides two emulator tools:

Syntax Checkers - These tools verify proper syntax, compilation completion, and machine compatibility, using the same compiler as the quantum computer they target. There are Syntax Checkers for both System Model H2 machines.

- Job type: `Simulation`
- Data Formats: `honeywell.openqasm.v1`, `honeywell.qir.v1`
- Target ID:
  - H2-1 Syntax Checker: `quantinuum.sim.h2-1sc`
  - H2-2 Syntax Checker: `quantinuum.sim.h2-2sc`
- Target Execution Profile: QIR Adaptive RI
- Pricing: Free ($0)

Emulators - These tools contain a detailed physical model and realistic noise model of the actual System Model H2 hardware. There are emulators for both H2 machines along with a cloud-based Quantinuum Emulator.

- Job type: `Simulation`
- Data Format: `honeywell.openqasm.v1, honeywell.qir.v1`
- Target ID:
  - H2-1 Emulator: `quantinuum.sim.h2-1e`
  - H2-2 Emulator: `quantinuum.sim.h2-2e`
- Target Execution Profile: QIR Adaptive RI

The Quantinuum Emulator is a System Model H-series based emulator available free-of-charge on the Code with Microsoft Quantum page. For more information, see the Quantinuum Emulator page.

For more information about all the Quantinuum emulators, see the Quantinuum provider page.

Rigetti

Rigetti provides their Quantum Virtual Machine (QVM), an open-source simulator for Quil. The QVM target accepts a Quil program as text and runs that program on QVM hosted in the cloud, returning simulated results.

- Job Type: `Simulation`

- Data Formats: `rigetti.quil.v1`, `rigetti.qir.v1`
- Target ID: `rigetti.sim.qvm`
- Target Execution Profile: QIR Base
- Pricing: Free ($0)

For more information, see the Rigetti provider page.

Last updated on 04/23/2026

IonQ provider

10/03/2025

IonQ's quantum computers perform calculations by manipulating the hyperfine energy states of Ytterbium ions with lasers. Atoms are nature's qubits — every qubit is identical within and between programs. Logical operations can also be performed on any arbitrary pair of qubits, enabling complex quantum programs unhindered by physical connectivity. Want to learn more? Read IonQ's trapped ion quantum computer technology overview.

- Publisher: IonQ
- Provider ID: `ionq`

The following targets are available from this provider:

Expand table

Target name	Target ID	Number of qubits	Description
Quantum simulator	ionq.simulator	29 qubits	IonQ's cloud-based idealized simulator. Free of cost.
IonQ Aria 1	ionq.qpu.aria-1	25 qubits	IonQ's Aria 1 trapped-ion quantum computer.
IonQ Forte 1	ionq.qpu.forte-1	36 qubits	IonQ's Forte 1 trapped-ion quantum computer.
IonQ Forte Enterprise 1	ionq.qpu.forte-enterprise-1	36 qubits	IonQ's Forte Enterprise 1 trapped-ion quantum computer.

IonQ's targets correspond to a **QIR Base** profile. For more information about this target profile and its limitations, see Understanding target profile types in Azure Quantum.

Quantum simulator

GPU-accelerated idealized simulator supporting up to 29 qubits, using the same set of gates IonQ provide on its quantum hardware—a great place to preflight jobs before running them on an actual quantum computer.

- Job type: `Simulation`
- Data Format: `ionq.circuit.v1`
- Target ID: `ionq.simulator`
- Target Execution Profile: QIR Base

IonQ Aria quantum computer

IonQ Aria is the flagship of IonQ's trapped-ion quantum computers, with a 25-qubit dynamically reconfigurable system. For more information, see IonQ Aria (ionq.com).

> **Important**
>
> *Debiasing* is enabled on Aria systems by default, and submitted jobs are subject to debiasing-based pricing. For more information about debiasing and how to disable/enable the service, see Error mitigation.

- Job type: `Quantum Program`
- Data Format: `ionq.circuit.v1`
- Target ID: `ionq.qpu.aria-1`
- Target Execution Profile: QIR Base

Expand table

Parameter Name	Type	Required	Description
`shots`	int	No	Number of experimental shots.

System timing

Expand table

Measure	Average time duration
T1	10-100 s
T2	1 s
Single-qubit gate	135 µs
Two-qubit gate	600 µs

System fidelity

Expand table

Operation	Average fidelity
Single-qubit gate	99.95% (SPAM corrected)
Two-qubit gate	99.6% (not SPAM corrected)
SPAM*	99.61%

* State Preparation and Measurement (SPAM): This measurement determines how accurately a quantum computer can set a qubit into its initial state and then measure the result at the end.

IonQ Aria is available through Pay As You Go plan. For more information, see Azure Quantum pricing.

IonQ Forte quantum computer

IonQ Forte is IonQ's highest-performing, commercially available trapped-ion quantum computer. With a 36-qubit software-configurable system. For more information, see IonQ Forte (ionq.com).

> **Important**
>
> *Debiasing* is enabled on the Forte system by default, and submitted jobs are subject to debiasing-based pricing. For more information about debiasing and how to disable/enable the service, see **Error mitigation**.

- Job type: `Quantum Program`
- Data Format: `ionq.circuit.v1`
- Target ID: `ionq.qpu.forte-1`
- Target Execution Profile: QIR Base

Expand table

Parameter Name	Type	Required	Description
`shots`	int	No	Number of experimental shots.

IonQ Forte Enterprise quantum computer

IonQ Forte Enterprise is IonQ's highest-performing, commercially available trapped-ion quantum computer. With a 36-qubit software-configurable system. For more information, see IonQ Forte Enterprise (ionq.com).

The Forte Enterprise is a version of the Forte class systems that has been adapted and ruggedized for deployment in a standard data center environment, making it more suitable for enterprise-level, production-oriented tasks. The base quantum computing hardware and performance are the same. The primary difference between the IonQ Forte and the IonQ Forte Enterprise lies in their deployment and intended use cases, not their core performance specifications. While both systems boast the same high-performance metrics, the Forte Enterprise is specifically designed for integration into a data center environment.

> **ⓘ Important**
>
> *Debiasing* is enabled on the Forte Enterprise system by default, and submitted jobs are subject to debiasing-based pricing. For more information about debiasing and how to disable/enable the service, see Error mitigation.

- Job type: `Quantum Program`
- Data Format: `ionq.circuit.v1`
- Target ID: `ionq.qpu.forte-enterprise-1`
- Target Execution Profile: QIR Base

Expand table

Parameter Name	Type	Required	Description
`shots`	int	No	Number of experimental shots.

Input format

In Q#, the output of a quantum measurement is a value of type `Result`, which can only take the values `Zero` and `One`. When you define a Q# operation, it can only be submitted to IonQ hardware if the return type is a collection of `Result`s, that is, if the output of the operation is the result of a quantum measurement. The reason for this is because IonQ builds a histogram from the returned values, so it restricts the return type to `Result` to simplify creating this histogram.

IonQ's targets correspond to the QIR Base profile. This profile can't run quantum operations that require the use of the results from qubit measurements to control the program flow.

Output format

When you submit a quantum program to the IonQ simulator, it returns the histogram created by the measurements. The IonQ simulator doesn't sample the probability distribution created by a quantum program but instead returns the distribution scaled to the number of shots. This is most apparent when you submit a single shot circuit. You will see multiple measurement results in the histogram for one shot. This behavior is inherent to IonQ simulator, while IonQ QPU actually runs the program and aggregates the results.

Additional Capabilities

Additional capabilities supported by IonQ hardware are listed here.

Expand table

Capability	Description
Error mitigation	Use debiasing to minimize noise and maximize algorithmic performance on IonQ hardware
Native gates support	Define and execute circuits directly on IonQ hardware-native gates
Noise model simulation	Simulate the noise profile that circuits will encounter when you run them on different IonQ hardware.

Users can take advantage of these additional capabilities via pass-through parameters in the Azure Quantum Q# and Qiskit providers.

Error mitigation

IonQ provides the option to enable *quantum error mitigation* when submitting jobs to IonQ hardware. Error mitigation is a compiler-level process that runs and executes multiple symmetric variations of a circuit, and then aggregates the outcomes while mitigating the impact of hardware errors and qubit decoherence. Unlike *quantum error correction* techniques, error mitigation does not require large gate and qubit overhead.

Debiasing is the process of creating slight variations of a given circuit that *should* be identical on an ideal noiseless machine, using techniques such as different qubit assignments, gate decompositions, and pulse solutions, and then executing those variations.

Sharpening and *Averaging* are options for aggregating the results of the variations. Averaging is based equally on all the variation results, whereas Sharpening filters out the erroneous results, and can be more reliable for certain types of algorithms.

For more information, see Debiasing and Sharpening. For error mitigation pricing, see IonQ pricing.

Enabling error mitigation

> **Note**
>
> *Debiasing* is enabled by default on Aria and Forte systems.

On Azure Quantum, error mitigation can be enabled or disabled for jobs submitted with Q# or with Qiskit.

To enable error mitigation, add an optional parameter for the target machine:

Python

```
option_params = {
    "error-mitigation": {
        "debias": True
    }
}
```

To disable error mitigation, set the parameter to `False`:

Python

```
option_params = {
    "error-mitigation": {
        "debias": False
    }
}
```

> **Note**
>
> If you are also using IonQ's noise model simulation, those parameters can be included here, for example:

Python

```
option_params = {
    "error-mitigation": {
        "debias": False
    },
    "noise": {
    "model": "aria-1",
    "seed": 100
```

```
    }
}
```

For more information, see Noise model simulation.

Running a job on Azure Quantum with error mitigation

This example uses a simple random number generator.

First, import the required packages and initiate the base profile:

Python

```
import qsharp
import azure.quantum
qsharp.init(target_profile=qsharp.TargetProfile.Base)
```

Next, define the function.

Q#

```
%%qsharp
import Std.Measurement.*;
import Std.Arrays.*;
import Std.Convert.*;

operation GenerateRandomBit() : Result {
    use target = Qubit();

    // Apply an H-gate and measure.
    H(target);
    return M(target);
}

and compile the operation:

```python
MyProgram = qsharp.compile("GenerateRandomBit()")
```

Connect to Azure Quantum, select the target machine, and configure the noise parameters for the emulator:

Python

```
MyWorkspace = azure.quantum.Workspace(
    resource_id = "",
    location = ""
```
```

```
)

MyTarget = MyWorkspace.get_targets("ionq.qpu.aria-1")
```

Specify the `error-mitigation` configuration

Python

```
option_params = {
    "error-mitigation": {
        "debias": True
    }
}
```

Pass in the error mitigation configuration when submitting the job:

Python

```
job = MyTarget.submit(MyProgram, "Experiment with error mitigation", shots = 10,
input_params = option_params)
job.get_results()
```

In Qiskit, you pass the optional parameters to the target machine configuration before submitting the job:

Python

```
circuit.name = "Single qubit random - Debias: True"
backend.options.update_options(**option_params)
job = backend.run(circuit, shots=500)
```

> ⓘ **Note**
>
> If you do not pass in the `error-mitigation` parameter, the target machine will use its default setting, which is *enabled* for Aria and Forte systems.

Native gates support and usage

By default IonQ allows you to specify a quantum circuit using an abstract set of quantum gates, called `qis`, which allows flexibility and portability when writing an algorithm without worrying

about optimization for the hardware.

However, in some advanced usage cases, you might want to define a circuit directly on native gates in order to be closer to the hardware and bypass optimization. The native gate set is the set of quantum gates that are physically executed in the quantum processor, and they map the circuit to those as part of the execution.

For more information, see Getting Started With Native Gates (ionq.com).

In order to use the native gate set when submitting Qiskit jobs to Azure Quantum, you specify the `gateset` parameter when initializing the backend as in the example below:

Python

```
# Here 'provider' is an instance of AzureQuantumProvider
backend = provider.get_backend("ionq.qpu.aria-1", gateset="native")
```

Expand table

Parameter Name	Type	Required	Description
`gateset`	string	No	Specifies the set of gates that will be used to define a circuit. A value of `qis` corresponds to the abstract gates (default behavior) and `native` to the IonQ hardware native gates.

For more information about Qiskit jobs, see Submit a circuit with Qiskit.

Noise model simulation

Even the best of today's quantum hardware has inherent noise, and knowing the noise characteristics of your target system can help you refine your algorithms and get a more realistic prediction of results when running the circuit on hardware. IonQ provides a *noise model simulation* that introduces noise into the circuit using a "noise fingerprint" specific to the target hardware. For more information, see Get Started with Hardware Noise Model Simulation.

Noise model parameters

Expand table

Parameter Name	Values	Description
`noise`	`model`, `seed`	Enables the noise model simulation
`model`	`ideal`, `aria-1`	Specifies the noise model for the target hardware. • `ideal` - No noise is introduced into the circuit. This is the same as not enabling the noise simulation. • `aria-1` - Uses the noise model for the IonQ Aria quantum computer.
`seed`	Integer between 1 and 2^{31} (2,147,483,648)	Allows you to specify a seed value for pseudo-random noise and shot-sampling, creating reproducible noisy results. If the parameter is not specified, a random `seed` value is created.

Shot awareness

Noise model simulation is *shot-aware*; that is, it samples measurements from the output state based on the number of shots provided. In Azure Quantum, the `shots` parameter is submitted with the job, and is required for `aria-1` noise models. If no `shot` value is specified, a default value of `1000` is used. If the `ideal` noise model is used, the `shots` parameter is ignored.

Qubit capacity

While the `ideal` noise model allows you to simulate up to 29 qubits with the IonQ quantum simulator, the hardware specific noise models are limited to the actual qubit capacity of the target hardware, which is 25 qubits for the `aria-1` noise model.

Enabling noise model simulation

On Azure Quantum, noise model simulation can be enabled or disabled for jobs submitted with Q# or with Qiskit.

To enable noise model simulation, add an optional parameter for the target machine, for example:

Python

```
option_params = {
    "noise": {
        "model": "aria-1",   # targets the Aria quantum computer
        "seed" : 1000         # If seed isn't specified, a random value is used
    }
```

```
}
```

> **Note**
>
> If you are also using IonQ's error mitigation, those parameters can be included here, for example:
>
> Python
>
> ```
> option_params = {
>     "error-mitigation": {
>         "debias": False
>     },
>     "noise": {
>     "model": "aria-1",
>     "seed": 1000
>     }
> }
> ```
>
> For more information, see Error mitigation.

Running a job with noise model simulation

You can use the same example program shown previously in error mitigation and add or replace the noise model configuration in `option_params`;

Python

```
option_params = {
    "error-mitigation": {
        "debias": True
    },
    "noise": {
    "model": "aria",
    "seed": 1000
    }
}
```

Then pass the optional parameters when you submit the job:

Python

```
job = MyTarget.submit(MyProgram, "Experiment with noise model simulation", shots =
10, input_params = option_params)
```

```
job.get_results()
```

In Qiskit, you pass the optional parameters to the target machine configuration before submitting the job:

Python

```
circuit.name = "Single qubit random - Debias: True"
backend.options.update_options(**option_params)
job = backend.run(circuit, shots=500)
```

Pricing

To see IonQ billing plan, visit Azure Quantum pricing.

Limits and quotas

IonQ quotas are tracked based on the QPU usage unit, which is *qubit-gate-shot (QGS)*. The resource usage is credited against your account.

Every quantum program consists of N quantum logical gates of one or more qubits, and is executed for a certain number of shots. The number of gate-shots is calculated by the following formula:

$$QGS = N \cdot C$$

where:

- N is the number of one- or two-qubit gates submitted
- C is the number of execution shots requested

IonQ status

For information about IonQ QPU job processing delays, see IonQ status page.

IonQ best practices and connectivity graph

To see recommended best practices for the IonQ QPU, see IonQ Best Practices (ionq.com).

Support policy for IonQ in Azure Quantum

09/18/2025

This article describes the Microsoft support policy that applies when you use the IonQ provider in Azure Quantum. The article applies to any of the targets under this provider.

If you're using the IonQ provider and hit any unexpected issues that you can't troubleshoot yourself, you can contact the Azure Support team for help by creating an Azure support case.

However, there are however some situations where the Azure Support team will need to redirect you to IonQ's support team. Or, you may receive a quicker response by reaching out to IonQ directly at the IonQ Support website.

Frequently asked questions

Q: What is the support policy for using IonQ offerings through Azure Quantum?

Microsoft will provide support for the Azure platform and the Azure Quantum service only. Support for IonQ hardware, simulators, and other products and targets will be provided directly by IonQ. For more information about Azure support, see Azure support plans. For information about IonQ support, please see the IonQ Support website.

Q: What happens if I raise a support issue with the Azure support team and it turns out that a third party provider (like IonQ) needs to troubleshoot the issue further?

The support engineer will create a redirection package for you. This is a PDF document that contains information about your case which you can provide to the IonQ support team. The support engineer will also give you advice and guidance on how to reach out to IonQ to continue troubleshooting. For more information, see IonQ status page.

Q: What happens if I raise a support issue with the IonQ team and it turns out that there is an issue with the Azure Quantum service?

The IonQ support team will help you reach out to Microsoft and provide you with a redirection package. This package is a PDF document that you can use when continuing your support inquiry with the Azure support team.

Third-party information disclaimer

The third-party products that this article discusses are manufactured by companies that are independent of Microsoft. Microsoft makes no warranty, implied or otherwise, about the performance or reliability of these products.

Third-party contact disclaimer

Microsoft provides third-party contact information to help you find additional information about this topic. This contact information may change without notice. Microsoft doesn't guarantee the accuracy of third-party contact information.

Pasqal provider

Pasqal's quantum computers control neutral atoms with optical tweezers, using laser light to manipulate quantum registers with up to a hundred qubits.

- Publisher: Pasqal
- Provider ID: `pasqal`

The following targets available from this provider:

Expand table

Target name	Target ID	Number of qubits	Description
EMU_SV	pasqal.sim.emu-sv	25 qubits 1D and 2D networks	Emulators are backends designed to emulate the dynamics of programmable arrays of neutral atoms.
EMU_MPS	pasqal.sim.emu-mps	80 qubits 1D and 2D networks	Emulators are backends designed to emulate the dynamics of programmable arrays of neutral atoms.
EMU_FREE	pasqal.sim.emu-free	12 qubits 1D and 2D networks	Emulators are backends designed to emulate the dynamics of programmable arrays of neutral atoms.
FRESNEL	pasqal.qpu.fresnel	100 qubits	FRESNEL is a hardware neutral atoms QPU - Orion Beta generation.
FRESNEL_CAN1	pasqal.qpu.fresnel-can1	100 qubits	FRESNEL_CAN1 is a hardware neutral atoms QPU - Orion Beta generation.

EMU_SV

Emulators are backends designed to emulate the dynamics of programmable arrays of neutral atoms.

EMU_SV is a Pulser backend emulating these dynamics using state vectors (SV). State vector representation provides a complete description of the quantum state, enabling highly accurate simulations with GPU acceleration if enabled.

For more information, see the Pasqal EMU_MPS documentation

- Job Type: `Simulation`
- Data Format: `application/json`
- Target ID: `pasqal.sim.emu-sv`

EMU_MPS

Emulators are backends designed to emulate the dynamics of programmable arrays of neutral atoms.

EMU_MPS is a Pulser backend emulating this dynamic with matrix product states (MPS). Matrix Product States (MPS) or tensor train (TT) are a specific class of tensor networks that provide a tractable parametrization of quantum states.

For more information, see the Pasqal EMU_MPS documentation

- Job Type: `Simulation`
- Data Format: `application/json`
- Target ID: `pasqal.sim.emu-mps`

EMU_FREE

Emulators are backends designed to emulate the dynamics of programmable arrays of neutral atoms.

EMU_FREE is a small Pulser backend on which you can emulate small systems (not more than 12 qubits).

- Job Type: `Simulation`
- Data Format: `application/json`
- Target ID: `pasqal.sim.emu-free`

FRESNEL

FRESNEL is a hardware neutral atoms QPU (Quantum Processing Unit) - Orion Beta generation. It is an optical machine at heart, utilizing light to trap and manipulate arrays of Rubidium atoms.

By making use of optical tweezers we can assemble an adjustable quantum register for the atoms which will serve as our computational basis. For the Pasqal machine one single trapped atom corresponds to one qubit.

- Job Type: `Quantum program`

- Data Format: `application/json`
- Target ID: `pasqal.qpu.fresnel`

FRESNEL_CAN1

FRESNEL_CAN1 is a hardware neutral atoms QPU (Quantum Processing Unit) - Orion Beta generation.

It is an optical machine at heart, utilizing light to trap and manipulate arrays of Rubidium atoms.

By making use of optical tweezers we can assemble an adjustable quantum register for the atoms which will serve as our computational basis. For the Pasqal machine one single trapped atom corresponds to one qubit.

- Job Type: `Quantum program`
- Data Format: `application/json`
- Target ID: `pasqal.qpu.fresnel-can1`

Pulser SDK

In Pasqal QPU, individual atoms are trapped at well-defined positions in 1D or 2D lattices. Pulser is a framework for composing, simulating and executing pulse sequences on neutral atoms quantum devices. For more information, see Pulser documentation.

To install Pulser SDK packages, run the following code:

Python

```
!pip -q install pulser-simulation #Only for using the local Qutip emulator included in Pulser
!pip -q install pulser-core
```

Input data format

Pasqal targets accept JSON files as input data format. To submit the pulse sequences, you need to convert the Pulser objects into a JSON string that can be used as input data.

Python

```
# Convert the sequence to a JSON string
def prepare_input_data(seq):
    input_data = {}
    input_data["sequence_builder"] = json.loads(seq.to_abstract_repr())
```

```
    to_send = json.dumps(input_data)
    #print(json.dumps(input_data, indent=4, sort_keys=True))
    return to_send
```

Before submitting your quantum job to Pasqal, you need to set proper input and output data format parameters. For example, the following code sets the input data format to `pasqal.pulser.v1` and the output data format to `pasqal.pulser-results.v1`.

Python

```
# Submit the job with proper input and output data formats
def submit_job(target, seq):
    job = target.submit(
        input_data=prepare_input_data(seq), # Take the JSON string previously defined
as input data
        input_data_format="pasqal.pulser.v1",
        output_data_format="pasqal.pulser-results.v1",
        name="Pasqal sequence",
        shots=100 # Number of shots
    )
```

For more information about how to submit jobs to the Pasqal provider, see Submit a circuit to Pasqal using Pulser SDK.

Pricing

To see the Pasqal billing plan, visit Azure Quantum pricing.

Limits and quotas

Pasqal quotas apply to the usage of the emulator and QPU and can be increased with a support ticket. To see your current limits and quotas, go to the **Operations** section and select the **Quotas** blade of your workspace on the Azure portal. Refer to Azure Quantum quotas for more information.

Last updated on 04/23/2026

Support policy for Pasqal in Azure Quantum

This article describes the Microsoft support policy that applies when you use the Pasqal provider in Azure Quantum. The article applies to any of the targets offered by Pasqal.

Azure support

In general, problems using Pasqal on Microsoft Azure Quantum are best addressed with the Azure Quantum support team directly by creating an Azure support case. This team has all the necessary details to isolate the issue you're experiencing and direct the report to the right point of contact. The Azure support team also has the means to reach Pasqal support outside of business hours for the most urgent issues.

Pasqal support

In some situations, the Azure Support team will need to redirect you to Pasqal's support team. You may receive a quicker response by reaching out to Pasqal directly.

Support timeline

Pasqal will always attempt to resolve issues with the Microsoft Azure integration as quickly as practical. Some issues will take longer to address than others, but the tables below indicate the time you may expect for an initial response from Pasqal after an issue is reported.

Expand table

Severity	Definition	Response SLA
High	Failure in connectivity, complete performance degradation	One business day
Medium	Performance degradation	Two business days
Low	Application related concerns and questions	Three business days

Last updated on 04/23/2026

Quantinuum provider

Quantinuum provides access to trapped-ion systems with high-fidelity, fully connected qubits, and the ability to perform mid-circuit measurement.

- Publisher: Quantinuum
- Provider ID: `quantinuum`

Targets

The following targets are available from this provider:

Expand table

Target name	Target ID	Number of qubits	Description
H2-1 Syntax Checker	quantinuum.sim.h2-1sc	56 qubits	Validates quantum programs against the H2-1 compiler before submitting to hardware or emulators on Quantinuum's platform. Free of cost.
H2-2 Syntax Checker	quantinuum.sim.h2-2sc	56 qubits	Validates quantum programs against the H2-2 compiler before submitting to hardware or emulators on Quantinuum's platform. Free of cost.
H2-1 Emulator	quantinuum.sim.h2-1e	56/32 qubits	Uses a realistic physical model and noise model of H2-1. 56 qubit simulation is only available as a stabilizer simulation
H2-2 Emulator	quantinuum.sim.h2-2e	56/32 qubits	Uses a realistic physical model and noise model of H2-2. 56 qubit simulation is only available as a stabilizer simulation
H2-1	quantinuum.qpu.h2-1	56 qubits	Quantinuum's H2-1 trapped ion device.
H2-2	quantinuum.qpu.h2-2	56 qubits	Quantinuum's H2-2 trapped ion device.

Quantinuum's targets correspond to a **QIR Adaptive RI** profile. For more information about this target profile and its limitations, see Understanding target profile types in Azure Quantum.

All of Quantinuum's targets now support Integrated hybrid circuits. For more information about submitting integrated hybrid jobs, see Integrated hybrid computing.

> **Tip**
>
> Quantum jobs submitted under a session have **exclusive access** to Quantinuum hardware as long as you queue jobs within one minute from each other. After that, any job is accepted and handled with the standard queueing and prioritization logic. For more information, see sessions in Azure Quantum.

Syntax Checkers

We recommend that users first use a Syntax Checker to validate their code. Syntax Checkers verify proper syntax, compilation completion, and machine compatibility. Syntax Checkers use the same compiler as the quantum computer that they target. For example, the H2-1 syntax checker uses the same compiler as H2-1. The full compilation stack is executed, except for the actual quantum operations. If the code compiles, then the syntax checker returns a `success` status and a result of all 0s. If the code doesn't compile, then the syntax checker returns a failed status and gives an error message to help users debug their circuit syntax. Syntax Checkers allow developers to validate their code at any time, even when machines are offline.

- Job type: `Simulation`
- Data Formats: `honeywell.openqasm.v1`, `honeywell.qir.v1`
- Target ID:
  - H2-1 Syntax Checker: `quantinuum.sim.h2-1sc`
  - H2-2 Syntax Checker: `quantinuum.sim.h2-2sc`
- Target Execution Profile: QIR Adaptive RI

Syntax Checkers usage is offered free-of-charge.

System Model H2 Emulator

After validating the syntax of their code with the H2-1 Syntax Checker, users can take advantage of Quantinuum's System Model H2 Emulator, an emulation tool which contains a detailed physical model and realistic noise model of the actual System Model H2 hardware. More information about the noise model can be found in the *System Model H2 Emulator Product Data Sheet* found on the System Model H2 page. The System Model H2 Emulator uses an identical API for job submission as the System Model H2 hardware, enabling seamless transition from emulation to hardware. To help maximize productivity and shorten development time, the H2 Emulator is available even while the hardware is offline.

- Job type: `Simulation`
- Data Format: `quantinuum.openqasm.v1`
- Target ID:
  - H2-1 Emulator: `quantinuum.sim.h2-1e`
  - H2-2 Emulator: `quantinuum.sim.h2-2e`
- Target Execution Profile: QIR Adaptive RI

System Model H2 Emulator usage is offered free-of-charge with a hardware subscription. For details, see Azure Quantum pricing.

System Model H2

The Quantinuum System Model H2 generation of quantum computers, Powered by Honeywell, is composed of a Quantum charge-coupled device (QCCD) with two connected linear sections and currently has 1 machine, the H2-1. More information can be found in the *System Model H2 Product Data Sheet* found on the System Model H2 page. Users are encouraged to test compatibility of their code by submitting jobs to a syntax checker and System Model H2 Emulator prior to submitting them to the target machines.

If a user submits a job to the H2-1 machine and the H2-1 machine is not available, the job remains in that machine's queue until the machine becomes available.

System Model H2 hardware is continuously upgraded throughout its product lifecycle. Users are given access to the most up-to-date, advanced, and capable hardware available.

- Job type: `Quantum Program`
- Data Format: `quantinuum.openqasm.v1`
- Target ID:
  - H2-1: `quantinuum.qpu.h2-1`
  - H2-2: `quantinuum.qpu.h2-2`
- Target Execution Profile: QIR Adaptive RI

System Model H2 Technical Specifications

Technical details for System Model H2 can be found in Quantinuum's product data sheets on the System Model H2 page alongside links to Quantinuum specification and quantum volume data repositories and how to cite usage of Quantinuum systems.

Additional Capabilities

Additional capabilities available via the Quantinuum API are listed here.

Expand table

Capability	Description
Mid-Circuit Measurement and Reset (MCMR)	Measure qubits in the middle of a circuit and reuse them
Arbitrary Angle ZZ Gates	Directly perform 2-qubit arbitrary angle gate rotations
General SU(4) Entangling Gate	Directly perform 2-qubit arbitrary angle gate rotations
Emulator Noise Parameters	Experiment with the noise parameters used in the Quantinuum emulators
TKET Optimizations in Quantinuum Stack	Experiment with turning on different levels of TKET optimizations in the Quantinuum stack

Users can take advantage of these additional capabilities via circuit functions or pass-through parameters in the Azure Quantum Q# and Qiskit providers.

Mid-circuit Measurement and Reset

Mid-circuit Measurement and Reset (MCMR) enables users to measure qubits in the middle of a circuit and reset them. This enables functionality for quantum error correction as well as the ability to reuse qubits within the circuit.

Due to the internal level structure of trapped-ion qubits, a mid-circuit measurement may leave the qubit in a non-computational state. All mid-circuit measurements should be followed by a reset if the qubit is to be used again in that circuit. The following code examples demonstrate this.

When a subset of qubits is measured in the middle of the circuit, the classical information from these measurements can be used to condition future elements of the circuit. The examples also highlight this usage.

For information on MCMR in Quantinuum systems, see the MCMR page on Quantinuum Docs.

MCMR with Q# Provider

In Q#, the `MResetZ` function can be used both to measure a qubit and reset it. For more information on this function, see MResetZ in the Q# documentation.

Q#

```
%%qsharp
import Std.Measurement.*;

operation ContinueComputationAfterReset() : Result[] {
    // Set up circuit with 2 qubits
    use qubits = Qubit[2];

    // Perform Bell Test
    H(qubits[0]);
    CNOT(qubits[0], qubits[1]);

    // Measure Qubit 1 and reset it
    let res1 = MResetZ(qubits[1]);

    // Continue additional computation, conditioned on qubits[1] measurement
outcome
    if res1 == One {
        X(qubits[0]);
    }
    CNOT(qubits[0], qubits[1]);

    // Measure qubits and return results
    let res2 = Measure([PauliZ, PauliZ], qubits);
    return [res1, res2];
}
```

Arbitrary Angle ZZ Gates

Quantinuum's native gate set includes arbitrary angle ZZ gates. This is beneficial for reducing the 2-qubit gate count for many quantum algorithms and gate sequences. For information on Arbitrary Angle ZZ gates in Quantinuum systems, Parameterized Angle ZZ Gates page on Quantinuum Docs.

Arbitrary Angle ZZ Gates with Q# Provider

In Q#, the arbitrary angle ZZ gate is implemented with the `Rzz` operation.

Q#

```
%%qsharp
import Std.Intrinsic.*;
import Std.Measurement.*;
import Std.Arrays.*;

operation ArbitraryAngleZZExample(theta : Double) : Result[] {
```

```
    // Set up circuit with 2 qubits
    use qubits = Qubit[2];

    // Create array for measurement results
    mutable resultArray = [Zero, size = 2];

    H(qubits[0]);
    Rz(theta, qubits[0]);
    Rz(theta, qubits[1]);
    X(qubits[1]);

    // Add Arbitrary Angle ZZ gate
    Rzz(theta, qubits[0], qubits[1]);

    // Measure qubits and return results
    for i in IndexRange(qubits) {
        resultArray w/= i <- M(qubits[i]);
    }

    return resultArray;
}
```

General SU(4) Entangling Gate

Quantinuum's native gate set includes a general SU(4) entangling gate. Note that quantum circuits submitted to the hardware are rebased to the fully entangling ZZ gate and the arbitrary angle RZZ gate. Circuits are only rebased to the General SU(4) Entangling gate if users opt into it. For information on the General SU(4) Entangler in Quantinuum systems, see the Parameterized Angle SU(4) Gates page on Quantinuum Docs.

General SU(4) Entangling Gate with Q# Provider

In Q#, the General SU(4) Entangling gate is implemented via Quantinuum's QIR profile. To use it, define a function with a custom intrinsic matching the QIR profile signature, and use this function within the `SU4Example` operation.

To ensure the circuit runs with the General SU(4) Entangling gate, pass the following options in the Quantinuum stack:

- `nativetq: Rxxyyzz` to prevent rebasing to other native gates.
- `noreduce: True` to avoid additional compiler optimizations (optional).

Q#

```
%%qsharp
import Std.Math.*;
```

```
operation __quantum__qis__rxxyyzz__body(a1 : Double, a2 : Double, a3 : Double, q1
: Qubit, q2 : Qubit) : Unit {
    body intrinsic;
}

operation SU4Example() : Result[] {
    use qs = Qubit[2];

    // Add SU(4) gate
    __quantum__qis__rxxyyzz__body(PI(), PI(), PI(), qs[0], qs[1]);

    MResetEachZ(qs)
}
```

Now compile the operation:

Python

```
from qdk import qsharp

MyProgram = qsharp.compile("GenerateRandomBit()")
```

Connect to Azure Quantum, select the target machine, and configure the noise parameters for the emulator:

Python

```
from qdk.azure import Workspace

MyWorkspace = Workspace(resource_id="")

MyTarget = MyWorkspace.get_targets("quantinuum.sim.h2-1e")

# Update TKET optimization level desired
option_params = {
    "nativetq": `Rxxyyzz`,
    "noreduce": True
}
```

Pass in the `noreduce` option when submitting the job:

Python

```
job = MyTarget.submit(MyProgram, "Submit a program with SU(4) gate", shots = 10,
input_params = option_params)
job.get_results()
```

Emulator Noise Parameters

Users have the option of experimenting with the noise parameters of the Quantinuum emulators. **Only a few of the available noise parameters are highlighted** here demonstrating how to pass through the parameters in the Azure Quantum providers.

For more information on the full set of noise parameters available, see the Quantinuum Emulators page on Quantinuum docs.

Emulator Noise Parameters with Q# Provider

First, import the required packages and initiate the base profile:

Python

```
from qdk import qsharp
from qdk.azure import Workspace

qsharp.init(target_profile=qsharp.TargetProfile.Base)
```

Next, define the function.

Q#

```
%%qsharp
import Std.Measurement.*;
import Std.Arrays.*;
import Std.Convert.*;

operation GenerateRandomBit() : Result {
    use target = Qubit();

    // Apply an H-gate and measure.
    H(target);
    return M(target);
}
```

and compile the operation:

Python

```
MyProgram = qsharp.compile("GenerateRandomBit()")
```

Connect to Azure Quantum, select the target machine, and configure the noise parameters for the emulator:

Python

```
MyWorkspace = Workspace(resource_id="")

MyTarget = MyWorkspace.get_targets("quantinuum.sim.h2-1e")

# Update the parameter names desired
# Note: This is not the full set of options available.
# For the full set, see the Quantinuum Emulators page
option_params = {
    "error-params": {
        "p1": 4e-5,
        "p2": 3e-3,
        "p_meas": [3e-3, 3e-3],
        "p_init": 4e-5,
        "p_crosstalk_meas": 1e-5,
        "p_crosstalk_init": 3e-5,
        "p1_emission_ratio": 6e-6,
        "p2_emission_ratio": 2e-4
    }
}
```

Pass in the emulator noise options when submitting the job:

Python

```
job = MyTarget.submit(MyProgram, "Experiment with Emulator Noise Parameters",
                      shots = 10,
                      input_params = option_params)
job.get_results()
```

To turn off the emulator noise model, set the `error-model` option to `False`. By default this is set to `True`.

Python

```
option_params = {
    "error-model": False
}
```

To use the stabilizer emulator, set the `simulator` option to `stabilizer`. By default this is set to `state-vector`.

Python

```
option_params = {
    "simulator": "stabilizer"
}
```

TKET Compilation in Quantinuum Stack

Circuits submitted to Quantinuum systems, **except for integrated hybrid submissions**, are automatically run through TKET compilation passes for Quantinuum hardware. This enables circuits to be automatically optimized for Quantinuum systems and run more efficiently.

More information on the specific compilation passes applied can be found in the pytket-quantinuum documentation, specifically the pytket-quantinuum Compilation Passes section.

In the Quantinuum software stack, the optimization level applied is set with the `tket-opt-level` parameter. *The default compilation setting for all circuits submitted to Quantinuum systems is optimization level 2.*

Users who would like to experiment with the TKET compilation passes and see what optimizations would apply to their circuits *before* submitting any jobs can see the *Quantinuum_compile_without_api.ipynb* notebook in the pytket-quantinuum Examples folder.

To turn TKET compilation in the stack off, a different option, `no-opt`, can be set to `True` inside `option_params`. For example, `"no-opt": True`.

For more information on `pytket`, see the following links:

- pytket
- pytket User Manual

TKET Compilation with Q# Provider

First, import the required packages and initiate the base profile:

Python

```
from qdk import qsharp
from qdk.azure import Workspace
```

```
qsharp.init(target_profile=qsharp.TargetProfile.Base)
```

Next, define the function.

Q#

```
%%qsharp
import Std.Measurement.*;
import Std.Arrays.*;
import Std.Convert.*;

operation GenerateRandomBit() : Result {
    use target = Qubit();

    // Apply an H-gate and measure.
    H(target);
    return M(target);
}
```

and compile the operation:

Python

```
MyProgram = qsharp.compile("GenerateRandomBit()")
```

Connect to Azure Quantum, select the target machine, and configure the noise parameters for the emulator:

Python

```
MyWorkspace = Workspace(resource_id="")

MyTarget = MyWorkspace.get_targets("quantinuum.sim.h2-1e")

# Update TKET optimization level desired
option_params = {
    "tket-opt-level": 1
}
```

Pass in the optimization option when submitting the job:

Python

```
job = MyTarget.submit(MyProgram, "Experiment with TKET Compilation", shots = 10,
input_params = option_params)
```

```
job.get_results()
```

Technical Specifications

Technical details for the System Model H2 and System Model H2 Emulators can be found on the Hardware User Guide page alongside Quantinuum specification and quantum volume data repositories and How to cite Quantinuum Systems.

Target Availability

The Quantinuum quantum computers are designed to be continuously upgraded, which allows customers to have access to the latest hardware capabilities as Quantinuum continually improves gate fidelities, memory errors, and system speed.

Quantinuum hardware cycles through commercial periods and development periods. During commercial periods, the hardware is available to process jobs via a queue system. During development periods, the hardware is offline as upgrades are applied.

Every month, a calendar is sent to Quantinuum users with information on the commercial and development periods. If you have not received this calendar, please email QCsupport@quantinuum.com.

A target's status indicates its current ability to process jobs. The possible states of a target include:

- **Available**: The target is online, processing submitted jobs and accepting new ones.
- **Degraded**: The target is accepting jobs, but not currently processing them.
- **Unavailable**: The target is offline, not accepting new job submissions.

For the Quantinuum quantum computer targets, **Available** and **Degraded** correspond to commercial periods, while **Unavailable** corresponds to development periods where the machine is offline for upgrades.

Current status information may be retrieved from the *Providers* tab of a workspace on the Azure portal.

Pricing

To see Quantinuum's billing plans, visit Azure Quantum pricing.

Limits and quotas

Quantinuum's quotas are tracked based on the QPU usage credit unit, *Hardware Quantum Credit (HQC)*, for jobs submitted to Quantinuum quantum computers, and emulator HQCs (eHQCs) for jobs submitted to emulators.

HQCs and eHQCs are used to calculate the cost of running a job, and they are calculated based on the following formula:

$$HQC = 5 + C(N_{1q} + 10N_{2q} + 5N_m)/5000$$

where:

- N_{1q} is the number of one-qubit operations in a circuit.
- N_{2q} is the number of native two-qubit operations in a circuit. Native gate is equivalent to CNOT up to several one-qubit gates.
- N_m is the number of state preparation and measurement (SPAM) operations in a circuit including initial implicit state preparation and any intermediate and final measurements and state resets.
- C is the shot count.

> **Note**
>
> The total cost in HQCs includes all gates and measurements across any conditional branches or control flows. This may have a higher impact on integrated hybrid jobs.

Quotas are based on plan selection and can be increased with a support ticket. To see your current limits and quotas, go to the **Operations** section and select the **Quotas** blade of your workspace on the Azure portal. For more information, see Azure Quantum quotas.

Last updated on 05/15/2026

Support policy for Quantinuum in Azure Quantum

Article • 02/14/2025

This article describes the Microsoft support policy that applies when you use the Quantinuum provider in Azure Quantum. The article applies to any of the targets under this provider.

If you're using the Quantinuum provider and hit unexpected issues, you can contact the Azure Support team for help by creating an Azure support case.

In some situations, the Azure Support team needs to redirect you to Quantinuum's support team. You may receive a quicker response by reaching out to Quantinuum directly at QCsupport@quantinuum.com.

Quantinuum support policy - Public Preview

The goal of support is to identify and remedy defects or malfunctions causing the Quantinuum hardware target to fail to perform in accordance with the agreed specifications and documentation (“Problems”). Support only covers the current released version generally available to customers. Although our quantum experts can help you troubleshoot algorithm issues, we're not responsible for any issues, problems, or defects with your algorithm. To minimize programming issues, users are able to run their algorithms through the syntax checker prior to running on hardware.

1.1 Contacting technical support within Quantinuum

To create a support request with Quantinuum, submit an incident report via email to QCsupport@quantinuum.com. The on-call engineer will respond within the appropriate SLA (Table 2). The Quantinuum team only receives incident reports that contain all of the following information and have the appropriate SLA resolution window. Reports without the necessary information don't trigger this response. All requests must be provided in the English language and are answered in the English language.

Table 1: Support contact information and standard operating hours

Expand table

Email	Standard operating hours
QCsupport@quantinuum.com	8am – 5pm MST on business days[1]

[1] *"Business days" means Monday to Friday but excluding national, legal, or bank holidays.*

The incident report requires the following information:

- **Description**: A detailed description of the issue you're facing, what you're observing, and any steps you have already taken to triage/understand the issue
- **Primary contact**: A primary contact name and phone number in case we require more information
- **Incident severity**: the severity of the incident, according to our severity definitions

Once a report is filed, the Quantinuum team is notified and responds within the required SLA window. You'll receive an acknowledgment of the issue once it has been received and the on-call engineer has been notified. The on-call engineer may request more information, or may call you at the contact number listed in the report depending on the severity.

During Reserved (dedicated) Sessions, Quantinuum provides at least one quantum specialist available to you for customer support for the entirety of your dedicated session. Quantinuum reserves the right to change out the specialist during your session, but will use commercially reasonable efforts to ensure the new specialist is prepared to continue providing support.

During queued runs, if your job is running in the general queue and you encounter an issue, you may submit your issue request online at: QCsupport@quantinuum.com. You can expect to receive an answer within three (3) business days.

1.2 Quantinuum severity index and response SLA

If the Quantinuum engineer determines that the issue doesn't fall within the bounds of our incident response commitments, or doesn't have the right severity classification, they may opt at their sole discretion to adjust the severity and/or end support if the new severity doesn't require it.

Certain problems may be easier to address than others, and we may not be able to completely resolve the problem with our initial response. If we're unable to reasonably resolve the problem, we'll make a good faith effort to give an assessment of the issue and an estimated time for resolution.

Table 2: Definition of severity index

Expand table

Severity	Meaning	Description
Severity 1	Significant impact	System performance degraded below minimum accepted level
Severity 2	Urgent or high impact	Job status hasn't updated; issues retrieving data
Severity 3	Not urgent	Miscellaneous

The following table shows the Response SLA that is adhered to by Quantinuum for incident reports of the corresponding severity. Note that these define how fast we'll respond to issues – resolution times aren't guaranteed.

Table 3: Response SLA for various severity

Expand table

Severity	Response SLA
Severity 1	Business hours response within 1 hour
Severity 2	Business hours response within 1 business day
Severity 3	Business hours response within 3 business days

Frequently asked questions

Q: What happens if I raise a support issue with the Azure support team and it turns out that a third party provider (like Quantinuum) needs to troubleshoot the issue further?

The support engineer creates a redirection package for you. This is a PDF document that contains information about your case which you can provide to the Quantinuum support team. The support engineer also gives you advice and guidance on how to reach out to Quantinuum to continue troubleshooting.

Q: What happens if I raise a support issue with the Quantinuum team and it turns out that there is an issue

with the Azure Quantum service?

The Quantinuum support team helps you reach out to Microsoft and provide you with a redirection package. This is a PDF document that you can use when continuing your support inquiry with the Azure support team.

Third-party information disclaimer

The third-party products that this article discusses are manufactured by companies that are independent of Microsoft. Microsoft makes no warranty, implied or otherwise, about the performance or reliability of these products.

Third-party contact disclaimer

Microsoft provides third-party contact information to help you find additional information about this topic. This contact information may change without notice. Microsoft doesn't guarantee the accuracy of third-party contact information.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Rigetti provider

Rigetti quantum processors are universal, gate-model machines based on tunable superconducting qubits. System features and device characteristics include enhanced readout capabilities, a speedup in quantum processing times, fast gate times for multiple entangling gate families, rapid sampling via active register reset, and parametric control.

- Publisher: Rigetti
- Provider ID: `rigetti`

The Rigetti provider makes the following targets available:

Expand table

Target name	Target ID	Number of qubits	Description
Quantum Virtual Machine (QVM)	rigetti.sim.qvm	-	Open-source simulator for Quil, Q#, and Qiskit programs. Free of cost.
Cepheus-1-108Q	rigetti.qpu.cepheus-1-108q	108 qubits	

> **Note**
>
> Rigetti simulators and hardware targets don't support Cirq programs.

Rigetti's targets correspond to a **QIR Base** profile. For more information about this target profile and its limitations, see Understanding target profile types in Azure Quantum.

Quantum computers

All of Rigetti's publicly available QPUs are available through Azure Quantum. This list is subject to change without advance notice.

Cepheus-1-108Q

A 108-qubit quantum processor built from an array of 9-qubit chips tiled together.

- Job Type: `Quantum Program`

- Data Format: `rigetti.quil.v1`, `rigetti.qir.v1`
- Target ID: `rigetti.qpu.cepheus-1-108q`
- Target Execution Profile: QIR Base

Simulators

The Quantum Virtual Machine (QVM) is an open-source simulator for Quil. The `rigetti.sim.qvm` target accepts a Quil program as text and runs that program on QVM hosted in the cloud, returning simulated results.

- Job Type: `Simulation`
- Data Formats: `rigetti.quil.v1`, `rigetti.qir.v1`
- Target ID: `rigetti.sim.qvm`
- Target Execution Profile: QIR Base
- Pricing: Free ($0)

Pricing

To see Rigetti's billing plan, visit Azure Quantum pricing.

Input format

All Rigetti targets currently accept two formats:

- `rigetti.quil.v1`, which is the text of a Quil program.
- `rigetti.qir.v1`, which is QIR bitcode.

All targets also take the optional `count` integer parameter for defining the number of shots to run. If omitted, the program only runs once.

Quil

All Rigetti targets accept the `rigetti.quil.v1` input format, which is the text of a Quil program, which stands for Quantum Instruction Language. By default, programs are compiled using quilc before running. However, quilc doesn't support the pulse level control features (Quil-T), so if you want to use those features you must provide a Native Quil program (also containing Quil-T) as input and specify the input parameter `skipQuilc: true`.

To make constructing a Quil program easier, you can use pyQuil along with the pyquil-for-azure-quantum package. Without this package, `pyQuil` can be used to *construct* Quil

programs but not to submit them to Azure Quantum.

QIR

All Rigetti hardware supports the execution of Quantum Intermediate Representation (QIR) compliant jobs with the QIR Base Profile, v1 as `rigetti.qir.v1`. QIR provides a common interface that supports many quantum languages and target platforms for quantum computation and enables communication between high-level languages and machines. For example, you can submit Q#, Quil, or Qiskit jobs to Rigetti hardware, and Azure Quantum automatically handles the input for you. For more information, see Quantum Intermediate Representation.

Selecting the right input format

Should you use Quil or another QIR compliant language? It comes down to your end use case. QIR is more accessible for many users, while Quil is more powerful today.

If you're using Qiskit, Q#, or another toolkit that supports QIR generation, and your application works on Rigetti targets via Azure Quantum, then QIR is right for you! QIR has a rapidly evolving specification, and Rigetti continues to increase support for more advanced QIR programs as time passes - what can't be compiled today may well compile tomorrow.

On the other hand, Quil programs express the full set of functionality available to users of Rigetti systems from any platform including Azure Quantum. If you're looking to tailor the decomposition of your quantum gates or write programs at the pulse level, then you'll want to work in Quil, because those capabilities aren't yet available through QIR.

Examples

The easiest way to submit Quil jobs is using the pyquil-for-azure-quantum package, as it allows you to use the tools and documentation of the pyQuil library.

You can also construct Quil programs manually and submit them using the `azure-quantum` package directly.

Use pyquil-for-azure-quantum

Python

```python
from pyquil.gates import CNOT, MEASURE, H
from pyquil.quil import Program
from pyquil.quilbase import Declare
from pyquil_for_azure_quantum import get_qpu, get_qvm

# Note that some environment variables must be set to authenticate with Azure
Quantum
qc = get_qvm()  # For simulation

program = Program(
    Declare("ro", "BIT", 2),
    H(0),
    CNOT(0, 1),
    MEASURE(0, ("ro", 0)),
    MEASURE(1, ("ro", 1)),
).wrap_in_numshots_loop(5)

# Optionally pass to_native_gates=False to .compile() to skip the compilation
stage
result = qc.run(qc.compile(program))
data_per_shot = result.readout_data["ro"]
# Here, data_per_shot is a numpy array, so you can use numpy methods
assert data_per_shot.shape == (5, 2)
ro_data_first_shot = data_per_shot[0]
assert ro_data_first_shot[0] == 1 or ro_data_first_shot[0] == 0

# Let's print out all the data
print("Data from 'ro' register:")
for i, shot in enumerate(data_per_shot):
    print(f"Shot {i}: {shot}")
```

Last updated on 04/10/2026

Support policy for Rigetti in Azure Quantum

Article • 09/16/2024

This article describes the Microsoft support policy that applies when you use the Rigetti provider in Azure Quantum. The article applies to any of the targets offered by the Rigetti provider.

Support points of contact

Azure support

In general, problems using the Rigetti provider on Microsoft Azure Quantum are best addressed with the Azure Quantum support team directly by creating an Azure support case. This team has all the necessary details to isolate the issue you're experiencing and direct the report to the right point of contact. The Azure support team also has the means to reach Rigetti support outside of business hours for the most urgent issues.

Rigetti support

In some situations, the Azure Support team will need to redirect you to Rigetti's support team. You may receive a quicker response by reaching out to Rigetti directly at https://rigetti.zendesk.com/hc/.

Support timeline

Rigetti will always attempt to resolve issues with the Microsoft Azure integration as quickly as practical. Some issues will take longer to address than others, but the tables below indicate the time you may expect for an initial response from Rigetti after an issue report.

Expand table

Severity	Definition	Response SLA
High	Failure in connectivity; complete performance degradation	1 business day
Medium	Performance degradation	2 business days

Severity	Definition	Response SLA
Low	Application-related concerns and questions	5 business days

Business days are defined as Monday through Friday, not counting US holidays. For example, if a medium-severity incident is reported on a Tuesday, a response will be returned by the end of the business day on Thursday, if the intervening days are also business days.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Azure Quantum preview API life cycle

The Azure Quantum service preview APIs (APIs that end in `-preview`) have a lifespan of approximately 90 days after a new preview API version is released. For example, the `2026-01-15-preview` Data Plane API will be deprecated about 90 days after the next Date Plane API is released.

After an API version is deprecated, that version no longer functions. To avoid issues with deprecated APIs, routinely do the following:

- Update your ARM/BICEP templates that use preview API versions to use the latest version of the preview API.
- Update your Azure Quantum preview CLI extension to the latest version.
- Update preview SDKs or other tools that are built on preview APIs to the latest version of the preview API.

Perform these updates at least every 6-9 months. If you don't update your preview API versions, then you receive a notification that you're using a soon-to-be deprecated version as the deprecation date approaches.

How to check what preview API versions you're using

If you're unsure what client or tool is using a specific API version, then check the activity logs using the following commands. Set the API version that you want to inspect for recent usage in the activity log. These examples checks for the `2024-10-01-preview` API version.

Bash

Bash

```bash
export API_VERSION="2024-10-01-preview"
az monitor activity-log list --offset 30d --max-events 10000 --namespace
microsoft.quantum --query "[?eventName.value == 'EndRequest' &&
contains(not_null(httpRequest.uri,''), '$API_VERSION')]"
```

Windows PowerShell

PowerShell

```
$env:API_VERSION = "2024-10-01-preview"
az monitor activity-log list --offset 30d --max-events 10000 --namespace
microsoft.quantum --query "[?eventName.value == 'EndRequest' &&
contains(not_null(httpRequest.uri,''), '$API_VERSION')]"
```

How to update to a newer version of the Data Plane API

Use the following table to update your Data Plane APIs.

Expand table

Tool	How to update
Azure SDKs	Update to a newer version of the SDK.
Azure CLI	Run `az upgrade` and `az extension update --name "quantum"` to update the CLI and the `quantum` extension.
Other tools	Update the tool to the latest version.

Upcoming deprecation schedule

Expand table

API type	API version	Announce Date	Deprecation Date
Data Plane	2026-01-15-preview		
Resource Manager	2026-06-15-preview		

Past deprecations

Expand table

API type	API version	Announce Date	Deprecation Date
Data Plane	2019-11-04-preview	June 1, 2026	August 1, 2026
Data Plane	2021-05-06-preview	June 1, 2026	August 1, 2026
Data Plane	2021-11-01-preview	June 1, 2026	August 1, 2026
Data Plane	2022-09-12-preview	June 1, 2026	August 1, 2026

API type	API version	Announce Date	Deprecation Date
Data Plane	2023-11-13-preview	June 1, 2026	August 1, 2026
Data Plane	2024-03-01-preview	June 1, 2026	August 1, 2026
Data Plane	2024-10-01-preview	June 1, 2026	August 1, 2026
Data Plane	2025-09-01-preview	June 1, 2026	August 1, 2026
Data Plane	2025-12-01-preview	June 1, 2026	August 1, 2026
Resource Manager	2019-11-04-preview	June 1, 2026	August 1, 2026
Resource Manager	2022-01-10-preview	June 1, 2026	August 1, 2026
Resource Manager	2023-11-13-preview	June 1, 2026	August 1, 2026
Resource Manager	2025-01-01-preview	June 1, 2026	August 1, 2026
Resource Manager	2025-08-11-preview	June 1, 2026	August 1, 2026
Resource Manager	2025-11-01-preview	June 1, 2026	August 1, 2026
Resource Manager	2025-12-15-preview	June 1, 2026	August 1, 2026

Last updated on 09/03/2026

Troubleshooting issues in Azure Quantum

When you work with the Azure Quantum service, you might experience connection or job-related issues. This article explains how to troubleshoot these issues.

Workspace connection issues

Issue: I can't authenticate to Azure Quantum with `pytket-azure`

When you try to authenticate to Azure Quantum with the `pytket-azure` package in a CI environment using the environment variables `"AZURE_TENANT_ID"`, `"AZURE_CLIENT_ID"`, and `"AZURE_CLIENT_SECRET"`, you might encounter the following error:

Windows Command Prompt

```
Code: InsufficientPermissions
Message: There are not enough permissions to perform this operation.
```

To resolve this issue, use a connection string and the environment variable `"AZURE_QUANTUM_CONNECTION_STRING"` to authenticate instead. For more information, see Connect with a connection string.

Python

```python
connection_string = "" # Add your connection string

import os

os.environ["AZURE_QUANTUM_CONNECTION_STRING"] = connection_string
```

Job submission issues

Issue: I can't find the target that I want to submit my job to

If the Azure Quantum target that you want to run your job on isn't in the available target list, then update to the latest version of the Quantum Development Kit (QDK) for Visual Studio Code (VS Code). For more information, see Update the QDK.

Issue: Operation returns an invalid status code 'Unauthorized'

To resolve this issue, follow these steps:

1. Sign in to the Azure portal and authenticate your account.
2. Go the Quantum workspace that you're trying to submit a job to.
3. In the workspace navigation pane, select **Access control (IAM)**.
4. Select the **View my access** button. The **assignments** pane opens.
5. In the **Role** column of the **Role assignments** table, check whether you have either the **Owner** or **Contributor** role.
6. If you don't have either role, ask your subscription administrator to assign you the **Owner** or **Contributor** role for this workspace.

Issue: "AuthorizationFailure - This request is not authorized to perform this operation"

If you get this message even when you have a valid connection to the Azure Quantum service, then the storage account might be configured to block public network access. The Azure Quantum service only supports storage accounts that have public internet access.

To check the storage account settings, follow these steps:

1. Sign in to Azure portal.
2. Go to your Quantum workspace.
3. In the **Overview** page, select the **Storage account**.
4. In the navigation pane, expand the **Security + networking** dropdown and select **Networking**.
5. In the **Public network access** section of the **Public access** tab, select the **Manage** button.
6. For the **Public network access** setting, select **Enable**. For the **Public network access scope** setting, select **Enable from all networks**.
7. Select the **Save** button.

Issue: "Failed to compile program" when you submit a Q# program from Azure CLI

When you submit a job with the `az quantum submit` Azure CLI command, you might encounter the following error message:

Azure CLI

```
az quantum job submit ...
Failed to compile program.
```

```
Command ran in 21.181 seconds (init: 0.457, invoke: 20.724)
```

This error occurs when there's a problem with the Q# program that causes the compilation to fail. Make sure that your Q# code has proper syntax.

Issue: Compiler error "Wrong number of gate parameters"

When you submit a job to Quantinuum from a local Jupyter Notebook or CLI environment, and you're using the legacy QASM (OPENQASM 2.0) translator, then you might encounter this error:

Windows Command Prompt

```
Job ID <jobId> failed or was cancelled with the message: 1000: Compile error: [<file, line>] Wrong number of gate parameters
```

This error occurs when a comma "," or another non-period character is used as a decimal separator. Replace all non-period decimal separators with periods ".". For example:

Q#

```
// replace this line:
rx(1,5707963267948966) q[0];

// with this:
rx(1.5707963267948966) q[0];
```

Issue: Compiler error "not available for the current compilation configuration"

When you run a Q# code cell in a Jupyter Notebook in VS Code, you might encounter the error:

Windows Command Prompt

```
<function name> not found. Found a matching item `<function name>' that is not available for the current compilation configuration
```

This error indicates that you set the QIR (quantum intermediate representation) target profile to **Base** when the function requires the **Unrestricted** target profile. If you don't specify a target profile type, then the compiler automatically sets the target to **Unrestricted**.

Issue: Operation returned an invalid status code 'Forbidden'

When you submit your first job, you might get a `'forbidden'` error code.

This issue occurs when you create a new workspace in the Azure portal and Azure Quantum fails to complete the role assignment that links the workspace to the specified storage account. This can happen when you close the tab or web browser before the workspace creation is completed.

To verify that you're running into this role assignment issue, follow these steps:

1. Go to your Quantum workspace in the Azure portal.
2. In the **Overview** page, select the **Storage account**.
3. In the navigation pane, select **Access Control (IAM)**.
4. Choose the **Role assignments** tab.
5. In the **Role** column, check whether your workspace's storage account has the **Storage Account Contributor** role and the **Storage Blob Data Contributor** role.

If the workspace doesn't have both of these roles on the storage account, then do one of the following:

- Create a new workspace and make sure that workspace creation completes before you close the web browser window or tab.
- Assign the **Storage Account Contributor** and **Storage Blob Data Contributor** roles to your workspace on the storage account.

Issue: Job fails with error code: QIRPreProcessingFailed

When you submit a job to a Rigetti target and the job fails, you might see the following error message in the **Job management** console for your Quantum workspace in the Azure portal:

Output

```
Error code: QIRPreProcessingFailed
Error message: No match found for output recording set converter from
outputrecordingset.v2.labeled to outputrecordingset.v1.nonlabeled
```

This error can be caused by a dependency conflict with a previous version of `pyqir` or `qiskit-qir`. Uninstall all versions of `pyqir`, `pyqir-*`, and `qiskit-qir` on your local machine, and then install or update the `qdk` Python library with the `azure` and `qiskit` extras:

Shell

```
pip install --upgrade "qdk[azure,qiskit]"
```

Issue: Retrieving basic information about failed jobs

When you submit a job to a hardware target, your job might sit in the queue for several hours or days before the job fails.

To retrieve more information about the job failure, do one of the following:

- To view the job output or the error message, use the get_results() method from the `qdk.azure` Python module:

  Python

  ```
  job.get_results()
  ```

- In your Quantum workspace in the Azure portal, go to the **Job Management** pane from the **Operations** dropdown, and then choose the job **Name** to open the **Job details** pane.

- In your Quantum workspace in the Azure portal, go to the **Providers** pane from the **Operations** dropdown. Verify that the target hardware is available. If the target status is **Degraded**, then jobs might stay in the queue longer than usual. Sometimes the jobs get processed, but sometimes they time out and return an error of **target unavailable**.

Issue: Azure Quantum asks me to authenticate when I programmatically connect to my workspace

If you're using the Azure Quantum Python SDK and you're connecting to your workspace using the `AzureQuantumProvider` class, then you might experience a pop-up to authenticate to Azure every time you run your script.

This pop-up happens because your security token gets reset each time you run the script.

To resolve this issue, run `az login` from the Azure CLI. For more information, see az login.

Azure Quantum workspace creation issues

You might experience the following issues when you create a Quantum workspace in the Azure portal.

Issue: You can't access the workspace creation form in the Azure portal, and you're asked to sign up for a subscription instead

This issue occurs because you don't have an active Azure subscription.

When you sign up for a 30 day free trial Azure subscription, you get free Azure credits. After you use all your free credits, or 30 days after you sign up, you need to upgrade to a pay-as-you-go subscription to continue using Azure Quantum services. When you have an active subscription, the Azure portal allows you to access the workspace creation form.

To see a list of your subscriptions and associated roles, see Check your subscriptions.

> **Note**
>
> Azure credits from the 30 day free trial Azure subscription aren't eligible to use on quantum hardware providers.

Issue: The Quick create option is unavailable

You must be an **Owner** of a subscription to use the **Quick create** option. To see a list of your subscriptions and associated roles, see Check your subscriptions. If you're a subscription **Contributor**, you can use the **Advanced create** option to create a workspace.

Issue: You're unable to create or select a resource group or storage account

This issue occurs because you don't have the authorization required at the subscription, resource group, or storage account level. For more information on required access levels, see Role requirements for creating a workspace.

Issue: A "Deployment Validation Failed" error message appears when you choose Create

This error message might include more details such as "The client does not have authorization to perform action."

This issue occurs because you don't have the authorization required at the subscription, resource group, or storage account level. For more information on required access levels, see Role requirements for creating a workspace.

If you were recently granted access, you might need to refresh the page. New role assignments can take up to one hour to take effect over cached permissions across the stack.

Issue: You don't see a specific quantum hardware provider on the Providers tab

This issue occurs because the provider doesn't support the billing region that your subscription is set in. For a list of providers and their availability by country/region, see Global availability of Azure Quantum providers.

Issue: Workspace creation or adding/removing providers fails with "ResourceDeploymentFailure" or "ProviderDeploymentFailure"

This issue might include more details such as "ResourceDeploymentFailure - The 'AzureAsyncOperationWaiting' resource operation completed with terminal provisioning state 'Failed'.", or "ProviderDeploymentFailure - Failed to create plan for provider: *<Name of the provider>*".

This failure occurs because the tenant didn't enable Azure Marketplace purchases. Follow the steps in Enabling Azure Marketplace purchases to enable Azure Marketplace purchases.

Issue: Deployment of a quantum workspace or storage account fails

When you try to deploy a Quantum workspace or storage account, you might get one of the following errors:

- **Workspace**: "The resource write operation failed to complete successfully, because it reached terminal provisioning state 'Failed'".
- **Storage account**: "The template deployment failed because of policy violation".

This issue might occur if your subscription security policy blocks the creation of storage accounts that have public access enabled. The Azure Quantum service only supports storage accounts that have public internet access.

To resolve this, work with your subscription administrator to get an exception for the storage account that you want to use.

Last updated on 06/15/2026

Release notes for the Microsoft Quantum Development Kit (QDK) and Azure Quantum

The release notes outline updates to the Microsoft Quantum Development Kit (QDK) and the Azure Quantum service.

For 'getting started' instructions, see Set up Azure Quantum. For instructions on how to update your QDK to the latest version, see Update the Microsoft Quantum Development Kit to the latest version.

- For the latest QDK and Azure Quantum release notes, see the GitHub qsharp repository.
- For the latest azure-quantum Python package release notes, see the GitHub azure-quantum-python repository.

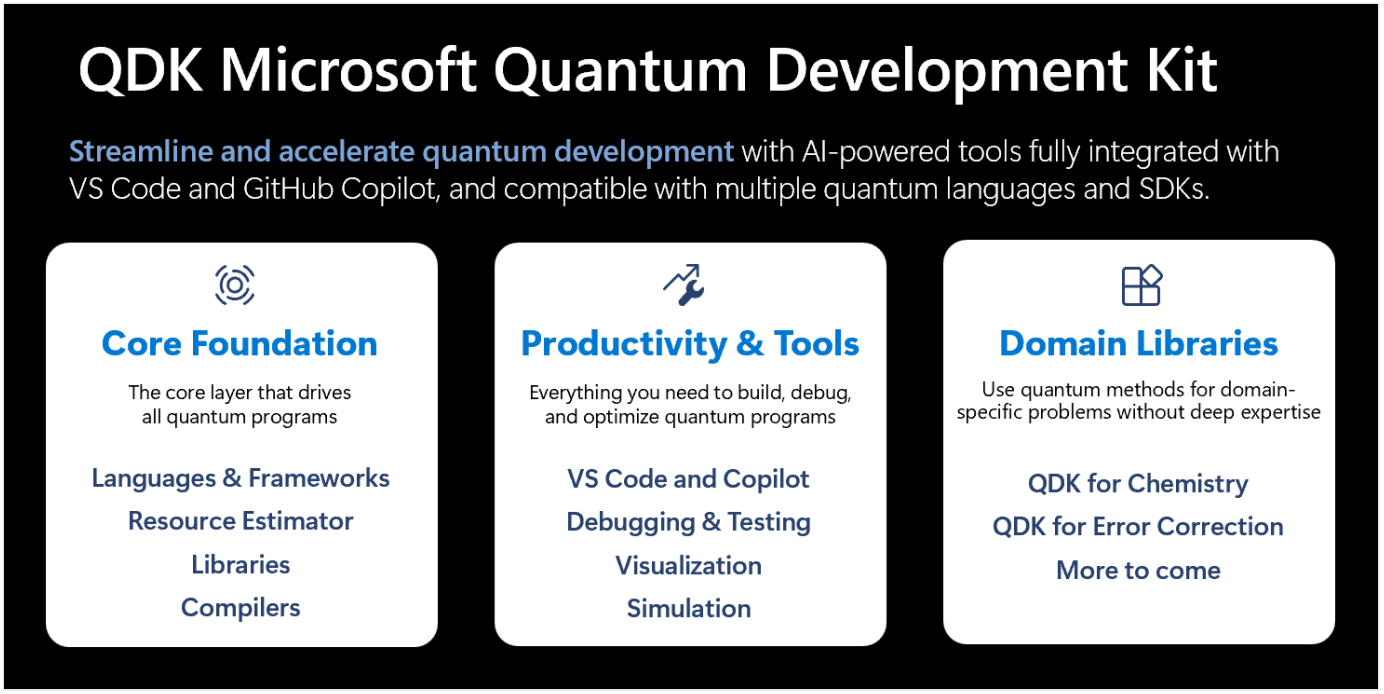
Last updated on 01/29/2026

Build quantum solutions with the Microsoft Quantum Development Kit

👉 Explore the latest QDK updates at quantum.microsoft.com

👉 Watch hands-on QDK tutorials

The Microsoft Quantum Development Kit (QDK) is a free open-source software development kit designed specifically for quantum program development and quantum computing. The QDK connects directly to Azure Quantum, so you can run programs on real quantum computers in the cloud.



With the QDK, you have everything you need to write, simulate, debug, and run quantum programs all in one toolkit.

Features of the QDK

Explore the following documentation to learn more about the main features of the QDK.

- Quantum language support
- Quantum simulators and noise models
- The circuit editor
- The quantum resource estimator
- Quantum error correction

- Chemistry and materials science applications with QDK/Chemistry
- Learn quantum computing with QDK Learning
- Copilot integration in VS Code

Get started with the QDK

The QDK consists of the following components.

Expand table

Component	Use for	Where to install
The QDK extension for Visual Studio Code	General quantum development	VS Code marketplace
The `qdk` Python package	General quantum development	Python Package Index
The `qdk-chemistry` Python library	Chemistry and material science applications	Python Package Index

To get started with the QDK, install the VS Code extension and Python libraries. You can use each component individually, or use them together for the same project. For example, you can quickly write and run Q# or OpenQASM programs in VS Code, then import the programs in Python to perform resource estimation for your program.

For instructions on how install the QDK extension for VS Code and the Python package, see Set up the QDK extension. To install the QDK/Chemistry library, see How to install QDK for chemistry.

👉 **Explore the latest QDK updates at quantum.microsoft.com**

👉 **Watch hands-on QDK tutorials**

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/22/2026

Set up the Microsoft Quantum Development Kit

In this article, you learn how to install the Microsoft Quantum Development Kit (QDK) extension for Visual Studio Code (VS Code), the QDK Python package with Jupyter Notebook support, and the Azure CLI.

Prerequisites

- The latest version of VS Code.
- If you want to submit jobs to Azure Quantum, then you need to have an Azure account with a quantum workspace. For more information, see Create an Azure Quantum workspace.

Install the QDK extension

To use the QDK in VS Code, install the QDK extension. You can also use the QDK in VS Code for the Web without installing the extension, but you won't have all the features of VS Code Desktop. For more information, see Different ways to run Q# programs.

You can now write, debug, and run Q# programs against the built-in quantum simulator. Or, if you have an Azure account, you can connect and submit Q# programs to quantum hardware, all from VS Code.

To test your setup, see Submit Q# jobs to Azure Quantum.

Install the QDK Python package

With Python support in VS Code, you can embed or call Q# code from your Python programs or Jupyter notebooks, and run them on the built-in quantum simulators. You can also connect to your Azure Quantum workspace and submit your jobs to run on real quantum hardware.

Prerequisites

- Install a Python environment (3.10 or greater, 3.11 recommended) with Python and Pip.
- Install the QDK extension in VS Code.

Install the required packages

To add Python and Jupyter Notebook support:

1. Install the Python and Jupyter extensions for VS Code.
2. Open the command line.
3. Install the `qdk` Python package with the `azure` extra:

   Windows Command Prompt

   ```
   python -m pip install "qdk[azure]"
   ```

4. For Qiskit version 1 and version 2 support, install the `qiskit` extra.

   Windows Command Prompt

   ```
   python -m pip install "qdk[qiskit]"
   ```

   > **Important**
   >
   > If you're updating from a previous Qiskit environment, then see **Update the qdk.azure module with Qiskit support**.

5. To work in Jupyter Notebook and display visualizations, install the following Python packages:

   Windows Command Prompt

   ```
   python -m pip install "qdk[jupyter]" ipykernel ipympl jupyterlab
   ```

6. For quantum error correction features, install the `ec` extra (requires Python 3.11 or later).

   Windows Command Prompt

   ```
   python -m pip install "qdk[ec]"
   ```

   For more information on how to install and use the `qdk.ec` module, see How to install QDK for error correction.

To test your setup, see Submit Q# jobs with Python or Submit Q# jobs with Jupyter Notebooks.

Add support for the Azure CLI

You have the option to use the Azure CLI to submit quantum jobs from a terminal window in VS Code.

1. Install the Azure CLI.
2. Open a Windows command prompt or a terminal in VS Code.
3. In the command prompt, run the following command to update to the latest Azure CLI `quantum` extension:

   Windows Command Prompt

   ```
   az extension add --upgrade -n quantum
   ```

To test your setup, see Submit Q# jobs to Azure Quantum.

Related content

- Different ways to run Q# programs
- Quickstart: Create your first Q# program
- Update the QDK to the latest version

Last updated on 10/22/2026

How to use agent mode in VS Code for the QDK

Use agent mode in Visual Studio Code (VS Code), powered by GitHub Copilot, to enhance your builder experience with the Microsoft Quantum Development Kit (QDK).

Agent mode is an AI-assisted development experience that helps you write and debug code, and complete other development tasks in VS Code. The QDK extension includes a `SKILL.md` file that GitHub Copilot agent reads to learn Agent Skills for specific tasks related to development in the QDK. The QDK skills help Copilot to perform the following tasks in agent mode:

- Write, debug, and explain Q# and OpenQASM code
- Develop with the `qdk` Python package
- Explain quantum computing concepts
- Give you information about your Azure Quantum jobs and workspaces

For more information about Agent Skills for GitHub Copilot in VS Code, see Use Agent Skills in VS Code on the VS Code website.

Prerequisites

- Install the latest version of VS Code.
- Install the QDK extension in VS Code.

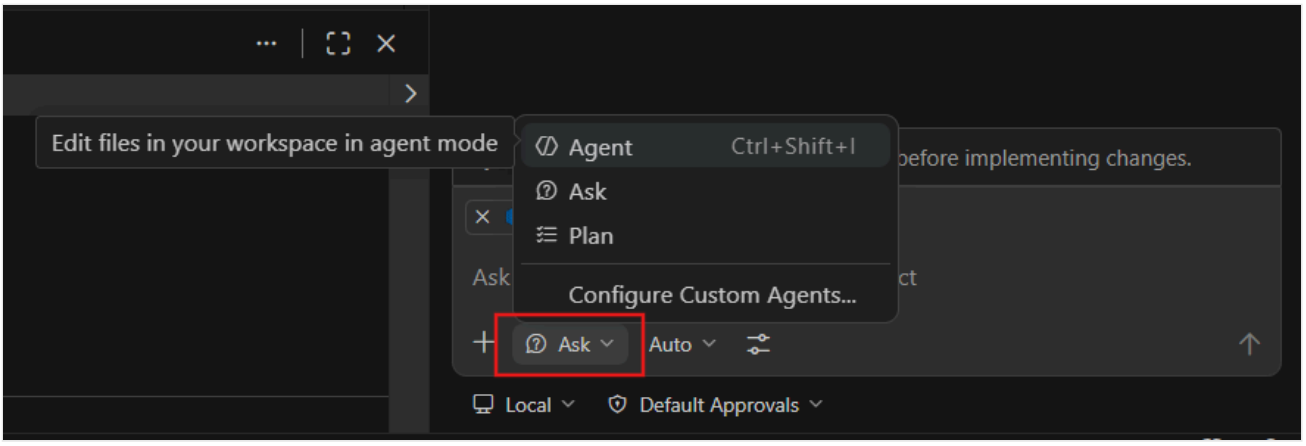
Set up Agent Skills from the QDK

To get started with the QDK Agent Skills for agent mode in VS Code, follow these steps:

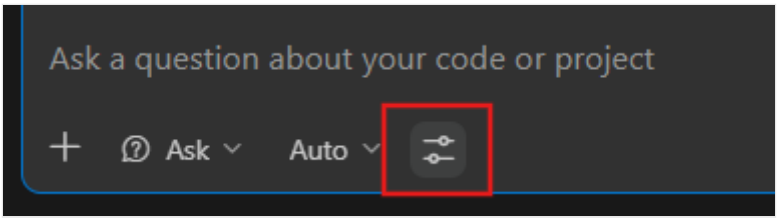
1. Open a Q# (`.qs`) or (`.qasm`) file in VS Code.

2. Open the Copilot Chat window in VS Code.

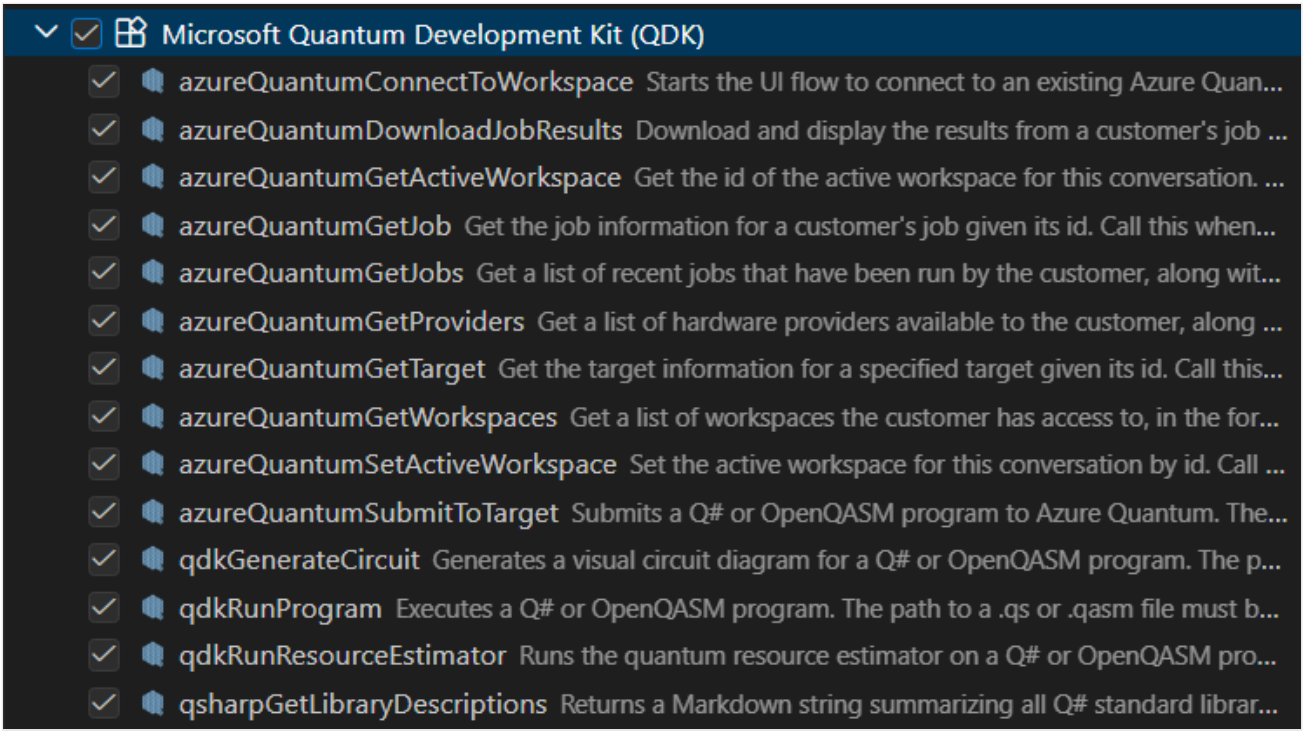
3. In the chat box, select the **Set Agent** icon and choose **Agent**.



4. In the chat box, select the **Configure Tools** icon. The **Configure Tools** window opens.



5. Select all the tools in the **Microsoft Quantum Development Kit (QDK)** dropdown section, and then select the **OK** button.



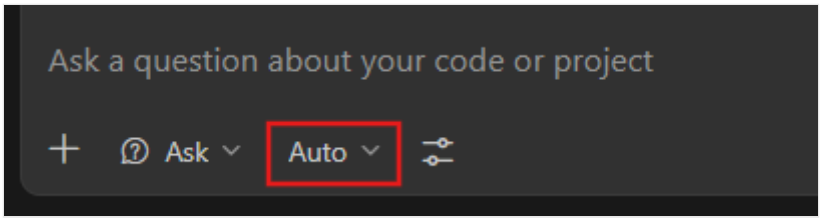
6. Enter a prompt for Copilot agent. For example, if your file already contains a program, ask what the code does. Or, if your file is empty, tell Copilot to write a teleportation algorithm.

Copilot reads the Agent Skills from the QDK and responds to your prompts with specialized knowledge about the QDK, Azure Quantum, and quantum computing.

Try different agent models

Copilot lets you choose from a set of different language models to use in agent mode. Different models have different strengths, so the best model for you depends on your use case.

To explore different models, select the **Pick Model** icon in the Copilot chat box and choose one of the available models from the list.



Try some prompts

Now that you're set up to leverage AI in your QDK projects, try out some prompts in the Copilot Chat.

To get started, here are a few example prompts:

> Simulate this program for 1,000 shots and show me a histogram.

> Submit this program to Azure Quantum.

> Add tests for this code.

> Show me my recent jobs on Azure Quantum.

> Hey, I've been meaning to learn a bit about quantum computing but I have no clue where to even start. Can you just help me write a very simple program, understand it, and maybe even run it on a real quantum computer?

Last updated on 07/31/2026

Reference: Microsoft Quantum Development Kit extension for Visual Studio Code

The Microsoft Quantum Development Kit (QDK) uses the standard functionality of Visual Studio Code (VS Code), along with language-specific features when you work with `.qs` or `.qasm` files. This reference guide describes the main QDK commands in the VS Code Command Palette, along with other features and links to more content. For general VS Code guidance, see the VS Code documentation.

To find QDK commands, select **View** > **Command Palette** and enter **QDK:** or **Microsoft Quantum:**. Some commands appear only when a supported file or view is active.

> **Tip**
>
> Create custom keyboard shortcuts for VS Code commands by using **Keyboard Shortcuts** or by pressing **Ctrl** + **K** + **S**. For more information, see Key Bindings for VS Code.

Commands

Many QDK commands for writing and running Q# and OpenQASM programs are only available when a `.qs`, `.qasm`, or `.qsc` file is active. Other commands are always available.

General commands

Expand table

Command	Action	Notes	Alternate user action
QDK: Create Q# project	Creates a Q# project in the active folder, including a `qsharp.json` manifest file and a `src` subfolder with a default `main.qs` file.	For more information about Q# projects, see Working with Q# projects.	In File Explorer, open the shortcut menu for the target folder and select **Create Q# project**.
QDK: Create a Microsoft	Opens a sample Jupyter Notebook that runs a Q# +	For more information about the QDK in Jupyter Notebook, see Submit jobs	N/A

Command	Action	Notes	Alternate user action
Quantum notebook	Python program and submits a job to Azure Quantum.	to Azure Quantum with the QDK Python package.	
QDK: Connect to an Azure Quantum workspace	Connects to an Azure Quantum workspace by using your Azure account or a connection string. After you're authenticated, select the **Microsoft Quantum** icon on the Activity Bar.	For more information about Azure Quantum connections, see Connect to your Azure Quantum workspace.	Hover over **Quantum Workspaces** and select the + icon.
QDK: Open QDK playground	Opens an online folder of Q# and OpenQASM sample programs in File Explorer. Edit and run the programs in the local quantum simulator, and set breakpoints and step through the code with the built-in debugger.	For more information, see the **README** file in the sample folder.	N/A
QDK: Refresh Azure Quantum workspaces	Syncs the latest changes from your connected quantum workspaces.	If there's a connection issue, an alert icon appears next to the workspace name.	Hover over **Quantum Workspaces** and select the refresh icon.
QDK: Create a Microsoft Quantum Python virtual environment	Creates a folder and installs a Python virtual environment with the selected packages.	Available in desktop VS Code, but not VS Code for the Web.	N/A
QDK: Bloch sphere	Opens the Bloch sphere visualizer.	For more information, see How to use the Bloch sphere visualizer in the QDK.	N/A
QDK: Show Changelog	Opens a changelog in a new tab that shows QDK updates for the current and all previous releases.	The changelog is also available on the open-source QDK GitHub repository.	N/A
Microsoft Quantum: Focus on Quantum Workspaces View	Opens the Microsoft Quantum sidebar and focuses the **Quantum Workspaces** view.	If no workspaces are connected, the view provides an option to connect to an existing workspace. For more information, see Connect to your Azure Quantum workspace.	Select the **Microsoft Quantum** icon on the Activity Bar, then expand **Quantum Workspaces**.

Command	Action	Notes	Alternate user action
Microsoft Quantum: Focus on QDK Learning View	Opens the Microsoft Quantum sidebar and focuses the **QDK Learning** welcome view, where you can select **Start learning**.	Available in desktop VS Code before you initialize QDK Learning in the workspace. For more information, see QDK Learning overview.	Select the **Microsoft Quantum** icon on the Activity Bar, then expand **QDK Learning**.
Microsoft Quantum: Focus on Learning View	Opens the Microsoft Quantum sidebar and focuses the **Learning** view, where you can access courses and continue your progress.	Available in desktop VS Code after you initialize QDK Learning in the workspace. For more information, see QDK Learning overview.	Select the **Microsoft Quantum** icon on the Activity Bar, then expand **Learning**.

Commands available based on the active file type

Expand table

Command	Program file types	Action	Notes	Alternate user action
QDK: Show circuit	`.qs`, `.qasm`	Shows a circuit diagram for the program before it runs.	For more information, see Visualize quantum circuit diagrams.	Select the code lens option **Circuit** from the menu next to the entry point operation or above each user-defined operation in the program.
QDK: Run file and show histogram	`.qs`, `.qasm`	Runs the current program and shows a histogram of the results in a new pane.	To access the sort and filter options, select the filter icon in the histogram pane.	Select the code lens option **Histogram** from the menu next to the entry point operation.
QDK: Get QIR for current QDK program	`.qs`, `.qasm`	Opens the QIR source for the current Q# or OpenQASM code in a new edit window. Your program must use the Base, Adaptive RI, or Adaptive RIF target profile to export QIR source.	For more information about QIR, see Quantum intermediate representation and the QDK developer blog.	N/A

Command	Program file types	Action	Notes	Alternate user action
QDK: Calculate Resource Estimates	`.qs`, `.qasm`	Runs the built-in version of the resource estimator. The VS Code extension uses a deprecated version of the resource estimator.	For more information, see Introduction to the Microsoft Quantum resource estimator.	N/A
QDK: Help	`.qs`	Opens a brief overview of the QDK extension in VS Code.	N/A	N/A
QDK: Run file and show circuit diagram	`.qs`, `.qasm`	Runs the current program and shows a circuit diagram of the program with outputs.	For more information, see Visualize quantum circuit diagrams.	N/A
QDK: Show API documentation	`.qs`	Opens the API documentation in a new pane. To search this pane, press **Ctrl** + **F**.	For more information, see the Q# API reference.	N/A
Debug: Start Debugging	`.qs`, `.qasm`, `.qsc`	Opens the current program in the debugger.	For more information, see Debugging and testing your quantum code.	Press **F5**, select the code lens option **Debug** next to the entry point operation, or select the **Run** icon in the upper-right corner and then select **Start Debugging**.
QDK: Run program	`.qs`, `.qasm`, `.qsc`	Runs the current program in the default quantum simulator.	For more information, see Get started with Q# programs.	Press **Ctrl** + **F5**, select the code lens option **Run** next to the entry point operation, or select the **Run** icon in the upper-right corner and then select **Run**.

Panel views

Q# and OpenQASM programs use the following views in the VS Code panel:

Expand table

View	Action
Debug console	Shows run output or debug output
Problems	Shows precompile error checks

Edit Q# and OpenQASM code

Most common code editing features in VS Code are also available for Q# and OpenQASM programs:

- Precompile error checks
- Go-to definition
- References
- Function signatures
- Parameter information
- Completion suggestions, including context-aware, namespace member, and type member completions
- Linting. For Q# files, configure linting per project in the manifest file. For more information, see Working with Q# projects.

For more information, see IntelliSense in the VS Code documentation.

Common tasks

The following tables list commands for common tasks in the QDK extension.

Work with Q# files and projects

Expand table

Task	Action	Notes
Create a new Q# file	Select **File** > **New file** > **Text file**. Save the file with a `.qs` extension.	If you set `Files: Default Language = qsharp`, new files automatically default to Q# formatting.
Create a Q# project	From an open folder in File Explorer, select **QDK: Create Q# project** from the Command Palette. Or, open the shortcut menu for the folder in File Explorer and select **Create Q# project**.	For more information about Q# projects, see Working with Q# projects.

Task	Action	Notes
Create a sample file	In a blank `.qs` or `.qasm` file, enter *samples*, and then select a sample program from the list.	You can also select **QDK: Open QDK playground** in the Command Palette to open an online folder of Q# and OpenQASM sample programs in File Explorer.

Connect to Azure Quantum

Expand table

Task	Action	Notes	Alternate user action
Connect to an Azure Quantum workspace	Select the **Microsoft Quantum** icon on the Activity Bar. From **Quantum Workspaces**, select **Connect to an existing workspace**. Follow the prompts to select a subscription and workspace.	You can connect to multiple workspaces. Select + next to **Quantum Workspaces** to connect another workspace. Workspace connections persist between your VS Code sessions.	From the Command Palette, select **QDK: Connect to an Azure Quantum workspace**.
Connect programmatically with a Python program	Open the shortcut menu for an existing workspace connection and select **Copy Python code to connect to workspace**. Paste the resulting code into your Python program.	For more information, see Submit jobs to Azure Quantum with the QDK Python package.	N/A

Run programs

Expand table

Task	Action	Notes	Alternate user action
Run a Q# or OpenQASM program on the local quantum simulator	In a `.qs` or `.qasm` file, select the **Run** icon in the upper-right corner, and then select **Run**.	For more information on the quantum simulator, see the Sparse quantum simulator.	Press **Ctrl** + **F5**, or choose either **QDK: Run file and show histogram** or **QDK: Run file and show circuit diagram** in the Command Palette, or choose the **Run** code lens option above the entry point operation.
Debug a program	In a Q# or OpenQASM program, select the **Run** icon in the upper-	For more information about	Press **F5**, or choose the code **Debug** code lens from the

Task	Action	Notes	Alternate user action
	right corner, and then select **Start Debugging**.	the Q# debugger in VS Code, see Debugging and testing your quantum code.	menu above the entry point operation.
View providers and targets in your workspaces	Select the **Microsoft Quantum** icon on the Activity Bar. From **Quantum Workspaces**, expand the workspace, and then expand **Providers** to see the available providers in the workspace. Expand an individual provider to view the available targets.	Hover over a target name to view its **Status** and **Queue time** before you submit a job.	N/A
Submit a job to Azure Quantum	With a Q# or OpenQASM program open, select a workspace, provider, and target. To submit the current program, select the arrow next to the target.	For more information, see Submit jobs to Azure Quantum with the QDK extension for VS Code.	N/A
View job results	Expand the workspace, and then expand **Jobs**. To open the job output from Azure Storage, select the cloud icon next to the job name.	Jobs are listed from newest to oldest.	N/A

ⓘ**Note:** The author created this article with assistance from AI. Learn more

Last updated on 11/06/2026

How to test quantum programs in the QDK

Testing helps you check that quantum programs behave as expected. This article shows you how to test Q# and OpenQASM programs with the Microsoft Quantum Development Kit (QDK).

The QDK provides Q# functions and operations that check your code as it runs, and you can write Q# unit tests to run in Visual Studio Code (VS Code) or Python. You can also run OpenQASM programs from Python and use Python assertions to check their results.

To learn how to debug Q# and OpenQASM programs, see How to debug quantum programs in the QDK.

Prerequisites

To run Q# unit tests in VS Code, install the following:

- The latest version of VS Code, or open VS Code for the Web.
- The latest version of the QDK extension in VS Code.

To test programs from Python, install the latest version of the `qdk` Python package. To run examples in a Jupyter notebook, also install the following:

- The Python extension and Jupyter extension in VS Code.
- The latest version of the `qdk` Python package with the `jupyter` extra. You need version 1.31.0 or later to run Q# unit tests from Python.

  Bash

  ```
  pip install --upgrade "qdk[jupyter]"
  ```

Test Q# code with the `fail` expression

The fail expression immediately ends your program. To test your code, use a fail expression inside a conditional statement.

The following example uses a `fail` statement to test that a qubit array contains exactly 3 qubits. The program ends with an error message when the test doesn't pass.

1. In VS Code, create and save a new `.qs` file with the following code:

Q#

```
operation Main() : Unit {
    use qs = Qubit[6];
    let n_qubits = Length(qs);

    if n_qubits != 3 {
        fail $"The system should have 3 qubits, not {n_qubits}.";
    }
}
```

2. Press **Ctrl + F5** to run the program. Your program fails and the following output appears in the **Debug Console**:

Output

```
Error: program failed: The system should have 3 qubits, not 6.
```

3. Edit your code from `Qubit[6]` to `Qubit[3]`, save your file, and then press **Ctrl + F5** to run the program again. The program runs without an error because the test passes.

Test Q# code with the `Fact` function

Use the Q# Fact function from the Std.Diagnostics namespace to test your code. The function takes a Boolean expression and an error message string. If the expression is true, the test passes and your program continues. Otherwise, Fact ends your program and shows the error message.

To test the array length with Fact, follow these steps:

1. In VS Code, create and save a new `.qs` file with the following code:

Q#

```
import Std.Diagnostics.Fact;

operation Main() : Unit {
    use qs = Qubit[6];
    let n_qubits = Length(qs);

    Fact(n_qubits == 3,  $"The system should have 3 qubits, not {n_qubits}.")
}
```

2. Press **Ctrl + F5** to run the program. The test condition in `Fact` doesn't pass and the error message appears in the **Debug Console**.

3. Edit your code from `Qubit[6]` to `Qubit[3]`, save your file, and then press **Ctrl + F5** to run the program again. The test condition in `Fact` passes and your program runs without an error.

Check qubit states in Q#

The `CheckZero` and `CheckAllZero` Q# operations check whether the current state of a qubit or qubit array is . The `CheckZero` operation takes a single qubit and returns `true` only when the qubit is in the state. The `CheckAllZero` operation takes a qubit array and returns `true` only when all qubits in the array are in the state. To use `CheckZero` and `CheckAllZero`, import them from the `Std.Diagnostics` namespace.

The following example uses both operations. The `CheckZero` operation tests that the `X` operation flips the first qubit from the state to the state, and the `CheckAllZero` operation tests that both qubits are reset to the state.

In VS Code, create and save a new `.qs` file with the following code, then run the program and examine the output in the **Debug Console**.

Q#

```
import Std.Diagnostics.*;

operation Main() : Unit {
    use qs = Qubit[2];
    X(qs[0]);

    if CheckZero(qs[0]) {
        Message("X operation failed");
    }
    else {
        Message("X operation succeeded");
    }

    ResetAll(qs);

    if CheckAllZero(qs) {
        Message("Reset operation succeeded");
    }
    else {
        Message("Reset operation failed");
    }
}
```

Write Q# unit tests with the `@Test()` annotation

In Q# programs, you can apply the `@Test()` annotation to a callable (function or operation) to turn the callable into a unit test. These unit tests appear in the **Testing** menu in VS Code so that you can take advantage of this VS Code feature. You can turn a callable into a unit test only when the callable doesn't take input parameters.

The following example wraps the array length test code in an operation and turns that operation into a unit test:

1. In VS Code, create and save a new `.qs` file with the following code:

   Q#

   ```
   import Std.Diagnostics.Fact;

   @Test()
   operation TestCase() : Unit {
       use qs = Qubit[3];
       let n_qubits = Length(qs);

       Fact(n_qubits == 3, $"The system should have 3 qubits, not {n_qubits}.");
   }
   ```

   The `@Test()` annotation on the line before the `TestCase` operation definition turns the operation into a VS Code unit test. A green arrow appears on the operation definition line.

2. Select the green arrow to run TestCase and view the test results.

3. To interact with your unit tests in the VS Code Test Explorer, choose the **Testing** flask icon in the Activity Bar.

4. Edit your code from `Qubit[3]` to `Qubit[6]` and run the unit test again to see how the test information changes.

You can write and run Q# unit tests in VS Code without an entrypoint operation in your program.

> **Note**
>
> Callables from the `Std.Diagnostics` namespace aren't compatible with QIR generation, so only include unit tests in Q# code that you run on simulators. If you want to generate QIR from your Q# code, then don't include unit tests in your code.

Test OpenQASM programs from Python

The `qdk.openqasm.run` function runs an OpenQASM program on the local simulator and returns its results. Use Python assertions to check those results, or put the assertions in a pytest test function to run them as part of your Python test suite.

The following example checks that an OpenQASM program applies an `x` gate and measures the qubit as `One` in all three shots:

Python

```python
from qdk import Result
from qdk.openqasm import run

qasm_code = """
    OPENQASM 3.0;
    include "stdgates.inc";
    qubit q;
    output bit result;
    x q;
    result = measure q;
"""

results = run(qasm_code, shots=3)
assert results == [Result.One, Result.One, Result.One]
```

The `qdk.test_utils.run_tests` function discovers Q# callables marked with `@Test()`. It doesn't discover OpenQASM programs directly. To use `run_tests` with OpenQASM code, import the program as a Q# callable and call it from a Q# unit test:

Python

```python
from qdk import qsharp
from qdk.openqasm import import_openqasm
from qdk.test_utils import run_tests

import_openqasm("""
    OPENQASM 3.0;
    include "stdgates.inc";
    qubit[1] qs;
    x qs[0];
""", name="ApplyX")

qsharp.eval("""
import Std.Diagnostics.Fact;

@Test()
operation TestApplyX() : Unit {
    use qs = Qubit[1];
    ApplyX(qs);
    Fact(MResetZ(qs[0]) == One, "Expected the OpenQASM program to return One.");
```

```
}
""")

run_tests()
```

Run Q# unit tests from Python

The `qdk.test_utils` Python module lets you run your Q# unit tests from Python. Use the `run_tests` function to find and run all the Q# callables that have the `@Test()` annotation. You can call `run_tests` from a Jupyter notebook, a Python script, or a Python test framework like pytest. For example, you can use `run_tests` to run your Q# tests in a continuous integration (CI) pipeline.

The `qdk.test_utils` module also includes helpers that test the effect of Q# operations on qubits.

Run Q# unit tests in a Jupyter notebook

The following example defines two Q# unit tests in a Jupyter notebook and then runs them with `run_tests`.

1. In VS Code, press **Ctrl** + **Shift** + **P** to open the **Command Palette**.
2. Enter **Create: New Jupyter Notebook** and press **Enter**. A new Jupyter Notebook tab opens.
3. In the first cell, copy and run the following code:

   Python

   ```
   from qdk import qsharp
   from qdk.test_utils import run_tests
   ```

4. Create a new code cell. Copy and run the following code to define two Q# unit tests:

   Python

   ```
   qsharp.eval("""
   import Std.Diagnostics.*;

   @Test()
   operation TestQubitCount() : Unit {
       use qs = Qubit[3];
       let n_qubits = Length(qs);
       Fact(n_qubits == 3, $"The system should have 3 qubits, not {n_qubits}.");
   }

   @Test()
   ```

```
operation TestXFlipsQubit() : Unit {
    use q = Qubit();
    X(q);
    Fact(not CheckZero(q), "X operation failed.");
    Reset(q);
}
""")
```

5. Create a new code cell. Copy and run the following code to run your tests:

Python

```
run_tests()
```

The `run_tests` function prints a dot for each test that passes, and then prints a summary:

Output

```
Running 2 tests...
..
Finished tests: 2 passed, 0 failed.
```

6. To see what happens when a test fails, create a new code cell. Copy and run the following code to add a test that fails, and then run all your tests again:

Python

```
qsharp.eval("""
@Test()
operation TestQubitCountFails() : Unit {
    use qs = Qubit[6];
    let n_qubits = Length(qs);
    Std.Diagnostics.Fact(n_qubits == 3, $"The system should have 3 qubits, not
{n_qubits}.");
}
""")

run_tests()
```

For each test that fails, `run_tests` prints the name of the test and the error message. After all the tests run, `run_tests` prints a summary and then raises a `RuntimeError`.

Choose which tests run and how much output you get

The `run_tests` function takes the following optional parameters:

Expand table

Parameter	Description
`context`	The `qdk.Context` to find and run tests in. If you don't pass a context, then `run_tests` uses the default context.
`seed`	The seed for the simulator's random number generator. Set a seed to get the same results each time you run tests that include measurements.
`regex`	A regular expression that filters tests by their fully qualified names, such as `MyNamespace.MyTest`. Only tests with names that match the expression run.
`verbose`	The amount of output to print. `0` prints nothing. `1` prints a dot for each test that passes, and is the default. `2` prints the name of each test. `3` or higher also prints the output from your Q# code, such as from `Message` calls.

For example, to run only `TestXFlipsQubit` and print the test names, create a new code cell, then copy and run the following code:

Python

```
run_tests(regex="XFlips", verbose=2)
```

The output shows the name of the test that ran:

Output

```
Running 1 tests...
Running TestXFlipsQubit...
PASSED: TestXFlipsQubit

Finished tests: 1 passed, 0 failed.
```

Run unit tests in a Q# project

To run the unit tests in a Q# project, load the project with `qsharp.init` and then call `run_tests`. Pass the path to the folder that contains your project's `qsharp.json` file. For more information about Q# projects, see Develop and Manage Q# Projects and Custom Libraries.

Python

```
from qdk import qsharp
from qdk.test_utils import run_tests
```

```
qsharp.init(project_root="./my_project")
run_tests()
```

To run your tests separately from other Q# code in your Python session, create a `Context` object for your project and pass it to `run_tests`:

Python

```
from qdk import Context
from qdk.test_utils import run_tests

run_tests(context=Context(project_root="./my_project"))
```

Tests in a Q# project have fully qualified names that include the namespace. By default, the namespace is the name of the file that contains the test. For example, a test named `TestAddition` in a file named `Tests.qs` has the fully qualified name `Tests.TestAddition`.

Run Q# unit tests with pytest

The `run_tests` function raises a `RuntimeError` when any test fails, so you can call `run_tests` inside a pytest test function. When any of your Q# tests fail, pytest reports that the Python test fails.

1. In the folder that contains your Q# project folder, create a file named `test_qsharp.py` with the following code:

   Python

   ```
   from pathlib import Path

   from qdk import Context
   from qdk.test_utils import run_tests

   PROJECT_ROOT = Path(__file__).parent / "my_project"

   def test_qsharp_project():
       run_tests(context=Context(project_root=str(PROJECT_ROOT)))
   ```

2. Install pytest and then run your tests:

   Bash

   ```
   pip install pytest
   pytest
   ```

Test arithmetic operations with `ArithmeticOpTester`

The `ArithmeticOpTester` class runs a Q# arithmetic operation on integer inputs. The class puts each input integer into a qubit register, applies the operation, and then reads each register back as an integer. The integers are unsigned and use little-endian order.

To create an `ArithmeticOpTester` object, pass the name of the operation and the number of qubits in each register. The operation must take one `Qubit[]` input for each register. Then, call `run` with one integer for each register.

For example, the `IncByLE` operation adds the integer in the first register to the integer in the second register. The following code tests `IncByLE` on two 8-qubit registers:

Python

```python
from qdk.test_utils import ArithmeticOpTester

tester = ArithmeticOpTester("Std.Arithmetic.IncByLE", [8, 8])
assert tester.run([5, 7]) == [5, 12]
assert tester.run([100, 27]) == [100, 127]
```

Test a Q# operation's state with `dump_operation_on_state`

The `dump_operation_on_state` function applies a Q# operation to a qubit register and returns the state vector of the register as a list of complex numbers. The operation must take a single `Qubit[]` input. You can also pass a compatible OpenQASM program after you import it with `import_openqasm`. By default, the qubits start in the state. To start in a different state, pass a list of real amplitudes to the `initial_state` parameter. The state vector uses big-endian order.

For example, the following code tests that an operation prepares the Bell state :

Python

```python
import math

from qdk import qsharp
from qdk.test_utils import dump_operation_on_state

qsharp.eval("operation PrepareBellState(qs : Qubit[]) : Unit { H(qs[0]); CNOT(qs[0], qs[1]); }")

state = dump_operation_on_state("PrepareBellState", 2)
expected = [1 / math.sqrt(2), 0, 0, 1 / math.sqrt(2)]
assert all(abs(a - b) < 1e-9 for a, b in zip(state, expected))
```

```
from qdk.openqasm import import_openqasm

import_openqasm("""
    OPENQASM 3.0;
    include "stdgates.inc";
    qubit[2] qs;
    h qs[0];
    cx qs[0], qs[1];
""", name="PrepareBellQasm")

qasm_state = dump_operation_on_state("PrepareBellQasm", 2)
assert all(abs(a - b) < 1e-9 for a, b in zip(qasm_state, expected))
```

For more examples of how to test Q# operations from Python, see Testing operations from the QDK samples.

Related content

- How to debug quantum programs in the QDK
- Develop and Manage Q# Projects and Custom Libraries

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 11/12/2026

How to debug quantum programs in the QDK

Debugging helps you understand how a quantum program runs and find problems in your code. This article shows you how to debug Q# and OpenQASM programs with the Microsoft Quantum Development Kit (QDK) in Visual Studio Code (VS Code), and how to inspect quantum states with QDK dump functions in Q# and Python.

The VS Code debugger supports both Q# and OpenQASM source files. The dump-function examples in this article use Q#; the Python `dump_operation` function can also inspect a compatible OpenQASM program after you import it as a Q# callable.

To learn how to test Q# and OpenQASM programs, see How to test quantum programs in the QDK.

Debug programs in VS Code

Use the debugger in the QDK extension for VS Code to step through functions and operations, track local variable values, and inspect qubit states.

The following example demonstrates how to use the debugger with a Q# program. For complete information on VS Code debuggers, see Debugging on the VS Code website.

1. In VS Code, create and save a new `.qs` file with the following code:

   Q#

   ```
   import Std.Arrays.*;
   import Std.Convert.*;

   operation Main() : Result {
       use qubit = Qubit();
       H(qubit);
       let result = M(qubit);
       Reset(qubit);
       return result;
   }
   ```

2. On the `H(qubit);` line, select the area to the left of the line number to set a breakpoint. A red circle marks the breakpoint.

3. In the Activity Bar, select the **Run and Debug** icon to open the **Run and Debug** view, and then select **Run and Debug** to start debugging. The Debug toolbar opens.

4. When the debugger pauses at the breakpoint, expand **Quantum State** under **Variables** in the **Run and Debug** view to inspect the qubit's initial state.

5. Press **F11** to step into the `H` operation. The source code for the `H` operation appears. Notice that **Quantum State** changes to a superposition as you step through the `H` operation.

6. Press **F10** to step over the `M` operation. Notice that **Quantum State** resolves to either or after the measurement. The `result` variable is listed under **Locals**.

7. Press **F10** again to step over the `Reset` operation. Notice that **Quantum State** is reset to .

When you finish exploring the debugger, press **Shift + F5** to exit the debugger.

Debug an OpenQASM program

You can set breakpoints and inspect qubit states while you debug an OpenQASM file.

1. In an open Q# project in VS Code, create and save a new `.qasm` file with the following code:

   qasm

   ```
   OPENQASM 3.0;
   include "stdgates.inc";

   qubit q;
   bit result;
   h q;
   result = measure q;
   ```

2. On the `h q;` line, select the area to the left of the line number to set a breakpoint.

3. In the code editor, select the **Debug** code lens to start debugging.

4. When the debugger pauses at the breakpoint, expand **Quantum State** in the **Variables** pane to inspect the qubit state. Press **F10** to step over `h q;` and inspect the state again.

5. Press **F5** to continue, or **Shift + F5** to stop debugging.

> ⓘ **Note**

The VS Code debugger works with Q# (`.qs`) and OpenQASM (`.qasm`) files. It doesn't debug Q# cells in Jupyter Notebook.

Inspect quantum state with QDK dump functions

The QDK provides Q# and Python functions that display information about a program's quantum state. The following `DumpMachine` and `dump_machine` examples inspect Q# programs. The Python `dump_operation` function can inspect Q# operations and compatible OpenQASM programs imported as Q# callables.

The Q# `DumpMachine` function

The Q# `DumpMachine` function prints information about the current state of the qubit system to the console without interrupting your program.

The following example calls `DumpMachine` at two points in a Q# program and explores the output.

1. In VS Code, create and save a new `.qs` file with the following code:

   Q#

   ```
   import Std.Diagnostics.*;

   operation Main() : Unit {
       use qubits = Qubit[2];
       X(qubits[1]);
       H(qubits[1]);
       DumpMachine();

       R1Frac(1, 2, qubits[0]);
       R1Frac(1, 3, qubits[1]);
       DumpMachine();

       ResetAll(qubits);
   }
   ```

2. Press **Ctrl + Shift + Y** to open the **Debug Console**.

3. Press **Ctrl + F5** to run your program. The following output from `DumpMachine` appears in the **Debug Console**:

   Output

   ```
   Basis | Amplitude      | Probability | Phase
   ------------------------------------------------
   ```

```
 |00⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
 |01⟩ | -0.7071+0.0000𝑖 |    50.0000% |  -3.1416

Basis | Amplitude      | Probability | Phase
-----------------------------------------------
 |00⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
 |01⟩ | -0.6533-0.2706𝑖 |    50.0000% |  -2.7489
```

The output from `DumpMachine` shows how the state of the qubit system changes after each set of gates.

> **Note**
>
> The output from `DumpMachine` uses big-endian ordering.

The Python `dump_machine` function

The dump_machine function is part of the `qdk.qsharp` Python module. This function returns a `StateDump` object that contains the qubit count and the sparse state amplitudes of the qubit system.

The following example uses `dump_machine` to inspect qubit states in a Jupyter notebook.

1. In VS Code, press **Ctrl** + **Shift** + **P** to open the **Command Palette**.
2. Enter **Create: New Jupyter Notebook** and press **Enter**. A new Jupyter Notebook tab opens.
3. In the first cell, copy and run the following code:

Python

```python
from qdk import qsharp
```

4. Create a new code cell, then copy and run the following Q# code:

Q#

```
%%qsharp

use qubits = Qubit[2];
X(qubits[0]);
H(qubits[1]);
```

5. Create a new code cell. Copy and run the following Python code to view the qubit state at this point in the program:

Python

```
dump = qsharp.dump_machine()
dump
```

The `dump_machine` function displays the following output:

Output

```
Basis State
(|ψ₁…ψₙ⟩)  Amplitude       Measurement Probability  Phase
|10⟩       0.7071+0.0000i   50.0000%                 ↑  0.0000
|11⟩       0.7071+0.0000i   50.0000%                 ↑  0.0000
```

6. Create a new code cell, then copy and run the following Q# code:

Q#

```
%%qsharp

R1Frac(1, 2, qubits[0]);
R1Frac(1, 3, qubits[1]);
```

7. Create a new code cell. Copy and run the following Python code to view the qubit state at this point in the program:

Python

```
dump = qsharp.dump_machine()
dump
```

The `dump_machine` function displays the following output:

Output

```
Basis State
(|ψ₁…ψₙ⟩)  Amplitude      Measurement Probability  Phase
|10⟩       0.5000+0.5000i  50.0000%                 ↗  0.7854
|11⟩       0.2706+0.6533i  50.0000%                 ↗  1.1781
```

8. To print an abbreviated version of the `dump_machine` output, create a new code cell and run the following Python code:

Python

```
print(dump)
```

9. To get the total number of qubits in the system, create a new code cell and run the following Python code:

Python

```
dump.qubit_count
```

10. To get the amplitudes of the and basis states, create a new code cell and run the following Python code:

Python

```
print(dump[2])
print(dump[3])
```

Inspect operations with Python `dump_operation`

The `dump_operation` function is available in the `qdk.qsharp` Python module. This function takes two inputs: a Q# operation or operation definition as a string and the number of qubits that the operation uses. The output from `dump_operation` is a nested list that represents the square matrix of complex numbers that corresponds to the given quantum operation. The matrix values are in the computational basis, and each sublist represents a row of the matrix.

The following example uses `dump_operation` to display matrices for single-qubit and two-qubit operations.

1. In VS Code, press **Ctrl + Shift + P** to open the **Command Palette**.
2. Enter **Create: New Jupyter Notebook** and press **Enter**. A new Jupyter Notebook tab opens.
3. In the first cell, copy and run the following code:

Python

```
from qdk import qsharp
from qdk.qsharp import dump_operation
```

4. To display the matrix elements of a single-qubit gate, call `dump_operation` and pass 1 for the number of qubits. For example, copy and run the following Python code in a new code cell

to get the matrix elements for an identity gate and a Hadamard gate:

Python

```
res = dump_operation("qs => ()", 1)
print("Single-qubit identity gate:\n", res)
print()

res = dump_operation("qs => H(qs[0])", 1)
print("Single-qubit Hadamard gate:\n", res)
```

5. You can also call the `qsharp.eval` function and then reference the Q# operation in `dump_operation` to get the same result. For example, create a new code cell, then copy and run the following Python code to print the matrix elements for a single-qubit Hadamard gate:

   Python

   ```
   qsharp.eval("operation SingleH(qs : Qubit[]) : Unit { H(qs[0]) }")

   res = dump_operation("SingleH", 1)
   print("Single-qubit Hadamard gate:\n", res)
   ```

6. To display the matrix elements of a two-qubit gate, call `dump_operation` and pass 2 for the number of qubits. For example, copy and run the following Python code in a new code cell to get the matrix elements for a Controlled Ry operation where the second qubit is the target qubit:

   Python

   ```
   qsharp.eval ("operation ControlRy(qs : Qubit[]) : Unit { Controlled Ry([qs[0]],
   (0.5, qs[1])); }")

   res = dump_operation("ControlRy", 2)
   print("Controlled Ry rotation gate:\n", res)
   ```

You can also inspect an OpenQASM program after you import it as a Q# callable. For example, the following code gets the matrix for an OpenQASM `x` gate:

Python

```
from qdk.openqasm import import_openqasm

import_openqasm("""
    OPENQASM 3.0;
    include "stdgates.inc";
```

```
    qubit[1] qs;
    x qs[0];
""", name="ApplyX")

res = dump_operation("ApplyX", 1)
print("OpenQASM X gate:\n", res)
```

For more examples of how to test and debug your code with `dump_operation`, see Testing operations from the QDK samples.

Related content

- How to test quantum programs in the QDK

Note: The author created this article with assistance from AI. Learn more

---

Last updated on 11/12/2026

Quantum programming language support in the QDK

The Microsoft Quantum Development Kit (QDK) supports multiple quantum programming languages, frameworks, and circuit formats through the QDK extension for Visual Studio Code (VS Code) and the QDK Python package. With the QDK, you can use one or more programming frameworks in your development workflow and submit supported programs to Azure Quantum.

Language support in the QDK extension for VS Code

The QDK extension for VS Code directly supports development in the Q# and OpenQASM quantum programming languages. When you open a Q# `.qs` file or OpenQASM `.qasm` file in VS Code, you can use QDK features such as resource estimation, circuit visualization, and job submission to Azure Quantum provider targets. You can also use VS Code features such as code completion, IntelliSense, and debugging.

The QDK extension also provides syntax highlighting for Stim `.stim` files. To compile and simulate Stim circuits, use the `qdk.stim` module in the QDK Python package.

The VS Code extension includes the QDK circuit editor, a graphical interface for building quantum circuits that you can use in Q# programs.

Language support in the QDK Python package

The QDK Python package supports development with the following quantum programming languages, frameworks, and circuit formats:

- Q#
- OpenQASM
- Qiskit
- Cirq
- Stim
- PennyLane

The QDK package includes modules that let you:

- Use Jupyter Notebook features, such as visualization tools and a magic command to write Q# code.

- Convert Q# and OpenQASM structs and callables into Python objects for use in other languages and frameworks. For more information about Q# callables, see How to use Q# callables in Python.
- Connect to your Azure Quantum workspace and submit supported programs.

Compare QDK language options

The following table compares the availability of languages, frameworks, and circuit formats in the QDK extension and Python package.

Expand table

Language, framework, or format	QDK VS Code extension	QDK Python package
Q#	Directly available	Available through the `qdk.qsharp` module and the `%%qsharp` magic command
OpenQASM	Directly available	Available through the `qdk.openqasm` module
Qiskit	Not available	Available through the `qdk.qiskit` module
Cirq	Not available	Available through the `qdk.cirq` module
PennyLane	Not available	Available through conversion to QIR
Stim	Syntax highlighting	Available through the `qdk.stim` module

Next steps

To get started with the quantum programming languages, frameworks, and circuit formats in the QDK, install the QDK extension for VS Code and the QDK Python package. For more information, see Set up the QDK.

To submit quantum programs to Azure Quantum, see the following guides:

- How to submit jobs to Azure Quantum
- Submit jobs to Azure Quantum with the QDK extension for VS Code
- Submit jobs to Azure Quantum with the QDK Python package
- Stim support for quantum error correction in the QDK

ⓘ**Note:** The author created this article with assistance from AI. Learn more

Last updated on 11/12/2026

Introduction to the quantum programming language Q#

👉 Learn more about the Microsoft Quantum platform

Q# is a high-level, open-source programming language developed by Microsoft for writing quantum programs. Q# is included in the Microsoft Quantum Development Kit (QDK). For more information, see Set up the Microsoft Quantum Development Kit.

As a quantum programming language, Q# meets the following requirements for language, compiler, and runtime:

- **Hardware agnostic:** Qubits in quantum algorithms aren't tied to a specific quantum hardware or layout. The Q# compiler and runtime handle the mapping from program qubits to physical qubits, allowing the same code to run on different quantum processors.
- **Integration of quantum and classical computing:** Q# allows for the integration of quantum and classical computations, which is essential for universal quantum computing.
- **Qubit management:** Q# provides built-in operations and functions for managing qubits, including creating superposition states, entangling qubits, and performing quantum measurements.
- **Respect the laws of physics:** Q# and quantum algorithms must follow the rules of quantum physics. For example, you can't directly copy or access the qubit state in Q#.

For more information about the origins of Q#, see the blog post Why do we need Q#?.

Structure of a Q# program

Before you start writing Q# programs, it's important to understand their structure and components. Consider the following Q# program, named `Superposition`, that creates a superposition state:

Q#

```
namespace Superposition {
    @EntryPoint()
    operation MeasureOneQubit() : Result {
        // Allocate a qubit. By default, it's in the 0 state.
        use q = Qubit();
        // Apply the Hadamard operation, H, to the state.
        // It now has a 50% chance of being measured as 0 or 1.
```

```
        H(q);
        // Measure the qubit in the Z-basis.
        let result = M(q);
        // Reset the qubit before releasing it.
        Reset(q);
        // Return the result of the measurement.
        return result;
    }
}
```

Based on the comments (`//`), the Q# program first allocates a qubit, applies an operation to put the qubit in superposition, measures the qubit state, resets the qubit, and finally returns the result.

Let's break this Q# program down into its components.

User namespaces

Q# programs can optionally start with a user-defined namespace, such as:

Q#

```
namespace Superposition {
    // Your code goes here.
}
```

Namespaces can help you organize related functionality. Namespaces are optional in Q# programs, meaning that you can write a program without defining a namespace.

For example, the `Superposition` program from the example can also be written without a namespace as:

Q#

```
@EntryPoint()
operation MeasureOneQubit() : Result {
    // Allocate a qubit. By default, it's in the 0 state.
    use q = Qubit();
    // Apply the Hadamard operation, H, to the state.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q);
    // Measure the qubit in the Z-basis.
    let result = M(q);
    // Reset the qubit before releasing it.
    Reset(q);
    // Return the result of the measurement.
```

```
    return result;
}
```

> ⓘ Note
>
> Each Q# program can have only one `namespace`. If you don't specify a namespace, the Q# compiler uses the filename as the namespace.

Entry points

Every Q# program must have an entry point, which is the starting point of the program. By default, the Q# compiler starts executing a program from the `Main()` operation, if available, which can be located anywhere in the program. Optionally, you can use the `@EntryPoint()` attribute to specify any operation in the program as the point of execution.

For example, in the `Superposition` program, the `MeasureOneQubit()` operation is the entry point of the program because it has the `@EntryPoint()` attribute before the operation definition:

Q#

```
@EntryPoint()
operation MeasureOneQubit() : Result {
    ...
}
```

However, the program could also be written without the `@EntryPoint()` attribute by renaming the `MeasureOneQubit()` operation to `Main()`, such as:

Q#

```
// The Q# compiler automatically detects the Main() operation as the entry point.

operation Main() : Result {
    // Allocate a qubit. By default, it's in the 0 state.
    use q = Qubit();
    // Apply the Hadamard operation, H, to the state.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q);
    // Measure the qubit in the Z-basis.
    let result = M(q);
    // Reset the qubit before releasing it.
    Reset(q);
    // Return the result of the measurement.
```

```
    return result;
}
```

Types

Types are essential in any programming language because they define the data that a program can work with. Q# provides built-in types that are common to most languages, including `Int`, `Double`, `Bool`, and `String`, and types that define ranges, arrays, and tuples.

Q# also provides types that are specific to quantum computing. For example, the `Result` type represents the result of a qubit measurement and can have two values: `Zero` or `One`.

In the `Superposition` program, the `MeasureOneQubit()` operation returns a `Result` type, which corresponds to the return type of the `M` operation. The measurement result is stored in a new variable that's defined using the `let` statement:

Q#

```
// The operation definition returns a Result type.
operation MeasureOneQubit() : Result {
    ...
    // Measure the qubit in the Z-basis, returning a Result type.
    let result = M(q);
    ...
}
```

Another example of a quantum-specific type is the `Qubit` type, which represents a quantum bit.

Q# also allows you to define your own custom types. For more information, see Type declarations.

Allocating qubits

In Q#, you allocate qubits using the `use` keyword and the `Qubit` type. Qubits are always allocated in the $|0\rangle$ state.

For example, the `Superposition` program defines a single qubit and stores it in the variable `q`:

Q#

```
// Allocate a qubit.
use q = Qubit();
```

You can also allocate multiple qubits and access each one through its index:

Q#

```
use qubits = Qubit[2]; // Allocate two qubits.
H(qubits[0]); // Apply H to the first qubit.
X(qubits[1]); // Apply X to the second qubit.
```

Quantum operations

After allocating a qubit, you can pass it to operations and functions. Operations are the basic building blocks of a Q# program. A Q# operation is a quantum subroutine, or a callable routine that contains quantum operations that change the state of the qubit register.

To define a Q# operation, you specify a name for the operation, its inputs, and its output. In the `Superposition` program, the `MeasureOneQubit()` operation takes no parameters and returns a `Result` type:

Q#

```
operation MeasureOneQubit() : Result {
    ...
}
```

Here's a basic example that takes no parameters and expects no return value. The `Unit` value is equivalent to `NULL` in other languages:

Q#

```
operation SayHelloQ() : Unit {
    Message("Hello quantum world!");
}
```

The Q# standard library also provides operations that you can use in quantum programs, such as the Hadamard operation, `H`, in the `Superposition` program. Given a qubit in the Z-basis, `H` puts the qubit into an even superposition, where it has a 50% chance of being measured as `Zero` or `One`.

Measuring qubits

While there are many types of quantum measurements, Q# focuses on projective measurements on single qubits, also known as Pauli measurements.

In Q#, the `Measure` operation measures one or more qubits in the specified Pauli basis, which can be `PauliX`, `PauliY`, or `PauliZ`. `Measure` returns a `Result` type of either `Zero` or `One`.

To implement a measurement in the computational basis $\{|0\rangle, |1\rangle\}$, you can also use the `M` operation, which measures a qubit in the Pauli Z-basis. This makes `M` equivalent to `Measure([PauliZ], [qubit])`.

For example, the `Superposition` program uses the `M` operation:

Q#

```
// Measure the qubit in the Z-basis.
let result = M(q);
```

Resetting qubits

In Q#, qubits **must** be in the $|0\rangle$ state when they're released to avoid errors in the quantum hardware. You can reset a qubit to the $|0\rangle$ state using the `Reset` operation at the end of the program. Failure to reset a qubit results in a runtime error.

Q#

```
// Reset a qubit.
Reset(q);
```

Standard library namespaces

The Q# standard library has built-in namespaces that contain functions and operations you can use in quantum programs. For example, the `Std.Intrinsic` namespace contains commonly used operations and functions, such as `M` to measure results and `Message` to display user messages anywhere in the program.

To call a function or operation, you can specify the full namespace or use an `import` statement, which makes all the functions and operations for that namespace available and makes your code more readable. The following examples call the same operation:

Q#

```
Std.Intrinsic.Message("Hello quantum world!");
```

Q#

```
// imports all functions and operations from the Std.Intrinsic namespace.
import Std.Intrinsic.*;
Message("Hello quantum world!");
```

```
// imports just the `Message` function from the Std.Intrinsic namespace.
import Std.Intrinsic.Message;
Message("Hello quantum world!");
```

> **Note**
>
> The `Superposition` program doesn't have any `import` statements or calls with full namespaces. That's because the Q# development environment automatically loads two namespaces, `Std.Core` and `Std.Intrinsic`, which contain commonly used functions and operations.

You can take advantage of the `Std.Measurement` namespace by using the `MResetZ` operation to optimize the `Superposition` program. `MResetZ` combines the measurement and reset operations into one step, as in the following example:

Q#

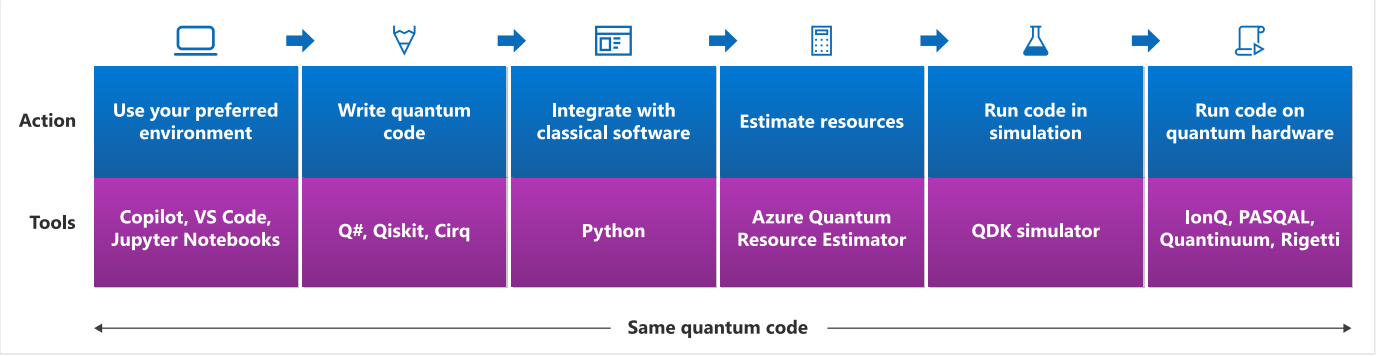
```
// Import the namespace for the MResetZ operation.
import Std.Measurement.*;

@EntryPoint()
operation MeasureOneQubit() : Result {
    // Allocate a qubit. By default, it's in the 0 state.
    use q = Qubit();
    // Apply the Hadamard operation, H, to the state.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q);
    // Measure and reset the qubit, and then return the result value.
    return MResetZ(q);
}
```

Learn to develop quantum programs with Q# and Azure Quantum

Q# and Azure Quantum are a powerful combination for developing and running quantum programs. With Q# and Azure Quantum, you can write quantum programs, simulate their behavior, estimate resource requirements, and run them on real quantum hardware. This integration allows you to explore the potential of quantum computing and develop innovative solutions for complex problems. Whether you are a beginner or an experienced quantum developer, Q# and Azure Quantum provide the tools and resources you need to unlock the power of quantum computing.

The following diagram shows the stages through which a quantum program passes when you develop it with Q# and Azure Quantum. Your program starts with the development environment and ends with the submission of the job to real quantum hardware.



Let's break down the steps in the diagram.

Choose your development environment

Run your quantum programs in your preferred development environment. You can use a local development environment with Visual Studio Code, or combine Q# code with Python code in Jupyter Notebook. For more information, see Different ways to run Q# programs.

Write your quantum program

You can write quantum programs in Q# using the QDK. To get started, see Quickstart: Create your first Q# program.

Besides Q#, the QDK offers support for other languages for quantum computing, such as Qiskit and Cirq.

Integrate with Python

You can use Q# by itself or together with Python in various IDEs. For example, you can use a Q# project with a Python host program to call Q# operations or integrate Q# with Python in Jupyter Notebooks. For more information, see Integration of Q# and Python.

The %%qsharp command

By default, Q# programs in Jupyter Notebooks use the `ipykernel` Python package. To add Q# code to a notebook cell, use the `%%qsharp` command, which is enabled with the `qsharp` Python package, followed by your Q# code.

When using `%%qsharp`, keep the following in mind:

- You must first run `from qdk import qsharp` to enable `%%qsharp`.
- `%%qsharp` scopes to the notebook cell in which it appears and changes the cell type from Python to Q#.
- You can't put a Python statement before or after `%%qsharp`.
- Q# code that follows `%%qsharp` must adhere to Q# syntax. For example, use `//` instead of `#` to denote comments and `;` to end code lines.

Estimate resources

Before running on real quantum hardware, you need to figure out whether your program can run on existing hardware, and how many resources it'll consume.

The Microsoft Quantum resource estimator allows you to assess architectural decisions, compare qubit technologies, and determine the resources needed to execute a given quantum algorithm. You can choose from pre-defined fault-tolerant protocols and specify assumptions of the underlying physical qubit model.

For more information, see Run your first resource estimate.

> **Note**
>
> The Microsoft Quantum resource estimator is free of charge and doesn't require an Azure account.

Run your program in simulation

When you compile and run a quantum program, the QDK creates an instance of the quantum simulator and passes the Q# code to it. The simulator uses the Q# code to create qubits (simulations of quantum particles) and apply transformations to modify their state. The results of the quantum operations in the simulator are then returned to the program. Isolating the Q# code in the simulator ensures that the algorithms follow the laws of quantum physics and can run correctly on quantum computers.

Submit your program to real quantum hardware

Once you've tested your program in simulation, you can run it on real quantum hardware. When you run a quantum program in Azure Quantum, you create and run a **job**. To submit a job to the Azure Quantum providers, you need an Azure account and quantum workspace. If you don't have a quantum workspace, see Create an Azure Quantum workspace.

Azure Quantum offers some of the most compelling and diverse quantum hardware. See Quantum computing providers for the current list of supported hardware providers.

Once you submit your job, Azure Quantum manages the job lifecycle, including job scheduling, execution, and monitoring. You can track the status of your job and view the results in the Azure Quantum portal.

Related content

- Different ways to run Q# programs
- Set up the Microsoft Quantum Development Kit
- Quickstart: Create your first Q# program

👉 **Learn more about the Microsoft Quantum platform**

Last updated on 05/15/2026

Develop OpenQASM programs in the Microsoft Quantum Development Kit

The Microsoft Quantum Development Kit (QDK) provides different development environments for OpenQASM programs with Azure Quantum integration. In this article, you learn how to develop and run OpenQASM code in:

- The QDK extension in Visual Studio Code (VS Code)
- The QDK Python library in Jupyter Notebook.

Developer tools like CodeLends, IntelliSense, code completion, and breakpoint debugging are available only in the VS Code extension. Other features are available only in the Python library. For example, you can pass Q# and OpenQASM callables as Python objects, and compile OpenQASM programs that take inputs into QIR.

Prerequisites

- Install the latest version of VS Code
- To work directly with OpenQASM files, install the latest version of the QDK extension in VS Code.
- To work with OpenQASM in Python and Jupyter Notebook, install the latest version of the Python extension and the Jupyter extension in VS Code.
- Install the latest version of the `qdk` Python library with the `jupyter` extra:

  Bash

  ```
  pip install --upgrade "qdk[jupyter]"
  ```

Work with OpenQASM in VS Code using the QDK extension

To write OpenQASM code in VS Code, open a Q# project and create an OpenQASM file with a `.qasm` extension. The QDK automatically recognizes the `.qasm` file as OpenQASM, and the QDK features for OpenQASM are available as you write your code.

The following QDK features are available for OpenQASM programs in VS Code:

- Code completion
- Code lenses (**Run**, **Histogram**, **Estimate**, **Debug**, and **Circuit**)
- Syntax highlighting and basic syntax features, such as brace matching
- Error checking in OpenQASM source files
- Breakpoint debugging and script execution for OpenQASM source files
- Debugging with live circuit generation
- Integration with Azure Quantum for quantum job submission

To run your OpenQASM program from VS Code, you can use either the QDK's local simulator or you can submit the code as a job to Azure Quantum.

Run your code on the QDK local simulator

To run your OpenQASM code on the local sparse quantum simulator in VS Code, open a `.qasm` file and then select the **Run** command from the code lens at the beginning of the file. The output from your program displays in the VS Code terminal.

> ⓘ **Note**
>
> The QDK automatically attempts to set a QIR target profile that supports your program. If the QIR target profile doesn't support operations in your program, you get compiler errors when you run your OpenQASM code. For more information about target profiles, see Different types of target profiles in Azure Quantum.

Built-in sample programs

The QDK comes with several built-in quantum algorithm samples for OpenQASM. To try out the sample algorithms, enter `sample` in an empty `.qasm` file. A completion list appears that contains the sample algorithms. Choose a sample and then run the generated code.

Debug locally in VS Code

The QDK includes a debugger for OpenQASM. The debugger supports breakpoint-style debugging and can render the program state.

To start the debugger, select the **Debug** command from the code lens.

In debug mode, you can set breakpoints in your code to help you debug. The state of your program is displayed in the **RUN AND DEBUG** pane. For example, you can view your local variable states and qubit states.

Run your code on Azure Quantum

To run OpenQASM programs on Azure Quantum targets, you need to have an Azure Quantum workspace. To create an Azure Quantum workspace, see Create an Azure Quantum workspace.

To submit OpenQASM jobs to Azure Quantum from VS Code, you need to connect to your Azure Quantum workspace. To connect to your workspace from VS Code, see Connect with a connection string.

To submit an OpenQASM job to Azure Quantum from VS Code, follow these steps:

1. Open the `qasm` file that contains your program in VS Code.
2. Select the **Microsoft Quantum** icon to open the Microsoft Quantum panel.
3. Expand the **QUANTUM WORKSPACES** dropdown and select your workspace.
4. Expand the **Providers** dropdown and choose a provider, and then choose the target that you want to run your job on. For example, submit your job to the `rigetti.svm.sim` target to run your program on Rigetti's backend simulator. For more information on provider and target availability, see Add or remove a provider in an existing workspace.
5. Select the play icon next to the target name.
6. To submit the job, enter values for the **Job name** prompt and the **Number of shots** prompt.

To view the times and status of a job, expand the **Jobs** dropdown and hover over the job name. To download the job results, select the histogram or text icons next to the job name.

Work with OpenQASM in Python and Jupyter Notebook

With the QDK Python library, you can run OpenQASM code directly and pass OpenQASM circuits as operations to Q# programs. Write your OpenQASM code as a Python string, and then use that string to run, import, or compile your OpenQASM code in the Python environment.

Run OpenQASM directly in Python

To run OpenQASM code directly, use the `run` function from the `qdk.openqasm` module. For example, the following code runs 10 shots on the local simulator of a simple OpenQASM program with noise:

Python

```
from qdk import BitFlipNoise
from qdk.openqasm import run

qasm_code = """
    include "stdgates.inc";
    qubit[2] q;
    reset q;
    h q[0];
    cx q[0], q[1];
    bit c = measure q[1];
    """

results = run(qasm_code, shots=10, noise=BitFlipNoise(0.1))
print(results)
```

Call an OpenQASM program in Q#

You can use the `import_openqasm` function to store an OpenQASM program as a Python object and then call that object later in your Python code. For example, the following cell stores an OpenQASM circuit in a Python object called `bell`:

Python

```
from qdk import qsharp
from qdk.openqasm import import_openqasm

qasm_code = """
    include "stdgates.inc";
    qubit[2] q;
    reset q;
    h q[0];
    cx q[0], q[1];
    bit c = measure q[1];
    """

import_openqasm(qasm_code, name="bell")
```

When `bell` is part of your environment, you can run it directly from a Q# cell:

Q#

```
%%qsharp

use qs = Qubit[2];
bell(qs)
```

You can also use the QDK package functionality with `bell`, such as noisy simulation, circuit rendering, and code generation.

Parameterized OpenQASM programs

The following cell creates an OpenQASM circuit that takes an angle `theta` as input, then calls the circuit as a Python function:

Python

```
from qdk.openqasm import import_openqasm, ProgramType

qasm_code = """
    include "stdgates.inc";
    input float theta;
    qubit[2] q;
    rx(theta) q[0];
    rx(-theta) q[1];
    bit[2] c;
    c = measure q;
    """

import_openqasm(qasm_code, name="parameterized_circuit",
program_type=ProgramType.File)

from qdk.code.qasm_import import parameterized_circuit

parameterized_circuit(1.57)
```

Pass an OpenQASM program to Q#

With the QDK Python library, you can directly pass an OpenQASM program to a Q# program. For example, the following code creates an OpenQASM program called `Entangle` and then uses `Entangle` as an operation in a Q# program:

Python

```
from qdk.openqasm import import_openqasm
from qdk import qsharp

qasm_code = """
    include "stdgates.inc";
    qubit[2] qs;
    h qs[0];
    cx qs[0], qs[1];
    """

import_openqasm(qasm_code, name="Entangle")
```

```
qsharp.eval("{ use qs = Qubit[2]; Entangle(qs); MResetEachZ(qs) }")
```

When you import Q# and import OpenQASM programs in a Python environment with `qsharp.eval` and `import_openqasm`, these programs become Python objects that you can call in your Python code. For example, the following code creates a Q# program called `TestAntiCorrelation` and passes the `Entangle` OpenQASM program to `TestAntiCorrelation`:

Python

```
qsharp.eval("""
    operation TestAntiCorrelation(entangler : Qubit[] => Unit) : Result[] {
        use qs = Qubit[2];
        X(qs[1]);
        entangler(qs);
        MResetEachZ(qs)
    }
    """)

from qdk.code import Entangle, TestAntiCorrelation

TestAntiCorrelation(Entangle)
```

Compile OpenQASM programs into QIR

You can compile OpenQASM code into Quantum Intermediate Representation (QIR) with the `compile` function.

For example, the following cell compiles OpenQASM code for a parameterized circuit into QIR:

Python

```
from qdk import TargetProfile
from qdk.openqasm import import_openqasm, ProgramType
from qdk import qsharp

qsharp.init(target_profile=TargetProfile.Base)

qasm_code = """
    include "stdgates.inc";
    input float theta;
    qubit[2] q;
    rx(theta) q[0];
    rx(-theta) q[1];
    bit[2] c;
    c = measure q;
    """

import_openqasm(qasm_code, name="parameterized_circuit",
```

```
program_type=ProgramType.File)

from qdk.code.qasm_import import parameterized_circuit

bound_compilation = qsharp.compile(parameterized_circuit, 1.57)
print(bound_compilation)
```

> ⓘ **Note**
>
> To compile OpenQASM programs that take input variables into QIR, you need to use the QDK Python library. You can't compile parameterized OpenQASM programs into QIR using the QDK extension in VS Code.

Submit OpenQASM programs to Azure Quantum

To submit an OpenQASM program to run on an Azure Quantum target, compile the OpenQASM code into QIR and then submit the QIR to run on Azure Quantum.

For example, compile a simple entanglement circuit into QIR.

Python

```python
from qdk.openqasm import compile

qasm_code = """
    include "stdgates.inc";
    bit[2] c;
    qubit[2] q;
    h q[0];
    cx q[0], q[1];
    c = measure q;
"""

qir_compilation = compile(qasm_code)
```

To run a QIR job to Azure Quantum, connect to your quantum workspace and submit the job to one of the available targets in your workspace. For example, submit the entanglement QIR to run on the IonQ backend simulator.

Python

```python
from qdk.azure import Workspace

workspace = Workspace(resource_id="") # Add the resource ID of your workspace

target = workspace.get_targets('ionq.simulator')
```

```
job = target.submit(qir_compilation, shots=100)

print("Job submitted. Waiting for results...")

results = job.get_results()
print(results)
```

Ongoing OpenQASM development in the QDK

Support for OpenQASM in the QDK is ongoing.

For a list of OpenQASM features that are under development in the QDK, see Known limitations and issues on the QDK GitHub repository wiki.

Last updated on 06/22/2026

Stim support for quantum error correction in the QDK

Stim is an open-source tool from Google Quantum AI for fast simulation and analysis of quantum stabilizer circuits, especially quantum error correction (QEC) circuits. QEC protocols commonly use repeated rounds of Clifford operations and Pauli measurements, and researchers often analyze QEC protocols with Pauli noise models. This circuit structure is much more efficient to simulate classically than a general quantum circuit, so researchers can test many runs of large error-correction circuits.

The Microsoft Quantum Development Kit (QDK) includes QDK-Stim, which extends Stim with additional features. You can write circuits with the Stim instruction set, compile them to quantum intermediate representation (QIR), and run them on the QDK simulators. QDK-Stim adds support for qubit loss, post-selection, repeat-until-success patterns, and non-Clifford operations.

Access QDK-Stim through the `qdk.stim` module. The module accepts Stim source code and runs the compiled program on QDK simulators. The QDK Python API doesn't have the full functionality of the `stim` Python package and doesn't expose classes such as `stim.Circuit`, compiled samplers, or detector error models.

For more information about Stim, see the Stim repository on GitHub.

Prerequisites

To use Stim circuits with the QDK, you need:

- Python 3.10 or later.
- `pip` or another Python package manager.
- A Python environment, such as a virtual environment or Jupyter Notebook.
- Basic familiarity with the Stim circuit format.

Install and use Stim with the QDK

Install or update the QDK Python package:

Bash

```
pip install --upgrade qdk
```

Import the `stim` module from `qdk`, define a Stim circuit as a string, and pass the circuit to `stim.run`. The following example prepares and measures a Bell state:

Python

```python
from qdk import stim

bell = """
H 0
CX 0 1
M 0 1
"""

results = stim.run(bell, shots=100, type="stabilizer")
print(results[:5])
```

Each item in `results` contains the measurement results from one shot. A standard result is `Result.Zero` or `Result.One`. QDK-Stim also includes `Result.Loss` for measurements of lost qubits.

For standard Stim stabilizer circuits, start with `type="stabilizer"`. For more information about the simulators, see Overview of quantum simulators in the QDK.

QDK features for Stim circuits

The QDK extends the Stim instruction set and connects it to QIR and the QDK simulators. These additions support simulation workflows that aren't available in Stim.

Compile Stim circuits to QIR

The QDK compiles Stim circuit source code to QIR before it runs the circuit. The `stim.run` function performs both steps in one call. Use `stim.compile` when you want to inspect the generated QIR or run it on another simulator or service.

The `stim.compile` function returns the QIR source code and the QDK `NoiseConfig` that the simulator needs. The following example compiles a Bell-state circuit and prints the QIR:

Python

```python
from qdk import stim

bell = """
H 0
CX 0 1
M 0 1
```

```
"""

qir, noise = stim.compile(bell)
print(qir)
```

Add noise to Stim circuits

You can add noise to a Stim circuit with Stim noise instructions or with a QDK `NoiseConfig`. Choose the method that best matches how you define the circuit and noise model:

Expand table

Noise method	How to use it	Use when
Stim noise instructions	Add instructions such as `X_ERROR`, `DEPOLARIZE1`, and `CORRELATED_ERROR` to the Stim code.	The noise is part of the Stim circuit definition.
QDK noise model	Define a `NoiseConfig` and pass it with the `noise` argument.	You want to apply an existing QDK noise model to a Stim circuit that doesn't contain noise instructions.

The `NoiseConfig` returned by `stim.compile` represents the noise instructions in the Stim circuit. It isn't a separate noise model.

> **⊗ Caution**
>
> Choose one noise method. Use either Stim noise instructions or a QDK `NoiseConfig`. If you use both, the simulator applies both noise models.

The following example defines a Stim circuit that has depolarizing noise:

Python

```
from qdk import stim

noisy_bell = """
H 0
CX 0 1
DEPOLARIZE2(0.01) 0 1
M 0 1
"""

results = stim.run(noisy_bell, shots=1000, type="stabilizer")
```

For more information, see How to build noise models for quantum simulations in the QDK.

Model qubit loss

The QDK adds loss instructions and targets to the Stim circuit format. Measurement of a lost qubit returns `Result.Loss` instead of `Result.Zero` or `Result.One`.

Expand table

Construct	Syntax	Behavior
Loss error	`LOSS_ERROR(p) 0 1`	Independently loses each target with probability `p`.
Correlated loss target	`L0`	Represents loss of a qubit in a `CORRELATED_ERROR` or `ELSE_CORRELATED_ERROR` event.
Loss inspection	`PEEK_LOSS 0 1`	Appends `1` for a lost qubit and `0` for a present qubit without measuring or resetting it.

The following example applies independent loss to two qubits, inspects their loss state, and then measures them:

Python

```
from qdk import Result, stim

loss_program = """
H 0
LOSS_ERROR(0.2) 0 1
PEEK_LOSS 0 1
M 0 1
"""

results = stim.run(loss_program, shots=100, type="stabilizer")

symbols = {
    Result.Zero: "0",
    Result.One: "1",
    Result.Loss: "L",
}

formatted = ["".join(symbols[value] for value in shot) for shot in results[:10]]

print(formatted)
```

Each formatted shot contains four characters. The first two characters are the `PEEK_LOSS` values, and the last two are the measurement results. A final `L` means that the corresponding qubit was

lost.

Use `L` targets to combine loss and Pauli errors in one correlated event:

text

```
CORRELATED_ERROR(0.1) L0 X1
ELSE_CORRELATED_ERROR(0.1) L1
```

> ⓘ Note
>
> In QDK-Stim, `CORRELATED_ERROR` and `ELSE_CORRELATED_ERROR` must come one immediately after the other. In Stim, they don't necessarily need to.

Post-select measurement results

The QDK adds `SELECT` blocks for post-selection and repeat-until-success patterns based on parity measurements and qubit-loss checks. If a condition in a `SELECT` block fails, the simulator reruns that block until all its conditions pass. A nested block restarts independently from its enclosing block.

Expand table

Construct	Syntax	Behavior
Selection block	`SELECT { ... }`	Reruns the block body until every selection condition passes.
Parity condition	`REQUIRE rec[-1]`	Requires the referenced measurement records to have even parity. A lost record also fails the condition.
Inverted record	`REQUIRE !rec[-1]`	Inverts a record before the parity check. With one record, this keeps measurement result `1`.
Loss-only condition	`NOTLEAKED rec[-1]`	Requires the referenced measurements to not be loss, but doesn't constrain their bit values.

List multiple `rec[...]` targets in one `REQUIRE` instruction to select their joint parity. The following program keeps only non-lost results with even parity, so the final measurements are `00` or `11`:

Python

```
from qdk import stim
```

```
post_selected = """
SELECT {
    H 0
    H 1
    LOSS_ERROR(0.1) 0 1
    M 0 1
    REQUIRE rec[-1] rec[-2]
}
"""

results = stim.run(post_selected, shots=1000, type="stabilizer")
```

Use `NOTLEAKED` when both `0` and `1` are valid outcomes, but you want to discard measurements of lost qubits:

text

```
SELECT {
    H 0
    LOSS_ERROR(0.2) 0
    M 0
    NOTLEAKED rec[-1]
}
```

Each condition must reference at least one record that the current `SELECT` block or a nested block produces. The compiler rejects a condition if a rerun of the block can't change any of its referenced records.

Simulate non-Clifford operations

Stim focuses on stabilizer circuits and doesn't support non-Clifford operations such as T and Toffoli gates. QDK-Stim adds operations similar to those supported by Clifft, a simulator for near-Clifford circuits.

Expand table

Category	Instructions	Target grouping
Phase gates	`T`, `T_DAG`	One qubit per target
Pauli-product phase gates	`TPP`, `TPP_DAG`	One Pauli product per target group
Controlled gates	`CH`	Consecutive pairs
Controlled gates	`CCX`, `CCZ`	Consecutive triples
Single-qubit rotations	`R_X(a)`, `R_Y(a)`, `R_Z(a)`	One qubit per target

Category	Instructions	Target grouping
General single-qubit gates	`U3(t, p, l)`, `U(t, p, l)`	One qubit per target
Two-qubit rotations	`R_XX(a)`, `R_YY(a)`, `R_ZZ(a)`	Consecutive pairs
Pauli-product rotations	`R_PAULI(a)`	One Pauli product per target group

By default, rotation arguments use half turns. For example, `R_X(0.5)` rotates by $\pi/2$ radians. Add the `rad` suffix to specify radians directly, such as `R_X(0.5rad)`. You can use either `U3` or `U` to mix angle units because both are the same.

The following circuit uses a Toffoli gate and an arbitrary Y rotation:

Python

```python
from qdk import stim

near_clifford = """
X 0 1
CCX 0 1 2
R_Y(0.25) 3
M 0 1 2 3
"""

results = stim.run(near_clifford, shots=1000, type="stabilizer")
```

The QDK stabilizer simulator branches the stabilizer decomposition when it encounters a non-Clifford operation. This approach remains efficient when non-Clifford operations are sparse, but the required state space grows exponentially with the number of non-Clifford operations. Use `type="cpu"` or `type="gpu"` for circuits that contain many non-Clifford operations.

For the complete non-Clifford gate definitions, see the Clifft supported gates reference.

Current limitations

The QDK supports most instructions in the Stim gate reference, with the following limitations. For the current implementation status, see the QDK-Stim missing-features tracking issue.

Expand table

Stim feature	Current QDK behavior
`DETECTOR` and `OBSERVABLE_INCLUDE`	Parsed and ignored. The QDK doesn't provide detector or observable sampling.

Stim feature	Current QDK behavior
`HERALDED_ERASE` and `HERALDED_PAULI_CHANNEL_1`	Compilation fails because the QDK doesn't support these instructions.
`MPAD`	Compilation fails because the QDK doesn't support these instructions.
Sweep-bit targets, such as `CX sweep[5] 7`	Compilation fails because the QDK doesn't support sweep-bit targets.
Pauli products that reduce to the identity, such as `MPP Z0*Z0`	Compilation fails because the QDK doesn't support the resulting identity target.

Use Stim when you need detector error models, compiled Stim samplers, Sinter, or other Stim Python APIs. Use `qdk.stim` when you want to compile Stim code to QIR and run the QIR on QDK simulators, or use the QDK loss, post-selection, and non-Clifford extensions.

Related content

- Overview of quantum simulators in the QDK
- How to build noise models for quantum simulations in the QDK
- Stim in the QDK sample notebook in the QDK repository on GitHub

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/21/2026

Overview of quantum simulators in the QDK

The Microsoft Quantum Development Kit (QDK) includes quantum simulators that run quantum programs on your local machine. Use these local simulators to test and develop your programs before you submit them to Azure Quantum targets.

The QDK provides four primary simulators to run quantum programs:

- The sparse simulator
- The stabilizer simulator
- The GPU simulator
- The CPU simulator

The sparse simulator

The sparse simulator represents qubit states as sparse vectors and uses sparse matrix algebra to quickly simulate programs that generate few nonzero amplitudes. It's the default simulator for Q# and OpenQASM programs in the QDK extension and for the `qsharp.run` and `openqasm.run` Python APIs. Use the sparse simulator when you want quick simulations of Q#, OpenQASM, or Qiskit programs, or when you want to display the quantum state of your program during simulation.

The sparse simulator is available in both the QDK extension for Visual Studio Code (VS Code) and the QDK Python package. This simulator is the only simulator available in the QDK extension, so the extension automatically uses it when you run Q# and OpenQASM programs in VS Code. In VS Code, use the Q# `DumpMachine` function or the debugger to display qubit state vectors at different points in the simulation.

Both the VS Code extension and the Python package support noise models for sparse simulations of Q# and OpenQASM programs. Only the Python package supports sparse simulation of Qiskit programs, but it doesn't support noise models for Qiskit simulations.

For more information about how to run simulations in the QDK extension, see Run quantum simulations in the QDK extension for Visual Studio Code.

The stabilizer simulator

The stabilizer simulator quickly and efficiently simulates Clifford-dominated programs with thousands of qubits. Clifford gates are common components of quantum error correction circuits. Clifford operations don't increase the number of stabilizer branches, so the simulator can apply them efficiently even in wide circuits. Common Clifford operations include:

- X, Y, and Z
- S and $S^\dagger$
- H, or Hadamard
- CX, CY, and CZ
- $SWAP$

The stabilizer simulator also uses stabilizer branching to simulate circuits with a limited number of non-Clifford operations. The simulator supports the following non-Clifford operations:

- T and $T^\dagger$
- Arbitrary-angle R_X, R_Y, R_Z, R_{XX}, R_{YY}, and R_{ZZ} rotations

Each non-Clifford operation can increase the number of stabilizer branches. Adding non-Clifford operations can increase memory usage and shot time exponentially. Use the stabilizer simulator for wide circuits with mostly Clifford operations and a modest number of non-Clifford operations. Use a sparse or full-state simulator for circuits that have a lot of non-Clifford operations.

The stabilizer simulator supports adaptive execution, Pauli noise, and qubit loss. It's available in the QDK Python package for Q#, OpenQASM, Qiskit, and QIR programs, but isn't available in the QDK extension for VS Code.

In Python APIs with a simulator selector, use either `"stabilizer"` or `"clifford"` to select the stabilizer simulator.

The GPU simulator

The GPU simulator is a full-state simulator that uses your machine's GPU to run many shots of your quantum program in parallel. The simulator supports all gate types but requires significant computing resources. The simulator can model up to 27 qubits. The power of your machine's GPU determines the performance, but performance is optimal for programs that have around 20 qubits and lots of shots.

The GPU simulator accepts QIR or Qiskit as input through certain QDK Python package APIs. The VS Code extension doesn't support the GPU simulator. There is also rich support for noise models on any type of quantum gate or operation.

The CPU simulator

Like the GPU simulator, the CPU simulator is a full-state simulator that runs programs with any type of quantum gate. But it's slower because it can't run multiple shots in parallel. This simulator scales up to around 25 qubits, but slows down exponentially with more qubits because of memory issues. Use the CPU simulator when your machine doesn't have a GPU, or when your program has few qubits and doesn't require many shots.

The CPU simulator accepts QIR or Qiskit input. You can use it through certain QDK Python package APIs, but not through the VS Code extension. There's also rich support for noise models on any type of quantum gate or operation.

What simulator should I use?

Consider these factors when you choose a simulator:

- Your development environment (VS Code or Python)
- The quantum programming framework that you use
- The complexity of your program and the number of shots
- The hardware capabilities of your local machine
- The Azure Quantum targets or other quantum hardware that you want to run your programs on
- The noise models that you want to build

The following table summarizes each QDK simulator's availability and supported quantum programming frameworks:

Expand table

Simulator	Availability	Supported frameworks
Sparse	VS Code extension and Python package	Q#, OpenQASM, Qiskit
Stabilizer	Python package	Q#, OpenQASM, Qiskit, QIR
GPU	Python package	Qiskit, QIR
CPU	Python package	Qiskit, QIR

Simulations in the QDK extension for VS Code

The QDK extension for VS Code supports only the sparse simulator. To use the sparse simulator, run your Q# or OpenQASM file in VS Code. You can add Pauli noise models to sparse simulations of Q# programs in VS Code, but not to OpenQASM programs.

Simulations in the QDK Python package

The QDK Python package supports multiple quantum programming frameworks and all four QDK simulators. Compatibility varies by framework and API.

The following table lists the simulators and programming frameworks that each Python API supports:

Expand table

Python API	Sparse	Stabilizer	GPU	CPU	Programming framework
`qsharp.run`	Supported	Supported	Not supported	Not supported	Q#
`openqasm.run`	Supported	Supported	Not supported	Not supported	OpenQASM
`qiskit.QSharpBackend`	Supported	Not supported	Not supported	Not supported	Qiskit
`simulation.NeutralAtomDevice`	Not supported	Supported	Supported	Supported	QIR
`qiskit.NeutralAtomBackend`	Not supported	Supported	Supported	Supported	Qiskit
`simulation.run_qir`	Not supported	Supported	Supported	Supported	QIR

Related content

- Run quantum simulations with the QDK Python package
- Run quantum simulations in the QDK extension for Visual Studio Code
- How to build noise models for quantum simulations in the QDK
- Backend quantum simulators from quantum providers

ⓘ**Note:** The author created this article with assistance from AI. Learn more

Last updated on 10/10/2026

Run quantum simulations with the QDK Python package

The Microsoft Quantum Development Kit (QDK) Python package includes local quantum simulators that you can use to test and develop quantum programs. This article explains how to install the simulators and run Q#, OpenQASM, Qiskit, and QIR programs in Python or Jupyter Notebook.

The QDK Python package includes four primary simulators:

- Sparse simulator
- Stabilizer simulator
- GPU simulator
- CPU simulator

For more information about the QDK quantum simulators, see Overview of quantum simulators in the QDK.

To run Q# or OpenQASM programs directly in the QDK extension for Visual Studio Code (VS Code), see Run quantum simulations in the QDK extension for Visual Studio Code.

Prerequisites

To follow the Jupyter Notebook examples in this article, install these tools:

- A Python environment with Python 3.10 or later and pip
- The latest version of Visual Studio Code (VS Code)
- The latest versions of the Python extension and Jupyter extension in VS Code

Install the simulators

To use the quantum simulators in Python and Jupyter Notebook, install the latest version of the `qdk` Python package with the `jupyter` extra:

Bash

```
pip install --upgrade "qdk[jupyter]"
```

The simulators don't require the `jupyter` extra, but the extra adds the `qdk.widgets` module to visualize simulation results in Jupyter Notebook.

Select a simulator based on the quantum programming framework and Python API you use.

Call the sparse simulator

The sparse simulator supports Q#, OpenQASM, and Qiskit programs. This table shows how to run 10 shots with the `qdk` package in each framework.

Expand table

Programming framework	Python API	Example call
Q#	`qsharp.run`	`qsharp.run("MyQsharpProgram()", shots=10)`
OpenQASM	`openqasm.run`	`openqasm.run(my_qasm_program, shots=10)`
Qiskit	`qiskit.QSharpBackend`	`QSharpBackend().run(my_qiskit_program, shots=10)`

> **Note**
>
> The sparse simulator supports noise models through the `noise` parameter for Q# and OpenQASM programs, but not for Qiskit programs.

For example, follow these steps to run the Bell pair Q# program in a Jupyter notebook in VS Code:

1. In VS Code, select **View** > **Command Palette**.
2. Enter **Create: New Jupyter Notebook**. An empty notebook file opens in a new tab.
3. In the first cell, import the `qsharp` module from the `qdk` package.

   Python

   ```python
   from qdk import qsharp
   ```

4. In a new cell, use the `%%qsharp` magic command to write the Q# code.

   Q#

   ```
   %%qsharp
   ```

```
operation Main() : (Result, Result) {
    use (q1, q2) = (Qubit(), Qubit());
    PrepareBellPair(q1, q2);
    (MResetZ(q1), MResetZ(q2))
}

operation PrepareBellPair(q1 : Qubit, q2 : Qubit) : Unit {
    H(q1);
    CNOT(q1, q2);
}
```

5. Create a new cell. To run the Q# program on the sparse simulator, call the `run` function from the `qsharp` module and specify the number of shots.

Python

```
qsharp.run("Main()", shots=10)
```

Call the stabilizer simulator

The stabilizer simulator efficiently runs wide, Clifford-dominated circuits. Stabilizer branching also supports a limited number of non-Clifford operations, such as T and $T^\dagger$ gates and arbitrary-angle Pauli rotations. Because the number of branches can grow exponentially with the number of non-Clifford operations, use a full-state simulator for circuits with substantial non-Clifford evolution.

The stabilizer simulator supports Q#, OpenQASM, Qiskit, and QIR programs. This table shows how to run 10 shots in each framework.

Expand table

Programming framework	Python API	Example call
Q#	`qsharp.run`	`qsharp.run("MyQsharpProgram()", type="stabilizer", shots=10)`
OpenQASM	`openqasm.run`	`openqasm.run(my_qasm_program, type="stabilizer", shots=10)`
Qiskit	`qiskit.NeutralAtomBackend`	`NeutralAtomBackend().run(my_qiskit_program, simulator_type="stabilizer", shots=10)`
QIR	`simulation.NeutralAtomDevice`	`simulation.NeutralAtomDevice().simulate(my_qir, type="stabilizer", shots=10)`

Programming framework	Python API	Example call
QIR	`simulation.run_qir`	`simulation.run_qir(my_qir, type="stabilizer", shots=10)`

> **Note**
>
> You can also use `simulation.run_qir` to run logical programs through a quantum error correction protocol. This workflow has additional protocol and decoder requirements. For more information, see Run and decode logical programs.

The `NeutralAtomDevice` and `NeutralAtomBackend` APIs are for simulations on neutral atom quantum computers specifically. For more information, see Neutral atom device simulator overview.

For example, this Python code runs an OpenQASM program with a non-Clifford T gate on the stabilizer simulator:

Python

```python
from qdk.openqasm import run

qasm_code = """
include "stdgates.inc";
qubit[1] q;
bit[1] result;

h q[0];
t q[0];
h q[0];
result = measure q;
"""

run(qasm_code, type="stabilizer", shots=1000)
```

> **Note**
>
> Both `"stabilizer"` and `"clifford"` select the stabilizer simulator. The API determines whether the parameter is `type` or `simulator_type`.

The T gate creates stabilizer branches. The final measurement samples the interference between those branches.

Call the GPU and CPU simulators

The GPU and CPU simulators support Qiskit and QIR programs. This table shows how to run 10 shots on the GPU simulator with the `qdk` package in each framework. To use the CPU simulator, replace `gpu` with `cpu` in the simulator type parameter.

Expand table

Programming framework	Python API	Example call
Qiskit	`qiskit.NeutralAtomBackend`	`NeutralAtomBackend().run(my_qiskit_program, simulator_type="gpu", shots=10)`
QIR	`simulation.NeutralAtomDevice`	`simulation.NeutralAtomDevice().simulate(my_qir, type="gpu", shots=10)`
QIR	`simulation.run_qir`	`simulation.run_qir(my_qir, type="gpu", shots=10)`

For example, this Python code converts a simple OpenQASM program to QIR and runs the QIR on the GPU simulator:

Python

```python
from qdk.openqasm import compile
from qdk.simulation import run_qir

qasm_code = """
    include "stdgates.inc";
    qubit[2] q;
    reset q;
    h q[0];
    cx q[0], q[1];
    bit c = measure q[1];
    """

qir = compile(qasm_code)
run_qir(qir, type="gpu", shots=10)
```

Related content

- Set up the Microsoft Quantum Development Kit
- How to install QDK for error correction

ⓘ**Note:** The author created this article with assistance from AI. Learn more

Last updated on 10/22/2026

Run quantum simulations in the QDK extension for Visual Studio Code

The Microsoft Quantum Development Kit (QDK) extension for Visual Studio Code (VS Code) automatically runs Q# and OpenQASM programs on its local sparse simulator.

This article explains how to run programs on the local simulator and how to add Pauli noise to Q# simulations. To use the other QDK simulators from Python, see Run quantum simulations with the QDK Python package.

How the local simulator works

The sparse simulator stores only the nonzero amplitudes of a quantum state. When most computational basis amplitudes are zero, this representation uses less memory and can simulate some programs with more qubits than a full-state simulator, which stores every amplitude.

The performance benefit depends on whether the quantum state remains sparse in the computational basis. For more information, see Jaques and Häner (arXiv:2105.01533).

Run a quantum program in VS Code

1. Install the QDK extension for VS Code.
2. Open a Q# (`.qs`) or OpenQASM (`.qasm`) file.
3. Select **Run** from the code lens at the start of the file or above the Q# `Main` operation.

The simulation output appears in the VS Code terminal. For Q# programs, select **Debug** to inspect the program's quantum state as it runs.

Add Pauli noise to Q# simulations

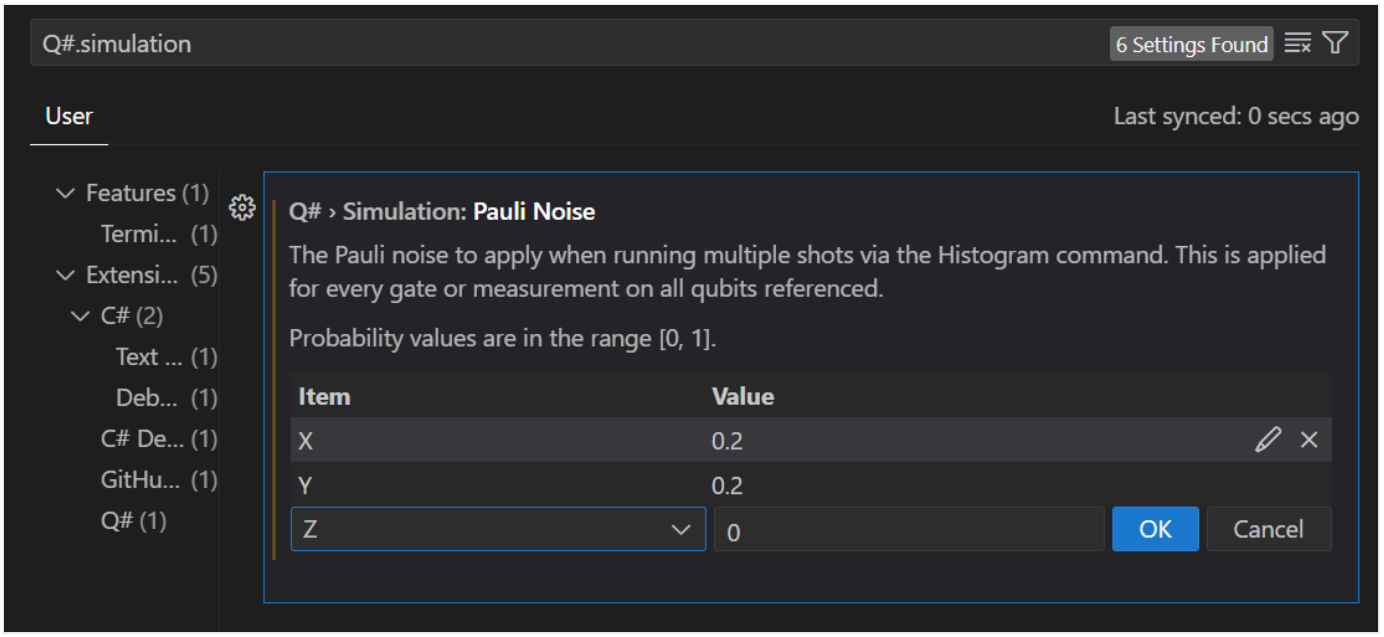
The sparse simulator supports Pauli noise for Q# programs in the QDK extension. Use `ConfigurePauliNoise` to set the probabilities of Pauli X, Y, and Z errors in a program. The QDK extension settings also support a global Pauli noise model.

> **Note**

You can't add a noise model to an OpenQASM simulation in the QDK extension. To build operation-specific noise models for Q# or OpenQASM programs, use the QDK Python package and the `NoiseConfig` API.

Add Pauli noise in the VS Code settings

- In VS Code, set global Pauli noise for Q# programs in the **Q#** > **Simulation:Pauli Noise** user setting for the QDK extension.



The noise settings apply to histogram results for all gates, measurements, and qubits in all Q# programs in VS Code.

Without noise, a histogram for this GHZ sample program shows $|00000\rangle$ for about half the measurements and $|11111\rangle$ for the other half.

Q#

```
import Std.Diagnostics.*;
import Std.Measurement.*;

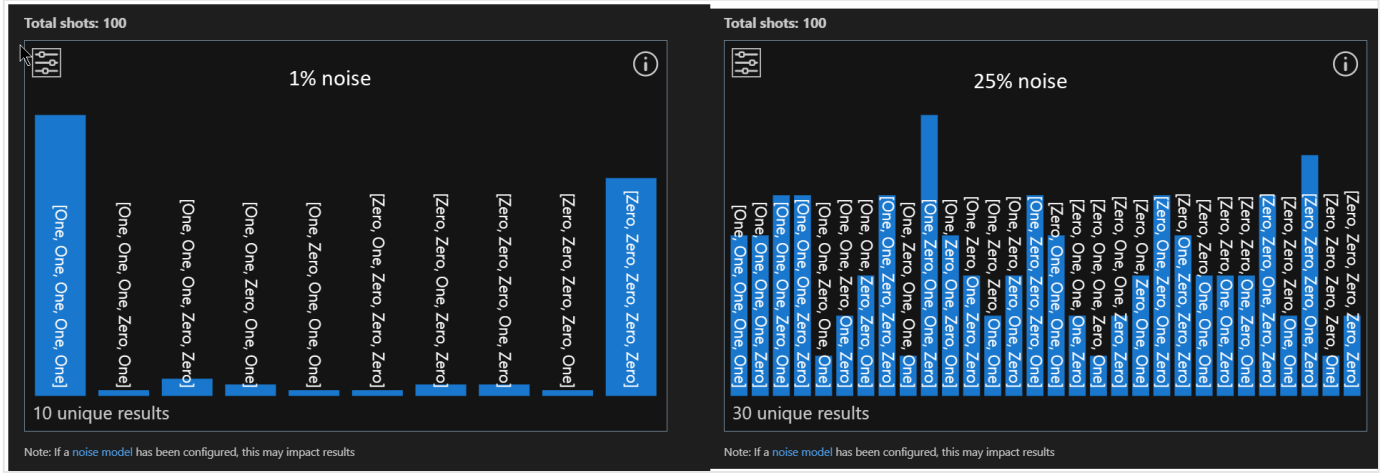
operation Main() : Result[] {
    let num = 5;
    return GHZSample(num);
}

operation GHZSample(n: Int) : Result[] {
    use qs = Qubit[n];
    H(qs[0]);
    ApplyCNOTChain(qs);
    let results = MeasureEachZ(qs);
    ResetAll(qs);
```

```
    return results;
}
```



At a 1% bit-flip noise rate, the results start to spread. At 25%, the histogram is indistinguishable from pure noise.



Add Pauli noise to individual Q# programs

Use `ConfigurePauliNoise` to set or change a Q# program's noise model and control when and where noise occurs.

> **Note**
>
> The VS Code settings apply noise to all Q# programs, but `ConfigurePauliNoise` overrides those settings for the program that calls it.

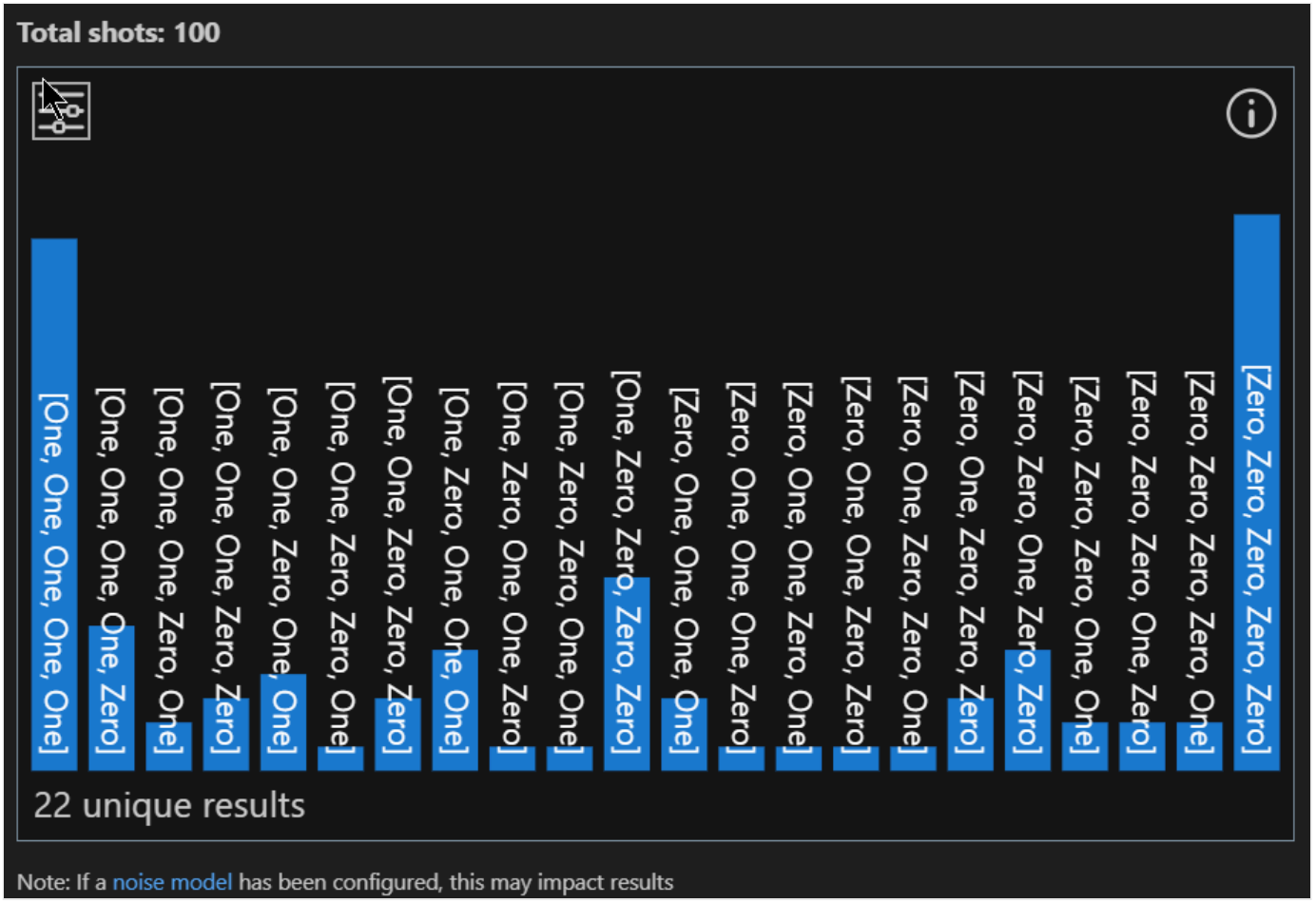
In the previous program, add noise immediately after qubit allocation:

Q#

```
operation GHZSample(n: Int) : Result[] {
    use qs = Qubit[n];

    // 5% bit-flip noise applies to all subsequent operations.
    ConfigurePauliNoise(0.05, 0.0, 0.0);

    H(qs[0]);
    ApplyCNOTChain(qs);
    let results = MeasureEachZ(qs);
    ResetAll(qs);
    return results;
}
```



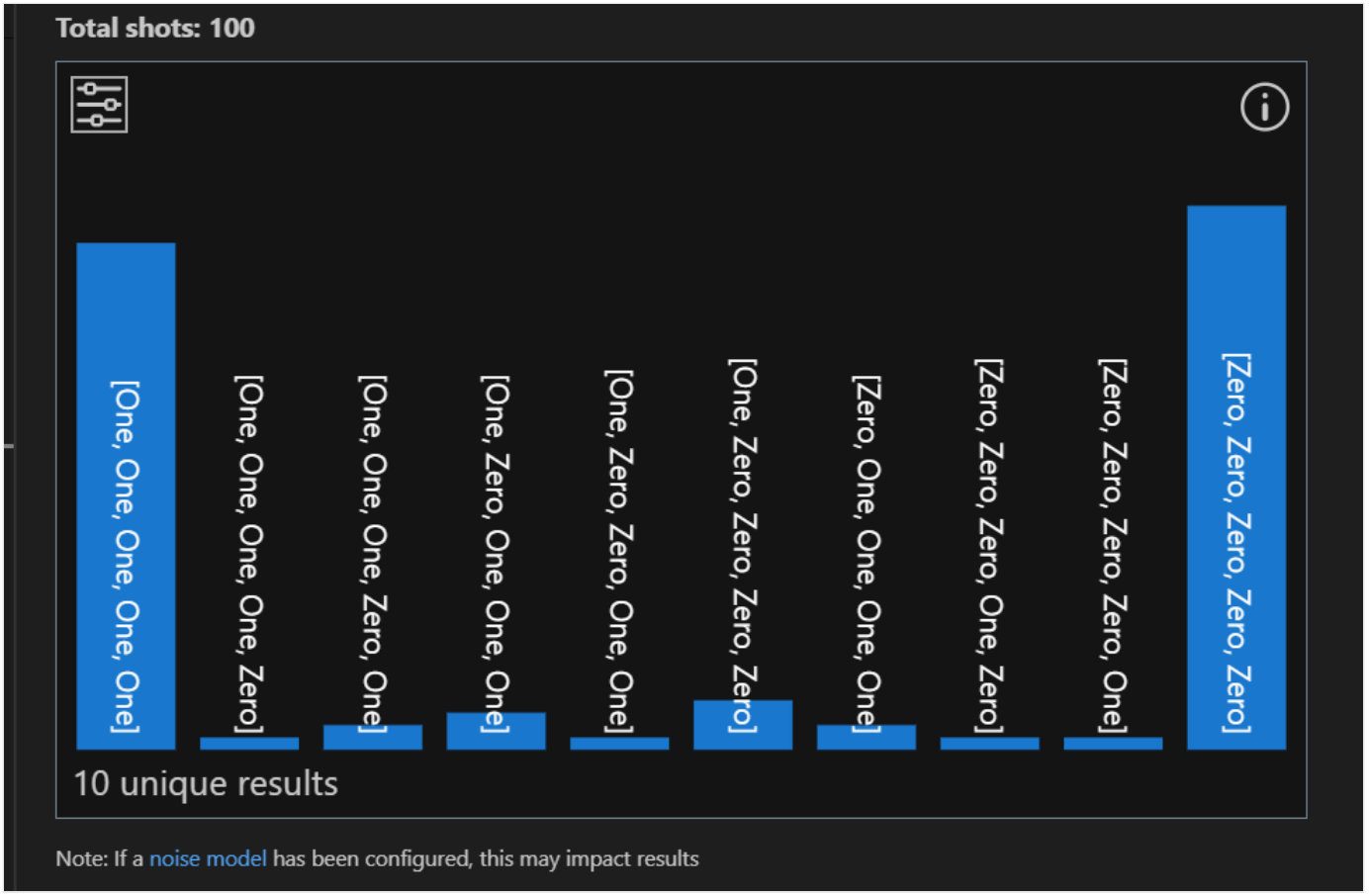
To apply noise only to the measurement operation, set the noise model immediately before measurement and clear it afterward:

Q#

```
operation GHZSample(n: Int) : Result[] {
    use qs = Qubit[n];
    H(qs[0]);
    ApplyCNOTChain(qs);

    // Noise applies only to the measurement operation.
    ConfigurePauliNoise(0.05, 0.0, 0.0);

    let results = MeasureEachZ(qs);
    ResetAll(qs);
    return results;
}
```



To change or clear the noise configuration at different points in your program, call `ConfigurePauliNoise` multiple times. For example, apply 5% bit-flip noise to the Hadamard gate and then clear the noise configuration for the rest of the program.

Q#

```
operation GHZSample(n: Int) : Result[] {
    use qs = Qubit[n];
```

```
    // Noise applies to the H operation.
    ConfigurePauliNoise(0.05, 0.0, 0.0);

    H(qs[0]);

    // Clear the noise settings.
    ConfigurePauliNoise(0.0, 0.0, 0.0);

    ApplyCNOTChain(qs);
    let results = MeasureEachZ(qs);
    ResetAll(qs);
    return results;
}
```

Other Q# noise functions

`ConfigurePauliNoise` supports any Pauli noise model. Use these functions and operations from the `Std.Diagnostics` namespace to configure or apply noise:

Expand table

Function	Description	Example
`ConfigurePauliNoise`	Sets Pauli noise for a simulation. Pass either the probabilities of Pauli X, Y, and Z errors or a Pauli noise model. The configuration applies to all subsequent gates, measurements, and qubits in a Q# program and overrides the VS Code extension noise settings. Subsequent calls override earlier configurations.	`ConfigurePauliNoise(0.1, 0.0, 0.5)` or `ConfigurePauliNoise(BitFlipNoise(0.1))`
`BitFlipNoise`	Returns a noise model with only Pauli X errors at the specified probability.	10% bit-flip noise: `ConfigurePauliNoise(BitFlipNoise(0.1))` ≡ `ConfigurePauliNoise(0.1, 0.0, 0.0)`
`PhaseFlipNoise`	Returns a noise model with only Pauli Z errors at the specified probability.	10% phase-flip noise: `ConfigurePauliNoise(PhaseFlipNoise(0.1))` ≡ `ConfigurePauliNoise(0.0, 0.0, 0.1)`
`DepolarizingNoise`	Returns a noise model with equal probabilities for Pauli X, Y, and Z errors.	6% depolarizing noise: `ConfigurePauliNoise(DepolarizingNoise(0.06))` ≡ `ConfigurePauliNoise(0.02, 0.02, 0.02)`

Function	Description	Example
`NoNoise`	Returns a model with zero error probabilities. Pass it to `ConfigurePauliNoise` to clear the current noise configuration.	`ConfigurePauliNoise(NoNoise())` ≡ `ConfigurePauliNoise(0.0, 0.0, 0.0)`

`ApplyIdleNoise`	Applies configured noise to a single qubit during simulation.	`...`
		`use q = Qubit[2];`
		`ConfigurePauliNoise(0.1, 0.0, 0.0);`
		`ApplyIdleNoise(q[0]);`
		`...`

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/10/2026

Build noise models for quantum simulations in the QDK

The Microsoft Quantum Development Kit (QDK) includes a set of quantum simulators that model how your program runs on a quantum computer. Programs that you run on a quantum computer always include some type and degree of noise. The QDK Python package lets you build custom noise models to use in your simulations through the `NoiseConfig` API.

For instructions on how to install and use the QDK simulators, see Run quantum simulations with the QDK Python package.

Types of noise

Each operation or instruction in a quantum program can introduce noise. This table lists all the operations and instructions that you can set noise for.

Expand table

Noise source	Noise model parameter	Source description
X gate	`x`	Single-qubit Pauli gate, bit flip
Y gate	`y`	Single-qubit Pauli gate, bit flip and phase flip
Z gate	`z`	Single-qubit Pauli gate, phase flip
H gate	`h`	Single-qubit Hadamard gate, creates equal superposition state
S gate	`s`	Single-qubit gate, half-pi phase shift
$S^\dagger$ gate	`s_adj`	Single-qubit gate, adjoint of S
T gate	`t`	Single-qubit gate, quarter-pi phase shift
$T^\dagger$ gate	`t_adj`	Single-qubit gate, adjoint of T
S_X gate	`sx`	Single-qubit gate, half bit-flip rotation
$S_X^\dagger$ gate	`sx_adj`	Single-qubit gate, adjoint of S_X
R_X gate	`rx`	Single-qubit gate, general phase rotation about x axis

Noise source	Noise model parameter	Source description
R_Y gate	`ry`	Single-qubit gate, general phase rotation about y axis
R_Z gate	`rz`	Single-qubit gate, general phase rotation about z axis
CX gate	`cx`	Two-qubit gate, controlled-X gate
CY gate	`cy`	Two-qubit gate, controlled-Y gate
CZ gate	`cz`	Two-qubit gate, controlled-Z gate
R_{XX} gate	`rxx`	Two-qubit gate, analogous to R_X
R_{YY} gate	`ryy`	Two-qubit gate, analogous to R_Y
R_{ZZ} gate	`rzz`	Two-qubit gate, analogous to R_Z
$SWAP$ gate	`swap`	Two-qubit gate, swaps the states of the qubits
CCX gate	`ccx`	Three-qubit gate, two-qubit controlled-X gate
Qubit movement	`mov`	Qubit movement between device zones (for neutral atom device simulation)
Qubit measurement	`mz`	Single-qubit measurement in the Pauli-Z basis
Qubit measurement and reset	`mresetz`	Single-qubit measurement and reset to 0 state

With `NoiseConfig`, you can apply four different kinds of noise to the preceding operations with specific probabilities. This table lists the noise type parameters that you can set with `NoiseConfig`. It also includes a parameter for no noise.

Expand table

Noise type	Noise model parameter	Noise description	Example use	Example description
Pauli X noise	`x`	Bit flip	`noise.z.x = 0.03`	Bit flip occurs in 3% of Z operations
Pauli Y noise	`y`	Bit flip and phase flip	`noise.sx.y = 0.01`	Bit flip and phase flip occurs in 1% of S_X operations
Pauli Z noise	`z`	Phase flip	`noise.h.z = 0.02`	Phase flip occurs in 2% of H operations

Noise type	Noise model parameter	Noise description	Example use	Example description
No noise	`i`	Identity operation, no effect	`noise.cz.ix = 0.02`	Bit flip on only the target qubit occurs in 2% of CZ operations
Qubit loss	`l`	Qubit is lost from the device	`noise.mov.l = 0.03`	Qubit is lost in 3% of movements between device zones on a neutral atom quantum computer

The noise occurs after the source operation, not instead of the source operation. For example, `noise.z.x` means that the program applies the intended Z gate to the qubit and then applies an unintended X gate. Because the noise applies after the source, you can configure noise that has the same effect as the source. For example, `noise.x.x` applies an unintended bit flip after the intended bit flip.

> **Note**
>
> The neutral atom device simulation APIs support noise from a limited number of sources. For more information, see How to build noise models for neutral atom device simulations in the QDK.

Build a noise model

To build a noise model for a simulation and view the effects of that noise on the results of your quantum program, follow these steps.

1. In VS Code, open the **View** menu and choose **Command Palette**.
2. Enter **Create: New Jupyter Notebook**. An empty Jupyter Notebook file opens in a new tab.
3. In the first cell of the notebook, import the required Python objects.

```python
from qdk import init, TargetProfile
from qdk.openqasm import compile
from qdk.simulation import NoiseConfig, run_qir
from qdk.widgets import Histogram
```

4. In a new cell, set the device QIR target profile and compile your OpenQASM circuit into QIR.

Python

```
init(target_profile=TargetProfile.Base)

qasm_src = """
include "stdgates.inc";
qubit[2] qs;
bit[2] r;

h qs[0];
t qs[0];
cx qs[0], qs[1];
r = measure qs;
"""

qir = compile(qasm_src)
```

5. Create a `NoiseConfig` object and build your noise model.

Python

```
noise = NoiseConfig()

noise.t.z = 0.01
noise.cx.zi = 0.02
```

This code produces this noise model, where the noise rate is the probability that the source causes the corresponding type of noise.

Expand table

Noise source	Noise type	Noise rate
T gate	Phase flip	1%
CX gate	Phase flip on control qubit	2%

6. Run the simulator with the noise model and view a histogram of measurement results. This code simulates 1,000 shots on the stabilizer simulator. The T gate causes stabilizer branching, and the configured Z fault is sampled after the T gate.

Python

```
results = run_qir(qir, shots=1000, noise=noise, type="stabilizer")
Histogram(results, labels="kets")
```

7. To compare the noisy results with a noiseless simulation, run the simulation again with no noise model.

```
Python
results = run_qir(qir, shots=1000, type="stabilizer")
Histogram(results, labels="kets")
```

Set multiple types of noise on the same source

You can model different types of noise on the same source, with different probabilities for each type of noise. For example, this code sets a 1% chance that a bit flip occurs and a 3% chance that a phase flip occurs after a Hadamard gate.

```
Python
noise.h.x = 0.01
noise.h.z = 0.03
```

When you configure multiple noise types for the same operation, only one noise type can apply to each operation in your program. For example, each Hadamard gate can have either X noise or Z noise, but not both.

Noise model functions

Instead of noise model parameters, you can use these noise functions to build your noise model.

Set Pauli noise

To include Pauli noise in your model, call the `set_pauli_noise` function on a gate or movement operation.

For single-qubit operations, pass a one-character Pauli string and a noise rate. For example, this code sets a 1% chance that a bit flip occurs during qubit movement.

```
Python
# Equivalent to: noise.mov.x = 0.01
noise.mov.set_pauli_noise('X', 0.01)
```

For two-qubit operations, pass a two-character Pauli string and a noise rate. The first character of the Pauli string corresponds to noise on the control qubit and the second character corresponds

to noise on the target qubit. For example, this code sets correlated phase flips after 1% of CX operations.

Python

```
# Equivalent to: noise.cx.zz = 0.01
noise.cx.set_pauli_noise('ZZ', 0.01)
```

Set depolarizing noise

The `set_depolarizing` function sets equal rates for all three types of Pauli noise. For example, this code sets a 3% chance that Pauli noise occurs after an H operation, distributed evenly as a 1% chance for each of the three Pauli noise types.

Python

```
# Equivalent to:
#     noise.h.x = 0.01
#     noise.h.y = 0.01
#     noise.h.z = 0.01
noise.h.set_depolarizing(0.03)
```

Set bit flip noise

To set the noise rate for bit flips, use the `set_bitflip` function on a gate or movement operation. For example, this code sets a 1% chance that a bit flip occurs after an R_Z operation.

Python

```
# Equivalent to: noise.rz.x = 0.01
noise.rz.set_bitflip(0.01)
```

Set phase flip noise

To set the noise rate for phase flips in an operation, use the `set_phaseflip` function on a gate or movement operation. For example, this code sets a 1% chance that a phase flip occurs after an R_Y operation.

Python

```
# Equivalent to: noise.ry.z = 0.01
noise.ry.set_phaseflip(0.01)
```

Related content

- Build noise models for multi-qubit gates
- How to build noise models for neutral atom device simulations in the QDK

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/12/2026

Build noise models for multi-qubit gates

The Microsoft Quantum Development Kit (QDK) Python package supports flexible noise models for multi-qubit gates in quantum program simulations. You can model correlated errors and other noise effects across multiple qubits to more accurately represent the behavior of quantum hardware.

For more information on noise models in the QDK, see How to build noise models for quantum simulations in the QDK.

Prerequisites

To build noise models in the QDK, install these tools.

- Visual Studio Code (VS Code) with QDK extension and Jupyter extension installed.
- The latest version of the `qdk` Python package with the `jupyter` extra.

Bash

```
pip install --upgrade "qdk[jupyter]"
```

Correlated noise

Multi-qubit gates can produce correlated noise, where the same noise pattern applies to all qubits that the gate operates on. To set correlated noise on multi-qubit gates, specify a noise parameter for each qubit.

For example, this Python code sets correlated bit flips on CX gates with a 2% probability. When noise occurs on a CX gate, an X gate applies to both the control qubit and the target qubit. The noise is correlated because the noise always applies to both qubits.

Python

```
from qdk.simulation import NoiseConfig

noise = NoiseConfig()

noise.cx.xx = 0.02
```

To make the noise uncorrelated, set multiple noise types with the identity parameter.

Python

```
noise.cx.xi = 0.02 # Bit flip on control qubit, do nothing to target qubit
noise.cx.ix = 0.02 # Do nothing to control qubit, bit flip on target qubit
```

In this model, each noise entry has a 2% probability. Because the simulator samples at most one configured noise entry for each gate operation, this model can't apply X noise to both qubits during the same CX operation.

To model the possibility of noise on both qubits, configure another noise setting that applies noise to both qubits.

Python

```
noise.cx.xi = 0.02 # Bit flip on control qubit, do nothing to target qubit
noise.cx.ix = 0.02 # Do nothing to control qubit, bit flip on target qubit
noise.cx.xx = 0.02 # Bit flip on both qubits
```

Noise on non-Clifford multi-qubit rotations

The stabilizer simulator supports arbitrary-angle R_{XX}, R_{YY}, and R_{ZZ} rotations through stabilizer branching. Noise configured on `rxx`, `ryy`, or `rzz` is applied after the ideal rotation. This behavior doesn't change if the rotation creates additional stabilizer branches.

Each non-Clifford rotation can increase the number of branches, so memory usage and shot time can grow exponentially. Use the stabilizer simulator for circuits with a limited number of non-Clifford rotations, and use the CPU or GPU full-state simulator when non-Clifford operations make up a substantial part of the circuit.

Qubit loss policies

Qubit loss is a type of noise where qubits are lost from the device. Loss policies define the behavior of a multi-qubit gate when one or more of the qubits is missing when the operation begins. The QDK supports these loss policies.

Expand table

Loss policy	Gates the policy can apply to	Effect on remaining qubits
`SKIP`	All gates	The gate has no effect on the remaining qubits
`PROPAGATE`	All gates	The other qubits are also lost

Loss policy	Gates the policy can apply to	Effect on remaining qubits
`DEGRADE`	`rxx`, `ryy`, and `rzz`	Apply as single-qubit gate to remaining qubit
`RESIDUAL_S_DAGGER`	All gates	Apply `s_adj` gate to remaining qubits
`APPLY_ANYWAY`	`swap`	Remaining qubit still swaps states with lost qubit

For loss policies to affect your simulation, include qubit loss in your noise model. The simulation tracks lost qubits. Loss policies apply when a gate operates on at least one lost qubit.

To include loss policies in your noise model, use `LossPolicy` from the `qdk.simulation` module. This code shows examples to set each policy.

Python

```
from qdk.simulation import NoiseConfig, LossPolicy

noise = NoiseConfig()

noise.x.l = 0.01 # 1% chance of qubit loss after X gates

noise.cz.on_loss   = LossPolicy.SKIP               # Don't apply CZ to remaining qubit
noise.cx.on_loss   = LossPolicy.PROPAGATE          # Both qubits are lost from CX gate
noise.rxx.on_loss  = LossPolicy.DEGRADE            # Apply Rx gate to remaining qubit
noise.ryy.on_loss  = LossPolicy.RESIDUAL_S_DAGGER  # Apply S-adjoint to remaining
qubit
noise.swap.on_loss = LossPolicy.APPLY_ANYWAY       # Perform swap anyway
```

For all loss policies, configured noise can still affect the remaining qubits. For example, this code can still apply bit flip noise on CZ gates even when one of the qubits in a CZ gate is missing.

Python

```
noise.z.l = 0.01 # Introduce qubit loss

noise.cz.xx = 0.02 # Set correlated bit flip noise on CZ gates
noise.cz.on_loss = LossPolicy.SKIP
```

Custom noise intrinsics

To build more complex noise models, the QDK has custom noise intrinsics for Q# and OpenQASM programs. Noise intrinsics behave like custom gates that you insert into your program to model correlated noise. You can use custom intrinsics to model noise on the gates that `NoiseConfig` supports, or on custom gates.

These examples show how to build a custom noise intrinsic that models crosstalk between three qubits. The noise intrinsic applies correlated bit flips to two of the qubits after a $CNOT$ gate is applied.

Add noise intrinsics to a Q# program

In Q# programs, use `@NoiseIntrinsic()` to declare a noise intrinsic. Then, use the `intrinsic` method from `NoiseConfig` to configure the noise intrinsic.

To configure and use the example noise intrinsic, follow these steps in a Jupyter notebook.

1. Import the required objects and set the QIR target profile.

   Python

   ```
   from qdk import init, TargetProfile
   from qdk import qsharp
   from qdk.simulation import run_qir, NoiseConfig
   from qdk.widgets import Histogram

   init(target_profile=TargetProfile.Adaptive_RIF)
   ```

2. Write a Q# program called `GHZ` that calls a noise intrinsic called `Crosstalk3Q` after each $CNOT$ gate.

   Q#

   ```
   %%qsharp

   // A noise intrinsic that represents crosstalk on 3 qubits.
   // In the ideal circuit this is a no-op; the simulator injects
   // Pauli errors according to the NoiseConfig.
   @NoiseIntrinsic()
   operation Crosstalk3Q(q0: Qubit, q1: Qubit, q2: Qubit) : Unit {
       body intrinsic;
   }

   // Prepare a GHZ state on 3 qubits, with crosstalk after each CNOT.
   operation GHZ() : Result[] {
       use qs = Qubit[3];
       H(qs[0]);
       CNOT(qs[0], qs[1]);
       Crosstalk3Q(qs[0], qs[1], qs[2]);  // crosstalk hits all 3 qubits
       CNOT(qs[1], qs[2]);
       Crosstalk3Q(qs[0], qs[1], qs[2]);  // crosstalk again
       MResetEachZ(qs)
   }
   ```

3. Configure the noise table for the intrinsic. Set the number of qubits, the types of noise, and the probability for each noise type.

Python

```
noise = NoiseConfig()
table = noise.intrinsic("Crosstalk3Q", num_qubits=3)
table.ixx = 0.10  # 10% XX on qubits 1-2
table.xxi = 0.05  #  5% XX on qubits 0-1
```

4. Compile the program to QIR.

Python

```
qir = qsharp.compile("GHZ()")
```

5. Run the simulation on the stabilizer simulator and plot a histogram of the results.

Python

```
result = run_qir(qir, shots=1000, noise=noise, type="stabilizer")
Histogram(result)
```

6. To compare the result to a simulation without noise, run the simulation again with no noise model.

Python

```
result = run_qir(qir, shots=1000, type="stabilizer")
Histogram(result)
```

Add noise intrinsics to an OpenQASM program

In OpenQASM programs, use `@qdk.qir.noise_intrinsic` to create a noise intrinsic as a custom gate definition. Then, use the `intrinsic` method from `NoiseConfig` to configure the noise intrinsic.

To write an OpenQASM program with a noise intrinsic called `crosstalk_3q` and compile the program into QIR, run this code in a Jupyter notebook.

Python

```
from qdk.openqasm import compile, OutputSemantics
from qdk import TargetProfile
```

```
from qdk.simulation import run_qir, NoiseConfig
from qdk.widgets import Histogram

qasm_source = """
OPENQASM 3.0;
include "stdgates.inc";

// A noise intrinsic that represents crosstalk on 3 qubits.
// In the ideal circuit this is a no-op; the simulator injects
// Pauli errors according to the NoiseConfig.
@qdk.qir.noise_intrinsic
gate crosstalk_3q q0, q1, q2 {}

qubit[3] qs;

// Prepare a GHZ state on 3 qubits, with crosstalk after each CNOT.
h qs[0];
cx qs[0], qs[1];
crosstalk_3q qs[0], qs[1], qs[2];  // crosstalk hits all 3 qubits
cx qs[1], qs[2];
crosstalk_3q qs[0], qs[1], qs[2];  // crosstalk again

bit[3] res = measure qs;
"""

qir_qasm = compile(
    qasm_source,
    output_semantics=OutputSemantics.OpenQasm,
    target_profile=TargetProfile.Base,
)
```

To configure the noise intrinsic and run the simulation, run this code.

Python

```
noise = NoiseConfig()
table = noise.intrinsic("crosstalk_3q", num_qubits=3)
table.ixx = 0.10  # 10% XX on qubits 1-2
table.xxi = 0.05  #  5% XX on qubits 0-1

result = run_qir(
    qir_qasm,
    shots=1000,
    noise=noise,
    type="stabilizer",
)
Histogram(result)
```

ⓘ**Note:** The author created this article with assistance from AI. Learn more

Last updated on 10/10/2026

Explore circuit diagrams in the QDK

Quantum algorithms are a mixture of quantum programming and classical programming. The quantum part includes qubits, quantum gates, and probabilistic measurements, while the classical part includes control flow constructs such as conditional logic and loops. Quantum circuit diagrams need to incorporate elements from both quantum and classical programming.

The Microsoft Quantum Development Kit (QDK) creates interactive circuit diagrams for Q# and OpenQASM programs that show how classical control groups affect the structure of your circuit.

For information on how to create circuit diagrams with the QDK in VS Code, see How to visualize quantum circuit diagrams with the QDK

Classical control groups in quantum circuit diagrams

In some quantum algorithms, the circuit changes depending on the result of a mid-circuit measurement. For example, the circuit applies a gate to a specific qubit only if the measurement result for another qubit is 1. Or, the circuit iterates over a loop until the measurement result for a specific qubit is 0.

For example, the following two-qubit Q# program puts a qubit into a superposition state, measures the qubit, and then applies a different gate to the other qubit depending on the measurement result of the first qubit.

Q#

```
import Std.Measurement.*;

operation Main() : Result[] {
    use q0 = Qubit();
    use q1 = Qubit();

    H(q0);

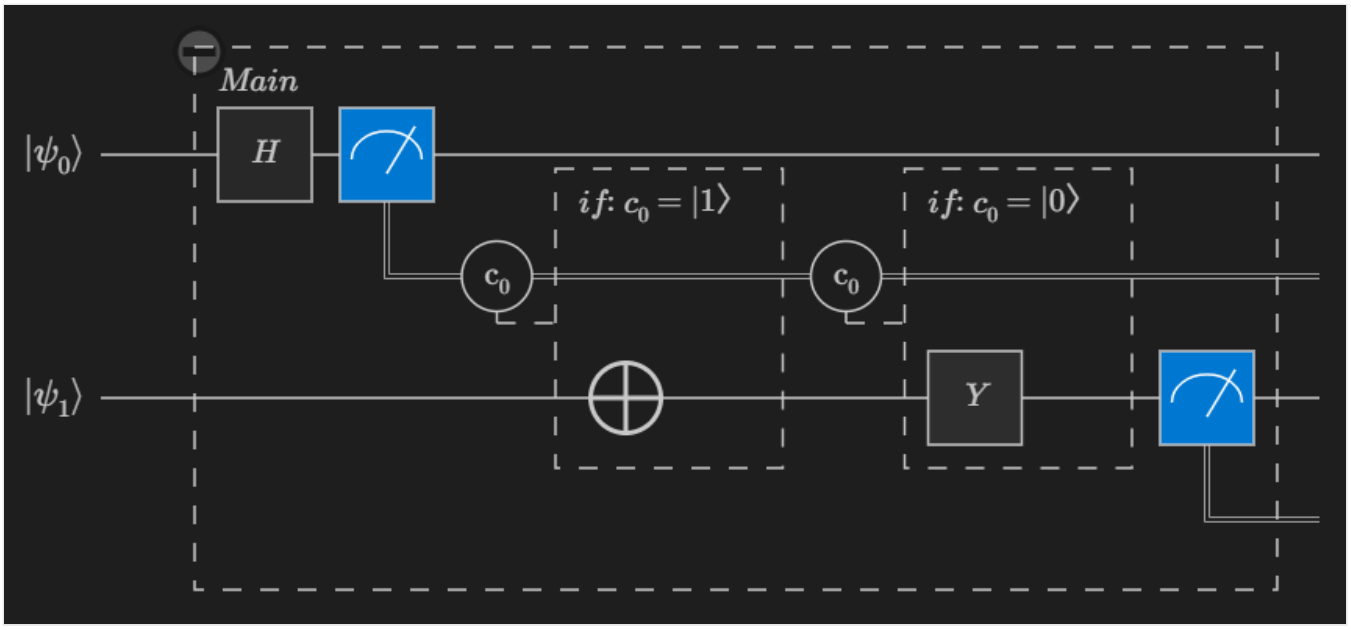
    let r = M(q0);

    if r == One {
        X(q1);
    } else {
        Y(q1);
    }

    let r1 = M(q1);
```

```
    [r, r1]
}
```

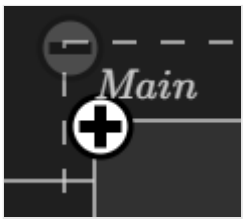
The circuit diagram for this program looks like this:



> **Tip**
>
> Select an element in the circuit diagram to highlight the code that creates the circuit element.

Circuit diagram block types

QDK circuit diagrams have two types of blocks, quantum operation blocks and classical control group blocks. Some blocks are collapsible. To collapse or expand a block, select the **+/-** icon at the top left corner of the block.



Quantum operation blocks

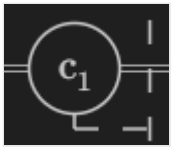
Each object in your program that applies quantum gates or measurements to qubits has its own block in the circuit diagram. For Q# programs, these objects include operations, functors, and lambdas. Each operation block shows the name of the Q# or OpenQASM object that corresponds to that block.

Classical control flow blocks

Each conditional statement and loop in your program has its own block in the circuit diagram.

Conditional blocks

Conditional blocks are preceded by a condition icon C_n, where each conditional statement in the program is identified by a unique subscript number.



Each conditional block is labeled with the condition that, if true, causes that part of the circuit to run during the program.

Loop blocks

Loop blocks are labeled with **loop: 1.. *N***, where ***N*** is the number of iterations of the loop. To view the circuit for each iteration of the loop, select the expansion icon in the loop block.

Related content

- How to visualize quantum circuit diagrams with the QDK
- Quantum circuit diagram conventions

Last updated on 04/01/2026

Create and visualize circuits with the circuit editor

Summarize this article for me

The circuit editor is a feature in the Quantum Development Kit (QDK) that provides a graphical interface where you can create, edit, and visualize quantum circuit diagrams inside your Q# projects. You can use the circuits that you build directly in your Q# programs as callable operations.

How to access the circuit editor

To get started with the circuit editor, follow these steps:

1. Create a new Q# project in VS Code, or open an existing project.
2. Create a new circuit file with the extension `.qsc`.
3. Open the circuit file in VS Code.

The circuit editor is the default view for `.qsc` files.

circuit editor features

Here's what you can do with the circuit editor:

Expand table

Feature	How to Use
Expand the State visualizer panel	Click **State Visualization** to expand the State Visualizer panel to view the resulting quantum state from running the circuit. The panel shows the probability density and phase for each basis state. You can use the Toolbox to edit the circuit and view panel updates live.
Insert a circuit element	Click-and-drag the element from the toolbox into the circuit diagram.
Remove a circuit element	Click-and-drag the element out of the circuit diagram. Or, right-click the element and choose **Delete** from the context menu.
Move a circuit element	Click-and-drag the element to a new location in the circuit diagram.
Copy a circuit element	Hold down the **Ctrl** key while you click-and-drag the element to a new location in the circuit diagram.

Feature	How to Use
Add a qubit	Place an element from the toolbox on a new wire in the circuit diagram. Or, move an existing element to a new wire in the diagram.
Remove a qubit	Click-and-drag the qubit icon out of the circuit diagram. The bottom-most qubit is automatically deleted when you remove all circuit elements from that qubit.
Rearrange the qubit order	Click-and-drag the qubit icons in the circuit diagram.
Add a control to a gate	Right-click the gate icon, choose **Add Control** from the context menu, and then choose the control qubit wire.
Remove a control from a gate	Right-click the gate icon, choose **Remove Control** from the context menu, and then select the icon for the control that you want to remove. You can also right-click the control icon directly and choose **Remove Control**, or drag the control icon off the diagram.
Convert a gate to its adjoint	Right-click the gate icon and choose **Toggle Adjoint** from the context menu.
Define an argument for a gate	When you place a gate that requires an argument, a prompt box appears. Enter a number or expression in the prompt box. To update the argument, right-click the gate and choose **Edit Argument**. Or, left-click the argument text on the gate icon.

How to use circuit editor circuits in your Q# projects

Circuit files with the `.qsc` extension define operations that you can reference from Q# code within the same Q# project. The circuits from the `.qsc` files appear like any other Q# operation and are supported by the Q# language service with features like completions, signature help, and go-to definition.

The circuit file defines a namespace that comes from the file name, similar to how Q# files work. The circuit in your `.qsc` file defines a Q# operation that has the same name as your file under that same namespace. For example, to reference a circuit from the file `Foo.qsc` in your Q# code, use the following import statement: `import Foo.Foo;`.

The following Q# program imports a circuit from the file `JointMeasurement.qsc`, and then calls the `JointMeasurement()` operation to apply the circuit to a system of qubits.

Q#

```
    import JointMeasurement.JointMeasurement;
```

```
    operation Main() : Result[] {
        use qs = Qubit[4];
        ApplyToEach(H, qs[0..2]);
        let results = JointMeasurement(qs);
        ResetAll(qs);
        results
}
```

Related content

- Visualize quantum circuit diagrams with Q#
- Quantum circuit diagram conventions

Last updated on 03/12/2026

How to use the Bloch sphere visualizer in the QDK

The Bloch sphere is a geometric representation of a single-qubit state. The north and south poles of the unit sphere represent the computational basis states $|0\rangle$ and $|1\rangle$. Other points on the surface of the sphere represent different superposition states of the qubit. Use the Bloch sphere to build intuition about how quantum gates affect the state of a qubit.

The Microsoft Quantum Development Kit (QDK) provides an interactive Bloch sphere visualizer in the QDK extension for Visual Studio Code (VS Code) and as a widget for Jupyter Notebook in the QDK Python package. Use the visualizer to build a gate sequence and display how the qubit state changes during the sequence.

For more information about the Bloch sphere and single-qubit states, see The qubit in quantum computing.

Prerequisites

To use the Bloch sphere visualizer in VS Code, install the following:

- The latest version of VS Code, or open VS Code for the Web.
- The latest version of the QDK extension in VS Code.

To use the Bloch sphere visualizer in Python with Jupyter Notebook, install the following:

- Python 3.10 or later and pip.
- The Python extension and Jupyter extension in VS Code.
- The `qdk` Python package with the `jupyter` extra.

```bash
pip install --upgrade "qdk[jupyter]"
```

For complete QDK setup instructions, see Set up the Microsoft Quantum Development Kit.

Open the Bloch sphere visualizer

Use the Bloch sphere visualizer in VS Code or Jupyter Notebook.

Open the visualizer in VS Code

1. In VS Code, select **View** > **Command Palette**.
2. Enter **QDK: Bloch sphere** and select the command.

The visualizer opens in a new tab and starts in the $|0\rangle$ state.

Open the visualizer in Jupyter Notebook

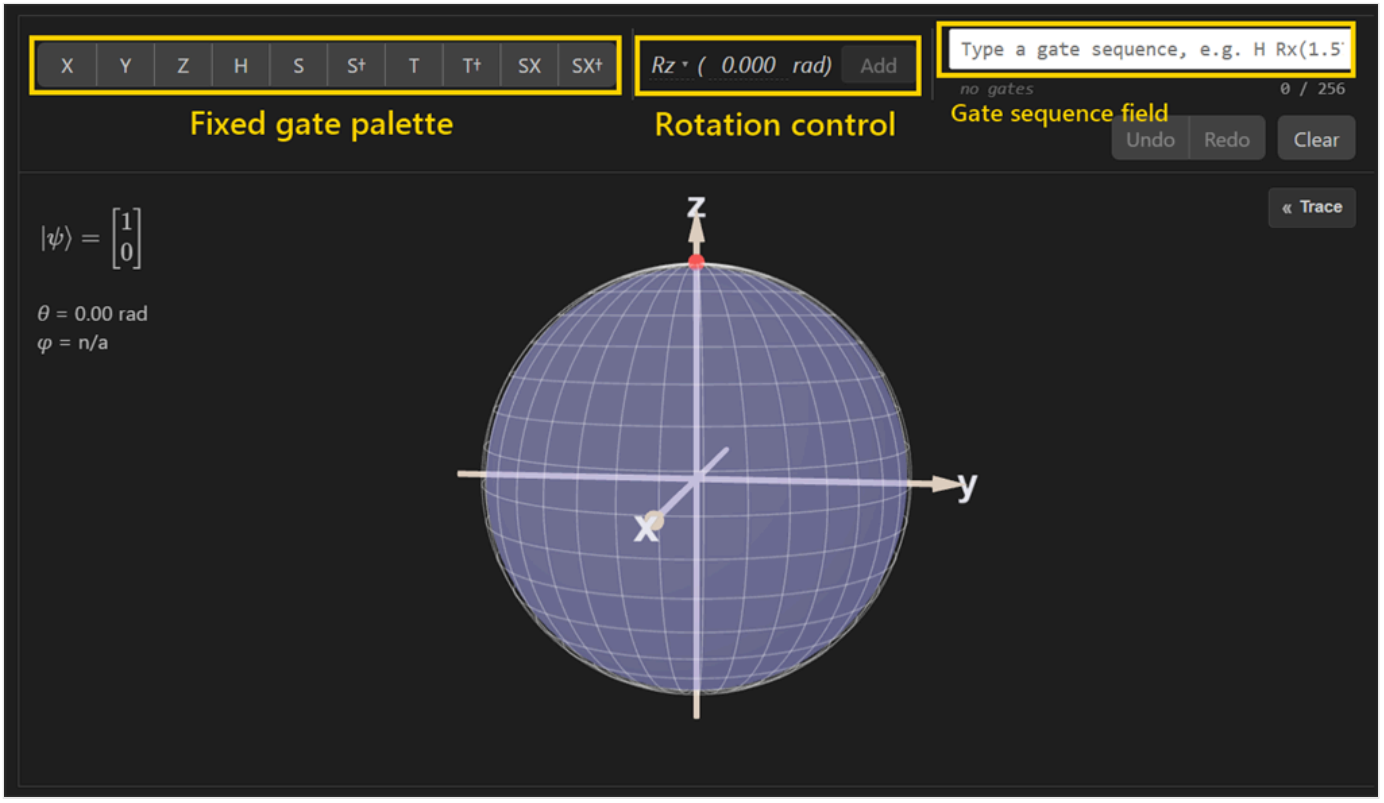
1. In VS Code, open the **View** menu and select **Command Palette**.
2. Enter **Create: New Jupyter Notebook** and select the command.
3. In a notebook cell, import `BlochSphere` and create the visualizer widget.

Python

```python
from qdk.widgets import BlochSphere

BlochSphere()
```

The widget opens in the cell output and starts in the $|0\rangle$ state.



To initialize the visualizer with a gate program, pass the program when you create the widget. Separate each gate with a space. For example, the following code initializes the visualizer with three gates.

Python

```
from IPython.display import display
from qdk.widgets import BlochSphere

display(BlochSphere("H T H"))
```

Build a gate sequence

Use the toolbar or the gate sequence field to build a gate sequence. The sphere and state information update as you add gates.

Add fixed gates

To append a gate to the sequence, select a gate from the fixed gate palette. The palette includes these gates:

- Pauli gates: `X`, `Y`, and `Z`
- Hadamard gate: `H`
- Phase gates and their adjoints: `S`, `S†`, `T`, and `T†`
- Square root of X gates: `SX` and `SX†`

Add rotation gates

To add a rotation gate, follow these steps:

1. In the rotation control, select `Rx`, `Ry`, or `Rz`.
2. Enter the rotation angle in radians or move the angle control to set the value.
3. Select **Add** to append the rotation gate to the gate sequence.

A popup in the rotation control shows the Clifford+T fixed gate decomposition that approximates the rotation. To append the synthesized fixed gate sequence instead of the native rotation, select **Add decomposition**. A gate sequence can include native rotations, synthesized Clifford+T sequences, or both.

Enter a gate sequence manually

Enter or edit a complete sequence in the gate sequence field.

The field supports the fixed gates `X`, `Y`, `Z`, `H`, `S`, `T`, and `SX`. To enter an adjoint gate, use an apostrophe: `S'`, `T'`, and `SX'`. Separate gates with a space.

For rotations, use `Rx(angle)`, `Ry(angle)`, or `Rz(angle)`, where the angle is in radians. For example:

text

```
H Rx(1.5708) S'
```

Explore the state and gate counts

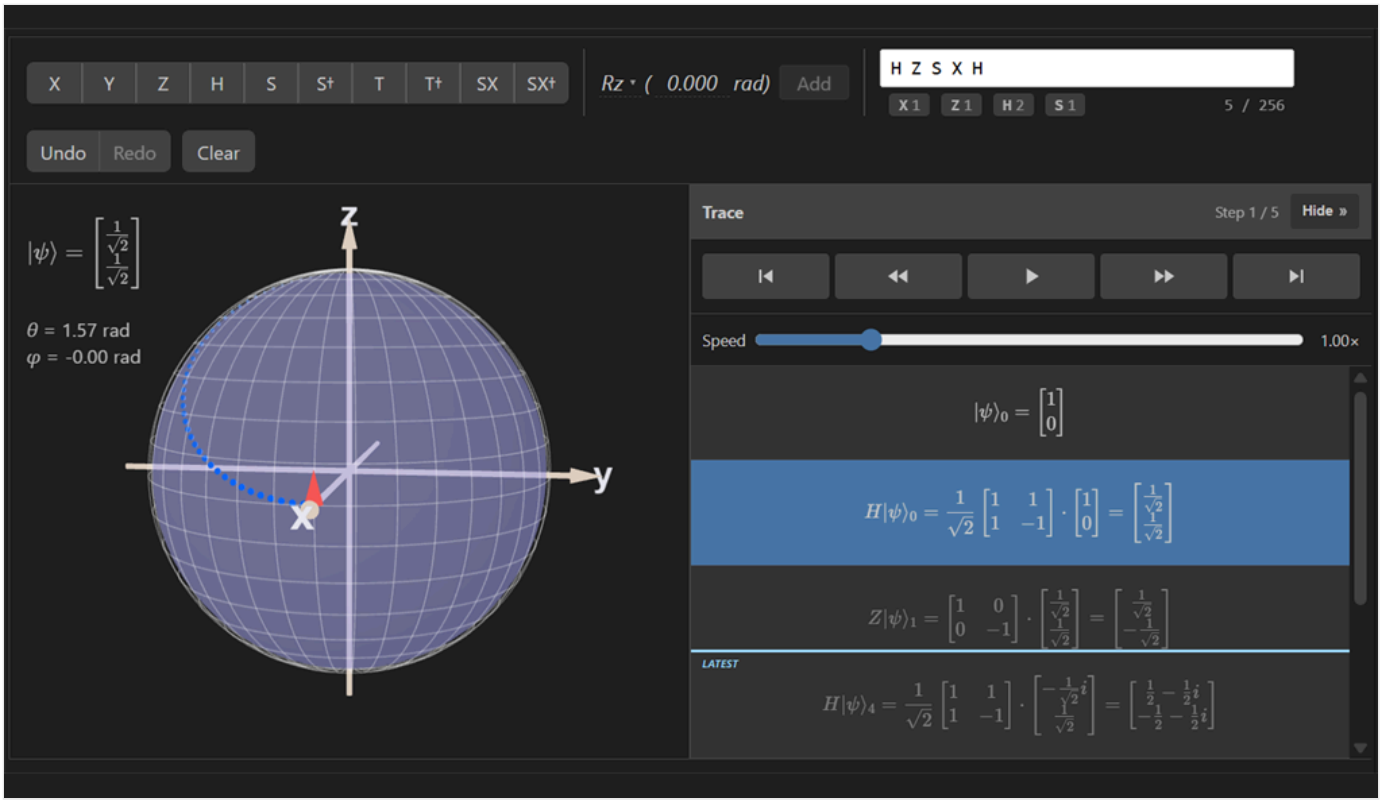
The interactive sphere includes the following information and controls.

Expand table

Element	Description
Bloch sphere	Move the camera around the sphere to view the state from different angles. The sphere shows the x, y, and z axes.
State marker and trail	The marker shows the current single-qubit state. The trail shows how the gates in the program move the state across the sphere.
State values	The visualizer shows the current state in ket notation and its theta (`θ`) and phi (`ϕ`) angles. At either pole, the visualizer shows `n/a` for `ϕ`. When you select an earlier trace row, these values and the sphere show that intermediate state.
Gate counts	The toolbar shows the count for each gate type, the T-count, and the total gate count out of the 256-gate limit.
Edit controls	Select **Undo** or **Redo** to move backward or forward through edits. Select **Clear** to remove all gates and return to the initial state.

Inspect and play the state trace

To open the trace window, select the **Trace** button. To collapse the window, select the **Hide** button.



The trace lists the initial state and one row for each gate in the sequence. Each row shows the gate matrix and the resulting qubit state. Select a row to inspect that intermediate state on the sphere. If you select an earlier row and then add a gate, the visualizer discards the later steps and appends the new gate at that point.

Use the playback controls to animate or step through the gate program.

Expand table

Control	Action
Start	Select the initial state.
Back	Move back one step.
Play, **Pause**, or **Replay**	Play the sequence, pause on the current step, or replay after the animation finishes.
Forward	Move forward one step.
End	Select the final state.

When you expand the trace, move the playback speed control to choose an animation speed.

Related content

- The qubit in quantum computing
- Visualize quantum circuit diagrams with the QDK

Last updated on 09/11/2026

Simulate quantum programs on neutral atom device hardware

The Microsoft Quantum Development Kit (QDK) provides a set of simulation tools that you can use to evaluate and iterate on your quantum programs before you run them on real quantum hardware. The neutral atom device simulation APIs model the types of noise and qubit operations that occur when programs run on neutral atom quantum computers, such as qubit loss and qubit movement. If you plan to run your quantum programs on neutral atom hardware, use the neutral atom device simulation APIs to test and refine your code.

How neutral atom quantum computers work

Neutral atom devices are one of many current quantum computer hardware technologies. Other technologies include superconducting qubits, trapped-ion qubits, and topological qubits. Each technology has its own strengths and drawbacks. Neutral atom quantum computers offer good scalability, flexible qubit connectivity, and long coherence times, but face challenges with gate fidelity and laser control complexity.

The exact qubit technology in a neutral atom device depends on the specific architecture. But in general, each qubit is a single electrically neutral atom. The qubit 0 and 1 states correspond to different energy states of the atoms. The atoms are arranged in 2D or 3D arrays on the device. Lasers trap the atoms in place and move the atoms between different zones on the device for storage, for measurement, and for interactions that perform quantum gates.

Neutral atom device simulation tools in the QDK

Neutral atom device simulation in the QDK models these properties of neutral atom quantum hardware:

- Devices contain only S_X, R_Z, and CZ gates.
- Qubits physically move between storage, interaction, and measurement zones on the device.

The format of your program determines which neutral atom device simulation API to use:

Expand table

API	QDK module	Input format	Supported simulators
`NeutralAtomDevice`	`qdk.simulation`	QIR	Stabilizer, GPU, and CPU
`NeutralAtomBackend`	`qdk.qiskit`	Qiskit	Stabilizer, GPU, and CPU

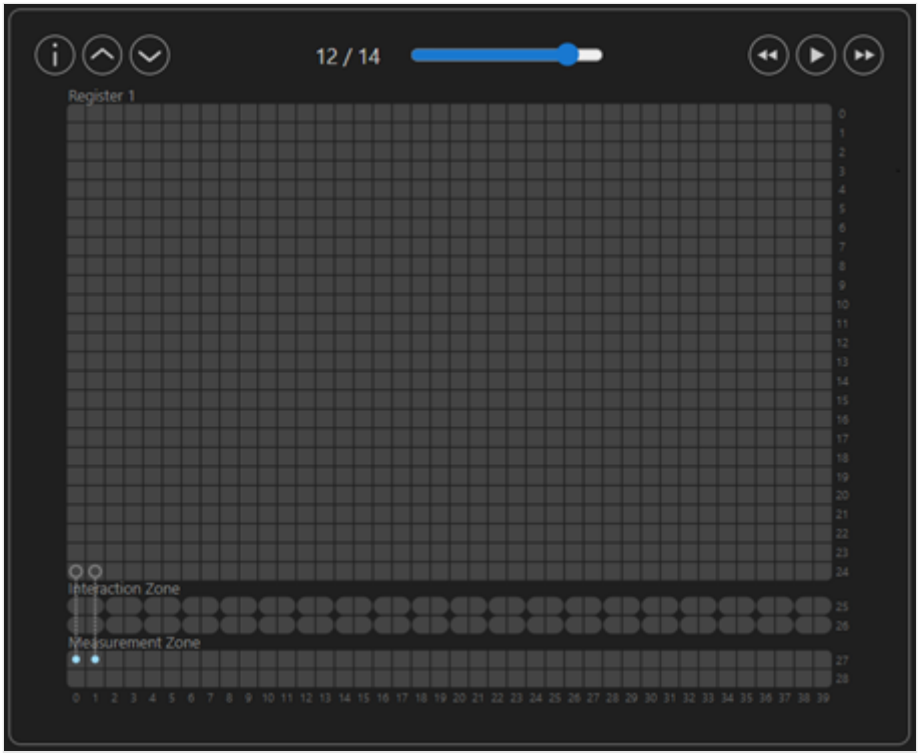
Both APIs compile your program into QIR that has qubit movement instructions and contains only the gates that neutral atom devices support.

Both `"stabilizer"` and `"clifford"` select the stabilizer backend. When you select this backend for a neutral atom device simulation, the compiled R_Z rotations must have Clifford-compatible angles. Use the CPU or GPU simulator for neutral atom circuits that contain other R_Z angles.

For more information on quantum simulation in the QDK, see Overview of quantum simulators in the QDK.

Neutral atom device visualizer

The `NeutralAtomDevice` API includes a neutral atom device visualizer for Jupyter Notebook through the `show_trace` method. The visualizer creates an interactive diagram that shows how qubits move through a basic neutral atom device as your program runs. The visualizer doesn't include the effects of qubit loss or other noise.



For more information on the neutral atom device visualizer, see How to use the neutral atom device visualizer.

Noise models for neutral atom device simulations

The quantum simulators in the QDK Python package use the `NoiseConfig` class from the `qdk.simulation` module to add noise models to simulations. When you perform simulations with the `NeutralAtomDevice` or `NeutralAtomBackend` APIs, your noise models only include noise from neutral atom device gates, qubit movement, and measurement.

For more information on neutral atom device noise models in the QDK, see How to build noise models for neutral atom device simulations in the QDK.

Get started with neutral atom device simulation

To get started with quantum simulation in the QDK Python package, see Run quantum simulations with the QDK Python package.

To learn how to use the neutral atom device visualizer, see How to use the neutral atom device visualizer.

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/10/2026

How to use the neutral atom device visualizer

The Microsoft Quantum Development Kit (QDK) offers several quantum simulators and a visualizer for neutral atom quantum computers. The neutral atom device visualizer produces an interactive diagram that lets you track how qubits move and get processed when your program runs on a basic neutral atom device. This article explains how to access and use the visualizer in Jupyter Notebook.

For instructions on how to install the simulators and visualizer, see How to install and run the QDK quantum simulators.

How to create a neutral atom qubit diagram

To create a qubit diagram with the neutral atom visualizer, follow these steps:

1. In VS Code, open the **View** menu and choose **Command Palette**.
2. Enter and select **Create: New Jupyter Notebook**. A new tab opens with an empty Jupyter Notebook file.
3. Import the necessary packages and objects. In the first cell, enter and run the following code:

   Python

   ```
   from qdk.openqasm import compile
   from qdk.simulation import NeutralAtomDevice
   ```

4. Write your OpenQASM circuit and compile the circuit into QIR. For example, the following circuit entangles two qubits:

   Python

   ```
   qasm_src = """
   include "stdgates.inc";
   qubit[2] qs;
   bit[2] r;

   h qs[0];
   cx qs[0], qs[1];
   ```

```
r = measure qs;
"""

qir = compile(qasm_src)
```

5. Create a simulator object using `NeutralAtomDevice`.

   Python

   ```
   simulator = NeutralAtomDevice()
   ```

6. To access the visualizer, call the `show_trace` method and pass the QIR for your program.

   Python

   ```
   simulator.show_trace(qir)
   ```

   The visualizer renders in the output cell.

> **Note**
>
> The visualizer doesn't include qubit loss or other noise.

How to interact with the visualizer

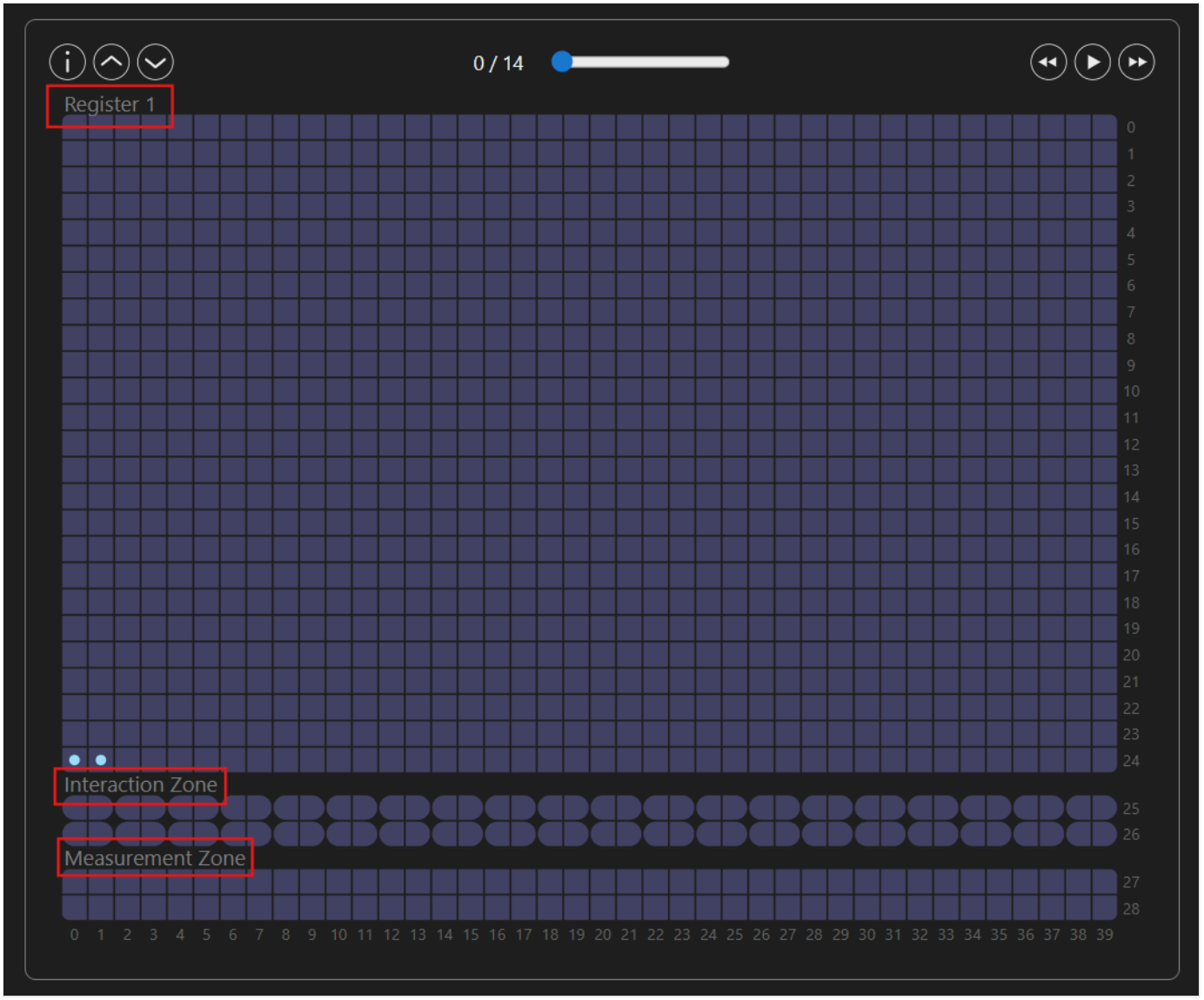
The visualizer has interactive elements that let you explore a simulation of how qubits behave as your program runs on a basic neutral atom quantum computer. The device diagram is a 2D grid with labeled rows and columns. Each dot in a grid position represents one neutral atom qubit on the device.

The diagram contains three zones:

Expand table

Zone	Description
Storage zone	This zone is labeled **Register 1**. The qubits start in the storage zone and stay there until they're ready for processing or measurement. Qubits always move back to the storage zone after operations and measurements.
Interaction zone	This zone is where quantum gates are applied to the qubits for processing.

Zone	Description
Measurement zone	This zone is where the qubits are measured.

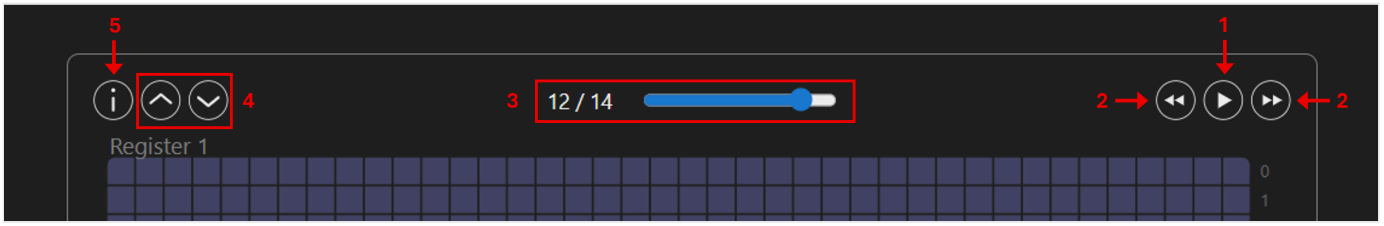


Interactive elements in the visualizer

To interact with the visualizer, use the elements at the top of the diagram. The diagram contains the following elements:

1. **Play button:** Select this button to play an animation of your program run. The animation goes through each step of the program until the program ends. During the animation, select this button again to pause the animation on the current step. When the animation ends, select this button again to start the animation from the beginning.
2. **Forward and backward buttons:** Select these buttons to go through the program one step at a time without playing the full animation.

3. **Progress slider:** This element shows the current step of the program. Move the slider to go through the program and choose a specific step. At each step, hover over a qubit to see where the qubit moved from in the previous step.
4. **Resize buttons:** Select the up arrow button to increase the size of the diagram, and select the down arrow button to decrease the size of the diagram.
5. **Information icon:** Hover over this icon to view a list of keyboard shortcuts that let you interact with the diagram. The keyboard shortcut **F** speeds up the animation and **S** slows down the animation when you select the **Play** button.



Last updated on 05/22/2026

Build noise models for neutral atom device simulations in the QDK

The Microsoft Quantum Development Kit (QDK) includes a set of neutral atom device simulation APIs that you can use to model how your program runs on a neutral atom quantum computer. These APIs model the types of noise that occur specifically on neutral atom hardware. The QDK Python package lets you build noise models for simulations with the `NoiseConfig` class.

For more information on noise models in QDK simulations, see How to build noise models for quantum simulations in the QDK.

For instructions on how to install and use the QDK simulators, see Run quantum simulations with the QDK Python package.

The neutral atom device simulation APIs

The QDK includes two neutral atom device simulation APIs, `NeutralAtomDevice` and `NeutralAtomBackend`. These APIs model the noise that occurs specifically on neutral atom quantum computers. For more information, see Simulate quantum programs on neutral atom device hardware.

Types of noise in neutral atom device simulators

In neutral atom devices, lasers physically move the qubits between different zones in the device. Noise can occur in these situations:

- Qubit movements between zones
- Quantum gate operations on qubits in the interaction zone
- Qubit measurements in the measurement zone

Neutral atom devices have a limited set of quantum gates. Neutral atom device simulations in the QDK support noise from these sources.

Expand table

Noise source	Noise model parameter	Source description
S_X quantum gate	`sx`	Single-qubit gate, half bit flip

Noise source	Noise model parameter	Source description
R_Z quantum gate	`rz`	Single-qubit gate, general phase rotation
CZ quantum gate	`cz`	Two-qubit gate, controlled-Z phase flip
Qubit measurement and reset	`mresetz`	Single-qubit measurement and reset to 0 state
Qubit movement	`mov`	Qubit movement between device zones

For these noise sources, you can configure all of the noise types that the QDK simulators support.

Your quantum program can contain any type of gate that the QIR target profile supports. The neutral atom simulation APIs compile the input QIR and decompose your program's gates into the set of gates that exist on neutral atom devices: S_X, R_Z, and CZ. When you use `NeutralAtomDevice` and `NeutralAtomBackend` for your simulations, don't configure noise on other gates because that noise doesn't affect the simulation. For example, if your program contains Hadamard gates and you configure noise for Hadamard gates, that noise doesn't show up in the simulation because the Hadamard gates are decomposed into neutral atom device gates.

The neutral atom simulation APIs convert all measurements in your program into measure-and-reset instructions. To include measurement noise, configure the noise on `mresetz`. Noise on `mz` isn't included in the simulation.

When you use the stabilizer backend, the compiled R_Z rotations must have Clifford-compatible angles. Use the CPU or GPU backend for neutral atom simulations that contain other R_Z angles.

Build a noise model for a neutral atom device simulation

To build a noise model for a neutral atom device simulation, follow these steps.

1. In Visual Studio Code, open the **View** menu and choose **Command Palette**.
2. Enter **Create: New Jupyter Notebook**. An empty Jupyter Notebook file opens in a new tab.
3. In the first cell of the notebook, import the required Python objects.

Python

```
from qdk import init, TargetProfile
from qdk.openqasm import compile
from qdk.simulation import NeutralAtomDevice, NoiseConfig
from qdk.widgets import Histogram
```

4. In a new cell, set the device QIR target profile and compile an OpenQASM program into QIR.

Python

```
init(target_profile=TargetProfile.Base)

qasm_src = """
include "stdgates.inc";
qubit[2] qs;
bit[2] r;

h qs[0];
cx qs[0], qs[1];
r = measure qs;
"""

qir = compile(qasm_src)
```

5. Create a `NoiseConfig` object and build your noise model. For example, run this code in a new cell.

Python

```
noise = NoiseConfig()

noise.sx.x = 0.01
noise.rz.z = 0.01
noise.cz.iy = 0.02
noise.mov.l = 0.005
```

This code produces this noise model, where the noise rate is the probability that the source causes the corresponding type of noise.

Expand table

Noise source	Noise type	Noise rate
S_X gate	Bit flip	1%
R_Z gate	Phase flip	1%
CZ gate	Bit flip and phase flip on target qubit	2%
Qubit movement	Qubit loss	0.5%

6. Run the simulator with the noise model and view a histogram of measurement results. For example, run this code in a new cell to run 1,000 shots on the stabilizer simulator.

Python

```
device = NeutralAtomDevice()

results = device.simulate(
    qir,
    shots=1000,
    noise=noise,
    type="stabilizer",
)
Histogram(results, labels="kets")
```

> **Note**
>
> Both `type="stabilizer"` and `type="clifford"` select the stabilizer simulator.

Note: The author created this article with assistance from AI. Learn more

---

Last updated on 10/10/2026

Welcome to QDK for chemistry

Get started here

QDK for chemistry (QDK/Chemistry) is an open-source Python package within the Microsoft Quantum Development Kit (QDK). It provides a set of tools and libraries to enable cutting-edge quantum chemistry workflows on quantum computers by bridging the gap between classical electronic structure calculations and quantum circuit execution.

Quantum applications pipelines depend on robust, efficient classical data preparation and post-processing. QDK/Chemistry addresses these needs through a modular architecture that separates data representations from computational methods. This modularity allows researchers to compose flexible workflows from interchangeable components. The toolkit provides native implementations of key quantum-classical algorithms and integrates seamlessly with widely used open-source quantum chemistry packages and quantum computing frameworks via a plugin system. By offering a unified interface to diverse methods and tools, QDK/Chemistry serves as a platform for innovation and a foundation for reproducible, scalable quantum chemistry experiments.

Why use QDK for chemistry?

Integration / Familiarity

QDK/Chemistry is seamlessly integrated with first- and third-party quantum chemistry tools and quantum programming languages, enabling flexible use of preferred tools and environments. AI-assisted coding and debugging can be carried out in VS Code, while development with Python can be performed in Jupyter Notebook.

Accessibility

QDK/Chemistry is accessible to quantum chemistry researchers of every level. For beginners, the toolkit offers tools and algorithms with robust default settings and parameters that you can use to get results for a large variety of molecular systems right out of the box. Advanced users can take advantage of the extensive capabilities and customizability of QDK/Chemistry tools to tackle more complex research questions.

Modularity

QDK/Chemistry simplifies the development process with a modular toolset. Each part of the quantum chemistry workflow functions independently, so the same tools can be reused across

different workflows and molecules.

Implementation

QDK/Chemistry offers a fast, intuitive, and seamless end-to-end solution for every stage of the quantum chemistry workflow. The toolset leverages state-of-the-art optimization techniques that maximizes performance, minimizes resource requirements, and scales to harness the power of next-generation quantum hardware.

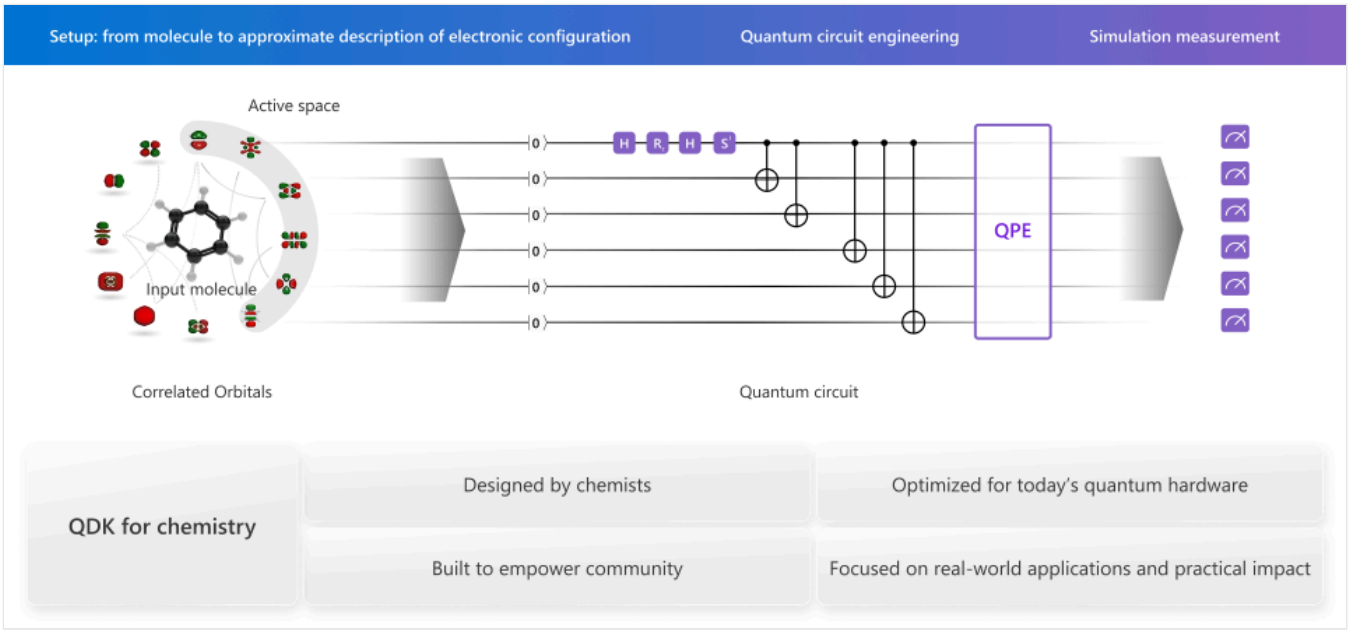
Documentation and samples

QDK/Chemistry comes with a suite of code samples to help get started with development. Samples include a variety of molecular systems, quantum programming languages, and tutorials that walk through end-to-end workflows from molecular structure to quantum algorithm execution. The documentation also includes a comprehensive API reference that explains how every tool and algorithm works, along with examples of how to use the tools.

QDK/Chemistry core features and technologies

- Robust SCF calculations and reliable automatic active space selection(LINK TO DEEP DIVE)
- Efficient quantum circuit preparation with GF2+X sparse isometry (LINK TO DEEP DIVE)
- Seamless integration with native QDK high-performance simulators and resource estimator
- VS code and Python extensions for QDK/Chemistry, powered by AI-assisted coding capabilities to help you write and understand quantum workloads

QDK: Mapping chemistry to a quantum computer



QDK/Chemistry resources

To get started with QDK/Chemistry, see How to install QDK for chemistry.

For an end-to-end example of a quantum chemistry state preparation circuit workflow, see the Quickstart guide in the QDK/Chemistry documentation on GitHub.

For detailed information on the available tools in QDK/Chemistry, see the Python API and C++ API references.

Last updated on 01/25/2026

Perform SCF calculations and active space selection to construct molecular orbitals

In quantum chemistry, electronic structure theory is a framework that describes the organization of electrons in molecular orbitals (MOs). Self-consistent field (SCF) methods are an electronic structure technique that you use to calculate the MOs of the electrons.

In quantum chemistry, the SCF method is known as the Hartree-Fock (HF) method, which represents the electronic wavefunction of the molecule as a single determinant that satisfies the Pauli exclusion principle. Each term in the determinant is a product of spin orbitals. Most other SCF methods are based on the HF method.

How SCF methods work

It's difficult to describe the exact electronic structure of a molecule because all of the electrons are correlated with each other. When a molecule has many electrons, it becomes computationally expensive to calculate the interactions between all the individual electrons. SCF methods treat many-body interactions as a single mean field interaction rather than individual interactions between particles. SCF calculations find a mean field that's due to all the electrons, and each individual electron interacts with this mean field.

An SCF calculation is an iterative procedure to find MO coefficients that are consistent with the mean field that the MOs generate. You make an initial guess at the MO coefficients, and then the SCF method builds an electron density and calculates the energy of the molecule from the initial guess. Then, the SCF method uses the electron density to build a new set of MO coefficients, which are used to calculate a new electron density and energy.

The process repeats until changes in the electron density and energy are smaller than the convergence criteria that you specify. This method to calculate the energy is called a variational method. For variational methods, the calculated energy is guaranteed to be greater than or equal to the exact electronic ground-state energy of the molecule.

Active space selection

For some molecular systems, a single-determinant wave function doesn't adequately model the electronic structure and correlation, even in the basis set limit. In these cases, a multi-determinant wavefunction is used to model the molecular orbitals (MOs) that are most involved with chemical activity or have the most correlation.

Active space selection is the process of choosing which MOs from the SCF calculation to include in the multi-determinant wavefunction. The QDK for chemistry (QDK/Chemistry) in

Microsoft's Quantum Development Kit (QDK) offers tools that automatically choose the best active space for your problem given a set of parameters that you specify, such as the number of electrons and number of orbitals.

Get started with SCF calculations and active space selection in QDK the chemistry libraries

To learn more about SCF calculations in the QDK/Chemistry, see Self-consistent field (SCF) solver.

To learn more about active space selection in the QDK/Chemistry, see Active space selection.

For code examples of SCF calculations and active space selection in the QDK/Chemistry, see Run a self-consistent field (SCF) calculation and Select an active space in the QDK/Chemistry quickstart guide.

Last updated on 01/25/2026

Build state preparation circuits for quantum chemistry calculations with sparse isometry

State preparation is a central subroutine in many quantum algorithms for quantum chemistry calculations, such as Quantum Phase Estimation (QPE).

The state preparation circuit puts the qubit system on the quantum computer into a state that represents the real quantum state of your molecule. Isometry techniques are used to create the state preparation circuit from the wavefunction that you calculate for your molecule. Typical isometry approaches usually result in deep circuits that contain a lot of quantum gates. Each gate in a circuit has the potential to introduce noise into the calculation on the quantum computer, so the ideal circuit is one that performs an accurate computation with the fewest gates.

Sparse isometry circuit generation with the QDK chemistry libraries

QDK for chemistry (QDK/Chemistry) in the Microsoft Quantum Development Kit (QDK) includes a novel sparse isometry technique that uses a small number of dense qubits to efficiently build an optimal state preparation circuit with less depth and fewer gates than traditional isometry encoding techniques. The circuit from sparse isometry prepares the same initial state as the larger circuit from regular isometry, but the sparse isometry circuit is less computationally expensive and introduces less noise into the calculation on the quantum computer.

The sparse isometry implementation in QDK/Chemistry allows you to encode ab initio wavefunctions in a consistent linear workflow to help you build optimal state preparation circuits for a wide variety of molecules and systems. To get started, see How to install QDK for chemistry.

Learn more

To learn more about state preparation circuits and sparse isometry in QDK/Chemistry, see State preparation in the QDK/Chemistry documentation on GitHub.

Last updated on 01/25/2026

How to install QDK for chemistry

In this article, you learn how to install QDK for chemistry (QDK/Chemistry), a Python library for quantum chemistry calculations in the Microsoft Quantum Development Kit (QDK).

Prerequisites

- Install the latest version of Visual Studio Code (VS Code).
- Install the Python and Jupyter extensions in VS Code.
- Install a Python interpreter (version 3.11, 3.12, or 3.13).

Install the `qdk-chemistry` library

QDK/Chemistry is distributed as the `qdk-chemistry` Python library through PyPI. To install the package, follow the steps for your operating system.

macOS / Linux

1. Create a local folder where you want to install QDK/Chemistry.
2. In a terminal, go to the folder that you created.
3. Run the following terminal command to create a Python virtual environment. For example, `myenv`.

   Bash

   ```
   python3 -m venv myenv
   ```

4. Activate the virtual environment.

   Bash

   ```
   source myenv/bin/activate
   ```

5. Install the `qdk-chemistry` library with all extras.

   Bash

```
python3 -m pip install "qdk-chemistry[all]"
```

The `all` extra installs all optional dependencies. For other installation methods and platform-specific notes, see Installation Instructions for QDK/Chemistry in the QDK/Chemistry repository on GitHub.

> **Note**
>
> To use QDK/Chemistry with Jupyter Notebook in VS Code, make sure that your notebook uses the Python interpreter in the virtual environment where you installed `qdk-chemistry`.

Related content

To learn more about quantum chemistry simulations and how to use QDK/Chemistry, see the following articles:

- Build state preparation circuits for quantum chemistry calculations with sparse isometry
- Perform SCF calculations and active space selection to construct molecular orbitals

---

Last updated on 09/22/2026

How to use the molecule visualizer with QDK for chemistry

The Microsoft Quantum Development Kit (QDK) includes a molecule visualizer to use with QDK for chemistry (QDK/Chemistry). You can use the visualizer to interact with the 3D structure of your molecule and overlay the molecular orbitals (MOs) in a Jupyter notebook.

Prerequisites

To use the molecule visualizer, install the following tools:

- Python environment (version 3.11, 3.12, or 3.13) with Python and Pip
- Visual Studio Code (VS Code) with the Jupyter extension
- The latest version of the `qdk` Python package with the `jupyter` extra, and the `qdk-chemistry` library with all extras:

  Bash

  ```
  pip install --upgrade "qdk[jupyter]" "qdk-chemistry[all]"
  ```

For more details on QDK/Chemistry installation, see How to install QDK for chemistry.

Create a `Structure` object

The molecule visualizer requires a `Structure` object from the `qdk-chemistry` library. A `Structure` object contains the three-dimensional coordinates and element type of each atom in the molecule. To create a `Structure` object, you can load data from a `.xyz` or `.json` structure file, or you can directly specify the coordinates and elements in your code.

For example, to load a `Structure` object for *p*-benzyne from a `.xyz` file, follow these steps:

1. In VS Code, open the folder where you want to save your files.
2. Create an empty text file named `para-benzyne.structure.xyz`.
3. Copy the following text into `para-benzyne.structure.xyz` and save the file.

plaintext

```
10
para benzyne
C  0.000000  1.396000  0.000000
C  1.209077  0.698000  0.000000
C  1.209077 -0.698000  0.000000
C  0.000000 -1.396000  0.000000
C -1.209077 -0.698000  0.000000
C -1.209077  0.698000  0.000000
H  2.151000  1.242000  0.000000
H  2.151000 -1.242000  0.000000
H -2.151000 -1.242000  0.000000
H -2.151000  1.242000  0.000000
```

4. In VS Code, open the **View** menu and choose **Command Palette**.
5. Enter and select **Create: New Jupyter Notebook**. A new tab opens with an empty Jupyter Notebook file.
6. In the first cell of your notebook, copy and run the following code to load the structure:

Python

```python
from qdk_chemistry.data import Structure

structure = Structure.from_xyz_file('para-benzyne.structure.xyz')

structure
```

For more information about `Structure` objects in the QDK/Chemistry, see Structure in the QDK/Chemistry documentation on GitHub.

View the molecular structure

To open the molecule visualizer, use the `to_xyz` method to pass an XYZ string to the `MoleculeViewer` widget. Copy and run the following code in a new cell:

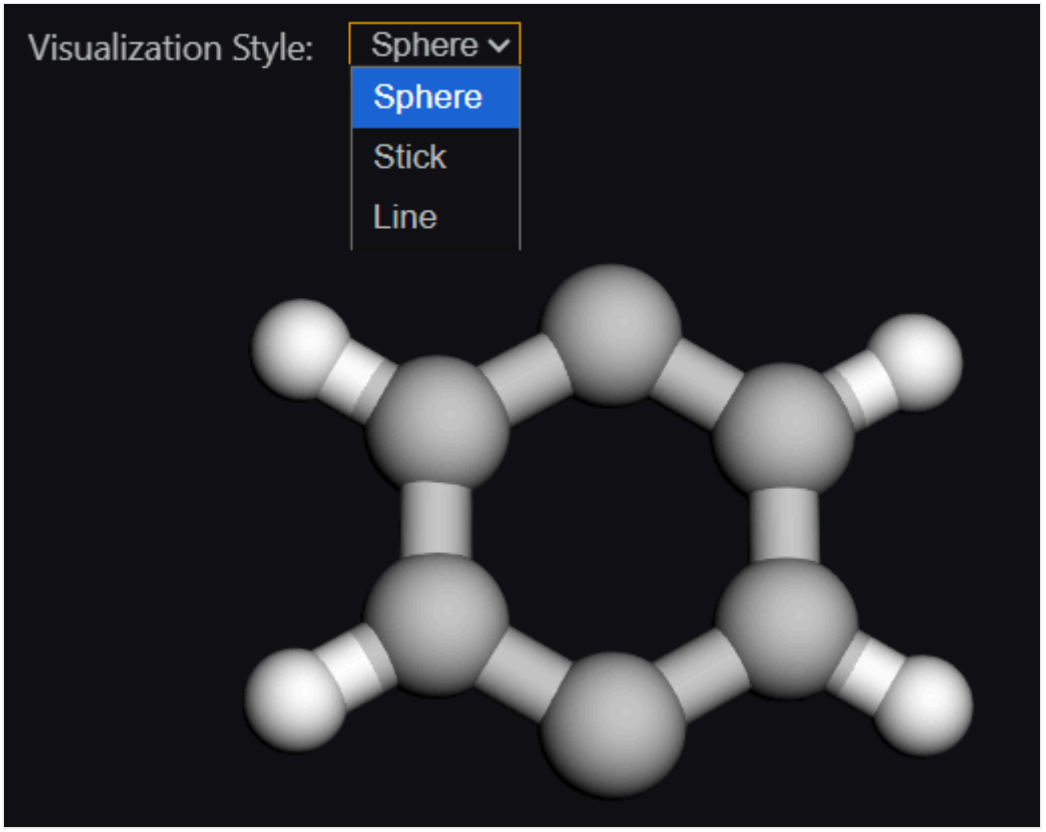
Python

```python
from qdk.widgets import MoleculeViewer

viewer = MoleculeViewer(molecule_data=structure.to_xyz())

viewer
```

The molecule visualizer opens in the output cell and renders a 3D representation of the molecular structure. The visualizer automatically draws bonds between atoms that are within a chemically reasonable distance threshold.



To change the visual representation, open the **Visualization Style** dropdown and choose one of the following options:

Expand table

Visualization style	Description
Sphere	Space-filling representation that shows atoms as spheres with element-specific size and color
Stick	Stick model that shows explicit bonds between atoms and element-specific colors
Line	Wireframe representation that shows bond connectivity but no color

Interact with the molecule visualizer

Use the following mouse and keyboard controls to interact with the molecule visualizer.

Expand table

Input	Action
Left-click and drag	Rotate the molecule in 3D space
Shift + left-click and drag	Zoom in (drag up) or zoom out (drag down)
Ctrl + left-click and drag	Translate the molecule

View the molecular orbitals

If you have `.cube` files that store the MO data for your molecule, you can load these files into the molecule visualizer to view the MOs overlaid on the molecule structure. Store the cube data in a Python dictionary with MO labels as keys, and then pass the dictionary to `MoleculeViewer`.

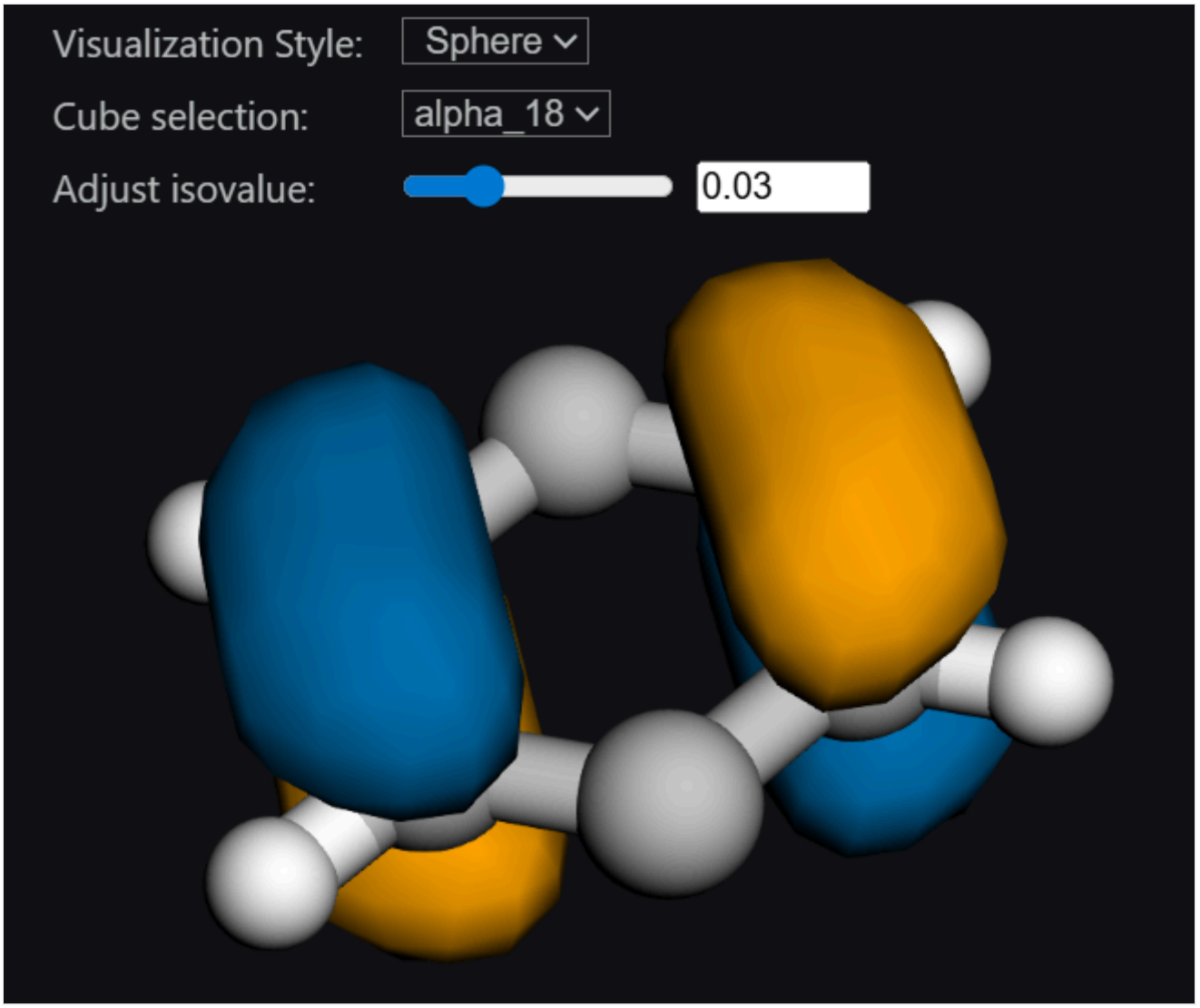
For example, the following code renders MOs for two active space orbitals in *p*-benzyne when you have the cube files in your working directory:

Python

```python
from pathlib import Path

cube_data = {
    "alpha_18": Path("MO_alpha_18.cube").read_text(),
    "alpha_19": Path("MO_alpha_19.cube").read_text(),
}

MoleculeViewer(molecule_data=structure.to_xyz(), cube_data=cube_data, isoval=0.03)
```



When you pass cube data to the `MoleculeViewer` widget, the visualizer displays the following UI elements.

Expand table

UI element	Description
Cube selection	Choose the MO that the visualizer displays. You can view only one MO at a time.
Adjust isovalue	Set the isovalue of the MO coefficients. The isovalue determines how the MOs are rendered.

For information on how to generate `.cube` files, see qdk_chemistry.utils.cubegen module in the QDK/Chemistry API reference on GitHub.

Display other MO information

The molecule visualizer can also display custom information about the MOs. To add custom information to the visualizer output, use dictionaries to define the cube data and include the

`"info"` key. The value for the `"info"` key is another dictionary that contains the custom information.

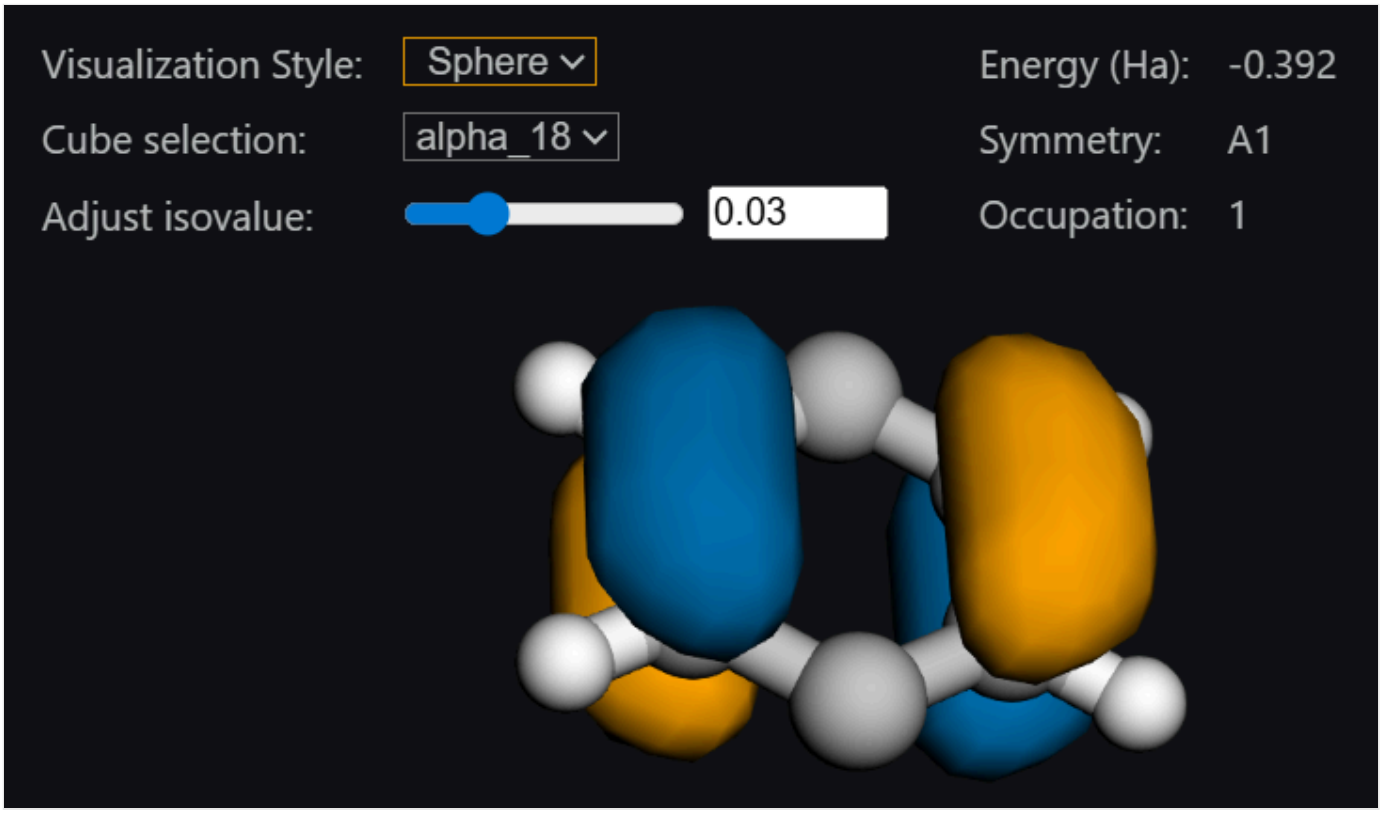
For example, the following code displays the energy, symmetry, and electron occupation of each MO:

Python

```
from pathlib import Path

cube_data = {
    "alpha_18": {"data": Path("MO_alpha_18.cube").read_text(),
                 "info": {"Energy (Ha)": -0.392, "Symmetry": "A1", "Occupation":
1.0}},
    "alpha_19": {"data": Path("MO_alpha_19.cube").read_text(),
                 "info": {"Energy (Ha)":  0.581, "Symmetry": "B2", "Occupation":
0.0}}
}

MoleculeViewer(molecule_data=structure.to_xyz(), cube_data=cube_data, isoval=0.03)
```



Last updated on 09/22/2026

What is the Microsoft Quantum resource estimator?

Quantum computers are prone to noise and errors because individual qubits are highly sensitive to changes in environmental conditions. Fault-tolerant quantum computing requires algorithms that use quantum error correction (QEC) codes. These codes build and use logical qubits from collections of individual physical qubits. Logical qubits correct errors while a program runs on a quantum computer so that quantum computations can provide reliable results.

The Microsoft Quantum resource estimator is an open-source tool in the Microsoft Quantum Development Kit (QDK) Python package that allows you to estimate the resources needed to run a quantum program on a fault-tolerant quantum computer. Error correction codes increase the number of physical qubits and run time of a quantum algorithm. The resource estimator determines how many physical qubits and how much time is needed for a quantum application to run on specific hardware with a given error correction scheme.

With the quantum resource estimator, you can compare qubit technologies, quantum error correction schemes, and hardware technologies to understand the impact on the resources needed to run a quantum program.

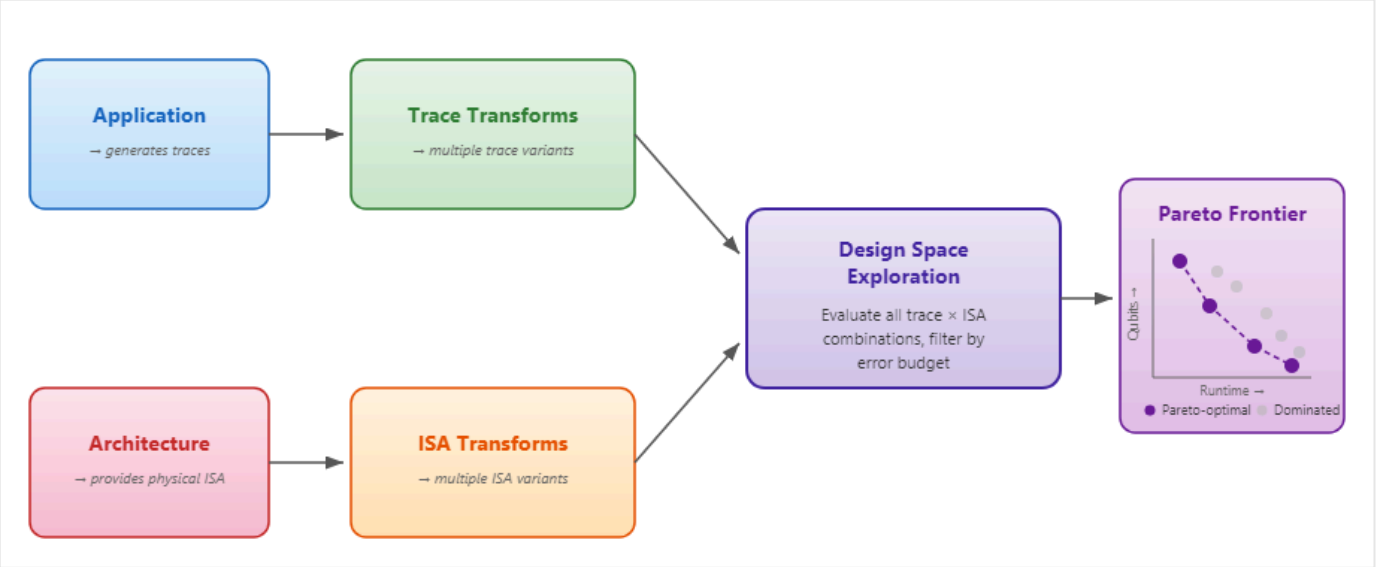
> **Note**
>
> The resource estimator uses quantum error correction models to estimate the resources needed to run fault-tolerant quantum applications. To develop error correction protocols and circuits, use the `qdk.ec` module. For more information, see Get started with QDK for error correction.

How does the quantum resource estimator work?

The resource estimator is designed around a layered modeling approach to be flexible and composable. You construct models for the application, the hardware, and the error correction scheme. You also specify how these models interact. All the models and instructions are independent of each other, so you can combine and compare different combinations of models.

The resource estimator uses intermediate representations to connect the layers. These representations are part of the estimator's implementation and aren't required for typical use.

You interact with the resource estimator through standard programming languages and high-level configuration models.



The quantum resource estimator requires four inputs:

- An application model that describes the quantum computation, such as a Q# program
- An architecture model that describes the target quantum hardware, such as gate-based superconducting qubits with specific gate times and error rates
- An error correction and factory distillation model that corresponds to the hardware architecture model
- An error budget, which is the maximum allowed error rate for operations on logical qubits

From the application model, the resource estimator generates one or more applications traces. These traces are compact representations of the instruction sequences that are applied to qubits. From the architecture model and error correction model, the resource estimator derives a physical instruction set (ISA) that specifies operation times, qubit costs, and error rates.

Both the application layer and the architecture layer are converted into configurable transforms. Trace transforms from the application layer include gate decompositions and layout routines. ISA transforms from the architecture layer include quantum error correction codes and magic state factories to build higher-fidelity logical instruction sets from the architecture's physical primitives.

The resource estimator explores a large combinatorial design space because there are many valid design choices at each layer. For example, different QEC code distances, different factory protocols, and different decomposition parameters. The resource estimator explores every combination of application trace and architecture ISA to evaluate physical qubit counts, run times, and accumulated error rates for each combination. The results are filtered down to a

Pareto-optimal frontier, which is the optimal set of configurations where no other result is simultaneously better in both qubits and runtime while staying within the specified error budget.

Application language support

The resource estimator accepts quantum applications in several in the following programming languages and intermediate formats:

- Q#
- Cirq
- OpenQASM
- QIR
- Logical counts
- Custom applications

With QIR and the ability to create custom applications, the resource estimator is language-agnostic and supports a broad range of quantum programming tools. Applications in different frameworks are analyzed using the same estimation pipeline. You can also build custom applications outside of these supported frameworks.

Architecture hardware support

Resource estimates depend on assumptions about the target hardware and the error correction code that protects logical operations from errors. The resource estimator includes built-in models for common architectures and error correction schemes. You can also define your own custom hardware and error correction models to explore hypothetical architectures, evaluate emerging technologies, or study how changes in physical parameters affect the resource requirements to run a program.

Compare estimates and visualize results

The resource estimator allows you to estimate the resources needed to run the same quantum algorithm for different configurations of target architecture models. You can plot the results for each configuration to compare the results. When you compare estimates, you can understand how the qubit architecture, QEC scheme, and other hardware parameters impact the overall resources required to run your program.

Get started with the resource estimator

To get started, see How to install and use the Microsoft Quantum resource estimator.

For more information and code samples, see the quantum resource estimator samples in the QDK repository on GitHub.

Last updated on 10/22/2026

How to install and use the Microsoft Quantum resource estimator

The Microsoft Quantum resource estimator is part of the Microsoft Quantum Development Kit (QDK). In this article, you learn how to install the quantum resource estimator and run an estimation for a simple Q# program with a default architecture model.

> ⚠ **Warning**
>
> The resource estimator in the QDK extension for VS Code will be deprecated soon. Use the `qdk.qre` Python module to perform resource estimation.

Prerequisites

To follow the steps in this article, you need to have the following installed:

- The latest version of Visual Studio Code (VS Code)
- The latest version of the QDK extension installed in VS Code
- The latest versions of the Python extension and Jupyter extension in VS Code

Install the quantum resource estimator

The resource estimator is available through the `qdk.qre` Python module. To use the resource estimator, install the latest version of the `qdk` Python package with the `qre` extra:

Bash

```
pip install --upgrade "qdk[qre]"
```

> 💡 **Tip**
>
> You don't need to have an Azure account to use the resource estimator.

Run a simple resource estimation with default settings

To perform a basic resource estimate, pass an application model, a hardware architecture model, and an error correction model to the resource estimator. Then, run the resource estimator to get the results. The following example runs the resource estimator from Jupyter Notebook in VS Code.

Create an application model

For this example, use the 1D Ising model algorithm from the Q# samples in the QDK VS Code extension. Or, use one of your existing Q# programs.

Write a Q# program

1. In VS Code, create a new text file and save the file as `ising-1d-sample.qs`.
2. On the first line of the file, enter **sample** to bring up the Q# sample program list.
3. Choose **Ising Model (Simple 1D) sample**, and then save the file.

Create an application

Use the Q# program to create an application model for the resource estimator.

1. To create a new Jupyter Notebook file in VS Code, open the **Command Palette** and enter **Create: New Jupyter Notebook**.

2. In the first notebook cell, evaluate the Q# program.

   Python

   ```
   from pathlib import Path
   from qdk import qsharp

   qsharp.eval(Path("ising-1d-sample.qs").read_text(encoding="utf-8"))
   ```

   This code creates a `Main` object in the `qdk.code` namespace that contains the Q# program.

3. In a new cell, create a resource estimator application from the Q# program.

   Python

   ```
   import qdk
   from qdk.qre.application import QSharpApplication

   app = QSharpApplication(qdk.code.Main)
   ```

Create an architecture model

For the hardware architecture, use the default `GateBased` model that the resource estimator comes with. The `GateBased` model requires inputs for error rate, gate time, and measurement time.

In a new cell, create a gate-based architecture with an error rate of 10^{-4}, a gate time of 100 ns, and a measurement time of 500 ns.

Python

```
from qdk.qre.models import GateBased

# Times are in units of nanoseconds
arch = GateBased(error_rate=1e-4, gate_time=100, measurement_time=500)
```

Run an estimation

To run an estimate, use the `estimate` function and pass the following inputs:

- Application
- Architecture
- Physical instruction set (ISA) query
- Maximum error rate

The ISA query tells the resource estimator what combinations of error correction code and T factory distillation protocols to evaluate. The resource estimator finds the Pareto frontier optimized set of results from those combinations. For this example, use the default `SurfaceCode` QEC and `RoundBasedFactory` distillation protocol that the resource estimator comes with.

Python

```
from qdk.qre import estimate
from qdk.qre.models import SurfaceCode, RoundBasedFactory

results = estimate(app, arch, isa_query=SurfaceCode.q() * RoundBasedFactory.q(),
max_error=0.01)```
```

View the results

To view the optimal combinations of number of physical qubits and program run time, use the `as_frame` method to display the estimation results in a pandas DataFrame.

Python

```
results.as_frame()
```

To plot the results, use the `plot_estimates` method and pass a unit for run time.

Python

```
from qdk.qre import plot_estimates

plot_estimates(results, figsize=(6, 4), runtime_unit="ms")
```

Last updated on 07/31/2026

How to build supported application models in the quantum resource estimator

The Microsoft Quantum resource estimator takes one or more quantum programs as input and converts the programs to applications and applications traces for resource estimation. You can write programs for the resource estimator in several quantum programming languages and frameworks. This article shows how to import programs and create applications for the default supported frameworks.

For instructions on how to install and use the resource estimator in the Microsoft Quantum Development Kit (QDK), see How to install and use the Microsoft Quantum resource estimator.

> ⚠ **Warning**
>
> The resource estimator in the QDK extension for VS Code will be deprecated soon. Use the `qdk.qre` Python module to perform resource estimation.

Build application models for supported frameworks

The resource estimator provides default tools to import quantum programs written in the following frameworks:

- Q#
- Cirq
- OpenQASM
- Quantum intermediate representation (QIR)

To build the following applications and run estimates for them, create a new Jupyter notebook in Visual Studio Code (VS Code):

1. In Visual Studio Code (VS Code), open the **View** menu and select **Command Palette**.
2. Enter **Create: New Jupyter Notebook**.

An empty Jupyter notebook opens in a new tab.

Build a Q# application

To import a Q# program, you can either load an existing `qs` Q# file or you can write a Q# directly in Python.

To write a Q# program directly in Jupyter Notebook and load it into the resource estimator as an application, follow these steps your empty Jupyter notebook:

1. Import the `qsharp` module from the QDK Python library.

   Python

   ```python
   from qdk import qsharp
   ```

2. In a new cell, write a Q# operation that performs ripple-carry addition on two registers of qubits.

   Q#

   ```
   %%qsharp

   import Std.Arithmetic.*;

   // Ripple-carry addition on two n-qubit registers
   operation Program(n : Int) : Unit {
       use a = Qubit[n];
       use b = Qubit[n];

       RippleCarryCGIncByLE(a, b);
   }
   ```

3. To create an application model, wrap the compiled program in `QSharpApplication` and pass the qubit register size as a runtime argument.

   Python

   ```python
   from qdk import code
   from qdk.qre.application import QSharpApplication

   # Wrap the compiled Q# entry point with a register size of 10 qubits
   qsharp_app = QSharpApplication(code.Program, args=(10,))
   ```

4. The application contains traces that the resource estimator uses to assess different configurations of application parameters. To view a summary of the ISA required to run the first trace, use the application's `enumerate_traces` method.

   Python

```
# Get the first generated application trace
trace = next(app.enumerate_traces())

# View the physical instruction set required to run the trace
trace.required_isa.as_frame()
```

The summary shows that the trace requires a CCX gate and a measurement in the Z basis.

Build a Cirq application

To build a Cirq application, import the `cirq` library to write a Cirq program, and then pass the program to `CirqApplication`. For example, use the built-in `qft` helper from `cirq` to create 10-qubit quantum Fourier transform program.

Python

```
import cirq
from qdk.qre.application import CirqApplication

# Build a 10-qubit QFT circuit and make it an application
circuit = cirq.Circuit(cirq.qft(*cirq.LineQubit.range(10)))
cirq_app = CirqApplication(circuit)
```

View the ISA summary for the first trace of the Cirq application:

Python

```
trace = next(cirq_app.enumerate_traces())
trace.required_isa.as_frame()
```

Build an OpenQASM application

To build an OpenQASM application, pass an OpenQASM 3 source string to `OpenQASMApplication`. For example, use Qiskit to generate an OpenQASM 3 program for a 4-qubit RGQFT multiplier, and then convert the program to an application.

Python

```
from qdk.qre.application import OpenQASMApplication
from qiskit.circuit.library import RGQFTMultiplier
from qiskit.qasm3.exporter import Exporter

# Export a Qiskit circuit to OpenQASM 3
qasm = Exporter().dumps(RGQFTMultiplier(num_state_qubits=4))
```

```
qasm_app = OpenQASMApplication(qasm)
```

View the ISA summary for the first trace of the OpenQASM application:

Python

```
trace = next(qasm_app.enumerate_traces())
trace.required_isa.as_frame()
```

Create a QIR application

QIR is an LLVM-based intermediate representation for quantum programs. Programs in most other quantum language frameworks can be compiled into QIR. To create an application from QIR, pass the QIR to `QIRApplication` as a string or bitcode.

For example, compile an OpenQASM Bell state preparation circuit into QIR, and then convert a string representation of the QIR into a resource estimator application.

Python

```
from qdk.openqasm import compile
from qdk.qre.application import QIRApplication

qasm_source = """
    include "stdgates.inc";
    bit[2] c;
    qubit[2] q;
    h q[0];
    cx q[0], q[1];
    c = measure q;
"""

qir = compile(qasm_source)

qir_app = QIRApplication(str(qir))
```

View the ISA summary for the first trace of the QIR application:

Python

```
trace = next(qir_app.enumerate_traces())
trace.required_isa.as_frame()
```

Run resource estimations for multiple applications

The preceding code created four resource estimator applications, each from a different programming framework. With the resource estimator, you can run estimates on all four applications and compare the results. To compare estimates, the applications don't need to be from the same programming framework.

To run estimates for all four applications with the same architecture models, follow these steps:

1. Create a gate-based architecture model with a 100 ns gate time and a 500 ns measurement time.

   Python

   ```
   from qdk.qre.models import GateBased

   arch = GateBased(gate_time=100, measurement_time=500)
   ```

2. Create and ISA query for a surface-based QEC and round-based factory distillation.

   Python

   ```
   from qdk.qre.models import SurfaceCode, RoundBasedFactory

   isa_query = SurfaceCode.q() * RoundBasedFactory.q()
   ```

3. Run estimates for each application.

   Python

   ```
   apps = [("Q# adder", qsharp_app), ("Cirq QFT", cirq_app), ("QIR Bell state",
   qir_app), ("OpenQASM multiplier", qasm_app)]

   estimates = []
   for name, app in apps:
       estimates.append(estimate(app, arch, isa_query, max_error=0.01, name=name))
   ```

4. Plot the Pareto frontier optimal estimates for each application.

   Python

   ```
   from qdk.qre import plot_estimates

   plot_estimates(estimates)
   ```

Custom applications

The quantum resource estimator offers the flexibility to build your own custom applications from a set of fixed program parameters and variable application parameters. For more information, see How to build custom application models in the quantum resource estimator.

Last updated on 06/17/2026

How to build custom application models in the quantum resource estimator

The Microsoft Quantum resource estimator has default application support for several quantum programming frameworks, such as Q# and Cirq. The parameters for these supported application types are predefined by the resource estimator. The resource estimator also lets you build custom applications for when you need finer control over the application parameters.

> ⚠ **Warning**
>
> The resource estimator in the QDK extension for VS Code will be deprecated soon. Use the `qdk.qre` Python module to perform resource estimation.

Custom application model parameters

To build your own applications, use the `Application` subclass. Custom application models consist of two types of parameters:

Expand table

Parameter type	Description	Examples
Instance parameters	Define the problem that the application solves. These parameters are fixed for a given estimation run.	The bit-size of an adder, the key length of an encryption scheme
Trace parameters	Algorithmic hyperparameters that control how the application solves the problem. The resource estimator explores all combinations of these parameters during estimation.	Choice of adder circuit, compute-to-memory ratio, eviction strategy

Example custom application

As an example custom application, build a configurable adder application with the following parameters:

- **Instance parameter**: Bit-size
- **Trace parameters**: Adder implementation, compute fraction, memory-management strategy

With a fixed bit-size, the resource estimator produces application traces for each combination of adder implementation, compute fraction, and memory-management strategy. The output is the Pareto-optimal result that minimizes the trade-offs between number of qubits and total run time for the algorithm.

To build the adder in with the custom parameters in Q#, follow these steps:

1. In VS Code, open the **View** menu and select **Command Palette**.
2. Enter **Create: New Jupyter Notebook**.
3. In the first cell, import `qsharp` to use the `%%qsharp` magic cell.

   Python

   ```
   from qdk import qsharp
   ```

4. In a new cell, run the Q# adder function code. Some of the variables, like `adder` and `strategy`, are defined in subsequent steps.

   Q#

   ```
   %%qsharp

   import Std.Arithmetic.*;
   import Std.Convert.*;
   import Std.Math.*;
   import Std.ResourceEstimation.*;

   // An adder operation that with custom instance and trace parameters for resource
   estimation
   operation Adder(n : Int, adder: (Qubit[], Qubit[]) => Unit, computeFraction:
   Double, strategy: Int) : Unit {
       use a = Qubit[n];
       use b = Qubit[n];

       if computeFraction > 0.0 {
           // Enables an automatic memory/compute architecture with a fixed number
   of compute qubits
           // (computed as a fraction of the total number of qubits).  The strategy
   controls how
           // compute qubits are deallocated if free compute qubits are needed
   (either least recently
           // used or least frequently used).
           EnableMemoryComputeArchitecture(Ceiling(computeFraction *
   IntAsDouble(n)), strategy);
       }

       // Run the adder
   ```

```
    adder(a, b);

    // Reset all qubits (if code is used in simulation)
    ResetAll(a + b);
}

// Re-export adders and strategies to use in Python, together with strings to
display the results
function CGAdder(): ((Qubit[], Qubit[]) => Unit, String) { (RippleCarryCGIncByLE,
"CG") }
function TTKAdder(): ((Qubit[], Qubit[]) => Unit, String) {
(RippleCarryTTKIncByLE, "TTK") }
function LRU(): (Int, String) { (LeastRecentlyUsed(), "LRU") }
function LFU(): (Int, String) { (LeastFrequentlyUsed(), "LFU") }
```

5. Define the trace parameters as a `dataclass` with `kw_only=True` because the resource estimator treats only keyword-only fields as variable trace parameters. Other fields are fixed during estimation.

Python

```
from dataclasses import dataclass, field
from typing import Any
from qdk import code

@dataclass(kw_only=True)
class AdderParameters:
    adder: tuple[Any, str] = field(default = code.CGAdder(), metadata={"domain":
[code.CGAdder(), code.TTKAdder()]})
    compute_fraction: float = field(default=1.0, metadata={"domain": [0.5, 1.0,
1.5, 2.0]})
    strategy: tuple[int, str] = field(default=code.LRU(), metadata={"domain":
[code.LRU(), code.LFU()]})
```

You need to specify a `"domain"` in each keyword's metadata dictionary because the domain tells the resource estimator what trace values to explore during estimation. You also need to specify a default value from the list of domain values for each parameter. The resource estimator evaluates the following trace parameters for the adder program:

Expand table

Parameter	Description	Values
`adder`	Adder implementation	Craig Gidney (CG) ripple-carry adder, Takahashi–Tani–Kunihiro (TTK) ripple-carry adder
`compute_fraction`	Ratio of qubits allocated to active	0.5, 1.0, 1.5, 2.0

Parameter	Description	Values
	computation versus memory	
`strategy`	Eviction policy when compute capacity is exceeded	Least Recently Used (LRU), Least Frequently Used (LFU)

6. To bind the trace parameter types, set `Application[AdderParameters]` as a subclass to `Adder` with the fixed instance parameters. Define instance parameters as regular (non-keyword) dataclass fields. In this case, `bitsize` is the only instance parameter.

Python

```
from qdk.qre import Application
from qdk.qre.interop import trace_from_entry_expr
from qdk.qre.property_keys import custom_properties

# Property keys are integers; we can use the custom_properties function to
# define custom properties that do not conflict with existing properties.
ADDER, COMPUTE_FRACTION, STRATEGY = custom_properties(3)

@dataclass
class Adder(Application[AdderParameters]):
    bitsize: int

    def __post_init__(self):
        # Disable parallel trace generation since passing the Q# adder operations
is not thread-safe.
        self.disable_parallel_traces()

    def get_trace(self, parameters: AdderParameters):
        # obtain the adder function and its name
        adder_fn, adder_name = parameters.adder
        # obtain the memory/compute strategy and its name
        strategy, strategy_name = parameters.strategy
        # generate a trace from the Q# entry point with the specified parameters
        trace = trace_from_entry_expr(code.Adder, self.bitsize, adder_fn,
parameters.compute_fraction, strategy)

        # Set trace properties for later analysis and display
        trace.set_property(ADDER, adder_name)
        trace.set_property(COMPUTE_FRACTION, parameters.compute_fraction)
        trace.set_property(STRATEGY, strategy_name)

        return trace
```

The `get_trace` builds an application trace for every combination of variable trace parameter value, with fixed instance parameter values for each trace. The `set_property` method sets custom metadata that you can use to filter estimation results.

7. Build an architecture model and physical instruction set (ISA) model for quantum error correction (QEC) and factory distillation protocol. For this example, use a gate-based hardware model and an ISA model that includes a 2D yoked surface code to account for memory qubits.

Python

```
from qdk.qre.models import GateBased, RoundBasedFactory, SurfaceCode,
TwoDimensionalYokedSurfaceCode

# Define the target architecture: gate-based with specified timing (ns)
arch = GateBased(gate_time=100, measurement_time=500)

# Explore surface code distances × round-based magic state factories x 2D yoked
surface code distances
isa_query = SurfaceCode.q() * RoundBasedFactory.q() *
TwoDimensionalYokedSurfaceCode.q(source=SurfaceCode.q())
```

8. Run the resource estimator and add custom results columns.

Python

```
from qdk.qre import estimate

results = estimate(Adder(64), arch, isa_query, max_error=0.01, name="Configurable
adder")

# Add results columns using default resource estimator functions
results.add_factory_summary_column()
results.add_qubit_partition_column()

# Add results columns for custom application parameters
results.add_property_column(ADDER, "adder")
results.add_property_column(COMPUTE_FRACTION, "compute_fraction")
results.add_property_column(STRATEGY, "strategy")

results.as_frame()
```

9. Plot the Pareto frontier estimation results.

Python

```
results.plot()
```

For more information on how to access results from the resource estimator, see How to work with results from the quantum resource estimator.

Related content

- What is the Microsoft Quantum resource estimator?
- How to build supported application models in the quantum resource estimator

---

Last updated on 06/17/2026

How to build hardware architecture models in the quantum resource estimator

The Microsoft Quantum resource estimator takes a hardware architecture model as one of its inputs to estimate the resources needed to run a program on a quantum computer. The architecture model describes the physical hardware characteristics of the computer. The resource estimator includes default hardware models and lets you build custom models.

> ⚠ **Warning**
>
> The resource estimator in the QDK extension for VS Code will be deprecated soon. Use the `qdk.qre` Python module to perform resource estimation.

Default architecture models

The quantum resource estimator provides three default architecture models:

- The `GateBased` class for general qubits with typical gate-based devices
- The `NeutralAtom` class for neutral atom qubit devices
- The `Majorana` class for Majorana qubits

The parameters for the default models apply to the entire architecture. For example, when you set a gate time, that time applies to all gates in the application. If you want different times for different types of gates, you need to build a custom architecture model.

Create a gate-based architecture

To build a default gate-based architecture model, use the `GateBased` class. The default gate-based model requires four parameters:

Expand table

Parameter	Units	Default value	Description
`error_rate`	Unitless	10^{-4}	Probability that an error occurs after a gate or measurement
`gate_time`	Nanoseconds	None	Time it takes to apply a single-qubit gate

Parameter	Units	Default value	Description
`measurement_time`	Nanoseconds	None	Time it takes to measure a qubit's state
`two_qubit_gate_time`	Nanoseconds	Value of `gate_time`	Time it takes to apply a two-qubit gate

The following code builds a gate-based architecture model with 200 ns gate times and 500 ns measurement times, and an overall error rate of 10^{-5}:

Python

```python
from qdk.qre.models import GateBased

gate_arch = GateBased(error_rate=1e-5, gate_time=200, measurement_time=500)
```

To view the properties of all the supported operations in the default gate-based architecture, run the following code:

Python

```python
gate_arch.provided_isa(gate_arch.context()).as_frame()
```

Each gate and measurement operation has the following default parameters:

Expand table

Parameter	Description
`encoding`	Specifies whether the operation acts on physical or logical qubits
`arity`	Number of inputs that the operation acts on
`space`	Number of physical qubits that the operation acts on
`time`	The time in nanoseconds that it takes for the operation to complete
`error`	The probability that an error occurs after the operation

Create a neutral atom device architecture

To build a default neutral atom device architecture model, use the `NeutralAtom` class. This model incorporates the physical characteristic of neutral atom devices, such as qubit spacing, qubit movement, and a specific gate set. The default neutral atom model requires ten parameters. For more information, see the API reference.

The following code builds a neutral atom architecture model with 5,000 ns measurement times, 5 µm atom spacing, and default values for the other parameters:

Python

```
from qdk.qre.models import NeutralAtom

atom_arch = NeutralAtom(measurement_time=5000, atom_spacing=5)
```

To view the properties of all the supported operations in the default neutral atom architecture, run the following code:

Python

```
atom_arch.provided_isa(gate_arch.context()).as_frame()
```

Create a Majorana architecture

To build a default Majorana qubit architecture model, use the `Majorana` class. The default Majorana model takes an error rate as the only input. All operation and measurement times are fixed at 1,000 ns.

The following code builds a default Majorana architecture model with an overall error rate of 10^{-6}:

Python

```
from qdk.qre.models import Majorana

maj_arch = Majorana(error_rate=1e-6)
```

To view the properties of all the supported operations in the default Majorana architecture, run the following code:

Python

```
maj_arch.provided_isa(maj_arch.context()).as_frame()
```

Build custom architecture models

An architecture model defines the physical operations that the hardware of a quantum computer can perform, with times and error rates for each operation. The default `GateBased` and `Majorana`

models have fixed operations and times. The resource estimator lets you build custom architecture models where you choose the set operations, and can set different times and error rates for each operation.

To create a custom architecture model, use the `Architecture` subclass and define its `provided_isa` method with the `ISAContext` parameter. For example, build a model where two-qubit gates have higher error rates and longer operation times than single-qubit gates, and measurement operations have a different time and error rate than gate operations.

The following code builds a configurable model with physical qubit operations called `ParameterizedArchitecture`:

Python

```python
from dataclasses import KW_ONLY, dataclass
from qdk.qre import Architecture, ISAContext, ISA
from qdk.qre.instruction_ids import H, S, S_DAG, T, T_DAG, CNOT, CZ, MEAS_X, MEAS_Y,
MEAS_Z

@dataclass
class ParameterizedArchitecture(Architecture):
    """
    A configurable architecture with separate timing and error parameters for
    single-qubit gates, two-qubit gates, and measurements.
    """

    _: KW_ONLY
    single_qubit_gate_time: int
    two_qubit_gate_time: int
    measurement_time: int
    single_qubit_error_rate: float
    two_qubit_error_rate: float
    measurement_error_rate: float

    def provided_isa(self, ctx: ISAContext) -> ISA:
        instructions = []

        # Single-qubit gates: Clifford gates and T gates
        for gate_id in [H, S, S_DAG, T, T_DAG]:
            instructions.append(
                ctx.add_instruction(
                    gate_id,
                    encoding=PHYSICAL,
                    arity=1,
                    time=self.single_qubit_gate_time,
                    error_rate=self.single_qubit_error_rate,
                )
            )

        # Measurements
```

```
        for meas_id in [MEAS_X, MEAS_Y, MEAS_Z]:
            instructions.append(
                ctx.add_instruction(
                    meas_id,
                    encoding=PHYSICAL,
                    arity=1,
                    time=self.measurement_time,
                    error_rate=self.measurement_error_rate,
                )
            )

        # Two-qubit entangling gates
        for gate_id in [CNOT, CZ]:
            instructions.append(
                ctx.add_instruction(
                    gate_id,
                    encoding=PHYSICAL,
                    arity=2,
                    time=self.two_qubit_gate_time,
                    error_rate=self.two_qubit_error_rate,
                )
            )

        return ctx.make_isa(*instructions)
```

The `add_instruction` method from `ISAContext` lets you define physical hardware characteristics, such as gate type, qubit encoding, and qubit arity.

To use the custom model, call `ParameterizedArchitecture` and set values for the custom parameters. For example, the following model describes hardware with fast but noisy operations:

Python

```
fast_noisy = ParameterizedArchitecture(
    single_qubit_gate_time=50,
    two_qubit_gate_time=100,
    measurement_time=200,
    single_qubit_error_rate=1e-4,
    two_qubit_error_rate=5e-4,
    measurement_error_rate=5e-4,
)

# Inspect the physical ISA provided by the fast-but-noisy architecture
fast_noisy.provided_isa(fast_noisy.context()).as_frame()
```

Run estimates for custom hardware architectures

To run an estimate on this model, pass the `fast_noisy` model to the `estimate` function as the architecture parameter along with an application model and an error correction model.

1. Import the necessary modules and objects.

Python

```
from qdk import qsharp
from qdk.qre import estimate, plot_estimates
from qdk.qre.application import QSharpApplication
from qdk.qre.models import SurfaceCode, RoundBasedFactory
```

2. Build an application model. For example, write an 8-bit adder program in Q# with single-qubit rotations.

Q#

```
%%qsharp

import Std.Arithmetic.*;
import Std.Math.*;
import Std.Convert.*;

/// An 8-bit adder with single-qubit rotations.
/// The rotations introduce T gates via synthesis, which exercises
/// the magic state factory during resource estimation.
operation AdderWithRotations() : Unit {
    use a = Qubit[8];
    use b = Qubit[8];
    for i in 0..7 {
        Ry(PI() / IntAsDouble(i + 2), a[i]);
    }
    RippleCarryCGIncByLE(a, b);
}
```

3. Convert the Q# program into an application model.

Python

```
from qdk import code

app = QSharpApplication(code.AdderWithRotations)
```

4. Build an error correction and factory distillation model. For example, use a surface error correction code and round-based factory distillation.

Python

```
isa_query = SurfaceCode.q() * RoundBasedFactory.q()
```

5. To run an estimate, pass the application model, architecture model, and error correction model to the `estimate` function. Set a 1% maximum error rate.

Python

```
result = estimate(app, fast_noisy, isa_query, max_error=0.01)

result.plot(figsize=(8, 5))
```

6. To compare estimates for different custom architecture parameter sets, build a list of estimates and plot the results. For example, compare the `fast_noisy` model to a `slow_clean` model that has longer operation times but less noise. Set the same maximum error rate for each estimate.

Python

```
slow_clean = ParameterizedArchitecture(
    single_qubit_gate_time=500,
    two_qubit_gate_time=1000,
    measurement_time=1000,
    single_qubit_error_rate=1e-6,
    two_qubit_error_rate=5e-6,
    measurement_error_rate=1e-5,
)

results = [
    estimate(app, fast_noisy, isa_query, max_error=0.01, name="Fast but noisy"),
    estimate(app, slow_clean, isa_query, max_error=0.01, name="Slow but clean")
]

plot_estimates(results, figsize=(8, 5), runtime_unit="ms")
```

Related content

- How to build custom application models in the quantum resource estimator
- How to build error correction models in the quantum resource estimator
- How to work with results from the quantum resource estimator

---

Last updated on 06/17/2026

How to build error correction models in the quantum resource estimator

The Microsoft Quantum resource estimator takes a quantum error correction (QEC) code transform and magic state factory transform model as one of its core inputs. The QEC and factory transforms determine how the physical instruction set (ISA) from the hardware architecture model converts to a logical instruction set for error correction.

From the QEC code model and magic state factory model, you build an ISA query that you pass to the resource estimator. The ISA query defines the combinations of QEC codes and magic state factories that the resource estimator explores. Each combination of QEC code and magic state factory determines how the resource estimator converts the physical ISA into a logical ISA. In this article, you learn how to build ISA queries from the default QEC code models and default magic state factory models, and how to build your own custom models.

> ⚠ **Warning**
>
> The resource estimator in the QDK extension for VS Code will be deprecated soon. Use the `qdk.qre` Python module to perform resource estimation.

Build ISA queries from the default QEC code and magic state factory models

The quantum resource estimator includes four default QEC code models and three default magic state factory models.

Default QEC code models

The quantum resource estimator includes the following default QEC models:

Expand table

QEC model	Description
SurfaceCode	Gate-based rotated surface code
SurfaceCodeLowMove	Gate-based rotated surface code for neutral atom hardware that has

QEC model	Description
	mobile ancilla qubits
OneDimensionalYokedSurfaceCode	1D Yoked surface code that provides generic memory instructions
TwoDimensionalYokedSurfaceCode	2D Yoked surface code that provides generic memory instructions
ThreeAux	Pairwise measurement-based surface code with three auxiliary qubits per stabilizer measurement

Default magic state factory models

The quantum resource estimator includes the following default magic state factory models:

Expand table

Magic state factory model	Description
RoundBasedFactory	Magic state factory that uses round-based distillation pipelines to produce T gate instructions
Litinski19Factory	T and CCZ factories based on the paper arXiv:1905.06903
GSJ24CCXFactory	8T-to-CCZ factories based on the paper arXiv:2409.17595
GSJ24Factory	Magic state cultivation based on the paper arXiv:2409.17595
MagicUpToClifford	ISA transform that adds Clifford equivalent representations of magic states

Build an ISA query from default models

To build an ISA query, combine a QEC model with a magic state factory model using either the `+` or `*` operator. For example, build an ISA query from the gate-based surface code QEC model and the round-based magic state factory model.

Python

```
from qdk.qre.models import SurfaceCode, RoundBasedFactory

isa_query = SurfaceCode.q() * RoundBasedFactory.q()
```

Build ISA queries from custom QEC code and magic state factory models

The quantum resource estimator lets you build custom QEC and magic state factory models. These custom models use the `ISATransform` subclass to convert physical qubit instructions into logical qubit instructions.

To build these custom models, use the `ISATransform` subclass and define two methods:

- `required_isa`: A static method that defines what operations to use from the architecture model's ISA. This method can't reference fields, such as `error_correction_threshold`. Instead, use `ConstraintBound` with comparison operators to set limits or ranges on values.
- `provided_isa`: Computes logical instruction properties from physical instructions to produce one or more logical ISAs. This method references the custom fields in your model.

Build a custom QEC model

A QEC transform encodes qubits into an error correction code based on these main parameters:

Expand table

QEC code parameter	Description	Depends on
Space	The number of physical qubits per logical qubits	Code distance
Time	The time it takes to complete one logical operation	Syndrome extraction cycles
Error	The probability that an error occurs from a logical operation	Physical error rate, maximum error rate, and code distance

Custom QEC transforms need to define values for these parameters. For example, the following code builds a custom QEC model called `GenericQEC` that converts physical H gate, CNOT gate, and measurement instructions into a logical lattice surgery instruction set.

Python

```
from dataclasses import KW_ONLY, dataclass, field
from typing import Generator
from qdk.qre import (
    ISAContext,
    ConstraintBound,
    ISA,
    ISARequirements,
    ISATransform,
    LOGICAL,
    constraint,
```

```
    linear_function
)
from qdk.qre.instruction_ids import H, CNOT, MEAS_Z, LATTICE_SURGERY

@dataclass
class GenericQEC(ISATransform):
    """
    A configurable QEC code model with tunable threshold, overhead,
    and error suppression.

    This transform consumes physical H, CNOT, and MEAS_Z instructions and
    produces a logical LATTICE_SURGERY instruction. The resource formulas
    are parameterized so you can model different QEC code families.

    The logical error rate follows:

        error = crossing_prefactor × (p_physical / threshold) ^ ⌊(d+1)/2⌋

    Space per data qubit is:

        qubits_per_data_qubit × d²
    """

    crossing_prefactor: float = 0.03
    error_correction_threshold: float = 0.01
    qubits_per_data_qubit: int = 2
    syndrome_extraction_depth: int = 4
    _: KW_ONLY
    distance: int = field(default=3, metadata={"domain": range(3, 22, 2)})

    @staticmethod
    def required_isa() -> ISARequirements:
        # required_isa is static, so we cannot reference instance fields here.
        # Instead, we use ConstraintBound to express fixed constraints on the
        # implementation ISA. ConstraintBound supports .lt(), .le(), .eq(),
        # .gt(), and .ge() comparisons.
        return ISARequirements(
            constraint(H, error_rate=ConstraintBound.lt(0.01)),
            constraint(CNOT, arity=2, error_rate=ConstraintBound.lt(0.01)),
            constraint(MEAS_Z, error_rate=ConstraintBound.lt(0.01)),
        )

    def provided_isa(
        self, impl_isa: ISA, ctx: ISAContext
    ) -> Generator[ISA, None, None]:
        cnot = impl_isa[CNOT]
        h = impl_isa[H]
        meas_z = impl_isa[MEAS_Z]

        # Physical error rate is the worst case across all required gates
        physical_error_rate = max(
            cnot.expect_error_rate(),
            h.expect_error_rate(),
            meas_z.expect_error_rate(),
        )
```

```
        d = self.distance

        # Space: physical qubits per data qubit, scaled by distance².
        #
        # Because LATTICE_SURGERY has variable arity (it can operate on any
        # number of logical qubits), space and error_rate must be provided as
        # functions of arity rather than as fixed numbers. The estimator
        # provides several helpers for this:
        #
        #   linear_function(slope)          → f(n) = slope × n
        #   constant_function(value)        → f(n) = value
        #   block_linear_function(k, s, o)  → f(n) = s × ⌈n/k⌉ + o
        #   generic_function(callable)      → f(n) = callable(n)
        #
        # For QEC codes, resources scale linearly with the number of logical
        # qubits, so linear_function is the natural choice.
        space = linear_function(self.qubits_per_data_qubit * d**2)

        # Time: syndrome extraction cycle × code distance
        code_cycle_time = (
            h.expect_time()
            + self.syndrome_extraction_depth * cnot.expect_time()
            + meas_z.expect_time()
        )
        time = code_cycle_time * d

        # Error: exponential suppression below threshold
        logical_error = self.crossing_prefactor * (
            (physical_error_rate / self.error_correction_threshold)
            ** ((d + 1) // 2)
        )
        error = linear_function(logical_error)

        yield ctx.make_isa(
            ctx.add_instruction(
                LATTICE_SURGERY,
                encoding=LOGICAL,
                arity=None,
                space=space,
                time=time,
                error_rate=error,
                # transform=self records that this instruction was produced by
                # this QEC transform instance, and source=[...] records which
                # physical instructions it was derived from. Together, they
                # form the provenance chain that you can inspect in the
                # estimation results (see notebook 2).
                transform=self,
                source=[cnot, h, meas_z],
                # Extra keyword arguments (like distance) are stored as
                # properties on the instruction.  This lets you retrieve the
                # code distance used for each Pareto-optimal result when
                # analyzing the estimation output. Further, the property can be
                # required and read by a parent ISATransform to create new
                # logical instructions based on this one.
```

```
                distance=d,
            ),
        )
```

You can call `GenericQEC` with the default parameter values, or you can set other values when you call the model. For example, create a QEC model that has a high error threshold and low qubit overhead.

Python

```
custom_qec = GenericQEC.q(
    crossing_prefactor=0.01,
    error_correction_threshold=0.03,
    qubits_per_data_qubit=1,
)
```

Build a custom magic state factory model

A magic state factory model transform produces high-fidelity magic states from low-level physical instructions.

You can build a custom magic state factory model, or use a custom model to explore how established models behave with changes to specific parameters. For example, build a custom factory model called `ScaledLitinskiFactory` that's based on the default `Litinski19Factory` model. The custom model lets you modify the space factor for logical qubits and the number of cycles that the factory requires.

Python

```
from dataclasses import dataclass
from typing import Generator
from math import ceil
from qdk.qre import (
    ISAContext,
    ConstraintBound,
    ISA,
    ISARequirements,
    ISATransform,
    LOGICAL,
    constraint
)
from qdk.qre.instruction_ids import T, H, CNOT, MEAS_Z

# Reference data from Litinski (arXiv:1905.06903), Table 1.
# Assumes Clifford and T error rates at most 1e-4.
# Each tuple: (output_error_rate, space_in_physical_qubits, cycles)
_LITINSKI_TABLE = [
```

```
    (4.4e-8,    810, 18.1),
    (1.5e-9,    762, 36.2),
    (9.3e-10,  1150, 18.1),
    (1.9e-11,  2070, 30.0),
    (2.4e-15, 16400, 90.3),
    (6.3e-25, 18600, 67.8),
]

@dataclass
class ScaledLitinskiFactory(ISATransform):
    """
    A factory transform based on the Litinski (2019) distillation protocols,
    with configurable scaling factors for space and cycle costs.

    The base data is taken directly from Table 1 in arXiv:1905.06903.
    Each entry specifies a distillation protocol's output error rate, space
    footprint (in physical qubits), and time (in syndrome extraction cycles).
    The space already includes the QEC overhead within the factory itself.

    Scaling factors let you explore hypothetical improvements:

    - ``space_factor=0.5`` models a factory that uses half the qubits.
    - ``cycle_factor=0.5`` models a factory that runs twice as fast.

    Note:
        The error rates are kept fixed when scaling.  In practice, changing
        space or cycles would affect error rates too, but modelling that
        relationship requires protocol-specific analysis.
    """

    space_factor: float = 1.0
    cycle_factor: float = 1.0

    @staticmethod
    def required_isa() -> ISARequirements:
        return ISARequirements(
            constraint(T, error_rate=ConstraintBound.le(1e-4)),
            constraint(H, error_rate=ConstraintBound.le(1e-4)),
            constraint(CNOT, arity=2, error_rate=ConstraintBound.le(1e-4)),
            constraint(MEAS_Z, error_rate=ConstraintBound.le(1e-4)),
        )

    def provided_isa(
        self, impl_isa: ISA, ctx: ISAContext
    ) -> Generator[ISA, None, None]:
        cnot = impl_isa[CNOT]
        h = impl_isa[H]
        meas_z = impl_isa[MEAS_Z]
        t = impl_isa[T]

        # Syndrome extraction time from the physical ISA
        syndrome_extraction_time = (
            4 * cnot.expect_time() + h.expect_time() + meas_z.expect_time()
        )
```

```
        for error_rate, base_space, base_cycles in _LITINSKI_TABLE:
            # Apply scaling factors to space and time
            space = ceil(base_space * self.space_factor)
            time = ceil(syndrome_extraction_time * base_cycles * self.cycle_factor)

            yield ctx.make_isa(
                ctx.add_instruction(
                    T,
                    encoding=LOGICAL,
                    arity=1,
                    space=space,
                    time=time,
                    error_rate=error_rate,
                    transform=self,
                    source=[cnot, h, meas_z, t],
                ),
            )
```

With the custom `ScaledLitinskiFactory` model, you can lower `space_factor` to explore factories that use fewer physical qubits or lower `cycle_factory` to explore factories that need fewer cycles for distillation. For example, create a model that uses half the physical qubits as `LitinskiFactory`, and another model that needs half as many cycles.

Python

```
compact_factory = ScaledLitinskiFactory.q(space_factor=0.5)
fast_factory = ScaledLitinskiFactory.q(cycle_factor=0.5)
```

Build an ISA query from custom models

To build an ISA query from custom QEC and distillation models, set the custom parameter values and combine the models using either the `+` or `*` operator. For example, build an ISA query from the custom QEC model that you already built. Set the `GenericQEC` model to have a higher error threshold and set the `ScaledLitinskiFactory` model to require fewer cycles.

Python

```
custom_qec = GenericQEC.q(error_correction_threshold=0.02)
custom_factory = ScaledLitinskiFactory.q(cycle_factor=0.75)

custom_query = custom_qec * custom_factory
```

Run estimates for custom QEC and distillation models

To run estimates with your custom models, pass `custom_query` to the `estimate` function as the ISA query parameter along with an application model and a hardware architecture model.

1. Import the necessary modules and objects.

Python

```python
from qdk import qsharp
from qdk.qre.application import QSharpApplication
from qdk.qre.models import GateBased
from qdk.qre import estimate, plot_estimates
```

2. Build an application model. For example, write an 8-bit adder program in Q# with single-qubit rotations.

Q#

```
%%qsharp

import Std.Arithmetic.*;
import Std.Math.*;
import Std.Convert.*;

/// An 8-bit adder with single-qubit rotations.
/// The rotations introduce T gates via synthesis, which exercises
/// the magic state factory during resource estimation.
operation AdderWithRotations() : Unit {
    use a = Qubit[8];
    use b = Qubit[8];
    for i in 0..7 {
        Ry(PI() / IntAsDouble(i + 2), a[i]);
    }
    RippleCarryCGIncByLE(a, b);
}
```

3. Convert the Q# program into an application model.

Python

```python
from qdk import code

app = QSharpApplication(code.AdderWithRotations)
```

4. Build a hardware architecture model. For example, use the default gate-based model.

Python

```
arch = GateBased(error_rate=1e-4, gate_time=100, measurement_time=500)
```

5. To run an estimate, pass the application model, architecture model, and custom ISA query to the `estimate` function. Set a 1% maximum error rate.

   Python

   ```
   result = estimate(app, arch, custom_query, max_error=0.01)

   result.plot(figsize=(8, 5))
   ```

6. To compare estimates for different sets of QEC and magic state factory models, build a list of estimates and plot the results. For example, build a query from the default parameters of the custom models and compare that with models that have different parameters.

   Python

   ```
   baseline = GenericQEC.q() * ScaledLitinskiFactory.q()
   high_cross = GenericQEC.q(crossing_prefactor=0.05) * ScaledLitinskiFactory.q()
   compact_factory = GenericQEC.q() * ScaledLitinskiFactory.q(space_factor=0.5)
   high_compact = GenericQEC.q(crossing_prefactor=0.05) *
   ScaledLitinskiFactory.q(space_factor=0.5)

   results = [
       estimate(app, arch, baseline, max_error=0.02, name="Baseline"),
       estimate(app, arch, high_cross, max_error=0.02, name="High crossing
   prefactor"),
       estimate(app, arch, compact_factory, max_error=0.02, name="Compact factory"),
       estimate(app, arch, high_compact, max_error=0.02, name="High crossing
   prefactor & Compact factory"),
   ]

   plot_estimates(results, figsize=(8, 5), runtime_unit="ms")
   ```

Related content

- How to build custom application models in the quantum resource estimator
- How to build hardware architecture models in the quantum resource estimator
- How to work with results from the quantum resource estimator

---

Last updated on 06/17/2026

How to work with results from the quantum resource estimator

The quantum resource estimator in the Microsoft Quantum Development Kit (QDK) produces optimal estimates of the run time and number of qubits required to run a quantum program on a certain type of hardware. For each estimate result, you can get statistics, the instruction set architecture (ISA), magic state factories, and other properties. You can also build and plot custom result properties.

This article shows how to access and analyze results from the resource estimator, and how to customize the results. The code examples are meant to be run in a Jupyter notebook in the order they appear in the article.

For instructions on how to install and use the resource estimator, see How to install and use the Microsoft Quantum resource estimator.

> ⚠ **Warning**
>
> The resource estimator in the QDK extension for VS Code will be deprecated soon. Use the `qdk.qre` Python module to perform resource estimation.

Run the resource estimator

To run the resource estimator and work with the estimation results, build an application model, a hardware architecture model, and an error correction model. Then, pass the models to the resource estimator and run an estimate for a given maximum error rate.

Build an application model from a Q# program

For this example, write a Q# program that performs eight parallel 8-bit ripple-carry additions, with a single-qubit rotation that precedes each addition. The rotations introduce T-gate decomposition into the estimation results.

To build an application from the Q# adder program, follow these steps in a Jupyter notebook:

1. Import the necessary objects.

Python

```
from qdk import qsharp
from qdk.qre.application import QSharpApplication
```

2. In a new cell, write the Q# program.

Q#

```
import Std.Arithmetic.*;
import Std.Math.*;
import Std.Convert.*;

operation EstimateAdder() : Unit {
    for _ in 1..8 {
        use a = Qubit[8];
        use b = Qubit[8];
        for i in 0..7 {
            Ry(PI() / IntAsDouble(i + 2), a[i]);
        }
        RippleCarryCGIncByLE(a, b);
    }
}
```

3. Build the application model.

Python

```
from qdk.code import EstimateAdder

app = QSharpApplication(EstimateAdder)
```

Build a hardware architecture model

Build a gate-based architecture model using the default `GateBased` class. The architecture model has a 10^{-4} maximum error rate, 100 ns gate times, and 500 ns measurement times.

Python

```
from qdk.qre.models import GateBased

arch = GateBased(error_rate=1e-4, gate_time=100, measurement_time=500)
```

Build an error correction model

For the error correction model, use the default surface QEC code and default round-based factory distillation model. Combine the QEC code and distillation model to create an ISA query that defines the set of architecture ISAs that the resource estimator explores.

Python

```
from qdk.qre.models import SurfaceCode, RoundBasedFactory

isa_q = SurfaceCode.q() * RoundBasedFactory.q()
```

Get basic estimation results

To run the resource estimator, pass a maximum error threshold, application model, architecture model, and ISA query to the `estimate` function. To increase the number of optimal estimation points, also pass an optional trace query that defines the set of application traces that resource estimator explores. For this example, use the `PSSPC` trace transform and the `LatticeSurgery` trace transform with a set of slow down factors.

Set a 1% maximum error rate and run the resource estimator. To print the basic results for each Pareto optimal point, use the `as_frame` method.

Python

```
from qdk.qre import estimate, PSSPC, LatticeSurgery

results = estimate(
    app,
    arch,
    isa_q,
    trace_query=PSSPC.q() * LatticeSurgery.q(slow_down_factor=[1.0, 1.5, 2.0, 3.0,
4.0]),
    max_error=0.01
)

print(f"Number of Pareto-optimal results: {len(results)}")
results.as_frame()
```

The `estimate` function returns an `EstimationTable` that contains the following resource estimates for each optimal point:

Expand table

Results parameter	Description
`qubits`	The total number of physical qubits required to run the program

Results parameter	Description
`runtime`	The amount of time it takes for the program to finish running
`error`	The overall error rate after error correction

The basic result from the resource estimator is an optimal set of points for run time and physical qubit requirements within a maximum error threshold. This information helps you choose the best parameters to run your program on real quantum hardware given your constraints on time and physical qubit availability.

Plot the Pareto-optimal results

To view the results as a plot of the number of qubits versus the run time, use the `plot` method. For example, plot the six Pareto-optimal points with the run time in milliseconds.

Python

```
results.plot(figsize=(8, 5), runtime_unit="ms")
```

Get detailed estimation results

The resource estimator produces detailed information about the estimation process and results. To access and display more detailed information about the estimation that you performed, see the following code examples.

Get estimation statistics

The estimation result has a `stats` attribute that contains the following information about the estimation process:

Expand table

Estimation statistic	How to access
Number of application traces	`stats.num_traces`
Number of ISAs	`stats.num_isas`
Number of estimation jobs	`stats.total_jobs`
Number of estimates below the error threshold	`stats.successful_estimates`

Estimation statistic	How to access
Number of Pareto-optimal results	`stats.pareto_results`

The following code displays this information for the resource estimation that you did:

Python

```
stats = results.stats

print(f"Traces explored:        {stats.num_traces}")
print(f"ISAs explored:          {stats.num_isas}")
print(f"Total estimation jobs:  {stats.total_jobs}")
print(f"Successful estimates:   {stats.successful_estimates}")
print(f"Pareto-optimal results: {stats.pareto_results}")
```

The total number of possible estimation jobs is the product of the number of traces and the number of ISAs. However, the actual number of estimation jobs is lower because the resource estimator uses a pruning strategy that keeps only the more reasonable combinations.

Get details for individual Pareto-optimal results

The estimation result has an `EstimationTableEntry` item for each Pareto-optimal point. The table entries contain detailed information about the individual optimal estimates.

To access the estimation table entry for an individual point, use the index for that point. For example, run the following code to get the estimation results for the third optimal point.

Python

```
entry = results[2]
```

Basic results

To access the basic estimation results for an individual optimal point, use the `qubits`, `runtime`, and `error` attributes. For example, the following code displays the basic estimation results for the third optimal point.

Python

```
print(f"Total physical qubits: {entry.qubits}")
print(f"Runtime:               {entry.runtime} ns")
print(f"Error:                 {entry.error:.6f}")
```

Detailed properties

Each estimation table entry has a `properties` attribute that contains detailed information on factories and the distribution of qubits. To view all the properties, run the following code:

Python

```
from qdk.qre import property_name

props = {property_name(k): v for k, v in entry.properties.items()}

for name, value in props.items():
    print(f"  {name}: {value}")
```

You can also access each property individually. For example, get only the number of T gates per rotation.

Python

```
from qdk.qre.property_keys import NUM_TS_PER_ROTATION

print(f"Number of T gates per rotation:  {entry.properties.get(NUM_TS_PER_ROTATION,
'N/A')}")
```

Display qubit partition and factory information

The resource estimator provides tools that let you display more detailed information in the results table. For example, the `add_qubit_partition_column` method adds columns to the results table that show how the physical qubits are distributed. The `add_factory_summary_column` shows the number of T factories that each optimal point requires.

Python

```
results.add_qubit_partition_column()
results.add_factory_summary_column()

results.as_frame()
```

Analyze the ISA of an estimate

Each optimal estimation result has an instruction source graph that describes how logical instructions are implemented in terms of lower-level operations. To access the instruction source graph for a result, use the `source` attribute of the `EstimationTableEntry` for that result.

For example, print the instruction source graph for the third result in the `entry` variable that you created.

Python

```
print("Instruction source graph for the third result:\n")
print(entry.source)
```

The instruction source graph behaves like a Python dictionary where the instruction IDs are the dictionary keys. For example, run the following code to get information about T gate instructions.

Python

```
from qdk.qre import instruction_name
from qdk.qre.instruction_ids import T

if T in entry.source:
    t_node = entry.source[T]
    t_inst = t_node.instruction
    print("T instruction:")
    print(f"  ID:          {instruction_name(t_inst.id)}")
    print(f"  Encoding:    {'LOGICAL' if t_inst.encoding == 1 else 'PHYSICAL'}")
    print(f"  Arity:       {t_inst.arity}")
    print(f"  Space:       {t_inst.space()} physical qubits")
    print(f"  Time:        {t_inst.time()} ns")
    print(f"  Error rate: {t_inst.error_rate():.2e}")
    if t_node.transform is not None:
        print(f"  Transform:  {t_node.transform}")
```

Get factory information

Each estimation result entry has a `factories` attribute, which is a dictionary that contains the following information about the magic-state factories:

- Type of factory
- Number of copies of the factory
- Number of times the factory runs
- Number of magic states that the factory creates
- Error rate after distillation

For example, run the following code to view all the factory information for the third result:

Python

```
print("Factories in the third result:")
```

```
for inst_id, factory in entry.factories.items():
    print(f"\n  {instruction_name(inst_id)} factory:")
    print(f"    Copies:     {factory.copies}")
    print(f"    Runs:       {factory.runs}")
    print(f"    States:     {factory.states}")
    print(f"    Error rate: {factory.error_rate:.2e}")
```

Customize the resource estimator results

You can add custom columns to the `EstimationTable` output from the resource estimator using the `add_property_column` and `add_column` functions.

The `add_property_column` takes a property key as input and creates a column for that property in the results table. For example, use the `NUM_TS_PER_ROTATION` property key to add a column for this property.

Python

```
results.add_property_column(NUM_TS_PER_ROTATION)

results.as_frame()
```

The `add_column` function takes a column name and a function as input. The input function gets values from the `EstimationTableEntry` attributes for each result to populate the new column. For example, create columns for the T gate error rate, logical operation error rate, and number of T states.

Python

```
from qdk.qre.instruction_ids import LATTICE_SURGERY

results.add_column(
    "t_error_rate",
    lambda r: r.source[T].instruction.error_rate() if T in r.source else None,
)
results.add_column(
    "logical_error_rate",
    lambda r: r.source[LATTICE_SURGERY].instruction.error_rate(1) if LATTICE_SURGERY
in r.source else None,
)
results.add_column(
    "t_states",
    lambda r: r.factories[T].states if T in r.factories else 0,
)

results.as_frame()
```

Last updated on 06/17/2026

Get started with QDK for error correction

The Microsoft Quantum Development Kit (QDK) includes Python tools to develop and test quantum error correction (QEC) protocols. Use the `qdk.ec` module to analyze error correction codes, build executable protocols, validate protocol declarations, and study circuit faults. Use `qdk.simulation` to run logical programs through those protocols.

Understand the QDK error correction tools

The QDK error correction workflow uses these packages and modules:

Expand table

Component	Use it to
`qodec`	Define and save codes, instruction sets, gadgets, layers, and complete QEC protocols.
`qdk.ec`	Analyze codes and gadgets, build baseline qodecs, derive equations, and audit protocol declarations.
`qdk.simulation`	Run Quantum Intermediate Representation (QIR) programs through a qodec and apply a decoder.
`qdk.simulation.decoders`	Select the built-in syndrome decoder, track frames, or connect to the optional deq decoder.
`qdk.qre`	Estimate the space, time, and error costs of QEC models for fault-tolerant resource estimation.

A *qodec* is a formal description of an executable QEC protocol. It connects logical instructions to physical circuits through one or more layers. The `qodec` package defines this data model and its YAML representation. The `qdk.ec` module derives facts from the model and checks whether its declarations match its circuits.

For more information, see the qodec overview.

The `qdk.qre` module has a different purpose. It models the resource cost of an error correction scheme. It doesn't define, audit, or run qodec gadgets. For more information about quantum resource estimation in the QDK, see What is the Microsoft Quantum resource estimator?

Understand the qodec protocol components

The following components define a qodec protocol and support its execution.

Expand table

Component	Description
Code	A static set of stabilizers and logical operators that defines a code space.
Instruction set	The logical or physical operations that a qodec layer supports.
Gadget	A circuit that implements an instruction through operations in the next lower layer.
Layer	A mapping from one instruction set to a lower-level instruction set.
Encoding	The connection between logical blocks and circuit qubits at a gadget boundary.
Check	A parity equation that must equal zero when the declared relation holds.
Readout	An equation that maps circuit results and frame signs to a logical outcome or flag.
Flag	A named readout that reports whether an instruction met its acceptance condition.
Decoder	A component that uses syndrome evidence to infer corrections and logical results.

A qodec can contain several layers. The top layer can define logical operations over encoded blocks. The gadgets in the top layer can call instructions in the next lower layer. The bottom layer contains the physical operations that the simulator runs.

Example protocol workflow

A typical workflow has six steps. The following steps use functions and classes from the `qdk.ec` module.

1. **Define a code.** Create a `qodec.Code` with stabilizers and logical Pauli operators.
2. **Analyze the code.** Use the `CodeProfile` class to inspect logical operators, calculate syndromes, and compute code distance.
3. **Build or load a qodec.** Use the `build_qodec` function to create a baseline protocol, or load an existing qodec from YAML.
4. **Complete and validate the protocol.** Use the `filled` function to derive supported check and readout equations. Use the `audit` function to find inconsistent or incomplete declarations.
5. **Analyze circuit faults.** Use the `GadgetProfile` class to inspect logical actions, fault effects, and gadget distance.

6. **Run logical programs.** Compile Q# or OpenQASM code to QIR. Pass the QIR and qodec to `qdk.simulation.run_qir`, then select a decoder and noise model.

The following code shows the main objects in this workflow:

Python

```python
import qodec
from qdk import ec
from qdk.simulation import run_qir

code = qodec.Code(
    "C4",
    stabilizers=["X_0 X_1 X_2 X_3", "Z_0 Z_1 Z_2 Z_3"],
    x=["X_0 X_1", "X_0 X_2"],
    z=["Z_0 Z_2", "Z_0 Z_1"],
)

code_profile = ec.CodeProfile(code)
print("Code distance:", code_profile.distance())

protocol = ec.build_qodec(code, strategy="bare-css/v1", strict=False)
print(ec.audit(protocol))

# Compile a logical program to QIR before you call run_qir.
# results = run_qir(qir, qodec=protocol, shots=100)
```

The `bare-css/v1` strategy in this example creates a simple baseline protocol without flag qubits. It helps you inspect the workflow, but it doesn't guarantee fault-tolerant gadgets. Use gadget distance analysis before you choose a protocol implementation.

Separate protocol validation from fault analysis

Protocol audits and distance analysis assess different aspects of a QEC protocol.

- Protocol audits check whether the declared protocol matches its circuits and equations when no faults occur.
- Distance analysis determines the minimum number of allowed errors or circuit faults that can cause an undetected logical failure.

Use the `audit` function to check noiseless consistency. The audit checks code algebra, instruction declarations, gadget actions, checks, readouts, flags, frame relations, and references between qodec objects.

An audit that reports no issues doesn't guarantee fault tolerance. A circuit can implement the correct logical action without faults and still fail after one physical fault. Use the

`CodeProfile.distance` method to analyze the code and the `GadgetProfile.distance` method to analyze each circuit implementation.

Before you use a protocol for noisy simulations, run a protocol audit and analyze code and gadget distances.

Choose a construction strategy

The `build_qodec` function supports two baseline CSS construction strategies.

Expand table

Strategy	Description
`flagged-css/v1`	Uses the flagged CSS construction and selects the flag count from the code distance. This strategy is the default.
`bare-css/v1`	Uses no flag qubits and skips the distance calculation that selects a flag count. Use it for inspection and early development.

Both strategies try to create these logical instructions:

- `prepare_z_all`
- `prepare_x_all`
- `syndrome`
- `measure_z_all`
- `measure_x_all`
- `h_all`
- `cx_all`

Not every code supports every candidate. With the default setting `strict=True`, `build_qodec` raises an error when it can't build and verify an instruction. With `strict=False`, `build_qodec` keeps the supported instructions and records each skipped instruction in the qodec metadata.

Choose your next task

- To prepare a Python environment, see How to install QDK for error correction.
- To define a code and create a protocol, see Develop and validate a QEC protocol.
- To compile, simulate, and decode logical programs, see Run and decode logical programs.

For a complete notebook that follows the full workflow, see Develop and test a QEC protocol in the QDK repository.

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/22/2026

How to install QDK for error correction

Install the `qdk.ec` Python module from the Microsoft Quantum Development Kit (QDK) to develop and test quantum error correction (QEC) codes and protocols. The module uses the `qodec` package for protocol models and persistence.

Prerequisites

Install these tools before you start:

- Python 3.11 or later
- `pip` or another Python package manager

The base `qdk` package supports Python 3.10 or later. The `qdk.ec` module requires Python 3.11 or later because of its `qodec` dependency.

(Optional) Create a virtual environment

Use a virtual environment to keep the `qdk.ec` dependencies separate from other Python projects.

1. Create the virtual environment, for example `.venv`.

   Bash

   ```
   python -m venv .venv
   ```

2. Activate the environment.

   On Windows:

   PowerShell

   ```
   .venv\Scripts\Activate.ps1
   ```

   On Linux or macOS:

   Bash

   ```
   source .venv/bin/activate
   ```

3. Update pip in the active environment.

```Bash
python -m pip install --upgrade pip
```

Install QDK for error correction

Install the `qdk` Python package with the `ec` extra.

```Bash
python -m pip install --upgrade "qdk[ec]"
```

To use Jupyter Notebook and QDK widgets, install the `jupyter` extra too.

```Bash
python -m pip install --upgrade "qdk[ec,jupyter]" ipykernel
```

The `ec` extra installs `qodec`, the Stim parser on supported platforms, and the algebra and solver packages that `qdk.ec` uses. It also includes HiGHS for distance searches.

> **Note**
>
> The `qdk.ec` module uses the separate Stim Python package through qodec. It doesn't use the `qdk.stim` module.

Install on Linux Arm64 or Windows Arm64

Stim doesn't publish wheels for Linux Arm64 or Windows Arm64. On those platforms, `qdk[ec]` installs qodec without its Stim parser.

You can load and save YAML qodecs and use a custom circuit parser. Features that parse `format="stim"` circuits don't work on these platforms. This limitation includes `qdk.ec.build_qodec`.

To enable the parser, install Stim separately. Your system needs a C++20 compiler to install Stim.

```Bash
```

```
python -m pip install --upgrade stim
```

Install optional distance and decoder packages

The default distance solver uses HiGHS, which the `qdk.ec` module includes. To select the optional MWPF solver, install `mwpf` separately.

Bash

```
python -m pip install mwpf
```

To use the deq circuit-level decoder with `qdk.simulation.run_qir`, install `deq` and `deq-runtime`.

Bash

```
python -m pip install deq deq-runtime
```

The QDK tests the deq adapter with version 0.5.7 of `deq` and `deq-runtime`.

> **Note**
>
> The `deq-runtime` package doesn't publish Windows Arm64 wheels. The deq adapter also requires Stim, which doesn't publish Linux Arm64 or Windows Arm64 wheels.

Verify the installation

Create a Python file or Jupyter Notebook cell with the following code. The example defines the four-qubit C4 code and checks its distance.

Python

```python
import qodec
from qdk import ec

c4 = qodec.Code(
    "C4",
    stabilizers=["X_0 X_1 X_2 X_3", "Z_0 Z_1 Z_2 Z_3"],
    x=["X_0 X_1", "X_0 X_2"],
    z=["Z_0 Z_2", "Z_0 Z_1"],
)

distance = ec.CodeProfile(c4).distance()
```

```
print("Code distance:", distance)
assert distance.value == 2
```

The program prints this result:

Output

```
Code distance: 2
```

If the import and distance search succeed, the core `qdk.ec` dependencies are ready.

Verify Stim support

Run this check on a platform that supports the Stim parser.

Python

```
from qdk import ec

protocol = ec.build_qodec(c4, strategy="bare-css/v1", strict=False)
print(ec.audit(protocol))
```

A correctly generated protocol prints an audit summary. A protocol that passes all checks prints:

Output

```
audit: ok
```

If `build_qodec` reports that no parser exists for the `stim` format, confirm that you installed `qdk[ec]` in the active environment. On Linux Arm64 or Windows Arm64, install Stim from source and restart Python or the Jupyter kernel.

Continue with protocol development

To define a code, build a qodec, and validate its gadgets, see Develop and validate a QEC protocol.

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 10/22/2026

Develop and validate a QEC protocol

Use `qodec` and the `qdk.ec` module from the Microsoft Quantum Development Kit (QDK) to define a quantum error correction (QEC) code and turn it into an executable protocol. In this article, you define an error code, build a baseline qodec, validate its declarations, calculate gadget distance, and save the protocol.

The examples in this article use the $[[4, 2, 2]]$ C4 code, which stores two logical qubits in four physical qubits and detects any single physical Pauli error.

Prerequisites

Complete these tasks before you start:

- Install QDK for error correction.
- Use an environment with Python 3.11 or later.
- Confirm that the qodec Stim parser is available. On Linux Arm64 or Windows Arm64, install Stim from source, and then restart Python or the Jupyter kernel before you call `build_qodec`.

Define the code

Create a `qodec.Code` object with the code stabilizers and one logical X and Z representative for each logical qubit.

Python

```python
import qodec

c4 = qodec.Code(
    "C4",
    stabilizers=["X_0 X_1 X_2 X_3", "Z_0 Z_1 Z_2 Z_3"],
    x=["X_0 X_1", "X_0 X_2"],
    z=["Z_0 Z_2", "Z_0 Z_1"],
)
```

The `qodec` package uses sparse notation for Pauli strings. Each term has a Pauli letter and a physical qubit label. For example, `X_0 X_2` applies X to physical qubits 0 and 2.

The code definition is a static mathematical object. It doesn't include circuits that prepare, preserve, or measure encoded states.

Analyze the code

To inspect a snapshot of the code operators, create a `CodeProfile`.

Python

```python
from qdk import ec

profile = ec.CodeProfile(c4)

print("Physical support:", sorted(profile.support))
print("Physical qubits:", profile.length)
print("Logical qubits:", profile.logical_qubit_count)
print("Stabilizers:", profile.stabilizers)
print("Logical X operators:", profile.x)
print("Logical Z operators:", profile.z)
```

The profile doesn't change if you later change the original `qodec.Code` object.

Calculate the code distance

To find the minimum number of allowed physical errors that cause a nontrivial logical effect with zero syndrome, call `distance`.

Python

```python
distance = profile.distance()
print("Code distance:", distance)

error = distance.witness.product
syndrome = profile.syndrome_of(error)
logical_effect = profile.logical_effect_of(error)

print("Witness:", error)
print("Syndrome:", sorted(syndrome))
print("Logical effect:", logical_effect)

assert distance.value == 2
assert profile.is_logical(error)
```

By default, the search assigns cost one to each single-qubit X, Y, or Z error. The returned `Distance` has certified bounds and a witness for any finite upper bound.

When a complete exact search costs too much, use `distance_bounds`.

Python

```
bounds = profile.distance_bounds(upper_bound=4)
print("Distance bounds:", bounds)
print("Lower bound:", bounds.lower_bound)
print("Upper bound:", bounds.upper_bound)
```

The default solver is HiGHS. To use a different solver, set `solver="enumeration"`, or install MWPF and set `solver="mwpf"`.

Build a baseline qodec

A *qodec* describes how to run encoded instructions. It has instruction sets, codes, layers, gadgets, circuits, checks, and readout equations.

Use `build_qodec` to create a two-layer protocol from the C4 code.

Python

```
protocol = ec.build_qodec(
    c4,
    strategy="bare-css/v1",
    strict=False,
)
```

This example uses `bare-css/v1` to let you inspect a simple protocol without flag qubits.

> **⊗ Caution**
>
> The `bare-css/v1` strategy doesn't promise fault-tolerant circuits. Calculate each gadget's distance before you use the protocol for noisy experiments.

The builder tries to create these logical instructions:

- `prepare_z_all`
- `prepare_x_all`
- `syndrome`
- `measure_z_all`
- `measure_x_all`
- `h_all`
- `cx_all`

The `_all` suffix means that an instruction acts on every logical qubit in a block. For example, `cx_all` applies CNOT between corresponding logical qubits in two blocks.

Not every code supports every candidate instruction. With `strict=True`, which is the default, the builder raises an error when it cannot build and verify a candidate. With `strict=False`, the builder keeps the supported instructions and records omissions in `protocol.metadata["qdk.ec"]["build"]["omitted"]`.

Inspect the logical layer and its gadgets.

Python

```
logical_layer = protocol.layers[0]

print("Instructions:")
for mnemonic in logical_layer.instruction_set.instructions:
    print(" -", mnemonic)

print("Gadgets:")
for mnemonic in logical_layer.gadgets:
    print(" -", mnemonic)
```

The second layer contains the physical Stim instruction set that the generated circuits use.

Inspect a gadget

A logical instruction declares a contract. Its gadget gives the circuit that implements the contract.

Inspect the `prepare_z_all` gadget.

Python

```
gadget = logical_layer.gadgets["prepare_z_all"]
gadget_profile = ec.GadgetProfile(gadget)

print("Declared objective:")
print(gadget_profile.objective)

print("Computed action:")
print(gadget_profile.action)
```

Compare the action computed from the circuit with the declared instruction objective.

Python

```
objective = gadget_profile.objective
assert objective is not None

if gadget_profile.action.is_equivalent_to(objective):
    print("The gadget implements its declared instruction.")
```

```
else:
    print(objective.why_not_equivalent_to(gadget_profile.action))
```

A `GadgetProfile` takes a snapshot of its input, so it doesn't reflect later changes. After you edit a gadget, create a new profile.

Audit the protocol

Use `audit` to check the whole qodec.

Python

```
report = ec.audit(protocol)
print(report)
```

If the audit finds no errors or warnings, it prints:

Output

```
audit: ok
```

The audit checks these areas:

- Code algebra
- Instruction and layer structure
- Circuit calls and parameter bindings
- Gadget actions
- Checks and readout equations
- Flag equations
- Input and output frame relations
- Reference bounds and protocol completeness

To act on specific severities, use the report collections.

Python

```
if report.errors:
    for diagnostic in report.errors:
        print(diagnostic)
    raise ValueError("The qodec has audit errors.")

if report.warnings:
```

```
    for diagnostic in report.warnings:
        print(diagnostic)
```

The `report.ok` value means that the report has no errors. Warnings don't change that property.

Derive missing equations

Use `filled` when a draft gadget or qodec omits checks or logical readout equations that exact simulation can derive.

The following example creates an incomplete copy of the protocol.

Python

```
draft = qodec.Qodec.loads(protocol.dumps())
draft_gadgets = draft.layers[0].gadgets

draft_gadgets["measure_x_all"].readouts = []
draft_gadgets["cx_all"].checks = []

print(ec.audit(draft))
```

Fill in the supported equations, and then audit the result.

Python

```
completed = ec.filled(draft)
print(ec.audit(completed))
```

The `filled` function returns a new gadget or qodec and doesn't change its input. It derives checks, observable readout equations, and frame relations. It preserves flag bindings because noiseless simulation can't identify which faults a flag should detect.

If the original protocol already has verified direct equations, `filled` keeps them when possible.

Analyze gadget faults

An audit with no errors proves noiseless consistency. It doesn't prove fault tolerance.

Call `GadgetProfile.distance` to find the minimum number of allowed circuit faults that causes a logical failure while all declared checks and flags remain zero. The combined fault must also leave each output inside its code space.

Python

```
protocol = completed

for mnemonic, gadget in sorted(protocol.layers[0].gadgets.items()):
    result = ec.GadgetProfile(gadget).distance()
    witness = result.witness if result.upper_bound is not None else "none"
    print(f"{mnemonic}: distance={result}, witness={witness}")
```

The default fault model includes these events after each circuit call:

- A Pauli error on the qubits that the call touches
- A flip of a readout bit that the call produces
- A correlated combination of those errors

Each event at one call costs one, even when it has a correlated multiqubit error.

A code can have distance two while one of its gadgets has distance one. For example, a single circuit fault can spread through an ancilla and create an undetected logical error. Add checks or flag circuits, then create a new `GadgetProfile` and calculate the distance again.

Inspect a specific fault

Use `FaultEvent.after` to define a fault after a zero-based circuit call index.

Python

```
gadget = protocol.layers[0].gadgets["syndrome"]
profile = ec.GadgetProfile(gadget)

fault = ec.FaultEvent.after(0, ec.Pauli("X_0"))
(effect,) = profile.effects_of([fault])

print("Fault:", fault)
print("Changed quantities:", effect)
```

A `FaultEffect` can include changed checks, logical readouts, and output frame signs. Use explicit fault sets when the default circuit-level model doesn't match your hardware assumptions.

Python

```
result = profile.distance(faults=[fault])
print(result)
```

Exact gadget searches can take a long time because the default fault set grows exponentially with each call's support. Use `distance_bounds`, an explicit fault set, or an `upper_bound` when you need to limit the search.

Save the protocol

Audit the final qodec before you save it.

Python

```
report = ec.audit(protocol)
if report.errors or report.warnings:
    raise ValueError(str(report))
```

Save the protocol as a single YAML bundle, and then load it again to check that it matches the original.

Python

```
from pathlib import Path

output_directory = Path("qodec-output")
output_directory.mkdir(exist_ok=True)

saved_path = protocol.save(str(output_directory), single_file=True)
loaded = qodec.Qodec.load(saved_path)

assert loaded == protocol
print("Saved qodec:", saved_path)
```

A single-file bundle stores the manifest, code definitions, instruction sets, gadgets, and circuit sources in one YAML file, which makes it easy to share and reload.

Run the protocol

After the qodec passes its audit and meets your distance requirements, compile a logical program to Quantum Intermediate Representation (QIR) and run it through the protocol. See Run and decode logical programs.

For a complete example that adds flag circuits and compares logical error rates, see the QEC protocol walkthrough on the QDK GitHub repository.

Note: The author created this article with assistance from AI. Learn more

Last updated on 10/22/2026

Run and decode logical programs

Use the Microsoft Quantum Development Kit (QDK) to compile a logical Q# or OpenQASM program to Quantum Intermediate Representation (QIR), and then use `qdk.simulation.run_qir` to run the program through a qodec. Set up a noise model, and then choose the built-in syndrome decoder or the optional deq circuit-level decoder.

Prerequisites

Complete these tasks before you start:

- Install QDK for error correction.
- Develop and validate a QEC protocol or get a complete qodec from another source.
- Audit the qodec by using `qdk.ec.audit`.

The Q# examples use a qodec named `protocol` that has instructions for the C4 code. To create one, see Develop and validate a QEC protocol.

Python

```python
import qodec
from qdk import ec

c4 = qodec.Code(
    "C4",
    stabilizers=["X_0 X_1 X_2 X_3", "Z_0 Z_1 Z_2 Z_3"],
    x=["X_0 X_1", "X_0 X_2"],
    z=["Z_0 Z_2", "Z_0 Z_1"],
)

protocol = ec.build_qodec(
    c4,
    strategy="bare-css/v1",
    strict=False,
)

report = ec.audit(protocol)
if report.errors or report.warnings:
    raise ValueError(str(report))
```

Audit the qodec before you run it. The runtime trusts the qodec's checks, readouts, flags, and frame equations, so invalid declarations can cause incorrect logical results or decoder failures.

This workflow uses `qdk.simulation.run_qir` with a qodec. The qodec tooling uses the separate Stim Python package to parse Stim-format gadget circuits. It doesn't call `qdk.stim.run`. On Linux Arm64 or Windows Arm64, install Stim from source and restart Python or the Jupyter kernel before you run a qodec that has Stim-format circuits.

Match QIR calls to qodec instructions

Match each quantum call in the QIR program to an instruction in the qodec's top instruction set. The instruction mnemonic must exactly match the QIR callee name.

The following table shows example mappings:

Expand table

Source call	Required qodec mnemonic
Q# `X(q)` or OpenQASM `x q;`	`__quantum__qis__x__body`
Q# `H(q)` or OpenQASM `h q;`	`__quantum__qis__h__body`
Q# `CNOT(a, b)` or OpenQASM `cx a, b;`	`__quantum__qis__cx__body`
Q# `Adjoint S(q)` or OpenQASM `sdg q;`	`__quantum__qis__s__adj`
Q# `M(q)` or OpenQASM `c = measure q;`	`__quantum__qis__m__body`
Q# intrinsic operation `Foo`	`Foo`
OpenQASM intrinsic gate or `def` named `foo`	`foo`

QIR arguments bind to qodec instruction elements in this order:

1. Qubit pointer arguments bind to block operands.
2. Scalar arguments bind to instruction parameters.
3. Result pointer arguments receive instruction outcomes and any returned flags.

The compiler raises an error if a quantum call doesn't match an instruction in the top instruction set.

Declare logical instructions in Q#

A qodec built from the C4 code exposes custom instructions like `prepare_z_all`, `measure_z_all`, `syndrome`, and `cx_all`. Declare Q# intrinsic operations with the same names.

Initialize Q# with the adaptive target profile because custom measurement operations require it.

Python

```
from qdk import TargetProfile, qsharp

qsharp.init(target_profile=TargetProfile.Adaptive_RI)
```

Declare the C4 instructions as Q# operations.

Python

```
qsharp.eval("""
@Measurement()
operation prepare_z_all(block : Qubit) : Result { body intrinsic; }

@Measurement()
operation prepare_x_all(block : Qubit) : Result { body intrinsic; }

@Measurement()
operation measure_z_all(block : Qubit) : (Result, Result) { body intrinsic; }

@Measurement()
operation measure_x_all(block : Qubit) : (Result, Result) { body intrinsic; }

operation cx_all(control : Qubit, target : Qubit) : Unit { body intrinsic; }
operation syndrome(block : Qubit) : Unit { body intrinsic; }
""")
```

In this program, each Q# `Qubit` represents one encoded block. The C4 block has two logical qubits, so each measurement operation returns two logical results. Each preparation operation returns one rejection flag.

Compile a logical Q# experiment

Define a state preparation and measurement (SPAM) experiment.

Python

```
qsharp.eval("""
operation TestSpamZ() : (Result[], Result[]) {
    use block = Qubit();
    let reject = prepare_z_all(block);
    let (left, right) = measure_z_all(block);
    ([reject], [left, right])
}
""")
```

Compile the `TestSpamZ` operation to QIR.

Python

```
qir = str(qsharp.compile("TestSpamZ()"))
```

The QIR keeps the calls to `prepare_z_all` and `measure_z_all`. When the program runs, the qodec runtime matches these names against the top instruction set and replaces each logical call with its gadget.

Run with the built-in decoder

Call `run_qir` and pass the qodec. If `decoder` is `None`, the runtime uses the built-in minimum weight Pauli decoder.

Python

```
from qdk.simulation import run_qir

results = run_qir(
    qir,
    qodec=protocol,
    type="stabilizer",
    shots=100,
    seed=42,
)

print(results[:5])
```

To run a qodec, use these simulator options:

Expand table

`type` value	Behavior
`None` or `"stabilizer"`	Use the stabilizer backend
`"cpu"`	Use the state vector backend for supported programs
`"gpu"`	Not supported with a qodec

The runtime prepares each program qubit before its first use unless the first instruction creates its block. A preparation instruction has output blocks, no input blocks, and an action that stabilizes logical `Z`.

Single-qubit Pauli decoder corrections and qodec frame updates use a noiseless Pauli frame, not noisy physical gates.

Configure simulator noise

Define noise on simulator operations, not qodec instruction names. The runtime translates each bottom-layer instruction into one or more simulator operations, so noise settings apply to those operations.

The C4 walkthrough uses these mappings:

Expand table

Bottom-layer instruction	Simulator operation	`NoiseConfig` field
`R`	Prepare or reset to Z+	`mresetz`
`H`	Hadamard	`h`
`CX`	Controlled X	`cx`
`M`	Z-basis measurement	`mresetz`

Create a simple 1% noise model:

Python

```python
from qdk.simulation import NoiseConfig

noise = NoiseConfig()
for gate in (noise.h, noise.cx):
    gate.set_depolarizing(0.01)
noise.mresetz.x = 0.01
```

Run the experiment with noise:

Python

```python
results = run_qir(
    qir,
    qodec=protocol,
    noise=noise,
    type="stabilizer",
    shots=1000,
    seed=42,
```

```
    on_shot_failure="discard",
)
```

Noise on `mresetz` acts on the qubit after measurement or preparation. It doesn't flip a measurement bit that the simulator already recorded. To study recorded bit flips, use qodec readout fault models and gadget analysis.

Control shot failures

Use `on_shot_failure` to choose how the runtime responds when a qodec flag, decoder, or readout doesn't produce an accepted shot.

Expand table

Value	Behavior
`"discard"`	Omit failed shots from the returned list. This value is the default.
`"raise"`	Stop when the first shot fails.
`"retry"`	Restart the failed shot, up to `max_retries` additional attempts.

With the `"discard"` policy, the `shots` argument counts attempted shots, so the returned list can have fewer items than `shots`.

A flag triggers the shot policy only when the logical program doesn't receive that flag. If a Q# `@Measurement()` operation returns an extra `Result` for the flag, the program receives the flag and decides whether to reject the shot.

In the `TestSpamZ` example, the program returns the preparation flag in the first array. So the `on_shot_failure="discard"` setting doesn't remove rows where the returned flag is `One`.

Select a decoder

To choose how errors are decoded, use the `decoder` parameter to pass a decoder preparation callable for each encoded layer.

The QDK has the following decoders:

Expand table

Decoder	Use it to
Default syndrome decoder	Infer minimum-weight Pauli corrections from syndrome evidence. Use `decoder=None`.
Frame decoder	Track noiseless frames without inferring physical faults. Use `prepare_frame_decoder`.
deq decoder	Decode circuit-level fault models with deq. Use `prepare_deq_decoder`.

Use the default deq configuration

Install `deq` and `deq-runtime`, then pass `prepare_deq_decoder`.

Python

```python
from qdk.simulation.decoders import prepare_deq_decoder

results = run_qir(
    qir,
    qodec=protocol,
    decoder=prepare_deq_decoder,
    noise=noise,
    type="stabilizer",
    shots=1000,
    seed=42,
    on_shot_failure="discard",
)
```

By default, the deq adapter uses relay belief propagation and a window coordinator. To change these settings, use a runtime factory.

Configure a monolithic deq runtime

To choose deq settings directly, pass a runtime factory to `prepare_deq_decoder`.

Python

```python
from functools import partial
from deq.runtime import Runtime
from qdk.simulation import decoders


def make_runtime(seed: int) -> Runtime:
    return Runtime(
        decoder="black-box-relay-bp",
        decoder_config={"parallel": 1, "seed": seed},
        coordinator="monolithic",
```

```
        controller="jit",
    )


results = run_qir(
    qir,
    qodec=protocol,
    decoder=partial(
        decoders.prepare_deq_decoder,
        runtime_factory=make_runtime,
    ),
    noise=noise,
    type="stabilizer",
    shots=1000,
    seed=42,
    on_shot_failure="discard",
)
```

Return a new `Runtime` from each factory call. The QDK starts and closes each runtime.

> **Note**
>
> The deq adapter requires:
>
> - One encoded layer
> - A measurement-independent Clifford execution trace
> - The stabilizer backend
> - A noise model without qubit loss
>
> The deq adapter doesn't support the retry policy. Unsupported programs raise an error instead of falling back to another decoder.

Reject low-confidence deq results

The deq adapter returns a score for each logical readout. Lower scores indicate a stronger preference for the chosen correction. The scores aren't calibrated failure probabilities.

To reject low-confidence results, pass `max_readout_score` when you prepare the decoder, like in the following example:

Python

```
from functools import partial
from qdk.simulation import decoders
```

```
scored_decoder = partial(
    decoders.prepare_deq_decoder,
    max_readout_score=0.2,
)

results = run_qir(
    qir,
    qodec=protocol,
    decoder=scored_decoder,
    noise=noise,
    type="stabilizer",
    shots=1000,
    on_shot_failure="discard",
)
```

The runtime accepts a shot only when every returned logical readout has a score at or below the threshold. The threshold doesn't apply to raw flags or unused intermediate readouts.

Score-based rejection can lower the logical error rate of accepted shots. But it also reduces the number of accepted shots and adds decoder work.

Compile OpenQASM programs

Standard OpenQASM operations use the same QIR mnemonic rules as standard Q# operations.

Python

```
from qdk import TargetProfile, openqasm

source = """
OPENQASM 3.0;
include "stdgates.inc";
qubit q;
bit c;
x q;
c = measure q;
"""

qir = openqasm.compile(
    source,
    target_profile=TargetProfile.Adaptive_RI,
)

print(qir)
```

To run this program through a qodec, make sure the qodec's top instruction set has `__quantum__qis__x__body` and `__quantum__qis__m__body`, with compatible block operands and one measurement outcome.

An OpenQASM intrinsic gate or subroutine named `foo` uses the mnemonic `foo`. Qubit arguments bind to block operands. The `float`, `int`, and `bool` arguments bind to instruction parameters in declaration order.

OpenQASM has these limitations for qodec calls:

- Custom intrinsics can't return a QIR `Result`. To get one measurement result, use the standard `measure` statement.
- OpenQASM programs can't run custom measurements, receive several outcomes from one call, or receive returned flags. For these cases, use Q# `@Measurement()` operations.
- The runtime controls any flags that the OpenQASM program can't receive, and these flags can reject a shot.

For detailed mapping examples, see the following QDK notebooks:

- Call qodec instructions from Q#
- Call qodec instructions from OpenQASM

Review requirements and limitations

Review these requirements, limitations, and result differences before you run a large experiment.

Requirements:

- Every QIR quantum call must resolve to a top-layer qodec instruction.
- Audit the protocol before you run it. The qodec runtime trusts the equations that you write and doesn't check them.

Limitations:

- GPU simulation doesn't support qodec execution.
- The default gadget fault set can grow exponentially, so exact distance searches can take a long time.
- The deq and Stim packages have platform limitations. For more information, see Install QDK for error correction.

Result differences:

- Exact output counts can vary by decoder.
- Discard and retry policies can change the returned result distribution.

For a complete Q# experiment with noise sweeps, deq decoding, and confidence-based rejection, see the QEC protocol walkthrough.

Note: The author created this article with assistance from AI. Learn more

---

Last updated on 10/22/2026

QDK Learning overview

The Microsoft Quantum Development Kit (QDK) extension for Visual Studio Code (VS Code) includes QDK Learning, a collection of self-paced courses for quantum computing and quantum development. Each course combines explanations with interactive programming activities that you complete in VS Code.

Use QDK Learning to:

- Choose a course based on your interests and experience.
- Work through course units and activities at your own pace.
- Track and save your progress for each course.
- Ask the QDK Learning agent in GitHub Copilot for hints, explanations, and guidance.
- Keep your course code and notebooks.

Available courses

QDK Learning includes these courses.

Expand table

Course	What you learn	Course format
Quantum Katas	Quantum computing fundamentals and quantum programming with Q#	Guided lessons and Q# coding exercises
Ground-State Molecular Energies with QPE	How to estimate the ground-state energy of a molecule from classical chemistry methods and quantum phase estimation (QPE)	Python activities in Jupyter notebooks that use QDK/Chemistry and Q#

Start with the Quantum Katas if you're new to quantum computing or the Q# language. Choose the quantum chemistry course if you already know the basics of quantum chemistry and quantum computing.

Prerequisites

To use QDK Learning, you need:

- The latest version of VS Code.
- The QDK extension for VS Code.

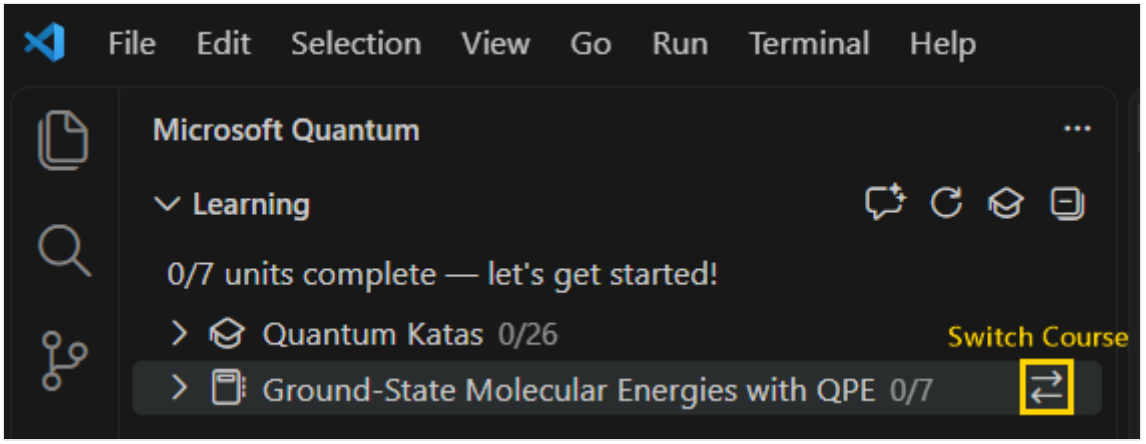
- A folder where QDK Learning can store your course files and progress.

To use AI-assisted learning, sign in to GitHub Copilot in VS Code. Check the prerequisites for each course for additional software requirements.

Start QDK Learning

To start QDK Learning for the first time, follow these steps.

1. Open VS Code.
2. Open the folder where you want to save your learning progress and course files.
3. On the **Activity Bar**, select the **Microsoft Quantum** icon.
4. In the **QDK Learning** view, select **Start learning**.
5. In the **Learning** view, select **Switch Course** for the course that you want to start. Or, tell the QDK Learning agent what course you want to open.



QDK Learning creates these items in your workspace.

- A `qdk-learning` folder that contains the files you use in the courses.
- A `qdk-learning.json` file that stores your current position and course progress.

To continue your progress in a later VS Code session, open the same workspace folder.

Move through QDK Learning

The **Learning** view organizes content into three levels:

- **Course:** A full learning experience, like the Quantum Katas or the quantum chemistry course.
- **Unit:** A group of related lessons and exercises.
- **Activity:** An single lesson, example, exercise, or notebook code cell.

Expand a course or unit to see its contents. The progress indicator for each course shows the number of completed units. Completed activities and units have a check mark.

Select a unit or activity to open it. Each course opens activities differently.

- The Quantum Katas open lessons in a **Lesson** tab and exercises in Q# files.
- Notebook courses open an editable Jupyter notebook for the selected unit.

Switch courses

QDK Learning saves your progress when you switch courses.

1. In the **Learning** view, find the course that you want to open.
2. Select **Switch Course** for the course.

QDK Learning opens your current position in that course. For a new course, QDK Learning opens the first unit.

Use Copilot to support your learning

When you start QDK Learning, the QDK Learning agent opens in Copilot Chat. The agent uses the context of your active course and activity to help you learn.

Ask the agent to:

- Explain a concept in more detail.
- Give you a hint for an exercise.
- Help you understand why a solution doesn't work.
- Answer questions about quantum computing, Q#, or QDK/Chemistry.

For example, try these prompts.

> What courses are available?

> Take me to the content about Grover's algorithm.

> Explain quantum phase estimation in a different way.

Course activities also include buttons that send the relevant lesson or exercise context to the QDK Learning agent.

Related content

- Learn quantum computing and Q# with the Quantum Katas
- Learn quantum chemistry with QDK Learning

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 09/18/2026

Learn quantum computing and Q# with the Quantum Katas

The Quantum Katas in QDK Learning are a self-paced course. QDK Learning is available in The Microsoft Quantum Development Kit (QDK) extension for Visual Studio Code (VS Code). The Katas course teaches quantum computing concepts and programming in Q#. Each kata includes explanations, examples, and coding exercises to help you apply what you learn.

Kata means "form" in Japanese and refers to a pattern for learning and practicing new skills.

Work through the Quantum Katas to:

- Learn the fundamentals of quantum computing and Q#.
- Run examples and check your solutions in VS Code.
- Choose the units that match your interests and experience.
- Save your progress and Q# exercise files in your workspace.
- Ask the QDK Learning agent for hints and explanations.

Prerequisites

To use the Quantum Katas, you need:

- The latest version of VS Code.
- The QDK extension for VS Code.
- A folder where QDK Learning can store your course files and progress.

To use the AI-assisted learning features, sign in to GitHub Copilot in VS Code.

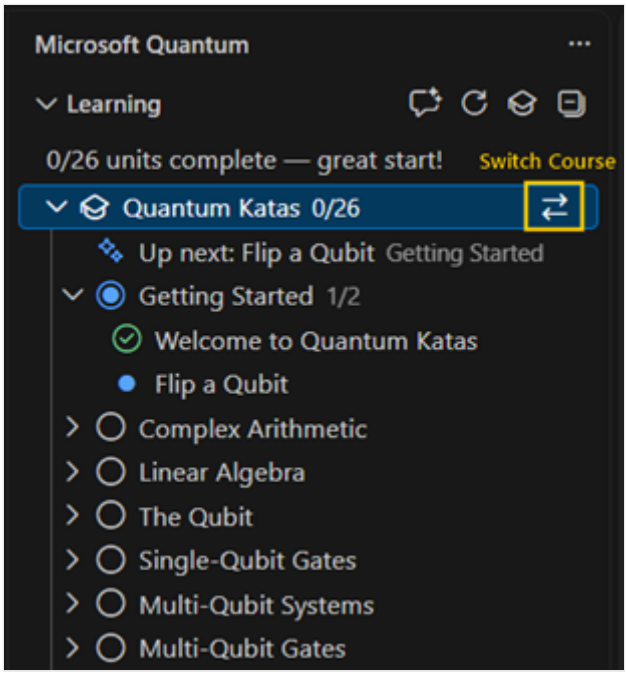
If you haven't used QDK Learning before, see QDK Learning overview.

Start the Quantum Katas

To start the Quantum Katas, follow these steps.

1. In VS Code, open the folder where you want to save your course files.
2. On the **Activity Bar**, select the **Microsoft Quantum** icon.
3. If the workspace isn't set up for QDK Learning, select **Start learning**.
4. In the **Learning** view, find **Quantum Katas** and then select **Switch Course**.

5. Expand the **Quantum Katas** course.
6. Select **Getting Started** to open the first lesson.



If you previously started the Quantum Katas in this workspace, QDK Learning returns to your saved position. To continue directly from that position, select **Up next** under the **Quantum Katas** course.

Go through the Quantum Katas

The Quantum Katas use two main areas in VS Code.

- The **Learning** view shows the course structure and your progress.
- The **Lesson** tab shows the current lesson or exercise and its available actions.

When an activity includes Q# code, QDK Learning also opens a `.qs` file in the editor. Edit and save the file like any other Q# file.

You don't have to complete the units in order. Select any unit in the **Learning** view to go to that unit.

Work with lessons and examples

Lessons explain quantum computing concepts and can include runnable Q# examples. The available actions depend on the lesson.

Expand table

Action	Description
Next	Mark the lesson as complete and go to the next activity.
Run	Run the Q# example for the lesson. After the example runs, **Next** becomes the primary action.
Explain	Ask the QDK Learning agent to explain the current concept in more detail.
Back	Go to the previous activity.

Complete coding exercises

Exercise activities open a Q# file with placeholder code. To complete an exercise, follow these steps.

1. Read the exercise instructions in the **Lesson** tab.
2. Complete the code in the `.qs` file.
3. Select **Check** in the **Lesson** tab, or select **Check Solution** above the exercise code.
4. If the solution is correct, select **Next** to continue.

Exercise activities include the following actions.

Expand table

Action	Description
Check	Check your code against the expected result.
Hint	Ask the QDK Learning agent for a hint about the current exercise.
Reset	Restore the original placeholder code and discard your changes to the exercise.
Back	Go to the previous activity.

If your solution isn't correct, a **Check failed** message provides more information. Select **What went wrong?** to ask the QDK Learning agent for help without revealing the full solution.

After you complete an exercise, **Next** becomes the primary action. To run the validation again, select **Check**. You can ask Copilot to show alternative solutions to some problems.

Use Copilot to support your learning

The QDK Learning agent uses your current kata, lesson, and code as context. Use the buttons in the **Lesson** tab or enter your own prompt in Copilot Chat.

For example, try the following prompts.

Navigate the course

> I'm interested in learning about Grover's algorithm. Take me to the relevant unit.

Get help with an exercise

> Help me understand why my solution doesn't work, but don't show me the complete answer.

Explore a concept

> Explain the difference between relative phase and global phase.

Save and reset your progress

QDK Learning automatically saves your place in the course and completed activities in the `qdk-learning.json` file at the root of your workspace. QDK Learning saves the Q# files that you edit in the `qdk-learning` folder.

To continue later:

1. In Visual Studio Code, open the same workspace folder.
2. Select the **Microsoft Quantum** icon.
3. Expand the **Learning** view and then the **Quantum Katas** course.
4. Select **Up next** to return to your current activity.

To reset an individual exercise:

1. Select **Reset** in the **Lesson** tab or **Reset Exercise** in the Q# code editor.
2. Confirm the reset. QDK Learning replaces your code.

You can also ask Copilot to reset a lesson for you.

Next steps

- QDK Learning overview
- Learn quantum chemistry with QDK Learning
- Introduction to Q#
- Create your first Q# program

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 09/18/2026

Learn quantum chemistry with QDK Learning

The **Ground-State Molecular Energies with QPE** course in QDK Learning shows how classical computational chemistry and quantum algorithms work together to estimate the ground-state electronic energy of a molecule. The Microsoft Quantum Development Kit (QDK) extension for Visual Studio Code (VS Code) includes QDK Learning.

In this course, you use the QDK for chemistry (QDK/Chemistry) library and quantum phase estimation (QPE) to estimate the energy of a stretched nitrogen molecule. The course uses Python notebooks to build and run Q# circuits on a local QDK simulator.

The course is appropriate for people with introductory knowledge of quantum computing and quantum chemistry.

What you learn

The course has seven units that guide you through the workflow.

Expand table

Unit	What you learn
Tutorial Overview	Review the course workflow, required environment, and quantum chemistry background.
Energy and Accuracy	Define the target energy and the accuracy comparisons used in the course.
Describing the Molecule	Define the stretched nitrogen molecule and construct a Hartree-Fock starting point.
Choosing the Active Space	Select the electrons and molecular orbitals for the correlated model.
Mapping the Problem to Qubits	Map the active-space Hamiltonian to qubits.
Preparing the Trial State	Prepare an approximation of the target ground state.
Iterative Quantum Phase Estimation	Estimate the energy with iterative QPE and compare the result with a classical reference.

The course uses a compact molecular model that also has a classical solution. This approach lets you compare each quantum step's output with the corresponding classical result.

Prerequisites

To work through the quantum chemistry course, you need:

- The latest version of VS Code.
- The QDK extension for VS Code.
- The Python extension for VS Code.
- The Python Environments extension for VS Code.
- The Jupyter extension for VS Code.
- Python 3.11, 3.12, or 3.13.
- QDK/Chemistry version 2.2.0 or later.
- A folder where QDK Learning can store your notebooks and progress.

To use the AI-assisted learning features, sign in to GitHub Copilot in VS Code.

You don't need an Azure subscription or an Azure Quantum workspace. The course runs locally on the QDK quantum simulators.

Create a Python environment

The QDK extension includes a command to create a Python virtual environment and install the packages required to run the quantum notebooks in this course. Install the Python Environments extension in VS Code to use the command.

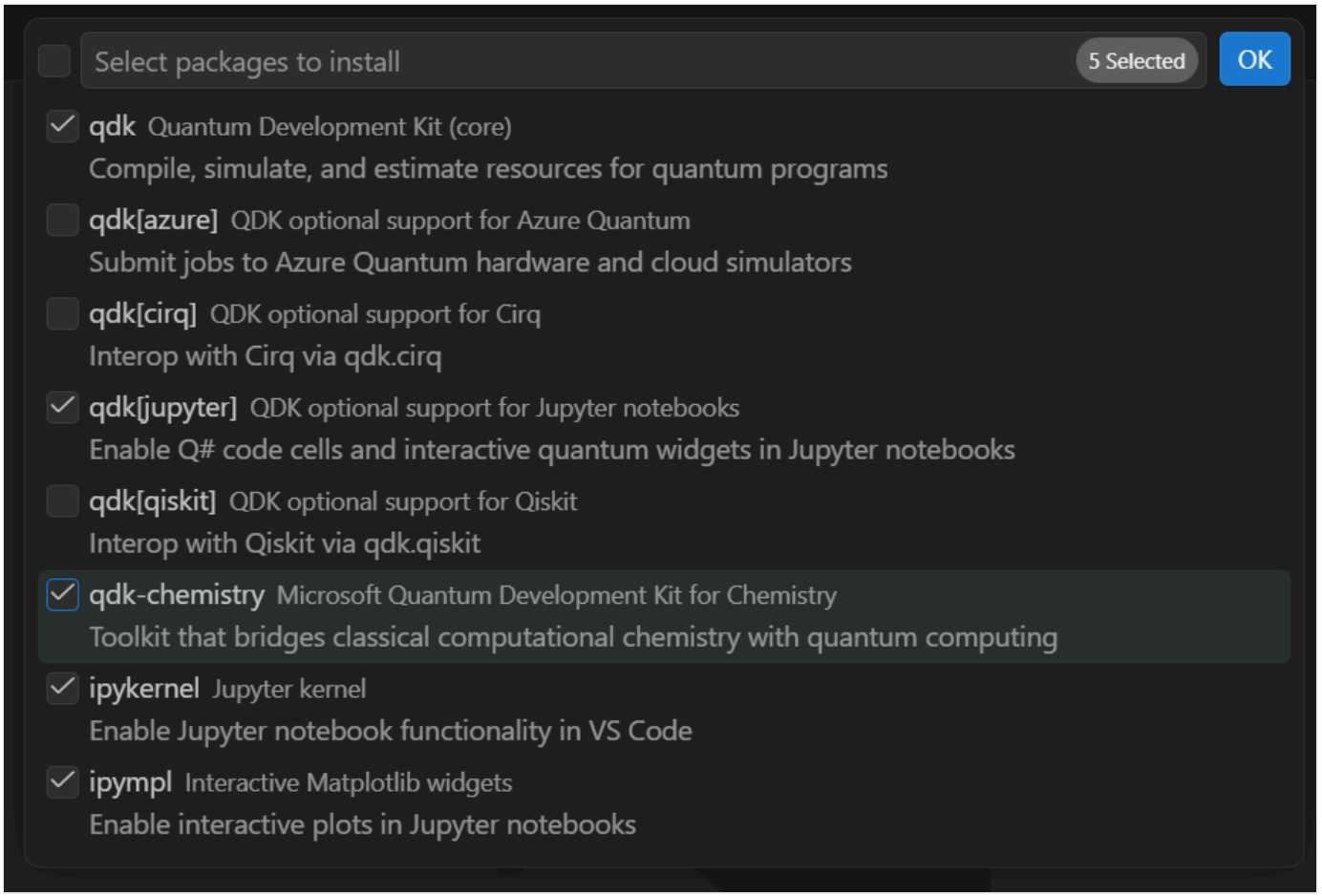
To create the environment, follow these steps:

1. In VS Code, open the folder where you want to complete the course.
2. Select **View** > **Command Palette**.
3. Enter and select **QDK: Create a Microsoft Quantum Python virtual environment**.
4. Keep the default package selections, and select **qdk-chemistry**.
5. Confirm the package selection.

The command creates a virtual environment named `.venv` in the current workspace. If the workspace already has a virtual environment, the command offers to update it.

> **Note**

The course requires `qdk-chemistry>=2.2.0`. The command installs an appropriate version in the new virtual environment.



Start the quantum chemistry course

To start the course, follow these steps:

1. On the **Activity Bar**, select the **Microsoft Quantum** icon.
2. If the workspace isn't set up for QDK Learning, select **Start learning**.
3. In the **Learning** view, find **Ground-State Molecular Energies with QPE** and select the **Switch Course** icon.
4. Expand the course to see its units.
5. Select **Tutorial Overview**.

QDK Learning creates an editable Jupyter notebook and opens it in the Jupyter notebook editor. The QDK attempts to set the notebook kernel to your Python virtual environment for the course.

Complete a notebook unit

Each unit uses a separate Jupyter notebook. To complete a unit, follow these steps:

1. Run the environment check near the beginning of the notebook. If the check reports missing packages, see Troubleshoot the Python environment.
2. Read the notebook and run the code cells in order.
3. Complete each exercise in the indicated code cell and run the cell to check your solution.
4. At the end of the notebook, select the link to the next unit or select another unit in the **Learning** view.

The exercise code includes automatic validation. If your solution is correct, the cell runs successfully, and QDK Learning marks the exercise as complete. If your solution is incorrect, the cell output explains that your solution doesn't work.

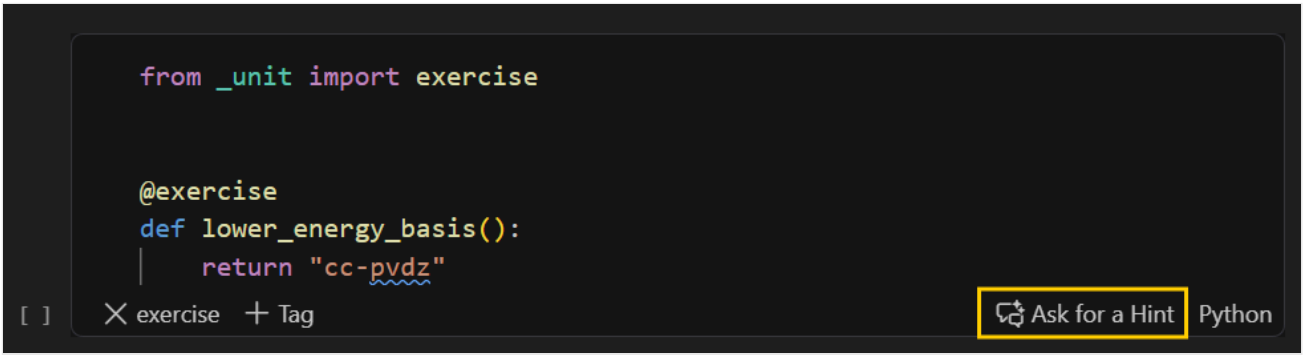
QDK Learning tracks the code cells that you successfully run, including cells that aren't exercises. The course and unit progress indicators update as you work.

> ⓘ **Note**
>
> The final circuit simulation in the **Iterative Quantum Phase Estimation** unit takes longer than the other required examples.

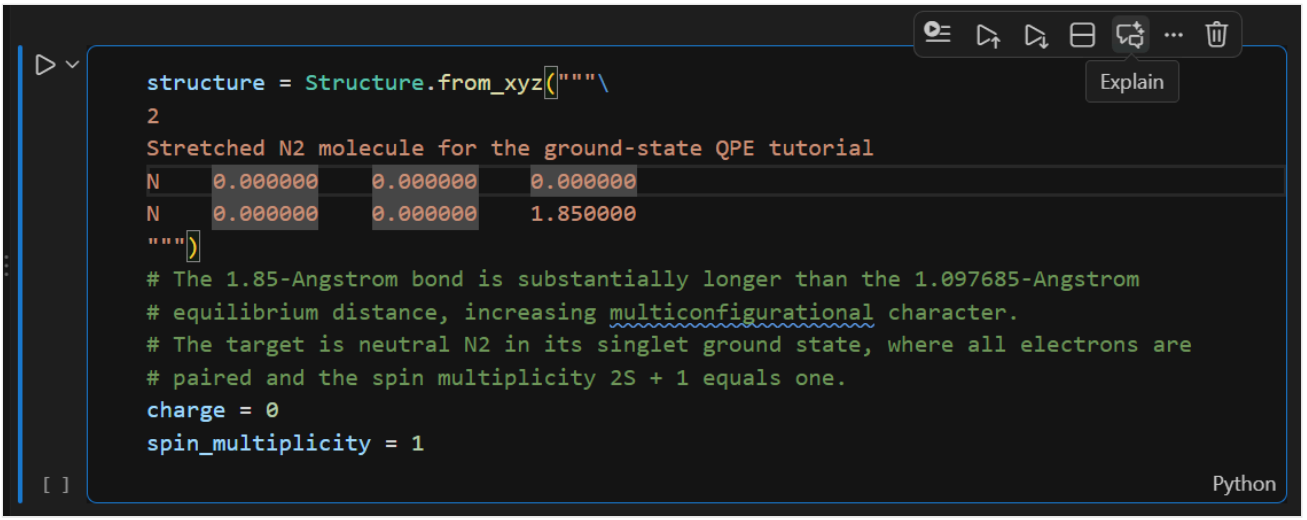
Get help from Copilot

The quantum chemistry notebooks include actions that send the current cell and course context to the QDK Learning agent.

- For an exercise cell, select **Ask for a Hint** in the cell status bar.



- To learn more about a notebook concept, select **Explain** in the cell toolbar.



- Enter your own question in Copilot Chat while the QDK Learning agent is active.

Try these example prompts:

> Why does the active-space selection affect the number of qubits?

> Give me a hint for this exercise without showing the complete solution.

> Explain how the trial state affects quantum phase estimation.

Reset a notebook unit

QDK Learning gives you an editable working copy of each notebook. The original course notebook remains available so that you can reset your work.

To reset a unit, follow these steps:

1. In the **Learning** view, open the shortcut menu for the unit that you want to reset.
2. Select **Reset Unit**.
3. In the confirmation message, select **Reset**.

You can select **Reset Unit** on the notebook toolbar instead.

Resetting a unit replaces the working notebook with a new copy, resets the unit's completion status, and deletes all your changes to the notebook.

Troubleshoot the Python environment

Use the following table to resolve issues with your Python environment.

Expand table

Problem	Resolution
The environment check can't import `qdk_chemistry`.	Run **QDK: Create a Microsoft Quantum Python virtual environment** again. Choose **Update existing environment**, and select **qdk-chemistry**.
The notebook uses the wrong Python kernel.	In the notebook toolbar, select the kernel name, and then choose the Python interpreter from the workspace `.venv` environment.
VS Code can't open or run the notebook.	Install and enable the Python and Jupyter extensions.
The installed QDK/Chemistry version is too old.	Update the workspace environment and confirm that it contains `qdk-chemistry` version 2.2.0 or later.

For more information about QDK/Chemistry requirements, see Install QDK for chemistry.

Save and resume your progress

QDK Learning stores your editable notebooks in the `qdk-learning/courses` folder and saves your progress in the `qdk-learning.json` file in the workspace root.

To resume the course, follow these steps:

1. Open the same workspace folder.
2. Select the **Microsoft Quantum** icon.
3. Select the unit where you want to continue.

Related content

- QDK Learning overview
- Welcome to QDK for chemistry
- QDK/Chemistry documentation

ⓘ**Note:** The author created this article with assistance from AI. Learn more

Last updated on 09/18/2026

How to test quantum programs in the QDK

Testing helps you check that quantum programs behave as expected. This article shows you how to test Q# and OpenQASM programs with the Microsoft Quantum Development Kit (QDK).

The QDK provides Q# functions and operations that check your code as it runs, and you can write Q# unit tests to run in Visual Studio Code (VS Code) or Python. You can also run OpenQASM programs from Python and use Python assertions to check their results.

To learn how to debug Q# and OpenQASM programs, see How to debug quantum programs in the QDK.

Prerequisites

To run Q# unit tests in VS Code, install the following:

- The latest version of VS Code, or open VS Code for the Web.
- The latest version of the QDK extension in VS Code.

To test programs from Python, install the latest version of the `qdk` Python package. To run examples in a Jupyter notebook, also install the following:

- The Python extension and Jupyter extension in VS Code.
- The latest version of the `qdk` Python package with the `jupyter` extra. You need version 1.31.0 or later to run Q# unit tests from Python.

  Bash

  ```
  pip install --upgrade "qdk[jupyter]"
  ```

Test Q# code with the `fail` expression

The fail expression immediately ends your program. To test your code, use a fail expression inside a conditional statement.

The following example uses a `fail` statement to test that a qubit array contains exactly 3 qubits. The program ends with an error message when the test doesn't pass.

1. In VS Code, create and save a new `.qs` file with the following code:

Q#

```
operation Main() : Unit {
    use qs = Qubit[6];
    let n_qubits = Length(qs);

    if n_qubits != 3 {
        fail $"The system should have 3 qubits, not {n_qubits}.";
    }
}
```

2. Press **Ctrl + F5** to run the program. Your program fails and the following output appears in the **Debug Console**:

Output

```
Error: program failed: The system should have 3 qubits, not 6.
```

3. Edit your code from `Qubit[6]` to `Qubit[3]`, save your file, and then press **Ctrl + F5** to run the program again. The program runs without an error because the test passes.

Test Q# code with the `Fact` function

Use the Q# Fact function from the Std.Diagnostics namespace to test your code. The function takes a Boolean expression and an error message string. If the expression is true, the test passes and your program continues. Otherwise, Fact ends your program and shows the error message.

To test the array length with Fact, follow these steps:

1. In VS Code, create and save a new `.qs` file with the following code:

Q#

```
import Std.Diagnostics.Fact;

operation Main() : Unit {
    use qs = Qubit[6];
    let n_qubits = Length(qs);

    Fact(n_qubits == 3,  $"The system should have 3 qubits, not {n_qubits}.")
}
```

2. Press **Ctrl + F5** to run the program. The test condition in `Fact` doesn't pass and the error message appears in the **Debug Console**.

3. Edit your code from `Qubit[6]` to `Qubit[3]`, save your file, and then press **Ctrl + F5** to run the program again. The test condition in `Fact` passes and your program runs without an error.

Check qubit states in Q#

The `CheckZero` and `CheckAllZero` Q# operations check whether the current state of a qubit or qubit array is . The `CheckZero` operation takes a single qubit and returns `true` only when the qubit is in the state. The `CheckAllZero` operation takes a qubit array and returns `true` only when all qubits in the array are in the state. To use `CheckZero` and `CheckAllZero`, import them from the `Std.Diagnostics` namespace.

The following example uses both operations. The `CheckZero` operation tests that the `X` operation flips the first qubit from the state to the state, and the `CheckAllZero` operation tests that both qubits are reset to the state.

In VS Code, create and save a new `.qs` file with the following code, then run the program and examine the output in the **Debug Console**.

Q#

```
import Std.Diagnostics.*;

operation Main() : Unit {
    use qs = Qubit[2];
    X(qs[0]);

    if CheckZero(qs[0]) {
        Message("X operation failed");
    }
    else {
        Message("X operation succeeded");
    }

    ResetAll(qs);

    if CheckAllZero(qs) {
        Message("Reset operation succeeded");
    }
    else {
        Message("Reset operation failed");
    }
}
```

Write Q# unit tests with the `@Test()` annotation

In Q# programs, you can apply the `@Test()` annotation to a callable (function or operation) to turn the callable into a unit test. These unit tests appear in the **Testing** menu in VS Code so that you can take advantage of this VS Code feature. You can turn a callable into a unit test only when the callable doesn't take input parameters.

The following example wraps the array length test code in an operation and turns that operation into a unit test:

1. In VS Code, create and save a new `.qs` file with the following code:

   Q#

   ```
   import Std.Diagnostics.Fact;

   @Test()
   operation TestCase() : Unit {
       use qs = Qubit[3];
       let n_qubits = Length(qs);

       Fact(n_qubits == 3, $"The system should have 3 qubits, not {n_qubits}.");
   }
   ```

   The `@Test()` annotation on the line before the `TestCase` operation definition turns the operation into a VS Code unit test. A green arrow appears on the operation definition line.

2. Select the green arrow to run TestCase and view the test results.

3. To interact with your unit tests in the VS Code Test Explorer, choose the **Testing** flask icon in the Activity Bar.

4. Edit your code from `Qubit[3]` to `Qubit[6]` and run the unit test again to see how the test information changes.

You can write and run Q# unit tests in VS Code without an entrypoint operation in your program.

> **Note**
>
> Callables from the `Std.Diagnostics` namespace aren't compatible with QIR generation, so only include unit tests in Q# code that you run on simulators. If you want to generate QIR from your Q# code, then don't include unit tests in your code.

Test OpenQASM programs from Python

The `qdk.openqasm.run` function runs an OpenQASM program on the local simulator and returns its results. Use Python assertions to check those results, or put the assertions in a pytest test function to run them as part of your Python test suite.

The following example checks that an OpenQASM program applies an `x` gate and measures the qubit as `One` in all three shots:

Python

```python
from qdk import Result
from qdk.openqasm import run

qasm_code = """
    OPENQASM 3.0;
    include "stdgates.inc";
    qubit q;
    output bit result;
    x q;
    result = measure q;
"""

results = run(qasm_code, shots=3)
assert results == [Result.One, Result.One, Result.One]
```

The `qdk.test_utils.run_tests` function discovers Q# callables marked with `@Test()`. It doesn't discover OpenQASM programs directly. To use `run_tests` with OpenQASM code, import the program as a Q# callable and call it from a Q# unit test:

Python

```python
from qdk import qsharp
from qdk.openqasm import import_openqasm
from qdk.test_utils import run_tests

import_openqasm("""
    OPENQASM 3.0;
    include "stdgates.inc";
    qubit[1] qs;
    x qs[0];
""", name="ApplyX")

qsharp.eval("""
import Std.Diagnostics.Fact;

@Test()
operation TestApplyX() : Unit {
    use qs = Qubit[1];
    ApplyX(qs);
    Fact(MResetZ(qs[0]) == One, "Expected the OpenQASM program to return One.");
```

```
}
""")

run_tests()
```

Run Q# unit tests from Python

The `qdk.test_utils` Python module lets you run your Q# unit tests from Python. Use the `run_tests` function to find and run all the Q# callables that have the `@Test()` annotation. You can call `run_tests` from a Jupyter notebook, a Python script, or a Python test framework like pytest. For example, you can use `run_tests` to run your Q# tests in a continuous integration (CI) pipeline.

The `qdk.test_utils` module also includes helpers that test the effect of Q# operations on qubits.

Run Q# unit tests in a Jupyter notebook

The following example defines two Q# unit tests in a Jupyter notebook and then runs them with `run_tests`.

1. In VS Code, press **Ctrl** + **Shift** + **P** to open the **Command Palette**.
2. Enter **Create: New Jupyter Notebook** and press **Enter**. A new Jupyter Notebook tab opens.
3. In the first cell, copy and run the following code:

   Python

   ```
   from qdk import qsharp
   from qdk.test_utils import run_tests
   ```

4. Create a new code cell. Copy and run the following code to define two Q# unit tests:

   Python

   ```
   qsharp.eval("""
   import Std.Diagnostics.*;

   @Test()
   operation TestQubitCount() : Unit {
       use qs = Qubit[3];
       let n_qubits = Length(qs);
       Fact(n_qubits == 3, $"The system should have 3 qubits, not {n_qubits}.");
   }

   @Test()
   ```

```
operation TestXFlipsQubit() : Unit {
    use q = Qubit();
    X(q);
    Fact(not CheckZero(q), "X operation failed.");
    Reset(q);
}
""")
```

5. Create a new code cell. Copy and run the following code to run your tests:

Python

```
run_tests()
```

The `run_tests` function prints a dot for each test that passes, and then prints a summary:

Output

```
Running 2 tests...
..
Finished tests: 2 passed, 0 failed.
```

6. To see what happens when a test fails, create a new code cell. Copy and run the following code to add a test that fails, and then run all your tests again:

Python

```
qsharp.eval("""
@Test()
operation TestQubitCountFails() : Unit {
    use qs = Qubit[6];
    let n_qubits = Length(qs);
    Std.Diagnostics.Fact(n_qubits == 3, $"The system should have 3 qubits, not
{n_qubits}.");
}
""")

run_tests()
```

For each test that fails, `run_tests` prints the name of the test and the error message. After all the tests run, `run_tests` prints a summary and then raises a `RuntimeError`.

Choose which tests run and how much output you get

The `run_tests` function takes the following optional parameters:

Expand table

Parameter	Description
`context`	The `qdk.Context` to find and run tests in. If you don't pass a context, then `run_tests` uses the default context.
`seed`	The seed for the simulator's random number generator. Set a seed to get the same results each time you run tests that include measurements.
`regex`	A regular expression that filters tests by their fully qualified names, such as `MyNamespace.MyTest`. Only tests with names that match the expression run.
`verbose`	The amount of output to print. `0` prints nothing. `1` prints a dot for each test that passes, and is the default. `2` prints the name of each test. `3` or higher also prints the output from your Q# code, such as from `Message` calls.

For example, to run only `TestXFlipsQubit` and print the test names, create a new code cell, then copy and run the following code:

Python

```
run_tests(regex="XFlips", verbose=2)
```

The output shows the name of the test that ran:

Output

```
Running 1 tests...
Running TestXFlipsQubit...
PASSED: TestXFlipsQubit

Finished tests: 1 passed, 0 failed.
```

Run unit tests in a Q# project

To run the unit tests in a Q# project, load the project with `qsharp.init` and then call `run_tests`. Pass the path to the folder that contains your project's `qsharp.json` file. For more information about Q# projects, see Develop and Manage Q# Projects and Custom Libraries.

Python

```
from qdk import qsharp
from qdk.test_utils import run_tests
```

```
qsharp.init(project_root="./my_project")
run_tests()
```

To run your tests separately from other Q# code in your Python session, create a `Context` object for your project and pass it to `run_tests`:

Python

```
from qdk import Context
from qdk.test_utils import run_tests

run_tests(context=Context(project_root="./my_project"))
```

Tests in a Q# project have fully qualified names that include the namespace. By default, the namespace is the name of the file that contains the test. For example, a test named `TestAddition` in a file named `Tests.qs` has the fully qualified name `Tests.TestAddition`.

Run Q# unit tests with pytest

The `run_tests` function raises a `RuntimeError` when any test fails, so you can call `run_tests` inside a pytest test function. When any of your Q# tests fail, pytest reports that the Python test fails.

1. In the folder that contains your Q# project folder, create a file named `test_qsharp.py` with the following code:

   Python

   ```
   from pathlib import Path

   from qdk import Context
   from qdk.test_utils import run_tests

   PROJECT_ROOT = Path(__file__).parent / "my_project"

   def test_qsharp_project():
       run_tests(context=Context(project_root=str(PROJECT_ROOT)))
   ```

2. Install pytest and then run your tests:

   Bash

   ```
   pip install pytest
   pytest
   ```

Test arithmetic operations with `ArithmeticOpTester`

The `ArithmeticOpTester` class runs a Q# arithmetic operation on integer inputs. The class puts each input integer into a qubit register, applies the operation, and then reads each register back as an integer. The integers are unsigned and use little-endian order.

To create an `ArithmeticOpTester` object, pass the name of the operation and the number of qubits in each register. The operation must take one `Qubit[]` input for each register. Then, call `run` with one integer for each register.

For example, the `IncByLE` operation adds the integer in the first register to the integer in the second register. The following code tests `IncByLE` on two 8-qubit registers:

Python

```python
from qdk.test_utils import ArithmeticOpTester

tester = ArithmeticOpTester("Std.Arithmetic.IncByLE", [8, 8])
assert tester.run([5, 7]) == [5, 12]
assert tester.run([100, 27]) == [100, 127]
```

Test a Q# operation's state with `dump_operation_on_state`

The `dump_operation_on_state` function applies a Q# operation to a qubit register and returns the state vector of the register as a list of complex numbers. The operation must take a single `Qubit[]` input. You can also pass a compatible OpenQASM program after you import it with `import_openqasm`. By default, the qubits start in the state. To start in a different state, pass a list of real amplitudes to the `initial_state` parameter. The state vector uses big-endian order.

For example, the following code tests that an operation prepares the Bell state :

Python

```python
import math

from qdk import qsharp
from qdk.test_utils import dump_operation_on_state

qsharp.eval("operation PrepareBellState(qs : Qubit[]) : Unit { H(qs[0]); CNOT(qs[0], qs[1]); }")

state = dump_operation_on_state("PrepareBellState", 2)
expected = [1 / math.sqrt(2), 0, 0, 1 / math.sqrt(2)]
assert all(abs(a - b) < 1e-9 for a, b in zip(state, expected))
```

```
from qdk.openqasm import import_openqasm

import_openqasm("""
    OPENQASM 3.0;
    include "stdgates.inc";
    qubit[2] qs;
    h qs[0];
    cx qs[0], qs[1];
""", name="PrepareBellQasm")

qasm_state = dump_operation_on_state("PrepareBellQasm", 2)
assert all(abs(a - b) < 1e-9 for a, b in zip(qasm_state, expected))
```

For more examples of how to test Q# operations from Python, see Testing operations from the QDK samples.

Related content

- How to debug quantum programs in the QDK
- Develop and Manage Q# Projects and Custom Libraries

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 11/12/2026

How to debug quantum programs in the QDK

Debugging helps you understand how a quantum program runs and find problems in your code. This article shows you how to debug Q# and OpenQASM programs with the Microsoft Quantum Development Kit (QDK) in Visual Studio Code (VS Code), and how to inspect quantum states with QDK dump functions in Q# and Python.

The VS Code debugger supports both Q# and OpenQASM source files. The dump-function examples in this article use Q#; the Python `dump_operation` function can also inspect a compatible OpenQASM program after you import it as a Q# callable.

To learn how to test Q# and OpenQASM programs, see How to test quantum programs in the QDK.

Debug programs in VS Code

Use the debugger in the QDK extension for VS Code to step through functions and operations, track local variable values, and inspect qubit states.

The following example demonstrates how to use the debugger with a Q# program. For complete information on VS Code debuggers, see Debugging on the VS Code website.

1. In VS Code, create and save a new `.qs` file with the following code:

   Q#

   ```
   import Std.Arrays.*;
   import Std.Convert.*;

   operation Main() : Result {
       use qubit = Qubit();
       H(qubit);
       let result = M(qubit);
       Reset(qubit);
       return result;
   }
   ```

2. On the `H(qubit);` line, select the area to the left of the line number to set a breakpoint. A red circle marks the breakpoint.

3. In the Activity Bar, select the **Run and Debug** icon to open the **Run and Debug** view, and then select **Run and Debug** to start debugging. The Debug toolbar opens.

4. When the debugger pauses at the breakpoint, expand **Quantum State** under **Variables** in the **Run and Debug** view to inspect the qubit's initial state.

5. Press **F11** to step into the `H` operation. The source code for the `H` operation appears. Notice that **Quantum State** changes to a superposition as you step through the `H` operation.

6. Press **F10** to step over the `M` operation. Notice that **Quantum State** resolves to either or after the measurement. The `result` variable is listed under **Locals**.

7. Press **F10** again to step over the `Reset` operation. Notice that **Quantum State** is reset to .

When you finish exploring the debugger, press **Shift + F5** to exit the debugger.

Debug an OpenQASM program

You can set breakpoints and inspect qubit states while you debug an OpenQASM file.

1. In an open Q# project in VS Code, create and save a new `.qasm` file with the following code:

   qasm

   ```
   OPENQASM 3.0;
   include "stdgates.inc";

   qubit q;
   bit result;
   h q;
   result = measure q;
   ```

2. On the `h q;` line, select the area to the left of the line number to set a breakpoint.

3. In the code editor, select the **Debug** code lens to start debugging.

4. When the debugger pauses at the breakpoint, expand **Quantum State** in the **Variables** pane to inspect the qubit state. Press **F10** to step over `h q;` and inspect the state again.

5. Press **F5** to continue, or **Shift + F5** to stop debugging.

> ⓘ **Note**

The VS Code debugger works with Q# (`.qs`) and OpenQASM (`.qasm`) files. It doesn't debug Q# cells in Jupyter Notebook.

Inspect quantum state with QDK dump functions

The QDK provides Q# and Python functions that display information about a program's quantum state. The following `DumpMachine` and `dump_machine` examples inspect Q# programs. The Python `dump_operation` function can inspect Q# operations and compatible OpenQASM programs imported as Q# callables.

The Q# `DumpMachine` function

The Q# `DumpMachine` function prints information about the current state of the qubit system to the console without interrupting your program.

The following example calls `DumpMachine` at two points in a Q# program and explores the output.

1. In VS Code, create and save a new `.qs` file with the following code:

   Q#

   ```
   import Std.Diagnostics.*;

   operation Main() : Unit {
       use qubits = Qubit[2];
       X(qubits[1]);
       H(qubits[1]);
       DumpMachine();

       R1Frac(1, 2, qubits[0]);
       R1Frac(1, 3, qubits[1]);
       DumpMachine();

       ResetAll(qubits);
   }
   ```

2. Press **Ctrl + Shift + Y** to open the **Debug Console**.

3. Press **Ctrl + F5** to run your program. The following output from `DumpMachine` appears in the **Debug Console**:

   Output

   ```
   Basis | Amplitude      | Probability | Phase
   ------------------------------------------------
   ```

```
 |00⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
 |01⟩ | -0.7071+0.0000𝑖 |    50.0000% |  -3.1416

Basis | Amplitude      | Probability | Phase
-----------------------------------------------
 |00⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
 |01⟩ | -0.6533-0.2706𝑖 |    50.0000% |  -2.7489
```

The output from `DumpMachine` shows how the state of the qubit system changes after each set of gates.

> **Note**
>
> The output from `DumpMachine` uses big-endian ordering.

The Python `dump_machine` function

The dump_machine function is part of the `qdk.qsharp` Python module. This function returns a `StateDump` object that contains the qubit count and the sparse state amplitudes of the qubit system.

The following example uses `dump_machine` to inspect qubit states in a Jupyter notebook.

1. In VS Code, press **Ctrl** + **Shift** + **P** to open the **Command Palette**.
2. Enter **Create: New Jupyter Notebook** and press **Enter**. A new Jupyter Notebook tab opens.
3. In the first cell, copy and run the following code:

Python

```python
from qdk import qsharp
```

4. Create a new code cell, then copy and run the following Q# code:

Q#

```
%%qsharp

use qubits = Qubit[2];
X(qubits[0]);
H(qubits[1]);
```

5. Create a new code cell. Copy and run the following Python code to view the qubit state at this point in the program:

Python

```
dump = qsharp.dump_machine()
dump
```

The `dump_machine` function displays the following output:

Output

```
Basis State
(|ψ₁…ψₙ⟩)  Amplitude       Measurement Probability  Phase
|10⟩       0.7071+0.0000i   50.0000%                 ↑  0.0000
|11⟩       0.7071+0.0000i   50.0000%                 ↑  0.0000
```

6. Create a new code cell, then copy and run the following Q# code:

Q#

```
%%qsharp

R1Frac(1, 2, qubits[0]);
R1Frac(1, 3, qubits[1]);
```

7. Create a new code cell. Copy and run the following Python code to view the qubit state at this point in the program:

Python

```
dump = qsharp.dump_machine()
dump
```

The `dump_machine` function displays the following output:

Output

```
Basis State
(|ψ₁…ψₙ⟩)  Amplitude      Measurement Probability  Phase
|10⟩       0.5000+0.5000i  50.0000%                 ↗  0.7854
|11⟩       0.2706+0.6533i  50.0000%                 ↗  1.1781
```

8. To print an abbreviated version of the `dump_machine` output, create a new code cell and run the following Python code:

Python

```
print(dump)
```

9. To get the total number of qubits in the system, create a new code cell and run the following Python code:

Python

```
dump.qubit_count
```

10. To get the amplitudes of the and basis states, create a new code cell and run the following Python code:

Python

```
print(dump[2])
print(dump[3])
```

Inspect operations with Python `dump_operation`

The `dump_operation` function is available in the `qdk.qsharp` Python module. This function takes two inputs: a Q# operation or operation definition as a string and the number of qubits that the operation uses. The output from `dump_operation` is a nested list that represents the square matrix of complex numbers that corresponds to the given quantum operation. The matrix values are in the computational basis, and each sublist represents a row of the matrix.

The following example uses `dump_operation` to display matrices for single-qubit and two-qubit operations.

1. In VS Code, press **Ctrl + Shift + P** to open the **Command Palette**.
2. Enter **Create: New Jupyter Notebook** and press **Enter**. A new Jupyter Notebook tab opens.
3. In the first cell, copy and run the following code:

Python

```
from qdk import qsharp
from qdk.qsharp import dump_operation
```

4. To display the matrix elements of a single-qubit gate, call `dump_operation` and pass 1 for the number of qubits. For example, copy and run the following Python code in a new code cell

to get the matrix elements for an identity gate and a Hadamard gate:

Python

```
res = dump_operation("qs => ()", 1)
print("Single-qubit identity gate:\n", res)
print()

res = dump_operation("qs => H(qs[0])", 1)
print("Single-qubit Hadamard gate:\n", res)
```

5. You can also call the `qsharp.eval` function and then reference the Q# operation in `dump_operation` to get the same result. For example, create a new code cell, then copy and run the following Python code to print the matrix elements for a single-qubit Hadamard gate:

   Python

   ```
   qsharp.eval("operation SingleH(qs : Qubit[]) : Unit { H(qs[0]) }")

   res = dump_operation("SingleH", 1)
   print("Single-qubit Hadamard gate:\n", res)
   ```

6. To display the matrix elements of a two-qubit gate, call `dump_operation` and pass 2 for the number of qubits. For example, copy and run the following Python code in a new code cell to get the matrix elements for a Controlled Ry operation where the second qubit is the target qubit:

   Python

   ```
   qsharp.eval ("operation ControlRy(qs : Qubit[]) : Unit { Controlled Ry([qs[0]],
   (0.5, qs[1])); }")

   res = dump_operation("ControlRy", 2)
   print("Controlled Ry rotation gate:\n", res)
   ```

You can also inspect an OpenQASM program after you import it as a Q# callable. For example, the following code gets the matrix for an OpenQASM `x` gate:

Python

```
from qdk.openqasm import import_openqasm

import_openqasm("""
    OPENQASM 3.0;
    include "stdgates.inc";
```

```
    qubit[1] qs;
    x qs[0];
""", name="ApplyX")

res = dump_operation("ApplyX", 1)
print("OpenQASM X gate:\n", res)
```

For more examples of how to test and debug your code with `dump_operation`, see Testing operations from the QDK samples.

Related content

- How to test quantum programs in the QDK

Note: The author created this article with assistance from AI. Learn more

---

Last updated on 11/12/2026

How to submit jobs to Azure Quantum

An Azure Quantum job runs a quantum program on a provider target, such as a quantum hardware system or simulator. Use the Microsoft Quantum Development Kit (QDK) to submit jobs interactively from Visual Studio Code (VS Code) or programmatically with Python.

Before you submit a job

To submit a job, you need:

- An Azure account with an active subscription.
- An Azure Quantum workspace. To create one, see Create an Azure Quantum workspace.
- A provider and target in your workspace that accept the input format and Quantum Intermediate Representation (QIR) target profile of your program.

Compare job submission methods

Choose a method based on how you develop your program and how much control or automation you need.

Expand table

Method	Languages and inputs	Use this method when	Consider another method when
QDK extension for VS Code	Q# and OpenQASM source files	You want an interactive workflow in VS Code with code completion, local simulation, debugging, circuit visualization, and job submission in one interface.	You use a Python quantum framework, need to generate jobs programmatically, or want to automate submissions.
QDK Python package	Q#, OpenQASM, Qiskit, Cirq, PennyLane, and compiled QIR	You work in a Python script or Jupyter notebook, want framework-native APIs, or need to generate, submit, and process jobs in code.	You only need to submit a Q# or OpenQASM file interactively, or you already have a compiled input file for an automation workflow.

Submit jobs with the QDK extension for VS Code

Use the QDK extension when you write Q# or OpenQASM directly in VS Code and want a visual, interactive workflow. The extension can:

- Run and debug your program on a local simulator.
- Display circuit diagrams and result histograms.
- Connect to an Azure Quantum workspace.
- List providers, targets, and submitted jobs.
- Submit the active Q# or OpenQASM program and download the results.

This method is good for exploration and interactive development. You don't need to write connection or submission code.

For instructions, see Submit jobs with the QDK extension for VS Code.

Submit jobs with the QDK Python package

Use the QDK Python package when your workflow starts in Python or a Jupyter notebook. The Python package provides:

- The `qdk.azure` module to connect to a workspace and submit jobs.
- Q# and OpenQASM compilation to QIR.
- Job submission APIs for Qiskit and Cirq.
- A QIR submission path for PennyLane programs.
- APIs to inspect job status, and retrieve and visualize results.

This method is good for parameterized experiments, repeated submissions, and application integration.

For instructions, see Submit jobs with the QDK Python package.

Use provider-specific formats

Requirements for job input formats and parameters can vary by provider and target, and can have different job submission workflows. For more information, see Submit formatted quantum circuits to Azure Quantum.

Use provider-specific formats only when your workflow or target requires them.

What happens after submission

All submission methods create an Azure Quantum job. The job moves through states such as **Waiting**, **Executing**, and a final state such as **Succeeded**, **Failed**, or **Cancelled**. You can monitor and manage all jobs from VS Code, Python, Azure CLI, or the Azure portal.

For information about job properties and output, see Work with Azure Quantum jobs.

Related content

- Connect to your Azure Quantum workspace
- Azure Quantum computing provider targets
- Quantum computing target profiles
- Submit jobs to Azure Quantum with Azure CLI

Last updated on 08/29/2026

Submit jobs to Azure Quantum with the QDK extension for VS Code

Use the Microsoft Quantum Development Kit (QDK) extension for Visual Studio Code (VS Code) to submit Q# and OpenQASM programs to Azure Quantum. You can test your program locally, connect to your Azure Quantum workspace, select a provider target, submit the active program, and view its results without writing submission code.

If you use Qiskit, Cirq, PennyLane, or a Python-based Q# or OpenQASM workflow, see Submit jobs with the QDK Python package.

Prerequisites

- An Azure account with an active subscription.
- An Azure Quantum workspace. To create one, see Create an Azure Quantum workspace.
- The latest version of VS Code.
- The latest version of the QDK extension.

For installation details, see Set up the Microsoft Quantum Development Kit.

> **Note**
>
> The examples in this article submit jobs to **simulator targets**. The targets available to you depend on the providers that you configure in your workspace.

Prepare and test your program

Choose the tab for the language of your source file.

Q#

1. In VS Code, select **File** > **New Text File**, and save the file as `Main.qs`.
2. Add the following Q# program, which uses the **Base** QIR target profile.

   Q#

```
@EntryPoint(Base)
operation Main() : Result {
    use q = Qubit();
    H(q);
    return MResetZ(q);
}
```

3. Select **Run** from the code lens above the `Main` operation to test the program on the local simulator.

Connect to your Azure Quantum workspace

> **⚠ Warning**
>
> Starting January 11, 2027, Azure Quantum will require users in a Quantum workspace to have the **Quantum Workspace Data Reader** or **Quantum Workspace Data Contributor** role to access job results. The **Quantum Workspace Data Contributor** role will be required to submit jobs. To avoid interruptions to your Azure Quantum workflows, assign these roles to the appropriate users before the deadline. For more information, see Assign data access roles for users in the workspace.

You can connect to a workspace using an Azure account.

1. In VS Code, select **View** > **Command Palette**.
2. Enter **QDK: Connect to an Azure Quantum workspace**.
3. Choose **Azure account**.
4. A VS Code message appears stating that QDK wants to sign in, press **Allow**.
5. Choose an account to use.
6. Select a **tenant**.
7. Select the **subscription**.
8. Confirm that your workspace appears under **Quantum Workspaces**.

For more information about connecting to workspaces, see Connect to your Azure Quantum workspace.

Select an Azure Quantum target

1. In the **Microsoft Quantum** panel, expand your workspace.

2. Expand **Providers**.
3. Expand a provider and review its available targets in your workspace.
4. Select a simulator target that accepts your program's input format and QIR target profile.

> **Tip**
>
> Test on a simulator target before you use a paid quantum hardware target. Before your submit to quantum hardware, review the provider pricing and estimate the job cost when the provider supports cost estimation.

Submit your program

1. In VS Code, open the Q# or OpenQASM file that you want to submit.
2. In the **Microsoft Quantum** panel, find the target that you selected.
3. Select the play icon next to the target.
4. Enter a name that identifies the job.
5. Enter the number of shots (the number of times to run the program).
6. Press **Enter** to submit the job.

VS Code displays a notification when you submit the job. The time required to complete the job depends on the target and its queue.

Monitor the job and view results

1. In the **Microsoft Quantum** panel, expand **Jobs**.
2. Find the job that you submitted. Hover over the job to view details such as its status and submission time.
3. After the job succeeds, select the histogram icon next to the job to display supported results as a histogram. To display or download the raw result, select the text icon.

Result formats differ by provider and target. If the result can't be displayed as a histogram, view the raw output.

For information about job states, cancellation, and output data, see Work with Azure Quantum jobs.

Troubleshoot job submission

If you experience issues with job submission, refer to the following guidance for help.

The target doesn't appear

Confirm that:

- The provider is configured in your workspace.
- The target is currently available.
- You connected to the intended Azure directory, subscription, and workspace.

For information about managing providers, see Add or remove a provider in an Azure Quantum workspace.

The program doesn't compile for the target

The program must use a QIR target profile and operations that the target supports. Review the compiler error, the program's target profile, and the target capabilities. For more information, see Quantum computing target profiles.

The job fails after submission

Open the job details and review the provider error message. Job failures can result from unsupported input, invalid job parameters, provider availability, or quota limits. For common resolutions, see Troubleshoot Azure Quantum.

Related content

- How to submit jobs to Azure Quantum
- Submit jobs with the QDK Python package
- Submit jobs to Azure Quantum with Azure CLI
- Work with Azure Quantum jobs
- Azure Quantum computing provider targets
- VS Code reference for the QDK

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 09/25/2026

Submit jobs to Azure Quantum with the QDK Python package

Use the Microsoft Quantum Development Kit (QDK) Python package to submit Q#, OpenQASM, Qiskit, Cirq, and PennyLane programs to Azure Quantum.

This article provides job submission examples for each supported quantum language or framework. For an interactive Q# or OpenQASM workflow that doesn't require Python code, see Submit jobs with the QDK extension for VS Code.

Prerequisites

- An Azure account with an active subscription.
- An Azure Quantum workspace. To create one, see Create an Azure Quantum workspace.
- Python 3.10 or later.
- If you use a Jupyter notebook, a Jupyter environment such as the Jupyter extension for VS Code and the `ipykernel` package.
- Install Azure CLI with the `quantum` extension.

  Azure CLI

  ```
  az extension add --upgrade --name quantum
  ```

- Install the latest version of the `qdk` Python package with the `azure` and `jupyter` extras.

  Bash

  ```
  pip install --upgrade "qdk[azure,jupyter]"
  ```

  If you want to submit a Qiskit or Cirq program, install the `cirq` and `qiskit` extras.

  Bash

  ```
  pip install --upgrade "qdk[qiskit,cirq]"
  ```

Connect to your Azure Quantum workspace

> ⚠ **Warning**
>
> Starting January 11, 2027, Azure Quantum will require users in a Quantum workspace to have the **Quantum Workspace Data Reader** or **Quantum Workspace Data Contributor** role to access job results. The **Quantum Workspace Data Contributor** role will be required to submit jobs. To avoid interruptions to your Azure Quantum workflows, assign these roles to the appropriate users before the deadline. For more information, see Assign data access roles for users in the workspace.

Before you submit a job, you need to connect to an Azure Quantum workspace. The `qdk.azure` module provides the `Workspace` object to connect to Azure Quantum workspaces. To connect to your workspace, follow these steps.

Sign in with Azure CLI

If you didn't already, sign in to Azure from a terminal before you open a Jupyter notebook.

Azure CLI

```
az login
```

If you don't sign in from the terminal, you need to authenticate every time you connect to a Quantum workspace through Python. If your account has access to multiple subscriptions, set the subscription that contains your Azure Quantum workspace.

Azure CLI

```
az account set --subscription <subscription-id>
```

Run `az login` again when the session expires or when you want to use a different Azure account.

Get the resource ID of your workspace

1. Sign in to the Azure portal.
2. Go to the Quantum workspace where you want to submit your job.
3. In the **Overview** page, find and copy **Resource ID**.

Connect to the workspace

1. To connect to your Quantum workspace, create a `Workspace` object with the resource ID that you copied.

   Python

   ```python
   from qdk.azure import Workspace

   workspace = Workspace(resource_id="") # Add your resource ID
   ```

   > **Note**
   >
   > Y

2. Verify the connection and view the targets available in the workspace.

   Python

   ```python
   for target in workspace.get_targets():
       print(target.name)
   ```

For other connection and authentication options, see Connect to your Azure Quantum workspace.

> **Note**
>
> The following examples submit jobs to simulator targets. The targets available in your workspace depend on your configured providers.

Prepare and submit your program

Choose the tab for your quantum language or framework.

Q#

Use the `qdk.qsharp` module to define or load Q# code, compile an entry point to QIR, and submit the QIR to an Azure Quantum target.

Define and compile the Q# program

1. Initialize Q# with a QIR target profile that the Azure Quantum target supports.

Python

```
from qdk import qsharp, TargetProfile

qsharp.init(target_profile=TargetProfile.Base)
```

For more information about QIR target profiles, see Azure Quantum QIR target profiles in the QDK.

2. Define a Q# operation. For example, use the following `RandomBit` operation.

Python

```
qsharp.eval("""
    operation RandomBit() : Result {
        use q = Qubit();
        H(q);
        return MResetZ(q);
    }
""")
```

3. Test the operation on the local simulator.

Python

```
print(qsharp.run("RandomBit()", shots=10))
```

4. Compile the operation to QIR.

Python

```
from qdk.qsharp import compile

program = compile("RandomBit()")
```

Submit the Q# job

1. Select a compatible target from your workspace.

Python

```
target = workspace.get_targets("rigetti.sim.qvm")
```

2. Submit the compiled program.

Python

```
job = target.submit(program, "qsharp-job", shots=100)
print("Job ID:", job.id)
```

3. Wait for the job to complete and retrieve its results.

Python

```
job.wait_until_completed()
print("Status:", job.details.status)

results = job.get_results()
print(results)
```

Related content

- How to submit jobs to Azure Quantum
- Submit jobs with the QDK extension for VS Code
- Submit jobs to Azure Quantum with Azure CLI
- Work with Azure Quantum jobs
- Quantum programming language support in the QDK
- Azure Quantum computing provider targets

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 09/25/2026

Azure Quantum QIR target profiles in the QDK

Quantum computing is still an emerging technology. Not all quantum hardware can run all quantum programs. For example, only certain hardware can perform mid circuit measurements, which is required to run programs with conditional branches based on qubit measurement results.

When you submit a program to run on Azure Quantum, your program is converted to quantum intermediate representation (QIR) format. QIR doesn't depend on the programming language or type of quantum hardware that your program runs on. The Microsoft Quantum Development Kit (QDK) supports several QIR target profiles for different hardware capabilities.

For more information about QIR, see Quantum intermediate representation.

Overview of target profiles

Azure Quantum and the QDK support several QIR target profiles. The type of target profile that you choose determines which of the following programming constructs your program can use.

- Conditional branches with `if` statements based on qubit measurement results
- Arithmetic operations on floating point numbers computed from qubit measurement results
- Fixed and unbounded loops based on qubit measurement results

The following table lists all the QIR target profiles in the QDK and the programming constructs that the target profiles support, in order from most to least restrictive.

Expand table

QIR target profile	Conditional branches	Floating point operations	Loops
Base	❌	❌	❌
Adaptive RI	✅	❌	❌
Adaptive RIF	✅	✅	❌
Adaptive	✅	✅	✅
Unrestricted	✅	✅	✅

Set target profiles in the QDK

To run a program on the QDK simulators or Azure Quantum target, you need to set a QIR target profile. If you don't manually set a target profile, the QDK tries to automatically set the appropriate profile for the chosen target.

when submitting programs to run through the Azure Quantum service, the QDK tries to automatically select the appropriate profile for the chosen execution target.

Base QIR profile

The base QIR target profile is the most restrictive one. Use the base profile for simpler programs that don't use classical programming structures like branches and loops. If your quantum hardware target can't perform mid-circuit measurements, then you probably need to use the base profile.

The following Azure Quantum targets can run programs that use the base QIR profile.

Expand table

Provider	Simulator	QPU
IonQ	`ionq.simulator`	`ionq.qpu.*`
Rigetti	`rigetti.sim.*`	`rigetti.qpu.*`

To learn more about these Azure Quantum providers, see IonQ provider and Rigetti provider.

Set the base profile

To set the base QIR target profile in the QDK extension for Visual Studio Code (VS Code), choose one of the following options.

- If you set up a Q# project, then add the following command to your project's `qsharp.json` file.

```
JSON
{
  "targetProfile": "base"
}
```

- If you're working in a `.qs` file that isn't part of a Q# project, then set the target profile directly in your Q# code. To do so, add `@EntryPoint(Base)` on the line before the entrypoint operation in your program.

  Q#

  ```
  @EntryPoint(Base)
  operation Main() : Unit {

  ...

  }
  ```

To set the base target profile in the QDK Python library, run the following code.

Python

```
from qdk import init, TargetProfile

init(target_profile=TargetProfile.Base)
```

Limitations on Q# programs with base profile

The base target profile can run a wide variety of Q# programs. The main constraint is that Q# programs can't perform logical comparisons with `Result` type values from measurement operations.

For example, you can't run the following `FlipQubitOnZero` operation on a base target because the program contains an `if` statement that uses a measurement result.

Q#

```
    @EntryPoint(Base)
    operation FlipQubitOnZero() : Unit {
        use q = Qubit();
        if M(q) == Zero {
            X(q);
        }
    }
```

Adaptive RI QIR profile

The adaptive RI target profile can run a wider variety of programs than the base profile, but still has some limitations. Adaptive RI targets support programs that use mid-circuit measurements in

conditional `if` statements. If your quantum hardware can perform mid-circuit measurements and your program uses `if` statements based on measurement results, then you probably need to use the adaptive RI profile.

For example, the following Q# program can run on an adaptive RI target.

Q#

```
@EntryPoint(Adaptive_RI)
operation MeasureQubit(q : Qubit) : Result {
    return M(q);
}

operation SetToZero(q : Qubit) : Unit {
     if MeasureQubit(q) == One { X(q); }
}
```

The following Azure Quantum targets can run programs that use the adaptive RI target profile.

Expand table

Provider	Simulator	QPU
Quantinuum	`quantinuum.sim.h2-1e`	`quantinuum.qpu.h2-1`
Quantinuum	`quantinuum.sim.h2-2e`	`quantinuum.qpu.h2-2`

For more information on Quantinuum in Azure Quantum, see Quantinuum provider.

Set the adaptive RI profile

To set the adaptive RI QIR target profile in the QDK extension for VS Code, choose one of the following options.

- If you set up a Q# project, then add the following command to your project's `qsharp.json` file:

  JSON

  ```
  {
    "targetProfile": "adaptive_ri"
  }
  ```

- If you're working in a `.qs` file that isn't part of a Q# project, then set the target profile directly in your Q# code. To do so, add `@EntryPoint(Adaptive_RI)` on the line before the

entrypoint operation in your program.

Q#

```
@EntryPoint(Adaptive_RI)
operation Main() : Unit {

...

}
```

To set the adaptive RI target profile in the QDK Python library, run the following code.

Python

```
from qdk import init, TargetProfile

init(target_profile=TargetProfile.Adaptive_RI)
```

Adaptive RIF QIR profile

The adaptive RIF target profile has the same capabilities as the adaptive RI profile, but also supports programs that contain floating point arithmetic operations. If your quantum hardware can perform mid-circuit measurements, and your program uses `if` statements and floating point numbers computed from measurement results, then you probably need to use the adaptive RIF profile.

For example, the following Q# program can run on an adaptive RIF target.

Q#

```
@EntryPoint(Adaptive_RIF)
operation DynamicFloat() : Double {
    use q = Qubit();
    H(q);
    mutable f = 0.0;
    if M(q) == One {
        f = 0.5;
    }
    Reset(q);
    return f;
}
```

Azure Quantum doesn't have adaptive RIF targets at this time, but you can run programs for adaptive RIF targets on the local QDK simulators. For more information on simulators in the QDK, see Overview of quantum simulators in the QDK.

Set the adaptive RIF profile

To set the adaptive RIF QIR target profile in the QDK extension for VS Code, choose one of the following options.

- If you set up a Q# project, then add the following command to your project's `qsharp.json` file:

  JSON

  ```
  {
    "targetProfile": "adaptive_rif"
  }
  ```

- If you're working in a `.qs` file that isn't part of a Q# project, then set the target profile directly in your Q# code. To do so, add `@EntryPoint(Adaptive_RIF)` on the line before the entrypoint operation in your program.

  Q#

  ```
  @EntryPoint(Adaptive_RIF)
  operation Main() : Unit {

  ...

  }
  ```

To set the adaptive RIF target profile in the QDK Python library, run the following code.

Python

```
from qdk import init, TargetProfile

init(target_profile=TargetProfile.Adaptive_RIF)
```

Adaptive QIR profile

The adaptive target profile has the same capabilities as the adaptive RIF profile, but also supports programs that use loops based on measurement results. Adaptive programs can use loops with a set number of iterations, and repeat-until-success (RUS) loops.

Azure Quantum doesn't have adaptive targets at this time, but you can run programs for adaptive targets on the local QDK simulators. For more information on simulators in the QDK, see Overview of quantum simulators in the QDK.

Set the adaptive profile

To set the adaptive QIR target profile in the QDK extension for VS Code, choose one of the following options.

- If you set up a Q# project, then add the following command to your project's `qsharp.json` file:

  JSON

  ```
  {
    "targetProfile": "adaptive"
  }
  ```

- If you're working in a `.qs` file that isn't part of a Q# project, then set the target profile directly in your Q# code. To do so, add `@EntryPoint(Adaptive)` on the line before the entrypoint operation in your program.

  Q#

  ```
  @EntryPoint(Adaptive)
  operation Main() : Unit {

  ...

  }
  ```

To set the adaptive target profile in the QDK Python library, run the following code.

Python

```
from qdk import init, TargetProfile

init(target_profile=TargetProfile.Adaptive)
```

Unrestricted QIR profile

The unrestricted target profile can run all quantum programs. No current quantum target supports unrestricted QIR, but you can use the unrestricted profile to run complex programs on the QDK simulators for quantum development.

Set the unrestricted profile

To set the unrestricted QIR target profile in the QDK extension for VS Code, choose one of the following options.

- If you set up a Q# project, then add the following command to your project's `qsharp.json` file:

  JSON

  ```
  {
    "targetProfile": "unrestricted"
  }
  ```

- If you're working in a `.qs` file that isn't part of a Q# project, then set the target profile directly in your Q# code. To do so, add `@EntryPoint(Unrestricted)` on the line before the entrypoint operation in your program.

  Q#

  ```
  @EntryPoint(Unrestricted)
  operation Main() : Unit {

  ...

  }
  ```

To set the unrestricted target profile in the QDK Python library, run the following code.

Python

```
from qdk import init, TargetProfile

init(target_profile=TargetProfile.Unrestricted)
```

Last updated on 07/02/2026

Connect to your Azure Quantum workspace with the QDK or Azure CLI

If you have an Azure Quantum workspace, you can use the `qdk.azure` Python module the connect to the workspace and submit your code to run on Azure Quantum. The `qdk.azure` module provides a Workspace class that represents an Azure Quantum workspace.

> ⚠ **Warning**
>
> Starting January 11, 2027, Azure Quantum requires users in a Quantum workspace to have the **Quantum Workspace Data Reader** or **Quantum Workspace Data Contributor** role to view job results. The **Quantum Workspace Data Contributor** role will be required to submit jobs. To avoid interruptions to your Azure Quantum workflows, assign these roles to the appropriate users before the deadline. For more information, see **Assign data access roles for users in the workspace**.

Prerequisites

To connect to your workspace with the `qdk.azure` module, you need:

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. If you don't have a workspace, see Create an Azure Quantum workspace.
- The latest version of the `qdk` Python package with the `azure` extra.

  Bash

  ```
  pip install --upgrade "qdk[azure]"
  ```

If you use the Azure CLI, update to the latest version. For installation instructions, see:

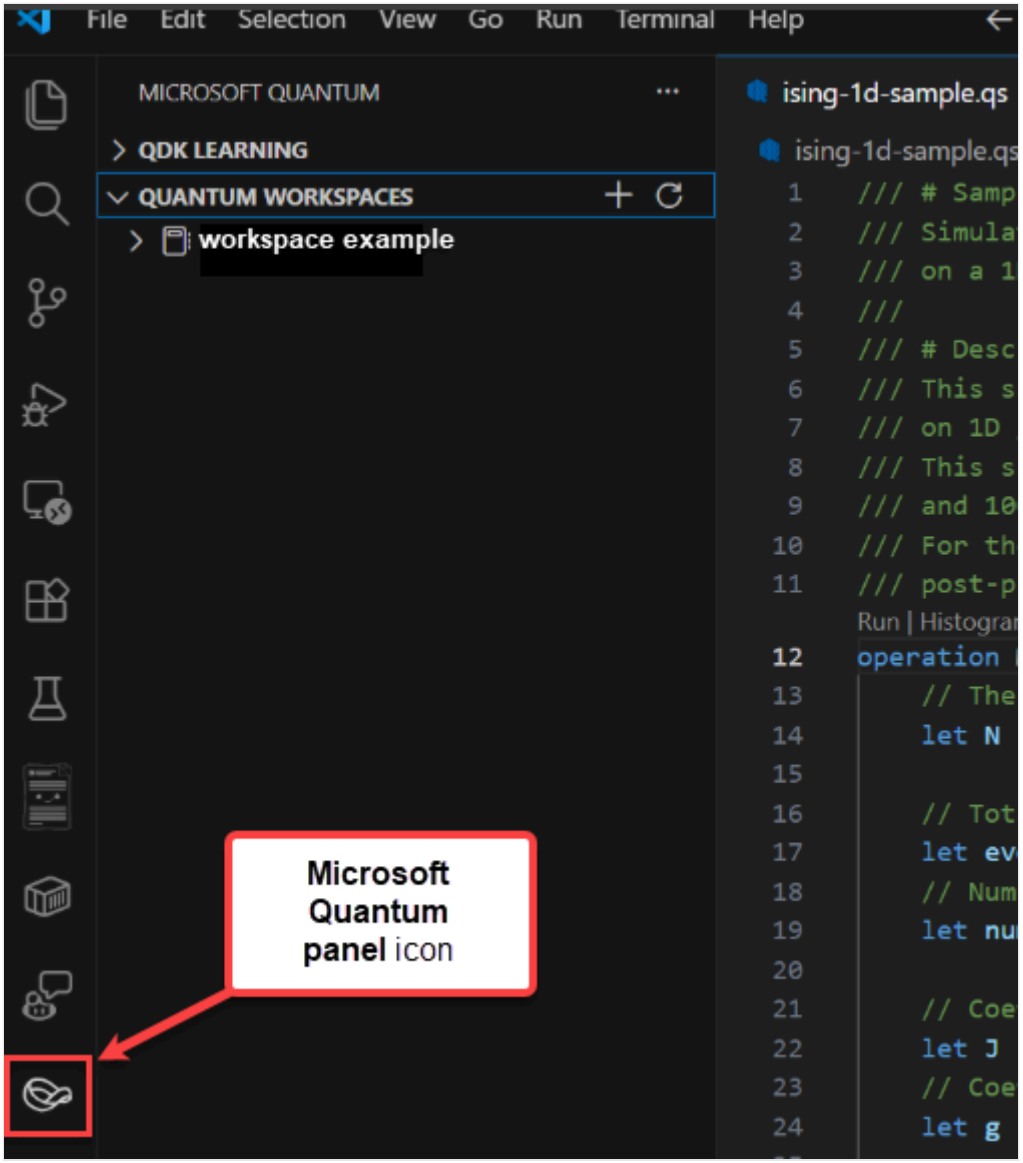
- Install Azure CLI on Windows
- Install Azure CLI on Linux
- Install Azure CLI on macOS

Connect with an Azure account

Use an Azure account to connect to your Azure Quantum workspace with the Microsoft Quantum Development Kit (QDK) extension in Visual Studio Code (VS Code).

1. Open VS Code.
2. Select **View** > **Command Palette**.
3. Enter and select **QDK: Connect to an Azure Quantum workspace**.
4. Choose **Azure account**.
5. When a VS Code message says that QDK wants to sign in, select **Allow**.
6. Select the account to use.
7. Select a tenant and subscription.

To view your Azure Quantum workspace, select the **Microsoft Quantum** icon on the Activity Bar.



Connect with a connection string

> **Note**
>
> Connection strings are less secure than an Azure account.

Use a connection string to specify the connection parameters for an Azure Quantum workspace. Connection strings are useful when:

- You want to share access to your workspace with others who don't have an Azure account.
- You want to share access to your workspace with others for a limited time.
- You can't use Microsoft Entra ID because of company policies.

> **Tip**
>
> Every Azure Quantum workspace has a primary key and a secondary key, and each key has its own connection string. To let others use your workspace, share the secondary key and keep the primary key for your own services. That way, you can replace the secondary key without causing downtime in your services. For more information about sharing access to your workspace, see Share your workspace access.

Copy the connection string

1. Sign in to the Azure portal.
2. Go to your Azure Quantum workspace.
3. In the workspace panel, expand the **Operations** dropdown and select **Access Keys**.
4. If the **Access Keys** slider is set to **Disabled**, set it to **Enabled**.
5. Select the **Copy** icon for the connection string that you want to copy. You can choose either the primary or secondary connection string.

> **Warning**
>
> Storing your account access keys or connection strings in clear text is a security risk. Store your account keys in an encrypted format, or migrate your apps to use Microsoft Entra authorization for your Azure Quantum workspace.

Use the connection string to access your Azure Quantum workspace

Use the connection string you copied to connect to your Azure Quantum workspace with the `qdk.azure` module or the QDK extension in VS Code.

Python

To connect to your Azure Quantum workspace, choose one of the following methods.

- Use the `from_connection_string` function to create a `Workspace` object.

  Python

  ```python
  from qdk.azure import Workspace

  connection_string = "<ConnectionString>" # Replace with your connection string
  workspace = Workspace.from_connection_string(connection_string)

  print(workspace.get_targets())
  ```

- Store your connection string in an environment variable.

  Python

  ```python
  import os
  from qdk.azure import Workspace

  connection_string = "<ConnectionString>" # Replace with your connection string
  os.environ["AZURE_QUANTUM_CONNECTION_STRING"] = connection_string

  workspace = Workspace()
  print(workspace.get_targets())
  ```

For more information about how to work with keys, see Authenticate in your workspace using an access key.

> **ⓘ Important**
>
> If you disable access keys for your workspace, you can't connect with connection strings. Use workspace parameters instead.

Connect to your workspace with workspace parameters

Every Azure Quantum workspace has a unique set of parameters that you use to connect to the workspace.

Expand table

Parameter	Description	Use when you connect with
`subscription_id`	The Azure subscription ID.	Azure CLI
`resource_group`	The Azure resource group name.	Azure CLI
`name`	The name of your Azure Quantum workspace.	Azure CLI
`resource_id`	The Azure resource ID of the Azure Quantum workspace.	Python

To find your workspace parameters, follow these steps:

1. Sign in to the Azure portal.
2. Go to your Azure Quantum workspace.
3. From your workspace panel, select **Overview**.
4. Expand the **Essentials** dropdown.
5. Copy the parameters from their corresponding fields.

> **Note**
>
> Ensure that you sign in to the correct tenant before you connect to your workspace. For more information about tenants, see How to manage Azure subscriptions with the Azure CLI.

Use workspace parameters to connect to your Azure Quantum workspace

To stay signed in when you run your Python scripts, open a terminal and run the following Azure CLI command to set the subscription for your workspace. Replace `<subscriptionId>` with your subscription ID.

Azure CLI

```
az account set --subscription <subscriptionId>
```

Use your workspace parameters to connect to your Azure Quantum workspace through the `qdk.azure` module or the Azure CLI.

Python

To connect to your Azure Quantum workspace, choose one of the following options.

- Specify the resource ID (recommended):

  Python

  ```python
  from qdk.azure import Workspace

  workspace = Workspace(resource_id="") # Add the resource ID of your
  workspace
  ```

- Specify the subscription ID, resource group, and workspace name:

  Python

  ```python
  from qdk.azure import Workspace

  workspace = Workspace(
      subscription_id="", # Add the subscription ID of your workspace
      resource_group="", # Add the resource group of your workspace
      name="" # Add the name of your workspace
      )
  ```

- Specify only the workspace name. This option might fail if you have multiple workspaces with the same name in the same tenant.

  Python

  ```python
  from qdk.azure import Workspace
  workspace = Workspace(name="") # Add the name of your workspace
  ```

Related content

- Manage quantum workspaces with Azure CLI
- Share access to your Azure Quantum workspace
- Manage quantum workspaces with the Azure Resource Manager
- Authenticate using a service principal
- Authenticate using a managed identity

Last updated on 11/06/2026

How to submit specific formatted circuits to Azure Quantum

Learn how to use the `qdk.azure` Python module to submit circuits in specific formats to the Azure Quantum service. This article shows you how to submit circuits in the following formats:

- Submit QIR circuits
- Submit provider-specific circuits

For more information, see Quantum circuits.

Prerequisites

To develop and run your circuits in Visual Studio Code (VS Code), you must have the following:

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. For more information, see Create an Azure Quantum workspace.
- A Python environment with Python and Pip installed.
- VS Code with the Microsoft Quantum Development Kit (QDK), Python, and Jupyter extensions installed.
- The `qdk` Python package with the `azure` extra, and the `ipykernel` package.

  Bash

  ```
  python -m pip install --upgrade "qdk[azure]" ipykernel
  ```

Create a new Jupyter notebook and connect to your Quantum workspace

To connect to your workspace in a Jupyter notebook in VS Code, follow these steps:

1. In VS Code, open the **View** menu and choose **Command Palette**.

2. Enter **Create: New Jupyter Notebook**. An empty Jupyter Notebook file opens in a new tab.

3. In the first cell of the notebook, run the following code. You can find the resource ID in the **Overview** pane for your workspace in the Azure portal.

Python

```
from qdk.azure import Workspace

workspace = Workspace (resource_id="") # Add your resource ID
```

Submit QIR-formatted circuits

Quantum Intermediate Representation (QIR) is an intermediate representation that serves as a common interface between quantum programming languages and targeted quantum computation platforms. For more information, see Quantum Intermediate Representation.

To submit a QIR-formatted circuit, follow these steps:

1. Create the QIR circuit. For example, run the following code in a new cell to create a simple entanglement circuit.

Python

```
QIR_routine = """%Result = type opaque
%Qubit = type opaque

define void @ENTRYPOINT__main() #0 {
  call void @__quantum__qis__h__body(%Qubit* inttoptr (i64 0 to %Qubit*))
  call void @__quantum__qis__cx__body(%Qubit* inttoptr (i64 0 to %Qubit*),
%Qubit* inttoptr (i64 1 to %Qubit*))
  call void @__quantum__qis__h__body(%Qubit* inttoptr (i64 2 to %Qubit*))
  call void @__quantum__qis__cz__body(%Qubit* inttoptr (i64 2 to %Qubit*),
%Qubit* inttoptr (i64 0 to %Qubit*))
  call void @__quantum__qis__h__body(%Qubit* inttoptr (i64 2 to %Qubit*))
  call void @__quantum__qis__h__body(%Qubit* inttoptr (i64 3 to %Qubit*))
  call void @__quantum__qis__cz__body(%Qubit* inttoptr (i64 3 to %Qubit*),
%Qubit* inttoptr (i64 1 to %Qubit*))
  call void @__quantum__qis__h__body(%Qubit* inttoptr (i64 3 to %Qubit*))
  call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 2 to %Qubit*),
%Result* inttoptr (i64 0 to %Result*)) #1
  call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 3 to %Qubit*),
%Result* inttoptr (i64 1 to %Result*)) #1
  call void @__quantum__rt__tuple_record_output(i64 2, i8* null)
  call void @__quantum__rt__result_record_output(%Result* inttoptr (i64 0 to
%Result*), i8* null)
  call void @__quantum__rt__result_record_output(%Result* inttoptr (i64 1 to
%Result*), i8* null)
```

```
  ret void
}

declare void @__quantum__qis__ccx__body(%Qubit*, %Qubit*, %Qubit*)
declare void @__quantum__qis__cx__body(%Qubit*, %Qubit*)
declare void @__quantum__qis__cy__body(%Qubit*, %Qubit*)
declare void @__quantum__qis__cz__body(%Qubit*, %Qubit*)
declare void @__quantum__qis__rx__body(double, %Qubit*)
declare void @__quantum__qis__rxx__body(double, %Qubit*, %Qubit*)
declare void @__quantum__qis__ry__body(double, %Qubit*)
declare void @__quantum__qis__ryy__body(double, %Qubit*, %Qubit*)
declare void @__quantum__qis__rz__body(double, %Qubit*)
declare void @__quantum__qis__rzz__body(double, %Qubit*, %Qubit*)
declare void @__quantum__qis__h__body(%Qubit*)
declare void @__quantum__qis__s__body(%Qubit*)
declare void @__quantum__qis__s__adj(%Qubit*)
declare void @__quantum__qis__t__body(%Qubit*)
declare void @__quantum__qis__t__adj(%Qubit*)
declare void @__quantum__qis__x__body(%Qubit*)
declare void @__quantum__qis__y__body(%Qubit*)
declare void @__quantum__qis__z__body(%Qubit*)
declare void @__quantum__qis__swap__body(%Qubit*, %Qubit*)
declare void @__quantum__qis__mz__body(%Qubit*, %Result* writeonly) #1
declare void @__quantum__rt__result_record_output(%Result*, i8*)
declare void @__quantum__rt__array_record_output(i64, i8*)
declare void @__quantum__rt__tuple_record_output(i64, i8*)

attributes #0 = { "entry_point" "output_labeling_schema"
"qir_profiles"="base_profile" "required_num_qubits"="4"
"required_num_results"="2" }
attributes #1 = { "irreversible" }

; module flags

!llvm.module.flags = !{!0, !1, !2, !3}

!0 = !{i32 1, !"qir_major_version", i32 1}
!1 = !{i32 7, !"qir_minor_version", i32 0}
!2 = !{i32 1, !"dynamic_qubit_management", i1 false}
!3 = !{i32 1, !"dynamic_result_management", i1 false}
"""
```

2. Create a `submit_qir_job` helper function to submit the QIR circuit to a target. In this example, the input and output data formats are `qir.v1` and `microsoft.quantum-results.v1`, respectively.

Python

```python
# Submit the job with proper input and output data formats
def submit_qir_job(target, input, name, count=100):
    job = target.submit(
        input_data=input,
```

```
        input_data_format="qir.v1",
        output_data_format="microsoft.quantum-results.v1",
        name=name,
        input_params = {
            "entryPoint": "ENTRYPOINT__main",
            "arguments": [],
            "count": count
            }
    )

    print(f"Queued job: {job.id}")
    job.wait_until_completed()
    print(f"Job completed with state: {job.details.status}")
    #if job.details.status == "Succeeded":
    result = job.get_results()

    return result
```

3. Submit the QIR circuit to a specific Azure Quantum target. For example, to submit the QIR circuit to the IonQ simulator target, run the following code:

Python

```
target = workspace.get_targets(name="ionq.simulator")
result = submit_qir_job(target, QIR_routine, "QIR routine")
result
```

Submit a circuit with a provider-specific format to Azure Quantum

Each Azure Quantum provider has its own format to represent quantum circuits. You can submit circuits to Azure Quantum in provider-specific formats instead of QIR languages, such as Q# or Qiskit.

- IonQ
- Pasqal
- Quantinuum
- Rigetti

Submit a circuit to IonQ in JSON format

IonQ uses JSON format to run circuits on their targets. For more information, see IonQ targets and the IonQ API documentation.

The following sample creates a superposition between three qubits in JSON format.

1. In a new cell, create a quantum circuit in JSON format.

Python

```
circuit = {
    "qubits": 3,
    "circuit": [
        {
        "gate": "h",
        "target": 0
        },
        {
        "gate": "cnot",
        "control": 0,
        "target": 1
        },
        {
        "gate": "cnot",
        "control": 0,
        "target": 2
        },
    ]
}
```

2. Submit the circuit to the IonQ target. The following example uses the IonQ simulator, which returns a `Job` object.

Python

```
target = workspace.get_targets(name="ionq.simulator")
job = target.submit(circuit)
```

3. When the job completes, get the results.

Python

```
results = job.get_results()
print(results)
```

Submit a circuit to Pasqal in Pulser SDK format

You can use the Pulser SDK to create pulse sequences and submit them to Pasqal targets.

Install the Pulser SDK

Pulser is a framework that allows you to create, simulate, and run pulse sequences for neutral-atom quantum devices. Pulser is designed by PASQAL as a pass-through to submit quantum

experiments to their quantum processors. For more information, see the Pulser documentation.

To submit the pulse sequences, first install the Pulser SDK packages:

Python

```
try:
    import pulser
    import pulser_azure
except ImportError:
    !pip -q install pulser pulser-azure --index-url https://pypi.org/simple
```

Create a quantum register

Define both a register and a layout. The register specifies where to arrange the atoms, and the layout specifies the positions of traps that capture and structure the atoms within the register.

For details on layouts, see the Pulser documentation.

Create a `devices` object to import the Pasqal quantum computer target, FRESNEL_CAN1.

Python

```
from pulser_azure import AzureConnection

connection = AzureConnection(resource_id="") # Add your resource ID
devices = connection.fetch_available_devices()
QPU = devices["pasqal.qpu.fresnel-can1"]
```

Set up your layout

Use a custom layout when the pre-calibrated layouts don't satisfy the requirements of your experiment.

For a given arbitrary register, a neutral-atom QPU places traps according to the layout, which must then be calibrated. Because each calibration takes time, it's a best practice to reuse an existing calibrated layout when possible.

To create an arbitrary layout, choose one of the following options:

- Automatically generate a layout based on a specified register. For large registers, this process can produce sub-optimal solutions. For example:

  Python

```
from pulser import Register
qubits = {
    "q0": (0, 0),
    "q1": (0, 10),
    "q2": (8, 2),
    "q3": (1, 15),
    "q4": (-10, -3),
    "q5": (-8, 5),
}

reg = Register(qubits).with_automatic_layout(QPU)
```

- To manually define a layout to create your register, see the Pulser documentation.

Write a pulse sequence

Neutral atoms are controlled with laser pulses. The Pulser SDK allows you to create pulse sequences to apply to the quantum register.

1. Define the pulse sequence attributes by declaring the channels that control the atoms. To create a `Sequence`, provide a `Register` instance along with the device where the sequence will be executed. For example, the following code declares one channel: `ch0`.

Python

```
from pulser import Sequence

seq = Sequence(reg, QPU)

# Print the available channels for your sequence
print(seq.available_channels)

# Declare a channel. For example, `rydberg_global`
seq.declare_channel("ch0", "rydberg_global")
```

> **Note**
>
> You can use the `QPU = devices["pasqal.qpu.fresnel-can1"]` device or import a virtual device from Pulser for more flexibility. The use of a `VirtualDevice` allows for sequence creation that's less constrained by device specifications, which lets you run on an emulator. For more information, see Pulser documentation.

2. Add pulses to your sequence. To do so, create and add pulses to the channels that you declared. For example, the following code creates a pulse and adds it to channel `ch0`:

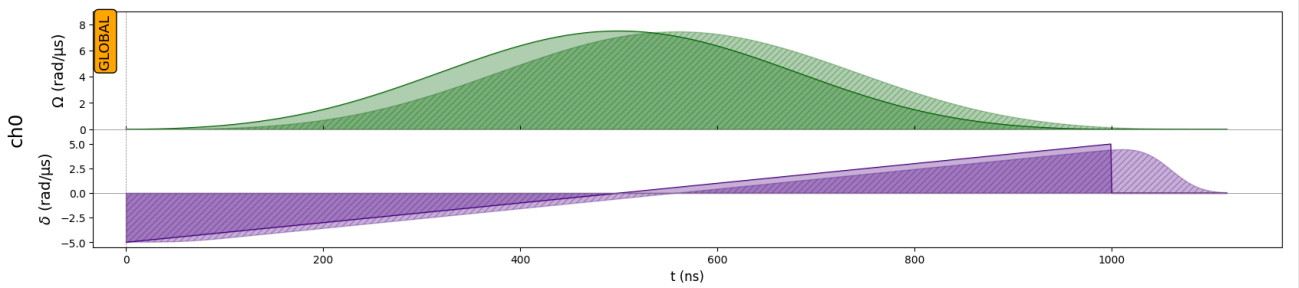
Python

```
from pulser import Pulse
from pulser.waveforms import RampWaveform, BlackmanWaveform
import numpy as np

amp_wf = BlackmanWaveform(1000, np.pi)
det_wf = RampWaveform(1000, -5, 5)
pulse = Pulse(amp_wf, det_wf, 0)
seq.add(pulse, "ch0")

seq.draw()
```

The following image shows the pulse sequence:



Convert the sequence to a JSON string

To submit the pulse sequences, convert the Pulser objects into a JSON string that can be used as input data.

Python

```
import json

# Convert the sequence to a JSON string
def prepare_input_data(seq):
    input_data = {}
    input_data["sequence_builder"] = json.loads(seq.to_abstract_repr())
    to_send = json.dumps(input_data)
    return to_send
```

Submit the pulse sequence to Pasqal target

1. Set the proper input and output data formats. For example, the following code sets the input data format to `pasqal.pulser.v1` and the output data format to `pasqal.pulser-results.v1`.

Python

```
# Submit the job with proper input and output data formats
def submit_job(target, seq, shots):
    job = target.submit(
        input_data=prepare_input_data(seq), # Take the JSON string previously
defined as input data
        input_data_format="pasqal.pulser.v1",
        output_data_format="pasqal.pulser-results.v1",
        name="Pasqal sequence",
        shots=shots # Number of shots
    )

    print(f"Queued job: {job.id}")
    return job
```

> **Note**
>
> The time required to run a job on the QPU depends on current queue times. You can view the average queue time for a target in the **Providers** pane of your workspace.

2. Submit the program to Pasqal. Before you submit your code to real quantum hardware, it's a best practice to test your code on the emulator `pasqal.sim.emu-mps` target.

Python

```
target = workspace.get_targets(name="pasqal.sim.emu-mps") # Change to
"pasqal.qpu.fresnel-can1" to use FRESNEL_CAN1 QPU
job = submit_job(target, seq, 10)

job.wait_until_completed()
print(f"Job completed with state: {job.details.status}")
result = job.get_results()
print(result)
```

Output

```
{
    "counter": {
        "1000000": 8,
        "0010000": 1,
        "0010101": 1
    }
}
```

Use pulser backends to execute the sequence

With `AzureConnection` from pulser-azure, you can directly use a QPU backend or a remote emulator backend to execute the sequence.

Python

```python
from pulser.backends import RemoteMPSBackend, QPUBackend

backend = RemoteMPSBackend(sequence=seq, connection=connection)  # Replace
RemoteMPSBackend with QPUBackend to execute the sequence on the QPU
results = backend.run(job_params=[{"runs": 10}], wait=True)

print(results.results[0].final_bitstrings)
```

Output

```
Counter({"1000000": 8, "0010000": 1, "0010101": 1})
```

Submit an OpenQASM circuit to Quantinuum

1. Create a quantum circuit in the OpenQASM representation. For example, the following code creates a Teleportation circuit:

   py

   ```
   circuit = """OPENQASM 2.0;
   include "qelib1.inc";
   qreg q[3];
   creg c0[3];
   h q[0];
   cx q[0], q[1];
   cx q[1], q[2];
   measure q[0] -> c0[0];
   measure q[1] -> c0[1];
   measure q[2] -> c0[2];
   """
   ```

   Or, load the circuit from an OpenQASM file:

   py

   ```
   with open("my_teleport.qasm", "r") as f:
       circuit = f.read()
   ```

2. Submit the circuit to a Quantinuum target. The following example submits the job to one of the Quantinuum simulator targets.

Python

```
target = workspace.get_targets(name="quantinuum.sim.h2-1sc")
job = target.submit(circuit, shots=500)
```

3. Wait until the job is complete and then fetch the results.

Python

```
results = job.get_results()
print(results)
```

> **Note**
>
> These results return 000 for every shot, which isn't random. This is because the API Validator only checks whether your code can run on Quantinuum hardware, but returns 0 for every quantum measurement. For a true random number generator, you need to run your circuit on quantum hardware.

Submit a Quil circuit to Rigetti

To submit a Quil job to a Rigetti target, use the `qdk.azure` Python module.

1. Load the required imports.

Python

```
from azure.quantum import Workspace
from azure.quantum.target.rigetti import Result, Rigetti, RigettiTarget,
InputParams
```

2. Create a `target` object and pass the name of the Rigetti target that you want to submit your job to. For example, the following code selects the `QVM` target.

Python

```
target = Rigetti(workspace=workspace, name=RigettiTarget.QVM)
```

3. Create a Quil program. For your program to be accepted, you must set the readout to `"ro"`.

Python

```
readout = "ro"
bell_state_quil = f"""
DECLARE {readout} BIT[2]

H 0
CNOT 0 1

MEASURE 0 {readout}[0]
MEASURE 1 {readout}[1]
"""

num_shots = 5
job = target.submit(
    input_data=bell_state_quil,
    name="bell state",
    shots=100,
    input_params=InputParams(skip_quilc=False)
)

print(f"Job completed with state: {job.details.status}")
result = Result(job)  # This throws an exception if the job failed
```

4. You can index a Result with the name of the readout. In the following code, `data_per_shot` is a list of length `num_shots`, and each item in the list is another list that contains the data for the register from that shot.

Python

```
data_per_shot = result[readout]

ro_data_first_shot = data_per_shot[0]
```

In this case, because the type of the register is BIT, the type is integer and the value either 0 or 1.

Python

```
assert isinstance(ro_data_first_shot[0], int)
assert ro_data_first_shot[0] == 1 or ro_data_first_shot[0] == 0
```

5. Print out all the data.

Python

```
print(f"Data from '{readout}' register:")
for i, shot in enumerate(data_per_shot):
    print(f"Shot {i}: {shot}")
```

> **Important**
>
> You can't submit multiple circuits on a single job. As a workaround you can call the `backend.run` method to submit each circuit asynchronously, and then fetch the results of each job. For example:

Python

```
jobs = []
for circuit in circuits:
    jobs.append(backend.run(circuit, shots=N))

results = []
for job in jobs:
    results.append(job.result())
```

Related content

- How to submit jobs to Azure Quantum
- Work with Azure Quantum jobs

Last updated on 08/29/2026

What is hybrid quantum computing?

Article • 08/06/2024

Hybrid quantum computing refers to the processes and architecture of a classical computer and a quantum computer working together to solve a problem. With the latest generation of hybrid quantum computing architecture available in Azure Quantum you can start programming quantum computers by mixing classical and quantum instructions together.

Azure Quantum embodies a forward-looking vision for hybrid quantum computing, where certain architectures are already operational, while others are actively being developed. This article describes the different approaches to hybrid quantum computing and how they can be used to optimize certain problems.

Grouping circuits with batch quantum computing

Batch quantum computing allows you to submit multiple quantum circuits as a single job to the quantum hardware.

Typically, quantum circuits are sent one at a time as single jobs to a quantum hardware target. When the client receives the result of one circuit, the next circuit is added as a new job to the queue. Batching multiple circuits into one job, however, eliminates the wait between job submissions, allowing you to run multiple jobs faster. Examples of problems that can take advantage of batch quantum computing include Shor's algorithm and simple quantum phase estimation.

With the batch computing model, you can also batch multiple predefined circuits into one job. The circuits are submitted to the quantum hardware as soon as the previous circuit is complete, reducing the wait between job submissions.

In this architecture, the state of the qubits is lost between each circuit submission.

> **Note**
>
> Azure Quantum currently doesn't support batch quantum computing.

Grouping jobs with sessions

Sessions allow you to organize multiple quantum computing jobs with the ability to run classical code between quantum jobs. You'll be able to run complex algorithms to better organize and track your individual quantum computing jobs. Plus, jobs grouped in sessions are prioritized over non-session jobs.

In this model, the client compute resource is moved to the cloud, resulting in lower latency and repeated execution of the quantum circuit with different parameters. Although sessions allow for shorter queue times and longer running problems, the qubit states don't persist between each iteration. Examples of problems that can use this approach are Variational Quantum Eigensolvers (VQE) and Quantum Approximate Optimization Algorithms (QAOA).

For more information, see Get started with sessions.

Running hybrid quantum computing

With hybrid quantum computing, the classical and quantum architectures are tightly coupled, allowing classical computations to be performed while physical qubits are coherent. Though limited by qubit life and error correction, this allows for quantum programs to move away from just circuits. Programs can now use common programming constructs to perform mid-circuit measurements, optimize and reuse qubits, and adapt in real-time to the QPU. Examples of scenarios that can take advantage of this model are adaptive phase estimation and machine learning.

For more information, see Integrated quantum computing.

Running distributed quantum computing

In this architecture, classical computation is working alongside logical qubits. With fully integrated classical control and longer lived logical qubits, the distributed quantum computing model enables real-time computations across quantum and distributed resources. The classical controls are no longer limited to loops, and allow for scenarios such as complex materials modeling or the evaluation of full catalytic reactions.

> **Note**
>
> Azure Quantum currently doesn't support distributed quantum computing.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Run hybrid quantum computing jobs with an adaptive target profile

Hybrid computing combines classical and quantum computing processes to solve complex problems.

In hybrid computing, the classical code controls the execution of quantum operations based on mid-circuit measurements while the physical qubits remain active. You can use common programming techniques, like nested conditionals, loops, and function calls, in a single quantum program to run complex programs with fewer shots. With qubit reuse techniques, larger programs can run on machines using a smaller number of qubits.

This article explains how to submit hybrid jobs to Azure Quantum using the Adaptive RI target profile. The Adaptive RI target profile supports mid-circuit measurements, measurement-based control flow, qubit reset, and classical integer computation.

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure account, register for free and sign up for a pay-as-you-go subscription.
- An Azure Quantum workspace. For more information, see Create an Azure Quantum workspace.

To submit Q# standalone programs, you also need the following prerequisites:

- Install Visual Studio Code (VS Code).
- Install the latest version of the Microsoft Quantum Development Kit (QDK) extension.

To submit Python and Q# programs, you also need the following prerequisites:

- A Python environment with Python and Pip installed.
- The latest `qdk` Python package with the optional `azure` extra.

  Bash

  ```
  pip install --upgrade "qdk[azure]"
  ```

Supported targets

To run hybrid quantum computing jobs, select a quantum provider that supports the Adaptive RI target profile.

Currently, Quantinuum targets support the Adaptive RI target profile in Azure Quantum.

Submitting adaptive RI jobs

To submit hybrid quantum computing jobs, set the target profile to **Adaptive RI**, where RI stands for "qubit Reset and Integer computations".

The Adaptive RI target profile supports mid-circuit measurements, measurement-based control flow, qubit reset, and classical integer computation.

Submit hybrid quantum jobs to Azure Quantum as Q# standalone programs or Python + Q# programs. To set the target profile for hybrid quantum jobs, see the following sections.

Q# in Visual Studio Code

To set the target profile for hybrid jobs in VS Code, choose one of the following options:

- If your Q# file isn't part of a Q# project, open the file and enter `@EntryPoint(Adaptive_RI)` before your program's entry point operation.
- If your Q# file is part of a Q# project, add the following code to your project's `qsharp.json` file:

  JSON

  ```
  {
    "targetProfile": "adaptive_ri"
  }
  ```

After you set your target profile to **Adaptive RI**, follow these steps to submit your Q# program as a hybrid quantum job to Quantinuum:

1. In VS Code, select **View** > **Command Palette**
2. Enter **QDK: Connect to an Azure Quantum workspace**.
3. Select **Azure account**, and follow the prompts to connect to your preferred directory, subscription, and workspace.
4. To view your Azure Quantum workspaces, select the **Microsoft Quantum** icon on the Activity Bar.
5. Expand your workspace and expand the **Quantinuum** provider.

6. Select an available Quantinuum target, like **quantinuum.sim.h2-1e**.
7. Select the play icon next to the target name to submit the current Q# program.
8. Enter a name to identify the job, and the number of shots.
9. Submit the job. The job status shows at the bottom of the screen.
10. Expand **Jobs** and hover over your job to see its times and status.

Supported features

The following table lists the supported features for hybrid quantum computing with Quantinuum in Azure Quantum.

Expand table

Supported feature	Notes
Dynamic values	Bools and integers whose values depend on a measurement result
Loops	Classically bounded loops only
Arbitrary control flow	Use of if/else branching
Mid-circuit measurement	Utilizes classical register resources
Qubit reuse	Supported
Real-time classical compute	64-bit signed integer arithmetic, utilizes classical register resources

The QDK provides target-specific feedback when Q# language features aren't supported for the selected target. If your Q# program has unsupported features when you run hybrid quantum jobs, you get an error message at design time. For more information, see the QIR wiki page.

> **Note**
>
> Select the **Adaptive RI** target profile to get feedback when you use Q# features that the target doesn't support.

To see the supported features in action, copy the following code into a Q# file and add the subsequent code snippets.

Q#

```
import Std.Measurement.*;
import Std.Math.*;
import Std.Convert.*;

operation Main() : Result {
    use (q0, q1) = (Qubit(), Qubit());
    H(q0);
    let r0 = MResetZ(q0);

    // Copy here the code snippets below to see the supported features
    // in action.
    // Supported features include dynamic values, classically-bounded loops,
    // arbitrary control flow, and mid-circuit measurement.

    r0
}
```

Quantinuum supports dynamic bools and integers, which are bools and integers that depend on measurement results. `r0` is a `Result` type that you can use to generate dynamic bool and integer values.

Q#

```
    let dynamicBool = r0 != Zero;
    let dynamicBool = ResultAsBool(r0);
    let dynamicInt = dynamicBool ? 0 | 1;
```

Quantinuum supports dynamic bools and integers, but it doesn't support dynamic values for other data types, like double. Copy the following code to see feedback about the limitations of dynamic values.

Q#

```
    let dynamicDouble = r0 == One ? 1. | 0.; // cannot use a dynamic double value
    let dynamicInt = r0 == One ? 1 | 0;
    let dynamicDouble = IntAsDouble(dynamicInt); // cannot use a dynamic double value
    let dynamicRoot = Sqrt(dynamicDouble); // cannot use a dynamic double value
```

Even though some data types don't support dynamic values, you can still use those data types with static values.

Q#

```
    let staticRoot = Sqrt(4.0);
    let staticBigInt = IntAsBigInt(2);
```

You can't use dynamic values of supported types in certain situations. For example, Quantinuum doesn't support dynamic arrays, which are arrays whose size depends on a measurement result. Quantinuum doesn't support dynamically bounded loops either. Copy the following code to see the limitations of dynamic values.

Q#

```
    let dynamicInt = r0 == Zero ? 2 | 4;
    let dynamicallySizedArray = [0, size = dynamicInt]; // cannot use a dynamically-
sized array
    let staticallySizedArray = [0, size = 10];
    // Loops with a dynamic condition are not supported by Quantinuum.
    for _ in 0..dynamicInt {
         Rx(PI(), q1);
    }

    // Loops with a static condition are supported.
    let staticInt = 3;
    for _ in 0..staticInt {
         Rx(PI(), q1);
    }
```

Quantinuum supports control flow, including `if/else` branching, with both static and dynamic conditions. Branching on dynamic conditions is also called branching based on measurement results.

Q#

```
    let dynamicInt = r0 == Zero ? 0 | 1;
    if dynamicInt > 0 {
        X(q1);
     }
    let staticInt = 1;
    if staticInt > 5 {
         Y(q1);
    } else {
         Z(q1);
    }
```

Quantinuum supports loops that have classical conditions and include `if` expressions.

Q#

```
    for idx in 0..3 {
        if idx % 2 == 0 {
            Rx(ArcSin(1.), q0);
            Rz(IntAsDouble(idx) * PI(), q1)
        } else {
```

```
            Ry(ArcCos(-1.), q1);
             Rz(IntAsDouble(idx) * PI(), q1)
        }
     }
```

Quantinuum supports mid-circuit measurement, that is, branching based on measurement results.

Q#

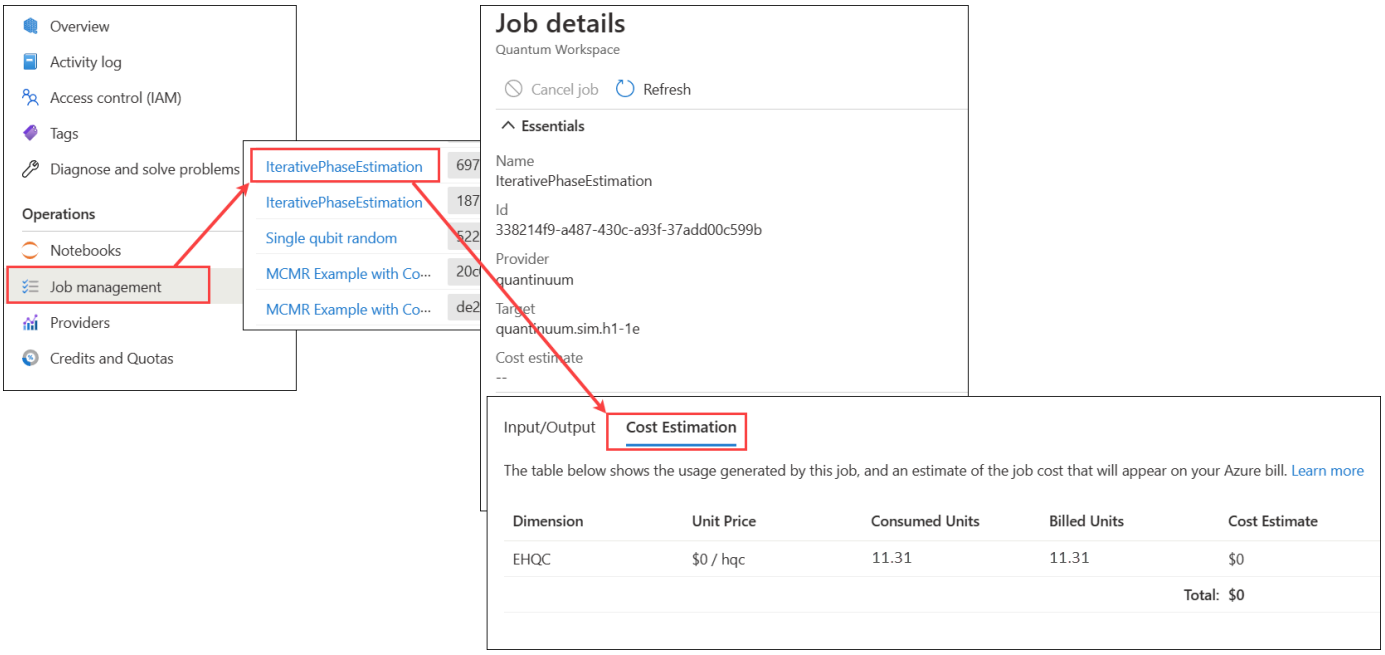
```
    if r0 == One {
        X(q1);
    }
    let r1 = MResetZ(q1);
    if r0 != r1 {
        let angle = PI() + PI() + PI()* Sin(PI()/2.0);
        Rxx(angle, q0, q1);
    } else {
        Rxx(PI() + PI() + 2.0 * PI() * Sin(PI()/2.0), q1, q0);
    }
```

Estimate the cost of a hybrid quantum computing job

Estimate the cost to run a hybrid quantum computing job on Quantinuum hardware by running the job on an emulator first.

After a successful run on the emulator:

1. In your Azure Quantum workspace, select **Job management**.
2. Select the job you submitted.
3. In the **Job details** popup, select **Cost Estimation** to view how many eHQCs (Quantinuum emulator credits) were used. This number corresponds to the number of HQCs (Quantinuum quantum credits) that you need to run the job on Quantinuum hardware.



> **Note**
>
> Quantinuum unrolls the entire circuit and calculates the cost on all code paths, whether or not they run conditionally.

Hybrid quantum computing samples

Find the following samples in the Q# code samples repository. These samples demonstrate the current feature set for hybrid quantum computing.

Three-qubit repetition code

This sample demonstrates how to create a three-qubit repetition code that detects and corrects bit flip errors.

It uses integrated hybrid computing features to count how many times error correction runs while the state of a logical qubit register is coherent.

Find the code in the three-qubit repetition code sample.

Iterative phase estimation

This sample program demonstrates iterative phase estimation in Q#. It uses iterative phase estimation to calculate an inner product between two 2-dimensional vectors encoded on a target qubit and an ancilla qubit. The program also initializes a control qubit, which is the only qubit used for measurement.

The circuit begins by encoding the pair of vectors on the target qubit and the ancilla qubit. It then applies an Oracle operator to the entire register, controlled off the control qubit, which is set up in the $|+\rangle$ state. The controlled Oracle operator generates a phase on the $|1\rangle$ state of the control qubit. To read this phase, apply an H gate to the control qubit, which makes the phase observable during measurement.

Find the code in the dot product via phase estimation sample.

> **Note**
>
> Members of the KPMG Quantum team in Australia wrote this sample code, which falls under an MIT License. The sample demonstrates expanded capabilities of Adaptive RI targets and uses bounded loops, classical function calls at run time, nested conditional `if` statements, mid-circuit measurements, and qubit reuse.

Related content

- Get started with sessions

Last updated on 11/06/2026

Introduction to the quantum programming language Q#

👉 Learn more about the Microsoft Quantum platform

Q# is a high-level, open-source programming language developed by Microsoft for writing quantum programs. Q# is included in the Microsoft Quantum Development Kit (QDK). For more information, see Set up the Microsoft Quantum Development Kit.

As a quantum programming language, Q# meets the following requirements for language, compiler, and runtime:

- **Hardware agnostic:** Qubits in quantum algorithms aren't tied to a specific quantum hardware or layout. The Q# compiler and runtime handle the mapping from program qubits to physical qubits, allowing the same code to run on different quantum processors.
- **Integration of quantum and classical computing:** Q# allows for the integration of quantum and classical computations, which is essential for universal quantum computing.
- **Qubit management:** Q# provides built-in operations and functions for managing qubits, including creating superposition states, entangling qubits, and performing quantum measurements.
- **Respect the laws of physics:** Q# and quantum algorithms must follow the rules of quantum physics. For example, you can't directly copy or access the qubit state in Q#.

For more information about the origins of Q#, see the blog post Why do we need Q#?.

Structure of a Q# program

Before you start writing Q# programs, it's important to understand their structure and components. Consider the following Q# program, named `Superposition`, that creates a superposition state:

Q#

```
namespace Superposition {
    @EntryPoint()
    operation MeasureOneQubit() : Result {
        // Allocate a qubit. By default, it's in the 0 state.
        use q = Qubit();
        // Apply the Hadamard operation, H, to the state.
        // It now has a 50% chance of being measured as 0 or 1.
```

```
        H(q);
        // Measure the qubit in the Z-basis.
        let result = M(q);
        // Reset the qubit before releasing it.
        Reset(q);
        // Return the result of the measurement.
        return result;
    }
}
```

Based on the comments (`//`), the Q# program first allocates a qubit, applies an operation to put the qubit in superposition, measures the qubit state, resets the qubit, and finally returns the result.

Let's break this Q# program down into its components.

User namespaces

Q# programs can optionally start with a user-defined namespace, such as:

Q#

```
namespace Superposition {
    // Your code goes here.
}
```

Namespaces can help you organize related functionality. Namespaces are optional in Q# programs, meaning that you can write a program without defining a namespace.

For example, the `Superposition` program from the example can also be written without a namespace as:

Q#

```
@EntryPoint()
operation MeasureOneQubit() : Result {
    // Allocate a qubit. By default, it's in the 0 state.
    use q = Qubit();
    // Apply the Hadamard operation, H, to the state.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q);
    // Measure the qubit in the Z-basis.
    let result = M(q);
    // Reset the qubit before releasing it.
    Reset(q);
    // Return the result of the measurement.
```

```
    return result;
}
```

> ⓘ Note
>
> Each Q# program can have only one `namespace`. If you don't specify a namespace, the Q# compiler uses the filename as the namespace.

Entry points

Every Q# program must have an entry point, which is the starting point of the program. By default, the Q# compiler starts executing a program from the `Main()` operation, if available, which can be located anywhere in the program. Optionally, you can use the `@EntryPoint()` attribute to specify any operation in the program as the point of execution.

For example, in the `Superposition` program, the `MeasureOneQubit()` operation is the entry point of the program because it has the `@EntryPoint()` attribute before the operation definition:

Q#

```
@EntryPoint()
operation MeasureOneQubit() : Result {
    ...
}
```

However, the program could also be written without the `@EntryPoint()` attribute by renaming the `MeasureOneQubit()` operation to `Main()`, such as:

Q#

```
// The Q# compiler automatically detects the Main() operation as the entry point.

operation Main() : Result {
    // Allocate a qubit. By default, it's in the 0 state.
    use q = Qubit();
    // Apply the Hadamard operation, H, to the state.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q);
    // Measure the qubit in the Z-basis.
    let result = M(q);
    // Reset the qubit before releasing it.
    Reset(q);
    // Return the result of the measurement.
```

```
    return result;
}
```

Types

Types are essential in any programming language because they define the data that a program can work with. Q# provides built-in types that are common to most languages, including `Int`, `Double`, `Bool`, and `String`, and types that define ranges, arrays, and tuples.

Q# also provides types that are specific to quantum computing. For example, the `Result` type represents the result of a qubit measurement and can have two values: `Zero` or `One`.

In the `Superposition` program, the `MeasureOneQubit()` operation returns a `Result` type, which corresponds to the return type of the `M` operation. The measurement result is stored in a new variable that's defined using the `let` statement:

Q#

```
// The operation definition returns a Result type.
operation MeasureOneQubit() : Result {
    ...
    // Measure the qubit in the Z-basis, returning a Result type.
    let result = M(q);
    ...
}
```

Another example of a quantum-specific type is the `Qubit` type, which represents a quantum bit.

Q# also allows you to define your own custom types. For more information, see Type declarations.

Allocating qubits

In Q#, you allocate qubits using the `use` keyword and the `Qubit` type. Qubits are always allocated in the $|0\rangle$ state.

For example, the `Superposition` program defines a single qubit and stores it in the variable `q`:

Q#

```
// Allocate a qubit.
use q = Qubit();
```

You can also allocate multiple qubits and access each one through its index:

Q#

```
use qubits = Qubit[2]; // Allocate two qubits.
H(qubits[0]); // Apply H to the first qubit.
X(qubits[1]); // Apply X to the second qubit.
```

Quantum operations

After allocating a qubit, you can pass it to operations and functions. Operations are the basic building blocks of a Q# program. A Q# operation is a quantum subroutine, or a callable routine that contains quantum operations that change the state of the qubit register.

To define a Q# operation, you specify a name for the operation, its inputs, and its output. In the `Superposition` program, the `MeasureOneQubit()` operation takes no parameters and returns a `Result` type:

Q#

```
operation MeasureOneQubit() : Result {
    ...
}
```

Here's a basic example that takes no parameters and expects no return value. The `Unit` value is equivalent to `NULL` in other languages:

Q#

```
operation SayHelloQ() : Unit {
    Message("Hello quantum world!");
}
```

The Q# standard library also provides operations that you can use in quantum programs, such as the Hadamard operation, `H`, in the `Superposition` program. Given a qubit in the Z-basis, `H` puts the qubit into an even superposition, where it has a 50% chance of being measured as `Zero` or `One`.

Measuring qubits

While there are many types of quantum measurements, Q# focuses on projective measurements on single qubits, also known as Pauli measurements.

In Q#, the `Measure` operation measures one or more qubits in the specified Pauli basis, which can be `PauliX`, `PauliY`, or `PauliZ`. `Measure` returns a `Result` type of either `Zero` or `One`.

To implement a measurement in the computational basis $\{|0\rangle, |1\rangle\}$, you can also use the `M` operation, which measures a qubit in the Pauli Z-basis. This makes `M` equivalent to `Measure([PauliZ], [qubit])`.

For example, the `Superposition` program uses the `M` operation:

Q#

```
// Measure the qubit in the Z-basis.
let result = M(q);
```

Resetting qubits

In Q#, qubits **must** be in the $|0\rangle$ state when they're released to avoid errors in the quantum hardware. You can reset a qubit to the $|0\rangle$ state using the `Reset` operation at the end of the program. Failure to reset a qubit results in a runtime error.

Q#

```
// Reset a qubit.
Reset(q);
```

Standard library namespaces

The Q# standard library has built-in namespaces that contain functions and operations you can use in quantum programs. For example, the `Std.Intrinsic` namespace contains commonly used operations and functions, such as `M` to measure results and `Message` to display user messages anywhere in the program.

To call a function or operation, you can specify the full namespace or use an `import` statement, which makes all the functions and operations for that namespace available and makes your code more readable. The following examples call the same operation:

Q#

```
Std.Intrinsic.Message("Hello quantum world!");
```

Q#

```
// imports all functions and operations from the Std.Intrinsic namespace.
import Std.Intrinsic.*;
Message("Hello quantum world!");
```

```
// imports just the `Message` function from the Std.Intrinsic namespace.
import Std.Intrinsic.Message;
Message("Hello quantum world!");
```

> **Note**
>
> The `Superposition` program doesn't have any `import` statements or calls with full namespaces. That's because the Q# development environment automatically loads two namespaces, `Std.Core` and `Std.Intrinsic`, which contain commonly used functions and operations.

You can take advantage of the `Std.Measurement` namespace by using the `MResetZ` operation to optimize the `Superposition` program. `MResetZ` combines the measurement and reset operations into one step, as in the following example:

Q#

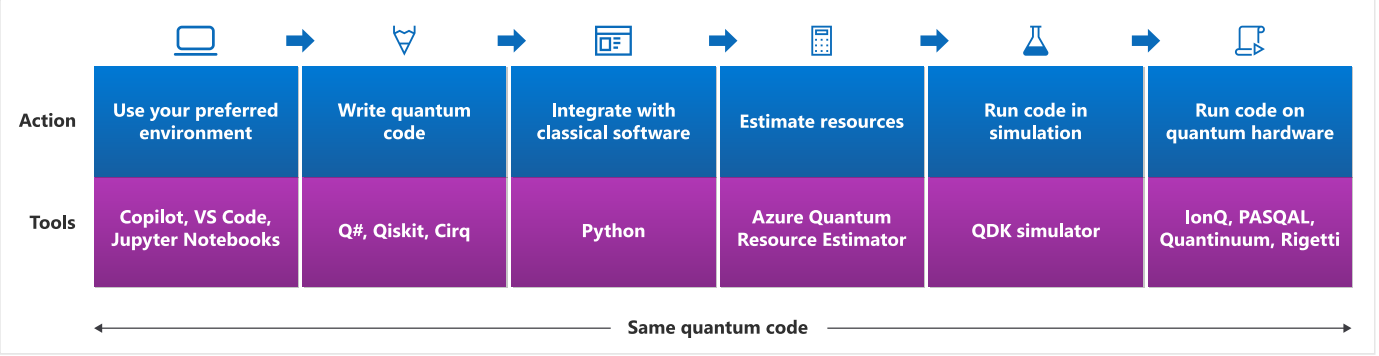
```
// Import the namespace for the MResetZ operation.
import Std.Measurement.*;

@EntryPoint()
operation MeasureOneQubit() : Result {
    // Allocate a qubit. By default, it's in the 0 state.
    use q = Qubit();
    // Apply the Hadamard operation, H, to the state.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q);
    // Measure and reset the qubit, and then return the result value.
    return MResetZ(q);
}
```

Learn to develop quantum programs with Q# and Azure Quantum

Q# and Azure Quantum are a powerful combination for developing and running quantum programs. With Q# and Azure Quantum, you can write quantum programs, simulate their behavior, estimate resource requirements, and run them on real quantum hardware. This integration allows you to explore the potential of quantum computing and develop innovative solutions for complex problems. Whether you are a beginner or an experienced quantum developer, Q# and Azure Quantum provide the tools and resources you need to unlock the power of quantum computing.

The following diagram shows the stages through which a quantum program passes when you develop it with Q# and Azure Quantum. Your program starts with the development environment and ends with the submission of the job to real quantum hardware.



Let's break down the steps in the diagram.

Choose your development environment

Run your quantum programs in your preferred development environment. You can use a local development environment with Visual Studio Code, or combine Q# code with Python code in Jupyter Notebook. For more information, see Different ways to run Q# programs.

Write your quantum program

You can write quantum programs in Q# using the QDK. To get started, see Quickstart: Create your first Q# program.

Besides Q#, the QDK offers support for other languages for quantum computing, such as Qiskit and Cirq.

Integrate with Python

You can use Q# by itself or together with Python in various IDEs. For example, you can use a Q# project with a Python host program to call Q# operations or integrate Q# with Python in Jupyter Notebooks. For more information, see Integration of Q# and Python.

The %%qsharp command

By default, Q# programs in Jupyter Notebooks use the `ipykernel` Python package. To add Q# code to a notebook cell, use the `%%qsharp` command, which is enabled with the `qsharp` Python package, followed by your Q# code.

When using `%%qsharp`, keep the following in mind:

- You must first run `from qdk import qsharp` to enable `%%qsharp`.
- `%%qsharp` scopes to the notebook cell in which it appears and changes the cell type from Python to Q#.
- You can't put a Python statement before or after `%%qsharp`.
- Q# code that follows `%%qsharp` must adhere to Q# syntax. For example, use `//` instead of `#` to denote comments and `;` to end code lines.

Estimate resources

Before running on real quantum hardware, you need to figure out whether your program can run on existing hardware, and how many resources it'll consume.

The Microsoft Quantum resource estimator allows you to assess architectural decisions, compare qubit technologies, and determine the resources needed to execute a given quantum algorithm. You can choose from pre-defined fault-tolerant protocols and specify assumptions of the underlying physical qubit model.

For more information, see Run your first resource estimate.

> **Note**
>
> The Microsoft Quantum resource estimator is free of charge and doesn't require an Azure account.

Run your program in simulation

When you compile and run a quantum program, the QDK creates an instance of the quantum simulator and passes the Q# code to it. The simulator uses the Q# code to create qubits (simulations of quantum particles) and apply transformations to modify their state. The results of the quantum operations in the simulator are then returned to the program. Isolating the Q# code in the simulator ensures that the algorithms follow the laws of quantum physics and can run correctly on quantum computers.

Submit your program to real quantum hardware

Once you've tested your program in simulation, you can run it on real quantum hardware. When you run a quantum program in Azure Quantum, you create and run a **job**. To submit a job to the Azure Quantum providers, you need an Azure account and quantum workspace. If you don't have a quantum workspace, see Create an Azure Quantum workspace.

Azure Quantum offers some of the most compelling and diverse quantum hardware. See Quantum computing providers for the current list of supported hardware providers.

Once you submit your job, Azure Quantum manages the job lifecycle, including job scheduling, execution, and monitoring. You can track the status of your job and view the results in the Azure Quantum portal.

Related content

- Different ways to run Q# programs
- Set up the Microsoft Quantum Development Kit
- Quickstart: Create your first Q# program

👉 **Learn more about the Microsoft Quantum platform**

Last updated on 05/15/2026

Quickstart: Create your first Q# program

Article • 10/29/2024

Learn how to write a basic Q# program that demonstrates entanglement, a key concept of quantum computing.

When two or more qubits are entangled, they share quantum information, which means whatever happens to one qubit also happens to the other. In this quickstart, you create a particular two-qubit entangled state called a Bell pair. In a Bell pair, if you measure one qubit in the $|0\rangle$ state, you know the other qubit is also in the $|0\rangle$ state without measuring it. For more information, see Quantum entanglement.

In this quickstart, you:

- ✓ Create a Q# file.
- ✓ Allocate a pair of qubits.
- ✓ Entangle the qubits.

Prerequisites

- The latest version of Visual Studio Code.
- The Azure Quantum Development Kit (QDK) extension. For installation details, see Set up the Quantum Development Kit.

Create a Q# file

1. Open Visual Studio Code.
2. Select **File** > **New Text File**.
3. Save the file as `Main.qs`. The .qs extension denotes a Q# program.

Write your Q# code

In your `Main.qs` file, follow these steps to entangle and measure a pair of qubits.

Import a quantum library

The QDK includes the Q# standard library with predefined functions and operations for your quantum programs. To use them, you must first import the relevant library.

In your program, use an `import` statement to open the `Microsoft.Quantum.Diagnostics` library. This gives you access to all its functions and operations, including `DumpMachine()`, which you later use to display the entangled state.

Q#

```
    import Microsoft.Quantum.Diagnostics.*;
```

Define an operation

After importing the relevant libraries, define your quantum operation and its input and output values. For this quickstart, your operation is `Main`. This is where you'll write the remaining Q# code to allocate, manipulate, and measure two qubits.

`Main` takes no parameters and returns two `Result` values, either `Zero` or `One`, which represent the results of the qubit measurements:

Q#

```
    operation Main() : (Result, Result) {
        // Your entanglement code goes here.
}
```

Allocate two qubits

The `Main` operation is currently empty, so the next step is to allocate two qubits, `q1` and `q2`. In Q#, you allocate qubits using the `use` keyword:

Q#

```
        // Allocate two qubits, q1 and q2, in the 0 state.
        use (q1, q2) = (Qubit(), Qubit());
```

> ⓘ **Note**
>
> In Q#, qubits are always allocated in the $|0\rangle$ state.

Put one qubit into superposition

The qubits `q1` and `q2` are in the $|0\rangle$ state. To prepare the qubits for entanglement, you must put one of them into an even superposition, where it has a 50% chance of being measured as $|0\rangle$ or $|1\rangle$.

You put a qubit into superposition by applying the *Hadamard*, `H`, operation:

Q#

```
        // Put q1 into an even superposition.
        H(q1);
```

The resulting state of `q1` is $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, which is an even superposition of $|0\rangle$ and $|1\rangle$.

Entangle the qubits

You're now ready to entangle the qubits using the controlled-NOT, `CNOT`, operation. `CNOT` is a control operation that takes two qubits, one acting as the control and the other as the target.

For this quickstart, you set `q1` as the control qubit and `q2` as the target qubit. This means `CNOT` flips the state of `q2` when the state of `q1` is $|1\rangle$.

Q#

```
        // Entangle q1 and q2, making q2 depend on q1.
        CNOT(q1, q2);
```

The resulting state of both qubits is the Bell pair $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

> **Tip**
>
> If you want to learn how the Hadamard and CNOT operations transform the state of the qubits, see **Creating entanglement with quantum operations**.

Display the entangled state

Before measuring the qubits, it's important to verify that your previous code successfully entangles them. You can use the `DumpMachine` operation, which is part of the `Microsoft.Quantum.Diagnostics` library, to output the current state of your Q# program:

Q#

```
        // Show the entangled state of the qubits.
        DumpMachine();
```

Measure the qubits

Now that you verified the qubits are entangled, you can use the `M` operation to measure them. Measuring `q1` and `q2` collapses their quantum states into `Zero` or `One` with even probability.

In Q#, you use the `let` keyword to declare a new variable. To store the measurement results of `q1` and `q2`, declare the variables `m1` and `m2`, respectively:

Q#

```
        // Measure q1 and q2 and store the results in m1 and m2.
        let (m1, m2) = (M(q1), M(q2));
```

Reset the qubits

Before being released at the end of each Q# program, qubits must be in the $|0\rangle$ state. You do this using the `Reset` operation:

Q#

```
        // Reset q1 and q2 to the 0 state.
        Reset(q1);
        Reset(q2);
```

Return the measurement results

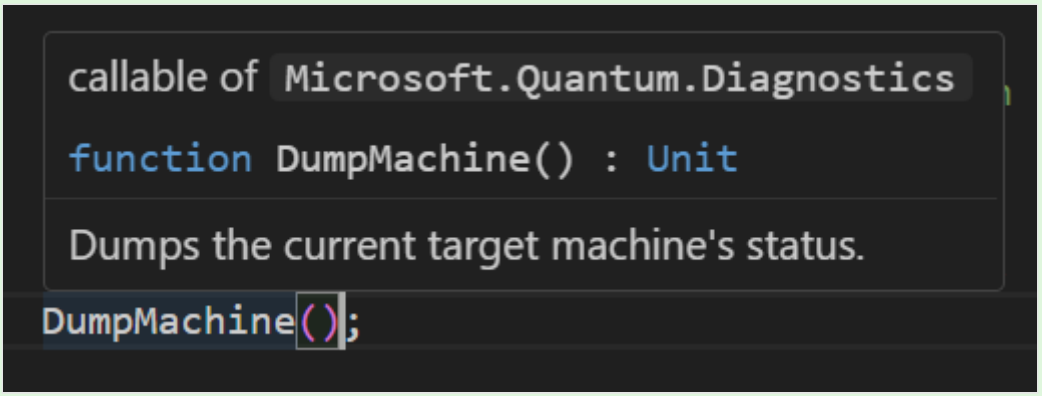
Finally, to complete the `Main` operation and observe the entangled state, return the measurement results of `m1` and `m2`:

Q#

```
        // Return the measurement results.
        return (m1, m2);
```

Tip

If you want to learn more about a Q# function or operation, hover over it.



Run your Q# code

Congratulations! You wrote a Q# program that entangles two qubits and creates a Bell pair.

Your final Q# program should look like this:

Q#

```
import Microsoft.Quantum.Diagnostics.*;

operation Main() : (Result, Result) {
    // Allocate two qubits, q1 and q2, in the 0 state.
    use (q1, q2) = (Qubit(), Qubit());

    // Put q1 into an even superposition.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q1);

    // Entangle q1 and q2, making q2 depend on q1.
    CNOT(q1, q2);

    // Show the entangled state of the qubits.
    DumpMachine();

    // Measure q1 and q2 and store the results in m1 and m2.
    let (m1, m2) = (M(q1), M(q2));

    // Reset q1 and q2 to the 0 state.
    Reset(q1);
    Reset(q2);

    // Return the measurement results.
    return (m1, m2);
}
```

To run your program and view the result of both qubits, select **Run** above the `Main` operation or press **Ctrl+F5**

```
import Microsoft.Quantum.Diagnostics.*;

Run | Histogram | Estimate | Debug | Circuit
operation Main() : (Result, Result) {
    // Allocate two qubits, q1 and q2, in the 0 state.
    use (q1, q2) = (Qubit(), Qubit());

    // Put q1 into an even superposition.
    // It now has a 50% chance of being measured as 0 or 1.
    H(q1);

    // Entangle q1 and q2, making q2 depend on q1.
    CNOT(q1, q2);

    // Show the entangled state of the qubits.
    DumpMachine();

    // Measure q1 and q2 and store the results in m1 and m2.
    let (m1, m2) = (M(q1), M(q2));

    // Reset q1 and q2 to the 0 state.
    Reset(q1);
```

You can run the program several times, each with a different result in the debug console. This demonstrates the probabilistic nature of quantum measurements and the entanglement of the qubits.

For example, if the result is `Zero`, your debug console should look like this:

Result

```
DumpMachine:

 Basis | Amplitude      | Probability | Phase
 -------------------------------------------------
  |00⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
  |11⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000

Result: "(Zero, Zero)"
```

Next step

To learn more about quantum entanglement with Q#, see Tutorial: Explore quantum entanglement with Q#. This tutorial expands on the concepts covered in this quickstart and helps you write a more advanced entanglement program.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Different ways to run Q# programs

The Microsoft Quantum Development Kit (QDK) offers two development options to write and run Q# programs. Both options interface with Azure Quantum so that you can run Q# programs on simulators and quantum hardware from Azure Quantum providers. The QDK also includes several local simulators to run Q# programs on your personal computer.

Options to develop and run Q# programs

Develop Q# programs and manage Q# jobs that you submit to Azure Quantum through the following development environments:

- **Visual Studio Code:** Write, run, and debug Q# code in your local environment, and submit jobs to Azure Quantum with the QDK extension in Visual Studio Code (VS Code). Installation required.
- **Jupyter Notebook:** Develop Q# code and submit jobs to Azure Quantum in Jupyter Notebook with the `qdk.qsharp` Python module. Installation required.
- **Azure portal:** Manage your Azure subscription and Azure Quantum workspace, and access information about your quantum providers and job submissions. Requires an Azure account.

The option that you choose depends on your coding experience, quantum knowledge, and goals. Each option has different features and functionality, so you can use them together. For example, write Q# programs with the QDK extension in VS Code and manage your quantum workspace in the Azure portal.

Visual Studio Code

VS Code is a free, open-source code editor from Microsoft. With the QDK extension for VS Code, you can create Q# programs and load built-in Q# samples. The QDK in VS Code offers the following local development features and more for Q# programs (`.qs` files):

- Error messaging
- Syntax highlighting
- Debugging
- CodeLens
- IntelliSense
- Quantum computer resource estimation

> **Note**
>
> The QDK extension also provides language support for OpenQASM programs (`.qasm` files).

You don't need an Azure account to use the QDK in VS Code, but you do need an Azure account to submit jobs to Azure Quantum with the QDK. You can use the QDK to connect to your Azure Quantum workspace from VS Code and run Q# programs on the quantum computers and simulators from different Azure Quantum providers. For more information, see How to submit Q# programs with Visual Studio Code.

To get started with the QDK extension in VS Code, see Set up the QDK.

> **Note**
>
> The QDK extension is also available in VS Code for the Web, which provides the same Azure connectivity and Q# language features as the desktop version. However, the web doesn't support Python, Qiskit, or Cirq programs.

Is the QDK extension in VS Code right for me?

VS Code is a feature-rich environment that includes CodeLens and IntelliSense to help you write, run, and debug Q# and OpenQASM quantum programs. If you have coding experience and want to explore Q# in depth, then VS Code is for you.

The following table shows what you can and can't do in VS Code:

Expand table

You can:	You can't:	You need:
• Run Q# and OpenQASM programs. • Load code samples. • Debug your programs. • Save your programs and results. • View compiler error messages. • Connect to your Azure Quantum	• Manage your subscriptions and workspaces. • Manage your quantum jobs. • Choose quantum computing providers and plans.	• To install VS Code. • To install the QDK extension. • An Azure subscription and a quantum workspace (if you want to run programs on real hardware).

You can:	You can't:	You need:
workspace. • Visualize quantum circuits. • Use the resource estimator.		

Jupyter Notebook

The QDK has a rich `qdk` Python package that allows you to develop Q# programs in `.py` Python files or Jupyter Notebook. The QDK Python package also supports other quantum languages, such as Qiskit, Cirq, and PennyLane.

The `qdk` Python package includes several modules to help you develop quantum programs and manage Azure Quantum jobs. For example, the `qsharp` module lets you write Q# code in Jupyter Notebook, and the `azure` module lets you connect to your quantum workspace and submit jobs to Azure Quantum.

For an overview of the `qdk` Python package and module features, see the QDK project description on the PyPi website.

Is Python and Jupyter Notebook right for me?

Jupyter Notebook is convenient to write Python code and visualize output all in one development environment. If you prefer to develop in Python and want support for multiple quantum programming languages, then the QDK Python package and Jupyter Notebook are for you.

The following table shows what you can and can't do in Python and Jupyter Notebook:

Expand table

You can:	You can't:	You need:
• Develop in multiple quantum programming languages. • Save your programs and results. • Connect to your Azure Quantum workspace. • Visualize quantum circuits.	• Manage your subscriptions and workspaces. • Manage your quantum jobs. • Choose quantum computing providers and plans.	• To install Python and Pip. • To install Jupyter Notebook. • To install the QDK Python package. • An Azure subscription and a quantum workspace (if you want to run programs on real hardware).

You can:	You can't:	You need:
• Use the resource estimator.		

Azure portal

The Azure portal is the main interface of the Microsoft Azure cloud computing platform. From the portal, you can create an Azure Quantum workspace to run quantum programs, send jobs to quantum hardware providers, and store job results in an Azure Quantum storage account. You can also manage your subscriptions, activity, credit usage, quotas, and access control.

Is the Azure portal right for me?

From the Azure portal, you can grant a group of users, like your team members or students, access to your quantum workspace. If you want to manage your jobs and subscriptions, review your invoices, or try out different quantum providers, then the Azure portal is for you.

The following table shows what you can and can't do in the Azure portal:

Expand table

You can:	You can't:	You need:
• Create quantum workspaces. • Manage your subscriptions and workspaces. • Copy workspace access keys. • Manage your quantum jobs. • Save your programs and results. • Choose quantum computing providers and plans.	• Develop quantum programs • Visualize quantum circuits and results. • Calculate resource estimates for your programs.	• An Azure subscription. • An Azure Quantum workspace.

Q# learning resources

To learn and explore the Q# programming language, use the following resources:

- **Azure Quantum learning path**: If you're interested in quantum computing but don't know where to start, take this learning path. Through a series of interactive modules, you learn about quantum computing and how to develop quantum solutions in Azure Quantum using Q# and the QDK.

- **Quantum Katas**: Learn quantum computing and programming together with these self-paced tutorials, each with relevant theory and Q# exercises to test your knowledge.
- **Q# code samples**: Build your first quantum solution with these ready-to-use Q# samples. They cover four areas: quantum algorithms, resource estimation, language constructs, and Jupyter notebooks.
- **QDK playground**: Explore common quantum algorithms written in Q#. The playground is hosted on VS Code for the Web and comes preconfigured with the QDK, so you don't need to install anything.

Related content

- Set up the Microsoft Quantum Development Kit
- Quickstart: Create your first Q# program
- Reference: QDK extension for Visual Studio Code

Last updated on 07/31/2026

How to create and manage Q# projects and custom libraries

Summarize this article for me

In this article, you learn how to create, manage, and share Q# projects. A Q# project is a folder structure with multiple Q# files that can access each other's operations and functions. Projects help you logically organize your source code. You can also use projects as custom libraries that you can access from external sources.

Prerequisites

- An Azure Quantum workspace in your Azure subscription. To create a workspace, see Create an Azure Quantum workspace.
- Visual Studio Code (VS Code) with the Microsoft Quantum Development Kit (QDK) and Python extensions installed.
- To publish your external project to a public GitHub repository, you need to have a GitHub account.

To run Python programs, you also need:

- A Python environment with Python and Pip installed.
- The `qdk` Python library with the `azure` extra.

```bash
python -m pip install --upgrade "qdk[azure]"
```

How Q# projects work

A Q# project contains a Q# manifest file named `qsharp.json`, and one or more `.qs` and `.qsc` files in a specified folder structure. You can create a Q# project manually or directly in VS Code.

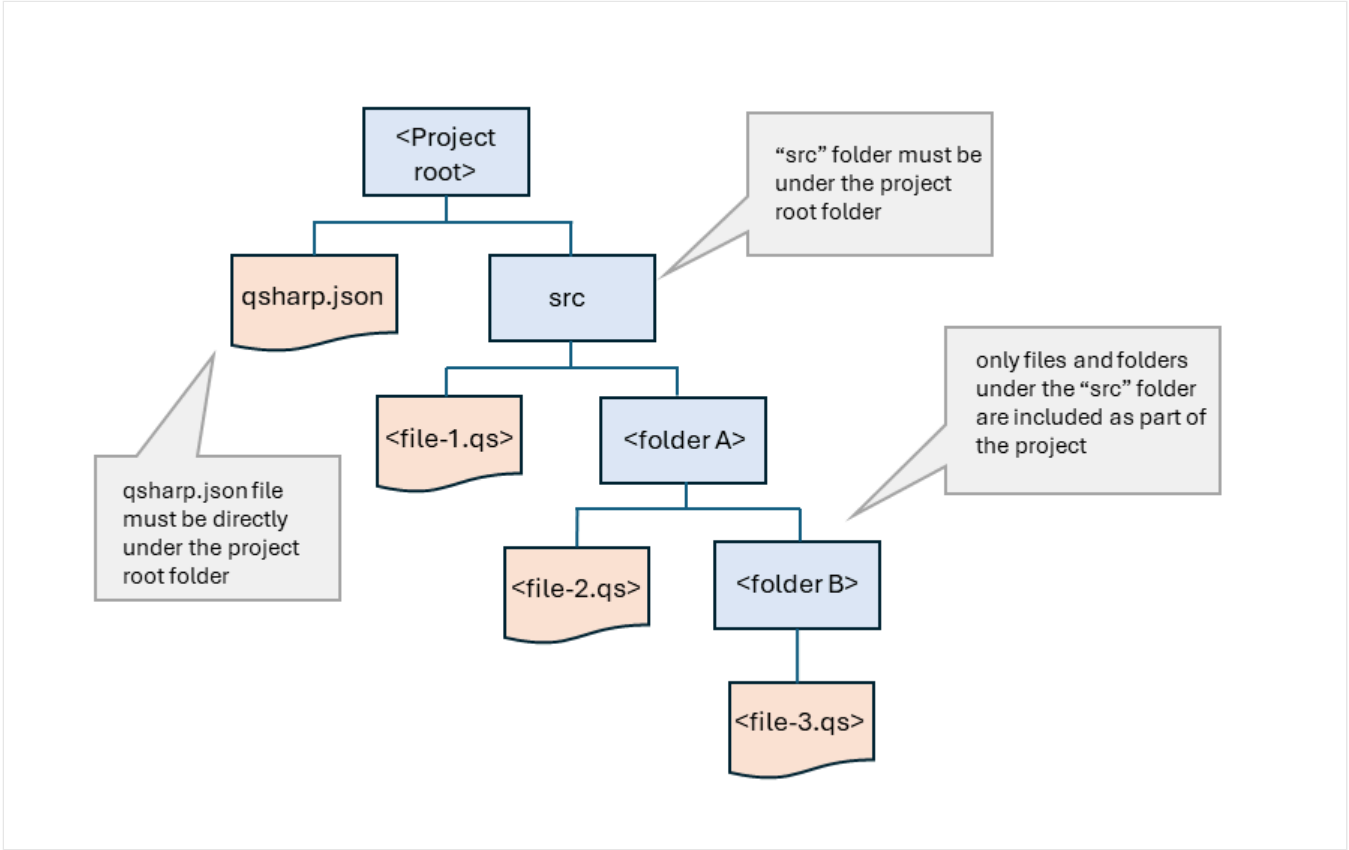
When you open a `.qs` or `.qsc` file in VS Code, the compiler searches the surrounding folder hierarchy for the manifest file and determines the project's scope. If the compiler doesn't find a manifest file, then the compiler operates in a single file mode.

When you set the `project_root` in a Jupyter Notebook or Python file, the compiler looks for the manifest file in the `project_root` folder.

An external Q# project is a standard Q# project that's located in another directory or on a public GitHub repository, and acts as a custom library. An external project uses `export` statements to define the functions and operations that are accessible by external programs. Programs define the external project as a dependency in their manifest file, and use `import` statements to access the items in the external project, such as operations, functions, structs, and namespaces. For more information, see Using projects as external dependencies.

Define a Q# project

A Q# project is defined by the presence of a `qsharp.json` manifest file and a `src` folder, both of which must be in the root folder of the project. The `src` folder contains the Q# source files. For Q# programs and external projects, the Q# compiler automatically detects the project folder. For Python programs and Jupyter Notebook files, you must specify the Q# project folder with a `qsharp.init` call. However, the folder structure for a Q# project is the same for all types of programs.



Use a Q# program

Define the project folder for Q# programs

When you open a `.qs` file in VS Code, the Q# compiler searches upward in the folder structure for a manifest file. If the compiler finds a manifest file, then the compiler includes all Q# files in the `/src` directory and its subdirectories. The items defined in each file become available to all other files within the project.

For example, consider the following folder structure:

- **Teleportation_project**
  - *qsharp.json*
  - **src**
    - *Main.qs*
    - **TeleportOperations**
      - *TeleportLib.qs*
      - **PrepareState**
        - *PrepareStateLib.qs*

When you open the file `/src/TeleportOperation/PrepareState/PrepareStateLib.qs`, the Q# compiler does the following:

1. Checks `/src/TeleportOperation/PrepareState/` for `qsharp.json`.
2. Checks `/src/TeleportOperation` for `qsharp.json`.
3. Checks `/src` for `qsharp.json`.
4. Checks `/Teleportation_project` for `qsharp.json` and finds the file.
5. Establishes `/Teleportation_project` as the root directory of the project, and includes all `.qs` and `.qsc` files under the `/src` directory in the project. If the manifest file.

> ⓘ **Note**
>
> If you includes explicit references to `.qs` and `.qsc` file paths in `qsharp.json`, the compiler loads those files and doesn't go through the automatic discovery process. Explicit file path references are necessary only when you define a library to be loadable from a Git reference.

Create a manifest file

A manifest file is a JSON file named `qsharp.json` that can include optional `author`, `license`, and `lints` fields. The minimum viable manifest file is the string `{}`. When you create a Q# project in VS Code, a minimal manifest file is created for you.

JSON

```
{}
```

Manifest file examples

The following examples show how manifest files define the scope of your Q# project.

- In this example, `author` is the only specified field, so all `.qs` files in this directory and its subdirectories are included in the Q# project.

  JSON

  ```
  {
      "author":"Microsoft"
  }
  ```

- Within a Q# project, you can also use the manifest file to fine-tune the VS Code Q# Linter settings. By default, the three Linter rules are:

  - `needlessParens`: default = `allow`
  - `divisionByZero`: default = `warn`
  - `redundantSemicolons`: default = `warn`

  You can set each rule in the manifest file to either `allow`, `warn`, or `error`. For example:

  JSON

  ```
  {
      "author":"Microsoft",
      "lints": [
          {
            "lint": "needlessParens",
            "level": "allow"
          },
          {
            "lint": "redundantSemicolons",
            "level": "warn"
          },
          {
            "lint": "divisionByZero",
            "level": "error"
          }
        ]
  }
  ```

- You can also use the manifest file to define an external Q# project as a dependency and remotely access operations and functions in that external project. For more information, see Using projects as external dependencies.

Q# project requirements and properties

The following requirements and configurations apply to all Q# projects.

- All `.qs` files that you want to include in the project must be under a folder named `src`, which must be under the root folder of the Q# project. When you create a Q# project in VS Code, the `/src` folder is automatically created.

- The manifest file should be at the same level as the `src` folder. When you create a Q# project in VS Code, a minimal manifest file is automatically created.

- Use `import` statements to reference operations and functions from other files in the project.

  Q#

  ```
  import MyMathLib.*;  //imports all the callables in the MyMathLib namespace

  ...

  Multiply(x,y);
  ```

  Or, reference them individually with the namespace.

  Q#

  ```
  MyMathLib.Multiply(x,y);
  ```

For Q# projects only

- You can define an entry point operation in only one `.qs` file in a Q# project, which is the `Main()` operation by default.
- You must put the `.qs` file with the entry point definition at a project directory level below the manifest file.
- All operations and functions in the Q# project that are cached from a `.qs` display in predictive text in VS Code.
- If the namespace for a selected operation or function isn't imported yet, then VS Code automatically adds the necessary `import` statement.

How to create a Q# project

To create a Q# project, follow these steps:

1. In the VS Code file explorer, go to the folder that you want to use as the root folder for the Q# project.
2. Open the **View** menu and choose **Command Palette**.
3. Enter **QDK: Create Q# project**. VS Code creates a minimal manifest file in the folder, and adds a `/src` folder with a `Main.qs` template file.
4. Edit the manifest file for your project. See Manifest file examples.
5. Add and organize your Q# source files under the `/src` folder.
6. If you're accessing the Q# project from a Python program or Jupyter Notebook, then set the root folder path with `qsharp.init`. This example assumes that your program is in the `/src` folder of the Q# project:

   Python

   ```
   qsharp.init(project_root = '../Teleportation_project')
   ```

7. If you're using only Q# files in VS Code, then the compiler searches for a manifest file when you open a Q# file and determines the root folder of the project. Then, the compiler scans `/src` and its subdirectories for `.qs` and `.qsc` files.

> **Note**
>
> You can manually create the manifest file and the `/src` folder instead.

Example project

This quantum teleportation program is an example of a Q# project that runs on the local simulator in VS Code. To run the program on Azure Quantum hardware or third-party simulators, see Get started with Q# programs and VS Code for steps to compile your program and connect to your Azure Quantum workspace.

This example has the following directory structure:

- **Teleportation_project**
  - *qsharp.json*

- **src**
  - *Main.qs*
  - **TeleportOperations**
    - *TeleportLib.qs*
    - **PrepareState**
      - *PrepareStateLib.qs*

The manifest file contains the **author** and **license** fields:

JSON

```
{
    "author":"Microsoft",
    "license":"MIT"
}
```

Q# source files

The main file `Main.qs` contains the entry point and references the `TeleportOperations.TeleportLib` namespace from `TeleportLib.qs`.

Q#

```
    import TeleportOperations.TeleportLib.Teleport; // references the Teleport
operation from TeleportLib.qs

    operation Main() : Unit {
        use msg = Qubit();
        use target = Qubit();

        H(msg);
        Teleport(msg, target); // calls the Teleport() operation from
TeleportLib.qs
        H(target);

        if M(target) == Zero {
            Message("Teleported successfully!");

        Reset(msg);
        Reset(target);
        }
    }
```

The `TeleportLib.qs` file defines the `Teleport` operation and calls the `PrepareBellPair` operation from the `PrepareStateLib.qs` file.

Q#

```
    import TeleportOperations.PrepareState.PrepareStateLib.*; // references the
namespace in PrepareStateLib.qs

    operation Teleport(msg : Qubit, target : Qubit) : Unit {
        use here = Qubit();

        PrepareBellPair(here, target); // calls the PrepareBellPair() operation
from PrepareStateLib.qs
        Adjoint PrepareBellPair(msg, here);

        if M(msg) == One { Z(target); }
        if M(here) == One { X(target); }

        Reset(here);
    }
```

The `PrepareStateLib.qs` file contains a standard reusable operation to create a Bell pair.

Q#

```
    operation PrepareBellPair(left : Qubit, right : Qubit) : Unit is Adj + Ctl {
        H(left);
        CNOT(left, right);
    }
```

Run the programs

Choose the tab for the environment where you run your program.

Run Q# code in VS Code

To run this program, open the `Main.qs` file in VS Code and choose **Run**.

Configure Q# projects as external dependencies

You can configure Q# projects as an external dependency for other projects, similar to a library, to make functions and operations in the external Q# project available to other Q# projects. An external dependency can reside on a drive share or be published to a public GitHub repository.

To use a Q# project as an external dependency, you must:

- Add the external project as a dependency in the manifest file of the calling project.

- If the external project is published to GitHub, then add the `files` property to the manifest file of the external project.
- Add `export` statements to the external project.
- Add `import` statements to the calling project.

Configure the manifest files

External Q# projects can reside on a local or network drive share, or you can publish them to a public GitHub repository.

The calling project manifest file

To add a dependency to an external project on a drive share, define the dependency in the manifest file of the calling project.

JSON

```
{
    "author": "Microsoft",
    "license": "MIT",
    "dependencies": {
        "MyDependency": {
            "path": "/path/to/project/folder/on/disk"
        }
    }
}
```

In the preceding manifest file, `MyDependency` is a user-defined string that identifies the namespace when you call an operation. For example, if you create a dependency named `MyMathFunctions`, then you can call a function from that dependency with `MyMathFunctions.MyFunction()`.

To add a dependency to a project that's published to a public GitHub repository, use the following example manifest file:

JSON

```
{
    "author": "Microsoft",
    "dependencies": {
        "MyDependency": {
            "github": {
                "owner": "GitHubUser",
                "repo": "GitHubRepoName",
                "ref": "CommitHash",
```

```
                "path": "/path/to/dependency"
            }
        }
    }
}
```

> **Note**
>
> For GitHub dependencies, `ref` refers to a GitHub refspec. Microsoft recommends that you always use a commit hash so that you can rely on a specific version of your dependency.

The external project manifest file

If your external Q# project is published to a public GitHub repository, then you must add the `files` property to the manifest file of the external project, including all files that the project uses.

JSON

```
{
    "author": "Microsoft",
    "license": "MIT",
    "files": [ "src/MyMathFunctions.qs", "src/Strings/MyStringFunctions.qs" ]
}
```

The `files` property is optional for an external project that you import via `"path"` with a local filepath-based import. The `files` property is required only for projects that are published to GitHub.

Use the `export` statement

To make functions and operations in an external project accessible to calling projects, use the `export` statement. You can export any or all of the callables in the file. You can't use wild card syntax, so you must specify each callable that you want to export.

Q#

```
operation Operation_A() : Unit {
...
}
operation Operation_B() : Unit  {
...
}
```

```
// makes just Operation_A available to calling programs
export Operation_A;

// makes Operation_A and Operation_B available to calling programs
export Operation_A, Operation_B, etc.;

// makes Operation_A available as 'OpA'
export Operation_A as OpA;
```

Use the `import` statement

To make items from an external dependency available, use `import` statements from the calling program. The `import` statement uses the namespace that you define for the dependency in the manifest file.

For example, consider the dependency in the following manifest file:

JSON

```
{
    "author": "Microsoft",
    "license": "MIT",
    "dependencies": {
        "MyMathFunctions": {
            "path": "/path/to/project/folder/on/disk"
        }
    }
}
```

Import the callables with the following code:

Q#

```
import MyMathFunctions.MyFunction;  // imports "MyFunction()" from the namespace

...
```

The `import` statement also supports wild card syntax and aliases.

Q#

```
// imports all items from the "MyMathFunctions" namespace
import MyMathFunctions.*;

// imports the namespace as "Math", all items are accessible via "Math.<callable>"
import MyMathFunctions as Math;
```

```
// imports a single item, available in the local scope as "Add"
import MyMathFunctions.MyFunction as Add;

// imports can be combined on one line
import MyMathFunctions.MyFunction, MyMathFunctions.AnotherFunction as Multiply;
```

Example external project

For this example, use the same teleportation program as the earlier example, but separate the calling program and the callables into different projects.

1. Create two folders on your local drive, for example `Project_A` and `Project_B`.
2. Create a Q# project in each folder. For details, see the steps in How to create a Q# project.
3. In `Project_A`, the calling program, copy the following code into the manifest file, but edit the path as needed for `Project_B`:

   JSON

   ```
   {
     "author": "Microsoft",
     "license": "MIT",
     "dependencies": {
       "MyTeleportLib": {
         "path": "/Project_B"
         }
       }
     }
   ```

4. In `Project_A`, copy the following code into `Main.qs`:

   Q#

   ```
   import MyTeleportLib.Teleport; // imports the Teleport operation from the
   MyTeleportLib namespace defined in the manifest file

   operation Main() : Unit {
       use msg = Qubit();
       use target = Qubit();

       H(msg);
       Teleport(msg, target); // calls the Teleport() operation from the
   MyTeleportLib namespace
       H(target);

       if M(target) == Zero {
           Message("Teleported successfully!");
   ```

```
    Reset(msg);
    Reset(target);
    }
}
```

5. In `Project_B`, copy the following code into `Main.qs`:

Q#

```
    operation Teleport(msg : Qubit, target : Qubit) : Unit {
        use here = Qubit();

        PrepareBellPair(here, target);
        Adjoint PrepareBellPair(msg, here);

        if M(msg) == One { Z(target); }
        if M(here) == One { X(target); }

        Reset(here);
    }

    operation PrepareBellPair(left : Qubit, right : Qubit) : Unit is Adj + Ctl
{
        H(left);
        CNOT(left, right);
    }

    export Teleport;       //  makes the Teleport operation available to
external programs
```

> **Note**
>
> You don't need to export the `PrepareBellPair` operation your program in `Project_A` doesn't directly call that operation. The `PrepareBellPair` operation is already accessible by the `Teleport` operation because `PrepareBellPair` is in the local scope of `Project_B`.

6. To run the program, open `/Project_A/Main.qs` in VS Code and choose **Run**.

Projects and implicit namespaces

In Q# projects, if you don't specify a namespace in a `.qs` program, then the compiler uses the file name as the namespace. Then, when you reference a callable from an external dependency, you use the syntax `<dependencyName>.<namespace>.<callable>`. However, if the file is named

`Main.qs`, then the compiler assumes that the namespace and the calling syntax is `<dependencyName>.<callable>`. For example, `import MyTeleportLib.Teleport`.

Because you might have multiple project files, you need to account for the correct syntax when you reference callables. For example, consider a project with the following file structure:

- **/src**
  - *Main.qs*
  - *MathFunctions.qs*

The following code makes calls to the external dependency:

Q#

```
import MyTeleportLib.MyFunction;        // "Main" namespace is implied

import MyTeleportLib.MathFunctions.MyFunction;   // "Math" namespace must be
explicit
```

For more information about namespace behavior, see User namespaces.

Last updated on 03/05/2026

How to use Q# callables in Python

With the Microsoft Quantum Development Kit (QDK), you can pass Q# functions and operations between Q# and Python as objects. A Q# function or operation is called a *callable*. Instead of building a Q# expression string for each combination of routines and parameters, you can select a callable in Python and pass it to another Q# callable.

You can also return a callable from Q# to Python, including a *closure* that captures values from the Q# code that created it. Use these callable objects as arguments to Q# routines or as entry points for QDK simulation, compilation, and circuit visualization.

The examples in this article use a state preparation workflow: choose an operation that prepares a qubit, and then pass it to a reusable measurement routine.

Prerequisites

- Python 3.10 or later.
- The QDK Python package. For installation instructions, see Set up the Microsoft Quantum Development Kit.
- A Jupyter Notebook environment, if you want to run the examples in a notebook instead of a Python file. This environment is optional.

Run the examples in order and in the same Python session.

Pass Q# callables from Python

When you define Q# callables, the QDK adds Python wrappers for them to the `qdk.code` module. Import the wrappers after you load the Q# definitions. You can call a wrapper directly or pass the callable itself as an argument without parentheses. A direct wrapper call runs the Q# callable once on the sparse simulator, just like a call through `qsharp.eval`.

To load callables from a Q# project instead of defining them with `qsharp.eval`, use a `qsharp.json` project. Then import the callables from the corresponding namespace in `qdk.code`. For project setup, see How to create and manage Q# projects and custom libraries.

Pass an operation

Initialize a QIR target profile and define two Q# operations. The `Base` target profile supports the QIR compilation example later in this article.

The `PreparePlus` operation applies a Hadamard gate to a qubit. The `RunPreparation` operation accepts a preparation operation, allocates a qubit, applies the preparation, and then measures and resets the qubit.

Python

```
from qdk import qsharp

qsharp.init(target_profile=qsharp.TargetProfile.Base)

qsharp.eval("""
    operation PreparePlus(q : Qubit) : Unit {
        H(q);
    }

    operation RunPreparation(prepare : Qubit => Unit) : Result {
        use q = Qubit();
        prepare(q);
        MResetZ(q)
    }
""")

from qdk.code import PreparePlus, RunPreparation

print(qsharp.run(RunPreparation, 5, PreparePlus))
```

This call passes `PreparePlus` to `RunPreparation` and runs five shots. Each shot returns one measurement result, with an equal probability of `Zero` or `One`.

The parameter type `Qubit => Unit` requires an operation that accepts a qubit and returns no value. Python passes the operation, not the qubit. The `RunPreparation` operation supplies the qubit inside Q#.

Pass a function

Functions use the same passing syntax. For example, here's how to convert a list of rotation angles from degrees to radians:

Python

```
qsharp.eval("""
    function DegreesToRadians(angle : Double) : Double {
        angle * Std.Math.PI() / 180.0
    }
```

```
    function ConvertAngles(convert : Double -> Double, angles : Double[]) : Double[] {
        Std.Arrays.Mapped(convert, angles)
    }
""")

from qdk.code import DegreesToRadians, ConvertAngles

print(ConvertAngles(DegreesToRadians, [0.0, 90.0, 180.0]))
```

The result is a list of angles in radians. `ConvertAngles` applies the supplied `DegreesToRadians` function to each angle.

Get a callable value from Q#

Use `qsharp.eval` to evaluate an expression whose result is a callable. For example, evaluate the name `PreparePlus` without calling it:

Python

```
prepare_plus = qsharp.eval("PreparePlus")

print(qsharp.run(RunPreparation, 5, prepare_plus))
```

`PreparePlus`, imported from `qdk.code`, is a Python wrapper for the Q# operation. `prepare_plus`, returned by `qsharp.eval("PreparePlus")`, is a reference to that same operation. You can pass either object to `RunPreparation`, but you can't call `prepare_plus` directly as a Python function.

The `qsharp.run` call runs `RunPreparation` for five shots and passes `prepare_plus` to its Q# parameter `prepare`. On each shot, `RunPreparation` allocates a qubit, invokes `PreparePlus` through `prepare(q)`, and then measures and resets the qubit.

A callable reference can also be an entry point when its inputs and output support Python conversion. Here, `RunPreparation` is the entry point because `PreparePlus` needs a qubit that's supplied inside Q#.

Return a closure to Python

A Q# function can return an operation that captures a value. Define `MakeRotation` to return a Y-rotation operation, then pass the returned closure to the same `RunPreparation` routine:

Python

```
qsharp.eval("""
    function MakeRotation(angle : Double) : Qubit => Unit {
        q => Ry(angle, q)
    }
""")

from qdk.code import MakeRotation

rotation = MakeRotation(0.5)
print(qsharp.run(RunPreparation, 5, rotation))
```

The returned operation captures the angle `0.5` radians. Calling `MakeRotation` creates the operation; it doesn't apply a gate. `RunPreparation` supplies a qubit and invokes the operation during each shot.

Like the callable reference from `qsharp.eval`, a returned closure isn't a directly callable Python function. This closure also needs a qubit, which Python can't supply. Pass it to `RunPreparation` so that qubit allocation and use stay inside Q#.

The closure captures only an angle, not a qubit, so it works with a fresh qubit on each shot.

Callable values can also appear in tuples, arrays, and Q# struct fields that pass between Q# and Python. These values remain references to Q# code, including any captured values.

Pass a partially applied operation

Partial application creates a callable by supplying some arguments and leaving others unspecified. In Q#, use `_` for an argument that the new callable receives later.

Create the same Y-rotation operation as `MakeRotation` without calling the factory function:

Python

```
rotation_from_expression = qsharp.eval("Ry(0.5, _)")

print(qsharp.run(RunPreparation, 5, rotation_from_expression))
```

The expression captures the rotation angle but leaves the qubit argument unspecified. The resulting operation accepts a qubit and returns `Unit`, which matches the parameter of `RunPreparation`.

Use a Q# lambda expression or partial application to build a callable in Q#, and then keep the result in Python. You don't need a new top-level Q# declaration for each combination of arguments.

Compile and visualize a program with callable arguments

Pass the same callable objects and Q# arguments to `qsharp.compile` and `qsharp.circuit`. These APIs don't take a shot count.

Python

```
program = qsharp.compile(RunPreparation, rotation)
circuit = qsharp.circuit(RunPreparation, rotation)

print(circuit)
```

These calls use the closure returned by `MakeRotation`. You can also pass `PreparePlus` or `rotation_from_expression`.

The `program` object has the compiled program in Quantum Intermediate Representation (QIR). Use `str(program)` to get the QIR text. The `circuit` object represents the preparation, measurement, and reset operations.

These calls don't submit a job to Azure Quantum. The program must still meet the requirements of the selected target profile and the destination target. For more information, see Azure Quantum QIR target profiles in the QDK.

To get logical resource counts, pass callable arguments to `qsharp.logical_counts`.

Usage guidelines

- Use callables defined in Q#. Import wrappers from `qdk.code`, or get Q# callable references and closures. Ordinary Python functions and lambdas can't serve as Q# callable arguments.
- Match the expected type. Check the callable's input and output types, plus any required `Adjoint` or `Controlled` support.
- Keep qubits and Q# ranges inside Q#. Pass the callable that uses them instead of the values themselves. When you call Q# code from Python, make sure its arguments and return value support Python conversion.
- Specify concrete types for generic routines. Define a Q# wrapper with specific input and output types instead of relying on Python to infer Q# type parameters.
- Reuse callables within their original Q# context. After another `qsharp.init` call, reload your Q# definitions and get fresh callable objects.

Related content

- Functions and operations in Q#
- Closures in Q#

ⓘ**Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 11/10/2026

How to visualize quantum circuit diagrams with the QDK

Quantum circuit diagrams are a visual representation of quantum algorithms. Circuit diagrams show the flow of qubits through a quantum program, including the gates and measurements that the program applies to the qubits.

In this article, you learn how to create circuit diagrams for Q# and OpenQASM programs with the Microsoft Quantum Development Kit (QDK) using Visual Studio Code (VS Code) and Jupyter Notebook.

For more information about quantum circuit diagrams, see Quantum circuit diagram conventions.

Prerequisites

To create circuit diagrams from Q# and OpenQASM files in VS Code, install the following:

- The latest version of VS Code, or open VS Code for the Web.
- The latest version of the QDK extension in VS Code.

To create circuit diagrams from Python programs in Jupyter Notebook, install the following:

- The Python extension and Jupyter extension in VS Code.
- The latest version of the `qdk` Python library with the `jupyter` extra.

  Bash

  ```
  pip install --upgrade "qdk[jupyter]"
  ```

Visualize quantum circuits in VS Code

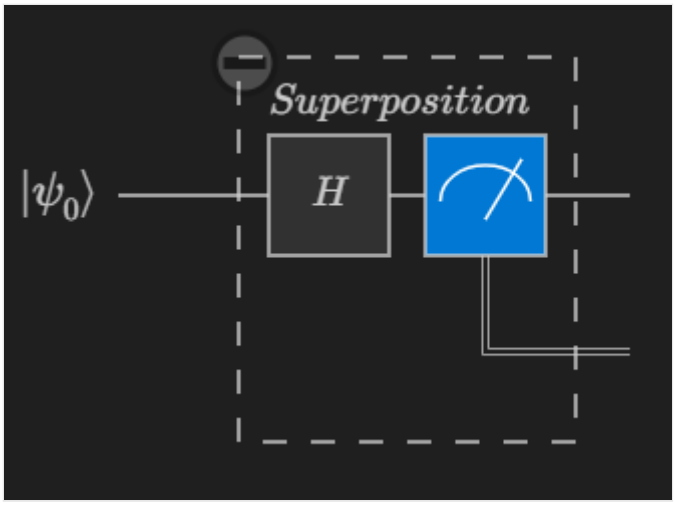
With the QDK extension in VS Code, you can create circuit diagrams for Q# (`.qs`) and OpenQASM (`.qasm`) files.

To view a circuit diagram in VS Code, follow these steps:

1. Open a Q# or OpenQASM file in VS Code, or load one of the quantum samples from the QDK.

2. Choose the **Circuit** command from the code lens that precedes your program.

The **QDK Circuit** window opens and displays the circuit diagram for your program. For example, the following circuit diagram corresponds to a program that produces a random bit. The circuit puts the qubit into a superposition state and then measures the qubit.



> **Tip**
>
> Select an element in the circuit diagram to highlight the code that creates the circuit element.

View circuit diagrams for individual Q# operations

To visualize the quantum circuit for an individual operation in a Q# file, choose the **Circuit** command from the code lens that precedes the operation.

```
42
    Circuit
43  operation PreparePhiPlus(register : Qubit[]) : Unit {
44      H(register[0]);                    // |+0⟩
45      CNOT(register[0], register[1]); // (|00⟩  + |11⟩)/sqrt(2)
46  }
47
48
    Circuit
49  operation PreparePhiMinus(register : Qubit[]) : Unit {
50      H(register[0]);                    // |+0⟩
51      Z(register[0]);                    // |-0⟩
52      CNOT(register[0], register[1]); // (|00⟩  - |11⟩)/sqrt(2)
53  }
54
```

View circuit diagrams when you're debugging

When you use the VS Code debugger in a Q# program, you can visualize the quantum circuit based on the state of the program at the current debugger breakpoint.

1. Choose the **Debug** command from the code lens that precedes your entry point operation.
2. In the **Run and Debug** pane, expand the **Quantum Circuit** dropdown in the **VARIABLES** menu. The **QDK Circuit** panel opens, which shows the circuit while you step through the program.
3. Set breakpoints and step through your code to see how the circuit updates as your program runs.

Visualize quantum circuits in Jupyter Notebook

In Jupyter Notebook, you can visualize quantum circuits for Q# and OpenQASM programs with the `qdk.qsharp` and `qdk.widgets` Python modules. The `widgets` module provides a widget that renders a quantum circuit diagram as an SVG image.

For more examples of circuit diagram generation in Jupyter Notebook, see the Circuits sample notebook on the QDK GitHub repository.

View circuit diagrams for Q# programs

To view a circuit diagram for a Q# program in Jupyter Notebook, follow these steps:

1. In VS Code, open the **View** menu and choose **Command Palette**.

2. Enter **Create: New Jupyter Notebook**. An empty Jupyter Notebook file opens in a new tab.

3. In the first cell of the notebook, import the `qdk.qsharp` module.

   Python

   ```
   from qdk import qsharp
   ```

4. Create a new cell and enter your Q# code. For example, the following code prepares a Bell State:

   Q#

   ```
   %%qsharp

   // Prepare a Bell State.
   operation BellState() : Unit {
       use register = Qubit[2];
       H(register[0]);
       CNOT(register[0], register[1]);
   }
   ```

5. To display a quantum circuit diagram, pass the Q# operation to the `qsharp.circuit` function. Run the following code in a new cell:

   Python

   ```
   qsharp.circuit("BellState()")
   ```

   The output looks like this:

   Output

   ```
   q_0    ── H ──── ● ──
   q_1    ─────────── X ──
   ```

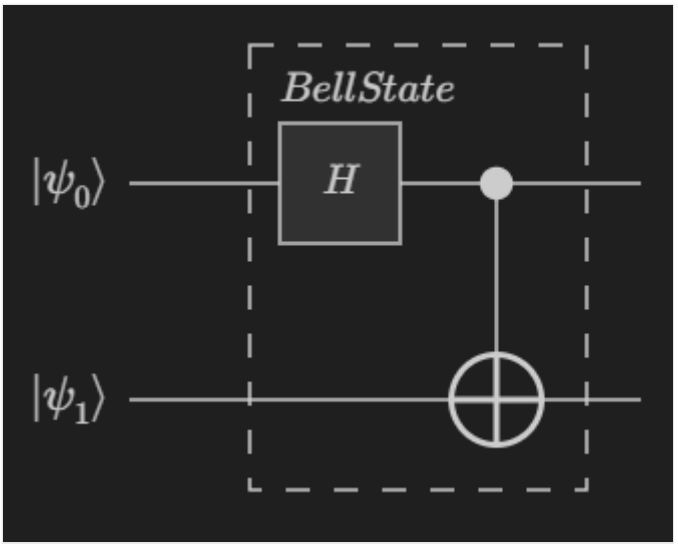
6. To render an SVG image of the quantum circuit, use the `widgets` module. Create a new cell, then run the following code to visualize the same circuit that you created in the previous cell.

   Python

```
from qdk.widgets import Circuit

Circuit(qsharp.circuit("BellState()"))
```

The circuit diagram looks like this:



View circuit diagrams for operations that take qubits as input

In the previous Bell state example, the operation `BellState` doesn't take qubits as input. If the operation takes qubits or qubit arrays as input, then omit the parentheses when you pass the operation to draw circuit diagrams.

For example, follow these steps to draw circuit diagrams for an operation that takes qubits:

1. In a new cell, run the following Q# code. This code prepares a cat state.

Q#

```
%%qsharp

operation PrepareCatState(register : Qubit[]) : Unit {
    H(register[0]);
    ApplyToEach(CNOT(register[0], _), register[1...]);
}
```

2. To draw the circuit as a text diagram, run the following code:

Python

```
Circuit(qsharp.circuit(operation="PrepareCatState"))
```

3. To render the circuit diagram as an SVG image, run the following code:

Python

```
Circuit(qsharp.circuit(operation="PrepareCatState"))
```

When the operation takes an array of qubits `(Qubit[])`, the circuit shows the array as a register of two qubits.

View circuit diagrams for OpenQASM programs

To view a circuit diagram for an OpenQASM program in Jupyter Notebook, follow these steps:

1. In VS Code, open the **View** menu and choose **Command Palette**.
2. Enter **Create: New Jupyter Notebook**. An empty Jupyter Notebook file opens in a new tab.
3. In the first cell of the notebook, run the following code to import the necessary objects to create and call OpenQASM functions with the QDK Python library:

   Python

   ```
   from qdk.openqasm import import_openqasm, ProgramType
   ```

4. In a new cell, write your OpenQASM program in a Python string and pass the string to the `import_openqasm` function. To call the program in your Python code, give the function a name and set `program_type` to `ProgramType.File`.

   Python

   ```
   source = """
       include "stdgates.inc";
       bit[2] c;
       qubit[2] q;
       h q[0];
       cx q[0], q[1];
       c = measure q;
   """

   import_openqasm(source, name="bell", program_type=ProgramType.File)
   ```

5. In a new cell, import the OpenQASM program as a Python function and call the program to get the measurement results.

   Python

```
from qdk.code.qasm_import import bell

bell()
```

6. To draw the circuit as a text diagram, run the following code in a new cell:

   Python

   ```
   from qdk.qsharp import circuit

   circuit(bell)
   ```

   The output looks like this:

   Output

   ```
   q_0    ── H ──── ● ──── M ──
                    │       ╘═══
   q_1    ──────────── X ──── M ──
                            ╘═══
   ```

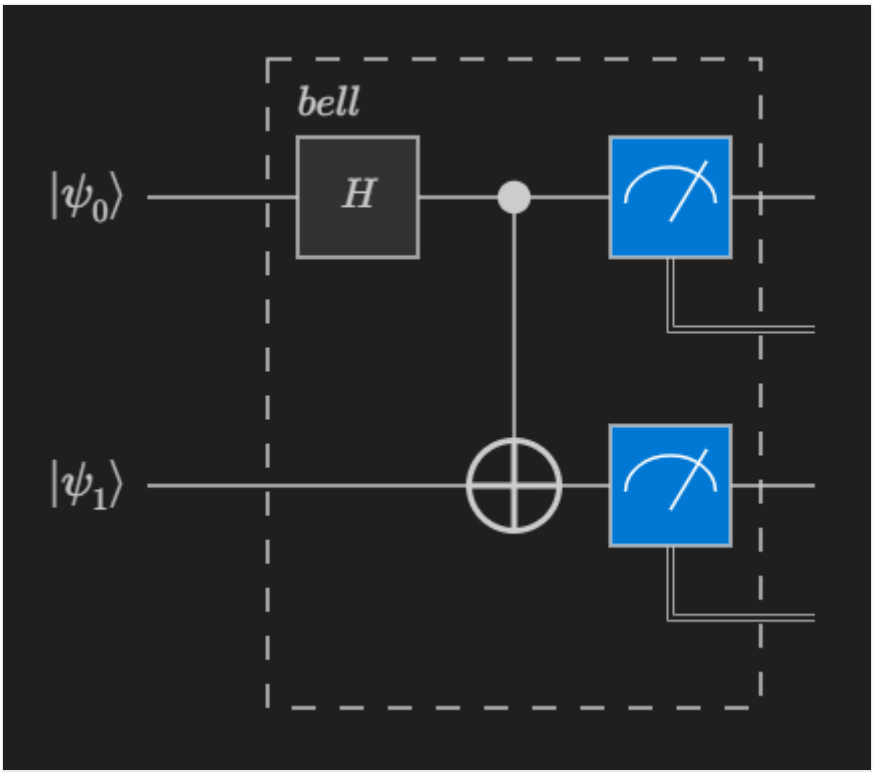
7. To render the circuit diagram as an SVG image, run the following code in a new cell:

   Python

   ```
   from qdk.widgets import Circuit

   Circuit(qsharp.circuit(bell))
   ```

   The circuit diagram looks like this:



> **Note**
>
> For OpenQASM programs, you can't view circuit diagrams for individual functions. You can view circuit diagrams for only the whole program.

Related content

- Explore circuit diagrams in the QDK
- Quantum circuit diagram conventions

Last updated on 06/22/2026

Program execution

Article • 10/24/2024

The following example gives a first glimpse at how a Q# program is implemented:

Q#

```
/// # Sample
/// Bell States
///
/// # Description
/// Bell states or EPR pairs are specific quantum states of two qubits
/// that represent the simplest (and maximal) examples of quantum
entanglement.
///
/// This Q# program implements the four different Bell states.

import Std.Diagnostics.*;
import Std.Measurement.*;

operation Main() : (Result, Result)[] {
    // Allocate the two qubits that will be used to create a Bell state.
    use register = Qubit[2];

    // This array contains a label and a preparation operation for each one
    // of the four Bell states.
    let bellStateTuples = [
        ("|Φ+⟩", PreparePhiPlus),
        ("|Φ-⟩", PreparePhiMinus),
        ("|Ψ+⟩", PreparePsiPlus),
        ("|Ψ-⟩", PreparePsiMinus)
    ];

    // Prepare all Bell states, show them using the `DumpMachine` operation
    // and measure the Bell state qubits.
    mutable measurements = [];
    for (label, prepare) in bellStateTuples {
        prepare(register);
        Message($"Bell state {label}:");
        DumpMachine();
        set measurements += [(MResetZ(register[0]), MResetZ(register[1]))];
    }
    return measurements;
}

operation PreparePhiPlus(register : Qubit[]) : Unit {
    ResetAll(register);             // |00⟩
    H(register[0]);                 // |+0⟩
    CNOT(register[0], register[1]); // 1/sqrt(2)(|00⟩ + |11⟩)
}

operation PreparePhiMinus(register : Qubit[]) : Unit {
```

```
    ResetAll(register);             // |00⟩
    H(register[0]);                 // |+0⟩
    Z(register[0]);                 // |-0⟩
    CNOT(register[0], register[1]); // 1/sqrt(2)(|00⟩ - |11⟩)
}

operation PreparePsiPlus(register : Qubit[]) : Unit {
    ResetAll(register);             // |00⟩
    H(register[0]);                 // |+0⟩
    X(register[1]);                 // |+1⟩
    CNOT(register[0], register[1]); // 1/sqrt(2)(|01⟩ + |10⟩)
}

operation PreparePsiMinus(register : Qubit[]) : Unit {
    ResetAll(register);             // |00⟩
    H(register[0]);                 // |+0⟩
    Z(register[0]);                 // |-0⟩
    X(register[1]);                 // |-1⟩
    CNOT(register[0], register[1]); // 1/sqrt(2)(|01⟩ - |10⟩)
}
```

This program implements the four basic Bell states of quantum entanglement, and is one of the sample programs included the Azure Quantum Visual Code extension.

You can run the program from the built-in simulator in the VS Code QDK extension and get standard output

Output

```
Message: Bell state |Φ+⟩:

DumpMachine:

 Basis | Amplitude      | Probability | Phase
 ------------------------------------------------
  |00⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
  |11⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000

Message: Bell state |Φ-⟩:

DumpMachine:

 Basis | Amplitude      | Probability | Phase
 ------------------------------------------------
  |00⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
  |11⟩ | -0.7071+0.0000𝑖 |    50.0000% |  -3.1416

Message: Bell state |Ψ+⟩:

DumpMachine:

 Basis | Amplitude      | Probability | Phase
```

```
 ------------------------------------------------
  |01⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
  |10⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000

Message: Bell state |Ψ-⟩:

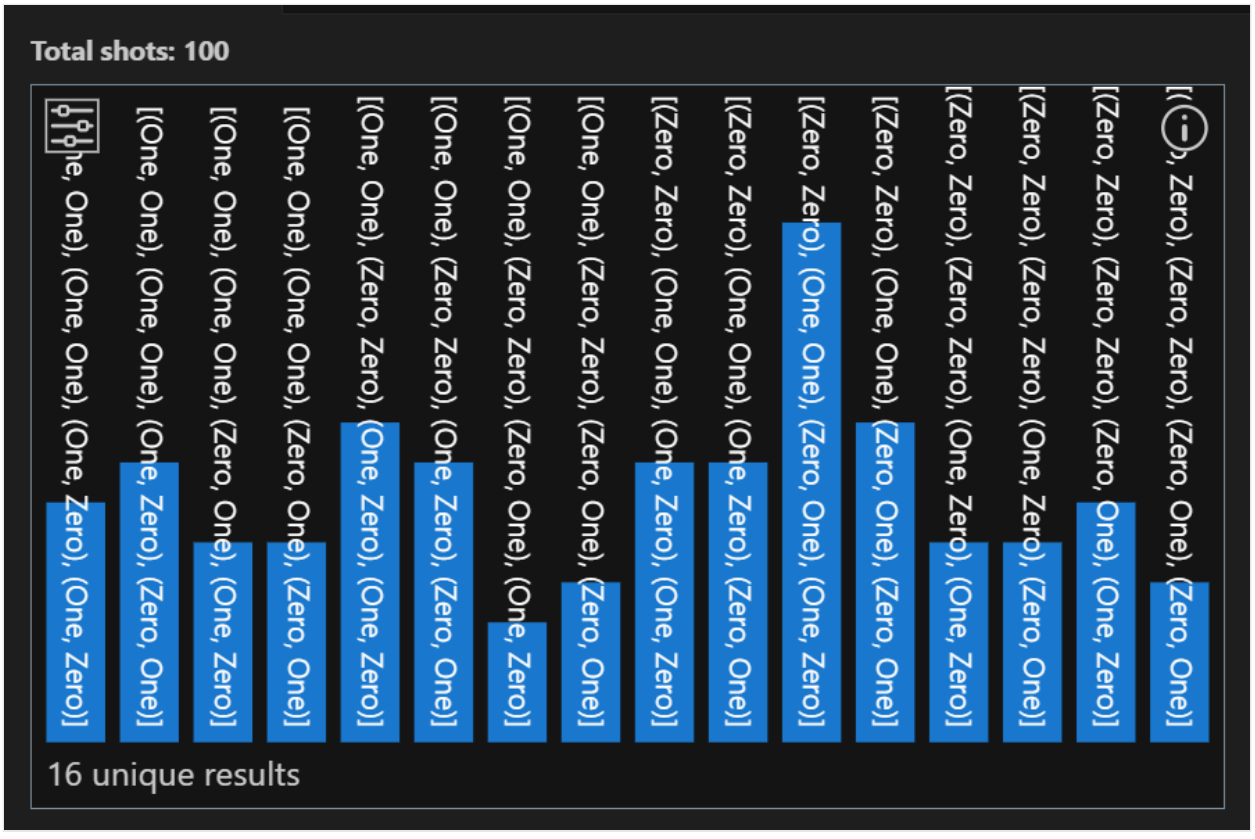
DumpMachine:

 Basis | Amplitude      | Probability | Phase
 ------------------------------------------------
  |01⟩ |  0.7071+0.0000𝑖 |    50.0000% |   0.0000
  |10⟩ | -0.7071+0.0000𝑖 |    50.0000% |  -3.1416

Result: "[(One, One), (Zero, Zero), (One, Zero), (Zero, One)]"
Finished shot 1 of 1

Q# simulation completed.
```

or run the simulator with a histogram output



To run the program on quantum hardware, the program first needs to be compiled and submitted to Azure Quantum, all of which can be done from inside VS Code. For the full end-to-end process, see Get started with Q# programs and Visual Studio Code.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Namespaces

Article • 02/18/2025

At its top-level, a Q# program consists of a set of namespaces. Aside from comments, namespaces are the only top-level elements in a Q# program, and any other elements must reside within a namespace. Each file may contain zero or more namespaces, and each namespace may span multiple files. Q# does not support nested namespaces.

> **Note**
>
> If you do not explicitly declare a namespace, the Q# compiler uses the filename as the namespace name.

A namespace block consists of the keyword `namespace`, followed by the namespace name, and the content of the block inside braces `{ }`. Namespace names consist of a sequence of one or more legal symbols separated by a dot (`.`). While namespace names may contain dots for better readability, Q# does not support relative references to namespaces. For example, two namespaces `Foo` and `Foo.Bar` are unrelated, and there is no notion of a hierarchy. In particular, for a function `Baz` defined in `Foo.Bar`, it is *not* possible to open `Foo` and then access that function via `Bar.Baz`.

Namespace blocks may contain import directives, operations, functions, and type declarations. These elements may occur in any order and are recursive by default, meaning they can be declared and used in any order and can call themselves. There is no need for the declaration of a type or callable to precede its use.

Import Directives

By default, everything declared within the same namespace can be accessed without further qualification. However, declarations in a different namespace can only be used by qualifying their name with the name of the namespace they belong to or by opening that namespace before it is used, as shown in the following example.

Q#

```
namespace Microsoft.Quantum.Samples {

    import Microsoft.Quantum.Arithmetic.*;
    import Microsoft.Quantum.Arrays.* as Array;
```

```
    // ...
}
```

> **Note**
>
> For namespaces in the standard Q# library, the root of the namespace may be simplified to `Std`. For example, the previous example could be rewritten as:
>
> Q#
>
> ```
> import Std.Arithmetic.*;
> import Std.Arrays.* as Array;
> ```

The example uses an `import` directive to import all types and callables declared in the `Microsoft.Quantum.Arithmetic` namespace. You can then refer to them by their unqualified name unless that name conflicts with a declaration in the namespace block or another opened namespace.

To avoid typing out the full name while still distinguishing where certain elements come from, you can define an alternative name, or *alias*, for a particular namespace. In this case, you can qualify all types and callables declared in that namespace by the defined short name instead. The previous example uses an alias for the `Microsoft.Quantum.Arrays` namespace. You can then use the function `IndexRange`, declared in `Microsoft.Quantum.Arrays`, for example, via `Array.IndexRange` within that namespace block.

Whether you are opening a namespace or defining an alias, `import` directives are valid throughout the namespace piece in that file only.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Type Declarations

Article • 02/21/2025

Q# supports user-defined `struct` types. `struct` types are similar to record types in F#; they're immutable but support a copy-and-update construct.

Struct types

`struct` types can only contain named items and don't support anonymous items. Any combination of named items is supported, although items can't be nested.

The following declaration, for example, defines a struct `Complex` which has two named items `Real` and `Imaginary`, both of type `Double`:

Q#

```
struct Complex {
    Real : Double,
    Imaginary : Double,
}
```

You can access the contained items via their name or by *deconstruction* (for more information, see item access). You can also access a tuple of all items where the shape matches the one defined in the declaration via the unwrap operator.

`struct` types are useful for two reasons. First, as long as the libraries and programs that use the defined types access items via their name rather than by deconstruction, the type can be extended to contain additional items later on without breaking any library code. Because of this, accessing items via deconstruction is discouraged.

Second, Q# allows you to convey the intent and expectations for a specific data type since there's no automatic conversion between values of two `struct` types, even if their item types are identical.

Struct constructors

The compiler automatically generates constructors for new `struct` types when it reads a `struct` definition. For the `Complex` struct in the previous example, you can create an instance with

Q#

```
let complexPair = Complex(1.4, 2.1);
```

You can also define instances with the `new` keyword, for example

Q#

```
let complexPair = new Complex { Real = 1.4, Imaginary = 2.1 };
```

You can copy an existing struct with the `...` syntax

Q#

```
let copyPair = new Complex { ...complexPair };
```

When copying, you can specify individual fields to change

Q#

```
let updatedPair = new Complex { ...complexPair, Real = 3.5 };
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Callable declarations

Article • 02/18/2025

Callable declarations, or *callables*, declared at a global scope are publicly visible by default; that is, they can be used anywhere in the same project and in a project that references the assembly in which they're declared. Access modifiers allow you to restrict their visibility to the current assembly only, such that implementation details can be changed later on without breaking code that relies on a specific library.

Q# supports two kinds of callables: operations and functions. The topic Operations and Functions elaborates on the distinction between the two. Q# also supports defining *templates*; for example, type-parameterized implementations for a certain callable. For more information, see Type parameterizations.

> **Note**
>
> Such type-parametrized implementations may not use any language constructs that rely on particular properties of the type arguments; there's currently no way to express type constraints in Q#, or to define specialized implementations for particular type arguments.

Callables and functors

Q# allows specialized implementations for specific purposes; for example, operations in Q# can implicitly or explicitly define support for certain *functors*, and along with it the specialized implementations to invoke when a specific functor is applied to that callable.

A functor, in a sense, is a factory that defines a new callable implementation that has a specific relation to the callable it was applied to. Functors are more than traditional higher-level functions in that they require access to the implementation details of the callable they have been applied to. In that sense, they're similar to other factories, such as templates. They can be applied to type-parameterized callables as well.

Consider the following operation, `ApplyQFT`:

qharp

```
    operation ApplyQFT(qs : Qubit[]) : Unit is Adj + Ctl {
        let length = Length(qs);
        Fact(length >= 1, "ApplyQFT: Length(qs) must be at least 1.");
        for i in length - 1..-1..0 {
```

```
            H(qs[i]);
            for j in 0..i - 1 {
                Controlled R1Frac([qs[i]], (1, j + 1, qs[i - j - 1]));
            }
        }
    }
```

This operation takes an argument of type `Qubit[]` and returns a value of type `Unit`. The annotation `is Adj + Ctl` in the declaration of `ApplyQFT` indicates that the operation supports both the `Adjoint` and the `Controlled` functor. (For more information, see Operation characteristics). The expression `Adjoint ApplyQFT` accesses the specialization that implements the adjoint of `ApplyQFT`, and `Controlled ApplyQFT` accesses the specialization that implements the controlled version of `ApplyQFT`. In addition to the original operation's argument, the controlled version of an operation takes an array of control qubits and applies the original operation on the condition that all of these control qubits are in a |1⟩ state.

In theory, an operation for which an adjoint version can be defined should also have a controlled version and vice versa. In practice, however, it may be hard to develop an implementation for one or the other, especially for probabilistic implementations following a repeat-until-success pattern. For that reason, Q# allows you to declare support for each functor individually. However, since the two functors commute, an operation that defines support for both also has to have an implementation (usually implicitly defined, meaning compiler-generated) for when both functors are applied to the operation.

There are no functors that can be applied to functions. Functions currently have exactly one body implementation and no further specializations. For example, the declaration

Q#

```
    function Hello (name : String) : String {
        $"Hello, {name}!"
    }
```

is equivalent to

Q#

```
    function Hello (name : String) : String {
        body ... {
            $"Hello, {name}!"
        }
    }
```

Here, `body` specifies that the given implementation applies to the default body of the function `Hello`, meaning the implementation is invoked when no functors or other factory mechanisms have been applied prior to invocation. The three dots in `body ...` correspond to a compiler directive indicating that the argument items in the function declaration should be copy and pasted into this spot.

The reasons behind explicitly indicating where the arguments of the parent callable declaration are to be copied and pasted are twofold: one, it's unnecessary to repeat the argument declaration, and two, it ensures that functors that require additional arguments, like the `Controlled` functor, can be introduced in a consistent manner.

When there's exactly one specialization defining the implementation of the default body, the additional wrapping of the form `body ... { <implementation> }` may be omitted.

Recursion

Q# callables can be directly or indirectly recursive and can be declared in any order; an operation or function may call itself, or it may call another callable that directly or indirectly calls the caller.

Stack space may be limited when running on quantum hardware, and recursions that exceed that stack space limit result in a runtime error.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Specialization declarations

Article • 02/21/2025

As explained in the section about callable declarations, there's currently no reason to explicitly declare specializations for functions. This topic applies to operations and elaborates on how to declare the necessary specializations to support certain functors.

It's quite a common problem in quantum computing to require the adjoint of a given transformation. Many quantum algorithms require both an operation and its adjoint to perform a computation. Q# employs symbolic computation that can automatically generate the corresponding adjoint implementation for a particular body implementation. This generation is possible even for implementations that freely mix classical and quantum computations. There are, however, some restrictions that apply in this case. For example, auto-generation isn't supported for performance reasons if the implementation makes use of mutable variables. Moreover, each operation called within the body generates the corresponding adjoint needs to support the `Adjoint` functor itself.

Even though one can't easily undo measurements in the multi-qubit case, it's possible to combine measurements so that the applied transformation is unitary. In this case, it means that, even though the body implementation contains measurements that on their own don't support the `Adjoint` functor, the body in its entirety is adjointable. Nonetheless, auto-generating the adjoint implementation fails in this case. For this reason, it's possible to manually specify the implementation. The compiler automatically generates optimized implementations for common patterns such as conjugations. Nonetheless, an explicit specialization may be desirable to define a more optimized implementation by hand. It's possible to specify any one implementation and any number of implementations explicitly.

> **Note**
>
> The compiler doesn't verify the correctness of such a manually specified implementation.

In the following example, the declaration for an operation `SWAP`, which exchanges the state of two qubits `q1` and `q2`, declares an explicit specialization for its adjoint version and its controlled version. While the implementations for `Adjoint SWAP` and `Controlled SWAP` are thus user-defined, the compiler still needs to generate the implementation for

the combination of both functors (`Controlled Adjoint SWAP`, which is the same as `Adjoint Controlled SWAP`).

```
Q#
```

```
    operation SWAP (q1 : Qubit, q2 : Qubit) : Unit
    is Adj + Ctl {

        body ... {
            CNOT(q1, q2);
            CNOT(q2, q1);
            CNOT(q1, q2);
        }

        adjoint ... {
            SWAP(q1, q2);
        }

        controlled (cs, ...) {
            CNOT(q1, q2);
            Controlled CNOT(cs, (q2, q1));
            CNOT(q1, q2);
        }
    }
```

Auto-generation directives

When determining how to generate a particular specialization, the compiler prioritizes user-defined implementations. This means that if an adjoint specialization is user-defined and a controlled specialization is auto-generated, then the controlled adjoint specialization is generated based on the user-defined adjoint and vice versa. In this case, both specializations are user-defined. As the auto-generation of an adjoint implementation is subject to more limitations, the controlled adjoint specialization defaults to generating the controlled specialization of the explicitly defined implementation of the adjoint specialization.

In the case of the `SWAP` implementation, the better option is to adjoint the controlled specialization to avoid unnecessarily conditioning the execution of the first and the last `CNOT` on the state of the control qubits. Adding an explicit declaration for the controlled adjoint version that specifies a suitable *generation directive* forces the compiler to generate the controlled adjoint specialization based on the manually specified implementation of the controlled version instead. Such an explicit declaration of a specialization generated by the compiler takes the form

```
```

```Q#
    controlled adjoint invert;
```

and is inserted inside the declaration of `SWAP`. On the other hand, inserting the line

```Q#
    controlled adjoint distribute;
```

forces the compiler to generate the specialization based on the defined (or generated) adjoint specialization. For more information, see this partial specialization inference proposal for more details.

For the operation `SWAP`, there's a better option. `SWAP` is *self-adjoint*, that is, it's its own inverse; the defined implementation of the adjoint merely calls the body of `SWAP` and is expressed with the directive

```Q#
    adjoint self;
```

Declaring the adjoint specialization in this manner ensures that the controlled adjoint specialization that the compiler automatically inserts merely invokes the controlled specialization.

The following generation directives exist and are valid:

Expand table

Specialization	Directives
`body` specialization:	-
`adjoint` specialization:	`self`, `invert`
`controlled` specialization:	`distribute`
`controlled adjoint` specialization:	`self`, `invert`, `distribute`

That all generation directives are valid for a controlled adjoint specialization isn't a coincidence; as long as functors commute, the set of valid generation directives for implementing the specialization for a combination of functors is always the union of the set of valid generators for each one.

In addition to the previously listed directives, the directive `auto` is valid for all specializations except `body`; it indicates that the compiler should automatically pick a suitable generation directive. The declaration

```
Q#

    operation DoNothing() : Unit {
        body ... { }
        adjoint auto;
        controlled auto;
        controlled adjoint auto;
    }
```

is equivalent to

```
Q#

    operation DoNothing() : Unit
    is Adj + Ctl { }
```

The annotation `is Adj + Ctl` in this example specifies the *operation characteristics*, which contain the information about what functors a particular operation supports.

While for readability's sake, we recommend that you annotate each operation with a complete description of its characteristics, the compiler automatically inserts or completes the annotation based on explicitly declared specializations. Conversely, the compiler also generates specializations that aren't declared explicitly but need to exist based on the annotated characteristics. We say the given annotation *implicitly declares* these specializations. The compiler automatically generates the necessary specializations if it can, picking a suitable directive. Q# thus supports inference of both operation characteristics and existing specializations based on (partial) annotations and explicitly defined specializations.

In a sense, specializations are similar to individual overloads for the same callable, with the caveat that certain restrictions apply to which overloads you can declare.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Attributes

Attributes attach extra information to a declaration that the Q# compiler uses to change how the declaration is processed. An attribute is written with a leading `@` symbol, followed by the attribute name and a parenthesized argument list, and it's placed immediately before the declaration it applies to.

Q#

```
@EntryPoint()
operation Main() : Result {
    use q = Qubit();
    H(q);
    MResetZ(q)
}
```

You can apply more than one attribute to the same declaration by placing each attribute on its own line before the declaration.

Q# supports the following built-in attributes.

@EntryPoint()

The `@EntryPoint()` attribute marks an operation or function as the starting point of a program's execution. If you don't specify an entry point, the compiler starts execution from the `Main()` operation, if one exists. You can place the `@EntryPoint()` attribute on any callable that takes no input parameters to make it the entry point instead.

Q#

```
@EntryPoint()
operation MeasureOneQubit() : Result {
    use q = Qubit();
    H(q);
    MResetZ(q)
}
```

When you work in a single `.qs` file that isn't part of a Q# project, you can also pass a target profile name to the attribute, such as `@EntryPoint(Base)`, `@EntryPoint(Adaptive_RI)`, `@EntryPoint(Adaptive_RIF)`, or `@EntryPoint(Unrestricted)`, to set the QIR target profile for the program. For more information, see Quantum computing target profiles.

@Config(...)

The `@Config(...)` attribute provides pre-processing information about when a declaration should be included in compilation, based on the capabilities of the current target profile. This lets a library provide different implementations of a callable for targets with different capabilities.

Valid arguments are `Base`, `Adaptive`, `IntegerComputations`, `FloatingPointComputations`, `BackwardsBranching`, `HigherLevelConstructs`, and `Unrestricted`. You can negate an argument with the `not` operator to include the declaration only when the capability *isn't* available.

Q#

```
@Config(Adaptive)
operation MeasureAndReset(q : Qubit) : Result {
    let result = M(q);
    if result == One {
        X(q);
    }
    result
}

@Config(not Adaptive)
operation MeasureAndReset(q : Qubit) : Result {
    MResetZ(q)
}
```

@Test()

The `@Test()` attribute turns a callable into a unit test. You can apply it only to a callable that takes no input parameters. Tests marked with this attribute appear in the **Testing** view in Visual Studio Code, where you can run them individually or as a group. For more information, see Testing and debugging Q# programs.

Q#

```
@Test()
operation TestCase() : Unit {
    Fact(1 + 1 == 2, "Expected 1 + 1 to equal 2.");
}
```

Attributes for intrinsic callables

The following attributes describe how an intrinsic callable is treated during Quantum Intermediate Representation (QIR) code generation. They're primarily used by library and

hardware authors who define low-level intrinsic operations.

@SimulatableIntrinsic()

The `@SimulatableIntrinsic()` attribute indicates that a callable should be treated as an intrinsic callable for QIR code generation, while the body that you provide is used only during simulation. This lets you supply a simulation implementation for an operation that's otherwise opaque to the target hardware.

@Measurement()

The `@Measurement()` attribute indicates that an intrinsic callable is a measurement. The operation is marked as *irreversible* in the generated QIR, and the output `Result` value is moved to the arguments of the generated function.

@Reset()

The `@Reset()` attribute indicates that an intrinsic callable is a reset. The operation is marked as *irreversible* in the generated QIR.

Last updated on 06/24/2026

Comments

Article • 08/20/2024

Comments begin with two forward slashes (`//`) and continue until the end of line. Such end-of-line comments may appear anywhere in the source code. Q# does not currently support block comments.

Documentation Comments

Comments that begin with three forward slashes, `///`, are treated specially by the compiler when they appear before a type or callable declaration. In that case, their contents are taken as documentation for the defined type or callable, as for other .NET languages.

Within `///` comments, text to appear as a part of API documentation is formatted as Markdown, with different parts of the documentation indicated by specially-named headers. As an extension to Markdown, cross-references to operations, functions, and struct types in Q# can be included using `@"<ref target>,"` where `<ref target>` is replaced by the fully qualified name of the code object being referenced. Optionally, a documentation engine may also support additional Markdown extensions.

For example:

Q#

```
/// # Summary
/// Given an operation and a target for that operation,
/// applies the given operation twice.
///
/// # Input
/// ## op
/// The operation to be applied.
/// ## target
/// The target to which the operation is to be applied.
///
/// # Type Parameters
/// ## 'T
/// The type expected by the given operation as its input.
///
/// # Example
/// ```Q#
/// // Should be equivalent to the identity.
/// ApplyTwice(H, qubit);
/// ```
///
/// # See Also
```

```
/// - Microsoft.Quantum.Intrinsic.H
operation ApplyTwice<'T>(op : ('T => Unit), target : 'T) : Unit {
    op(target);
    op(target);
}
```

Q# recognizes the following names as documentation comment headers.

- **Summary**: A short summary of a function or operation's behavior or the purpose of a type. The first paragraph of the summary is used for hover information. It should be plain text.
- **Description**: A description of a function or operation's behavior or the purpose of a type. The summary and description are concatenated to form the generated documentation file for the function, operation, or type. The description may contain in-line LaTeX-formatted symbols and equations.
- **Input**: A description of the input tuple for an operation or function. May contain additional Markdown subsections indicating each element of the input tuple.
- **Output**: A description of the tuple returned by an operation or function.
- **Type Parameters**: An empty section that contains one additional subsection for each generic type parameter.
- **Named Items**: A description of the named items in a struct type. May contain additional Markdown subsections with the description for each named item.
- **Example**: A short example of the operation, function, or type in use.
- **Remarks**: Miscellaneous prose describing some aspect of the operation, function, or type.
- **See Also**: A list of fully qualified names indicating related functions, operations, or struct types.
- **References**: A list of references and citations for the documented item.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Statements

Article • 02/29/2024

Q# distinguishes between statements and expressions. Q# programs consist of a mixture of classical and quantum computations, and the implementation looks much like any other classical programming language. Some statements, such as the `let` and `mutable` bindings, are well-known from classical languages, while others, such qubit allocations, are unique to the quantum domain.

The following statements are currently available in Q#:

- **Expression statement**
  Contains a Q# expression to be run, such as a call to an operation. If the last statement in a block is an expression statement, it may have its trailing semicolon omitted to give the block the evaluated value of the contained expression.

- **Variable declaration**
  Defines one or more local variables that are valid for the remainder of the current scope, and binds them to the specified values. Variables can be permanently bound or declared to be reassignable later on. See Variable declarations and reassignments for more details.

- **Qubit allocation**
  Instantiates and initializes qubits, or arrays of qubits, and binds them to the declared variables. The statement can optionally be used with a specified block of code, in which the qubit allocations are valid. Otherwise the allocations are valid for the enclosing scope. Qubits are automatically released at the end of the appropriate scope. See Quantum memory management for more details.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Visibility of Local Variables

Article • 02/21/2025

In general, symbol bindings in Q# become inoperative at the end of the statement block they occur in. However, there are some exceptions to this rule.

Visibility of local variables

Expand table

Scope	Visibility
Loop variables	Bindings of loop variables in a for loop are defined only for the body of the loop. They're inoperative outside of the loop.
use and borrow statements	Bindings of allocated qubits in `use` and `borrow` statements are defined for the body of the allocation, and are inoperative after the statement terminates. This action only applies to `use` and `borrow` statements that have an associated statement block.
repeat expressions	For `repeat` expressions, both the blocks and the condition are treated as a single scope, that is, symbols that are bound in the body are accessible in both the condition and in the `fixup` block.
Loops	Each iteration of a loop runs in its own scope, and all defined variables are bound anew for each iteration.

Bindings in outer blocks are visible and defined in inner blocks. A symbol may only be bound multiple times per block in which case the last binding in scope is the one used for logic. This scenario is known as "shadowing."

The following sequences are valid:

Q#

```
if a == b {
    ...
    let n = 5;
    ...             // n is 5
}
let n = 8;
...                 // n is 8
```

and

Q#

```
if a == b {
    ...
    let n = 5;
    ...             // n is 5
} else {
    ...
    let n = 8;
    ...             // n is 8
}
...                 // n is not bound to a value
```

Q#

```
let n = 5;
...                 // n is 5
let n = 8;
...                 // n is 8
```

and

Q#

```
let n = 8;
if a == b {
    ...             // n is 8
    let n = 5;
    ...             // n is 5
}
...                 // n is 8 again
```

For more information, see Variable Declarations and Reassignments.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Variable Declarations and Reassignments

Article • 02/21/2025

Values can be bound to symbols using the `let` and `mutable` statements. These kinds of bindings provide a convenient way to access a value via the defined handle. Despite the misleading terminology borrowed from other languages, handles declared on a local scope and containing values are called *variables*. This naming may be misleading because `let` statements define *single-assignment handles*, which are handles that remain bound to the same value for the duration of their validity. Variables that can be re-bound to different values at different points in the code need to be explicitly declared as such, and are specified using the `mutable` statement.

Q#

```
let var1 = 3;
mutable var2 = 3;
var2 = var2 + 1;
```

In this example, the `let` statement declares a variable named `var1` that can't be reassigned and always contains the value `3`. The `mutable` statement defines a variable `var2` that is temporarily bound to the value `3` but can be reassigned to a different value later on using an assignment expression, as shown in the last line. You can express the same statement with the shorter version `var2 += 1;`, as is common in other languages. For more information, see Evaluate and reassign statements.

To summarize:

- `let` is used to create an immutable binding.
- `mutable` is used to create a mutable binding.
- `=` without a `let` is used to change the value of a mutable binding.

For all three statements, the left-hand side consists of a symbol or a symbol tuple. If the right-hand side of the binding is a tuple, then that tuple may be fully or partially deconstructed upon assignment. The only requirement for deconstruction is that the shape of the tuple on the right-hand side matches the shape of the symbol tuple on the left side. The symbol tuple may contain nested tuples or omitted symbols, or both, indicated by an underscore. For example:

Q#

```
let (a, (_, b)) = (1, (2, 3)); // a is bound to 1, b is bound to 3
mutable (x, y) = ((1, 2), [3, 4]); // x is bound to (1, 2), y is bound to
[3, 4]
(x, _, y) = ((5, 6), 7, [8]);  // x is re-bound to (5,6), y is re-bound to
[8]
```

For more information on deconstruction using the unwrap (`!`) operator, see Item access for struct types.

All assignments in Q# obey the same deconstruction rules, including, for example, qubit allocations and loop-variable assignments.

For both kinds of bindings, the types of the variables are inferred from the right-hand side of the binding. The type of a variable always remains the same, and an assignment expression can't change it. Local variables can be declared as either being mutable or immutable. There are some exceptions, such as loop-variables in `for` loops, where the behavior is predefined and can't be specified. Function and operation arguments are always immutably bound. In combination with the lack of reference types, as discussed in the Immutability topic, being immutably bound means that a called function or operation can never change any values on the caller side.

Since the states of `Qubit` values aren't defined or observable from within Q#, this doesn't preclude the accumulation of quantum side effects, which are observable only via measurements. For more information, see Quantum data types.

Independent of how a value is bound, the values themselves are immutable. In particular, this is true for arrays and array items. In contrast to popular classical languages where arrays often are reference types, arrays in Q# - like all types - are value types and always immutable; that is, you can't modify them after initialization. Changing the values accessed by array-type variables thus requires explicitly constructing a new array and reassigning it to the same symbol. For more information, see Immutability and Copy and update expressions.

Evaluate-and-reassign statements

Statements of the form `intValue += 1;` are common in many other languages. Here, `intValue` needs to be a mutably bound variable of type `Int`. Such statements provide a convenient way of concatenation if the right-hand side consists of applying a binary operator and the result is re-bound to the left argument of the operator. For example, this code segment

Q#

```
mutable counter = 0;
for i in 1 .. 2 .. 10 {
    counter += 1;
    // ...
}
```

increments the value of the counter `counter` in each iteration of the `for` loop and is equivalent to

Q#

```
mutable counter = 0;
for i in 1 .. 2 .. 10 {
    counter = counter + 1;
    // ...
}
```

Similar statements exist for a wide range of operators. Such evaluate-and-reassign expressions exist for all operators where the type of the left-most subexpression matches the expression type. More precisely, they're available for binary logical and bitwise operators including right and left shift, arithmetic expressions including exponentiation and modulus, and concatenations, as well as copy-and-update expressions.

The following function example computes the sum of an array of Complex numbers:

Q#

```
function ComplexSum(values : Complex[]) : Complex {
    mutable res = Complex(0., 0.);
    for complex in values {
        res = new Complex { Re = res.Re + complex.Re, Im = res.Im +
complex.Im };
    }
    return res;
}
```

Similarly, the following function multiplies each item in an array with the given factor:

Q#

```
function Multiplied(factor : Double, array : Double[]) : Double[] {
    mutable res = new Double[Length(array)];
    for i in IndexRange(res) {
        res w/= i <- factor * array[i];
    }
```

```
    return res;
}
```

For more information, see Contextual expressions, which contains other examples where expressions can be omitted in a specific context when a suitable expression can be inferred by the compiler.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Quantum Memory Management

Article • 02/29/2024

A program always starts without qubits, meaning you cannot pass values of type `Qubit` as entry point arguments. This restriction is intentional since the purpose of Q# is to express and reason about a program in its entirety. Instead, a program allocates and releases qubits, or quantum memory, as it goes. In this regard, Q# models the quantum computer as a qubit heap.

Rather than supporting separate *allocate* and *release* statements for quantum memory, Q# supports quantum memory allocation in the form of *block statements*, where the memory is accessible only within the scope of that block statement. The statement block can be implicitly defined when allocating qubits for the duration of the current scope, as described in more detail in the sections about the `use` and `borrow` statements. Attempting to access the allocated qubits after the statement terminates results in a runtime exception.

Q# has two statements, `use` and `borrow`, that instantiate qubit values, arrays of qubits, or any combination thereof. You can only use these statements within operations. They gather the instantiated qubit values, bind them to the variables specified in the statement, and then run a block of statements. At the end of the block, the bound variables go out of scope and are no longer defined.

Q# distinguishes between the allocation of *clean* and *dirty* qubits. Clean qubits are unentangled and are not used by another part of the computation. Dirty qubits are qubits whose state is unknown and can even be entangled with other parts of the quantum processor's memory.

Use statement

Clean qubits are allocated by the `use` statement.

- The statement consists of the keyword `use` followed by a binding and an optional statement block.
- If a statement block is present, the qubits are only available within that block. Otherwise, the qubits are available until the end of the current scope.
- The binding follows the same pattern as `let` statements: either a single symbol or a tuple of symbols, followed by an equals sign `=`, and either a single tuple or a matching tuple of *initializers*.

Initializers are available either for a single qubit, indicated as `Qubit()`, or an array of qubits, `Qubit[n]`, where `n` is an `Int` expression. For example,

```
Q#
use qubit = Qubit();
// ...

use (aux, register) = (Qubit(), Qubit[5]);
// ...

use qubit = Qubit() {
    // ...
}

use (aux, register) = (Qubit(), Qubit[5]) {
    // ...
}
```

The qubits are guaranteed to be in a |0⟩ state upon allocation. They are released at the end of the scope and are required to be in a |0⟩ state upon release. This requirement is not compiler-enforced since this would require a symbolic evaluation that quickly gets prohibitively expensive. When running on simulators, the requirement can be runtime enforced. On quantum processors, the requirement cannot be runtime enforced; an unmeasured qubit may be reset to |0⟩ via unitary transformation. Failing to do so results in incorrect behavior.

The `use` statement allocates the qubits from the quantum processor's free qubit heap and returns them to the heap no later than the end of the scope in which the qubits are bound.

Borrow statement

The `borrow` statement grants access to qubits that are already allocated but not currently in use. These qubits can be in an arbitrary state and need to be in the same state again when the borrow statement terminates. Some quantum algorithms can use qubits without relying on their exact state, and without requiring that they are unentangled with the rest of the system. That is, they require extra qubits temporarily, but they can ensure that those qubits are returned exactly to their original state, independent of which state that was.

If there are qubits that are in use but not touched during parts of a subroutine, those qubits can be borrowed for use by such an algorithm instead of allocating additional quantum memory. Borrowing instead of allocating can significantly reduce the overall

quantum memory requirements of an algorithm and is a quantum example of a typical space-time tradeoff.

A `borrow` statement follows the same pattern described previously for the use statement, with the same initializers being available. For example,

```
Q#
```

```
borrow qubit = Qubit();
// ...

borrow (aux, register) = (Qubit(), Qubit[5]);
// ...

borrow qubit = Qubit() {
    // ...
}

borrow (aux, register) = (Qubit(), Qubit[5]) {
    // ...
}
```

The borrowed qubits are in an unknown state and go out of scope at the end of the statement block. The borrower commits to leaving the qubits in the same state as when they were borrowed; that is, their state at the beginning and the end of the statement block is expected to be the same.

The `borrow` statement retrieves in-use qubits that are guaranteed not to be used by the program from the time the qubit is bound until the last use of that qubit. If there aren't enough qubits available to borrow, then qubits are allocated from and returned to the heap like a `use` statement.

> **Note**
>
> Among the known use-cases of dirty qubits are implementations of multi-controlled CNOT gates that require very few qubits, and implementations of incrementers. This paper on factoring with qubits provides an example of an algorithm that utilizes borrowed qubits.

Feedback

Was this page helpful? Yes No

Expressions

Article • 02/21/2025

At the core, Q# expressions are either value literals or identifiers, where identifiers can refer to either locally declared variables or to globally declared callables (there are currently no global constants in Q#). Operators, combinators, and modifiers can be used to combine these identifiers into a wider variety of expressions.

- *Operators* in a sense are nothing but dedicated syntax for particular callables.

  > Even though Q# is not yet expressive enough to formally capture the capabilities of each operator in the form of a backing callable declaration, that should be remedied in the future.

- *Modifiers* can only be applied to certain expressions. One or more modifiers can be applied to expressions that are either
  - identifiers
  - array item access expressions
  - named item access expressions
  - an expression within parenthesis which is the same as a single item tuple. For more information, see singleton tuple equivalence). They can either precede (prefix) the expression or follow (postfix) the expression. They are thus special unary operators that bind tighter than function or operation calls, but less tight than any kind of item access. Concretely, functors are prefix modifiers, whereas the unwrap operator is a postfix modifier.
- Function, operation, and item access calls can be seen as a special type of operator, similar to modifiers. The're all subject to the same restrictions regarding where they can be applied; we refer to them as *combinators*.

The section on precedence and associativity contains a complete list of all operators as well as a complete list of all modifiers and combinators.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Precedence and associativity

Article • 08/15/2024

Precedence and associativity define the order in which operators are applied. Operators with higher precedence are bound to their arguments (operands) first, while operators with the same precedence bind in the direction of their associativity. For example, the expression `1+2*3` according to the precedence for addition and multiplication is equivalent to `1+(2*3)`, and `2^3^4` equals `2^(3^4)` since exponentiation is right-associative.

Operators

The following table lists the available operators in Q#, as well as their precedence and associativity. Additional modifiers and combinators are also listed, and bind tighter than any of these operators.

Expand table

Description	Syntax	Operator	Associativity	Precedence
copy-and-update operator	`w/ <-`	ternary	left	1
range operator	`..`	infix	left	2
conditional operator	`? |`	ternary	right	3
logical OR	`or`	infix	left	4
logical AND	`and`	infix	left	5
bitwise OR	`|||`	infix	left	6
bitwise XOR	`^^^`	infix	left	7
bitwise AND	`&&&`	infix	left	8
equality	`==`	infix	left	9
inequality	`!=`	infix	left	9
less-than-or-equal	`<=`	infix	left	10
less-than	`<`	infix	left	11
greater-than-or-equal	`>=`	infix	left	11

Description	Syntax	Operator	Associativity	Precedence
greater-than	`>`	infix	left	11
right shift	`>>>`	infix	left	12
left shift	`<<<`	infix	left	12
addition or concatenation	`+`	infix	left	13
subtraction	`-`	infix	left	13
multiplication	`*`	infix	left	14
division	`/`	infix	left	14
modulus	`%`	infix	left	14
exponentiation	`^`	infix	right	15
bitwise NOT	`~~~`	prefix	right	16
logical NOT	`not`	prefix	right	16
negative	`-`	prefix	right	16

Copy-and-update expressions necessarily need to have the lowest precedence to ensure a consistent behavior of the corresponding evaluate-and-reassign statement. Similar considerations hold for the range operator to ensure a consistent behavior of the corresponding contextual expression.

Modifiers and combinators

Modifiers can be seen as special operators that can be applied to certain expressions only. They can be assigned an artificial precedence to capture their behavior.

For more information, see Expressions.

This artificial precedence is listed in the following table, along with how the precedence of operators and modifiers relates to how tightly item access combinators (`[`,`]` and `::` respectively) and call combinators (`(`, `)`) bind.

Expand table

Description	Syntax	Operator	Associativity	Precedence
Call combinator	`(` `)`	n/a	left	17

Description	Syntax	Operator	Associativity	Precedence
Adjoint functor	`Adjoint`	prefix	right	18
Controlled functor	`Controlled`	prefix	right	18
Unwrap application	`!`	postfix	left	19
Named item access	`.`	n/a	left	20
Array item access	`[ ]`	n/a	left	20
Function lambda	`->`	n/a	right	21
Operation lambda	`=>`	n/a	right	21

To illustrate the implications of the assigned precedences, suppose you have a unitary operation `DoNothing` (as defined in Specialization declarations), a callable `GetStatePrep` that returns a unitary operation, and an array `algorithms` that contains items of type `Algorithm` defined as follows

Q#

```
struct Algorithm {
    Register : Qubit[],
    Initialize : Transformation,
    Apply : Transformation,
}
```

The following expressions, then, are all valid:

Q#

```
GetStatePrep()(arg)
Adjoint DoNothing()
Controlled Adjoint DoNothing(cs, ())
Controlled algorithms[0].Apply!(cs, _)
algorithms[0].Register![i]
```

Looking at the precedences defined in the table above, you can see that the parentheses around `(Transformation(GetStatePrep()))` are necessary for the subsequent unwrap operator to be applied to the `Transformation` value rather than the returned operation. However, parentheses are not required in `GetStatePrep()(arg)`; functions are applied left-to-right, so this expression is equivalent to `(GetStatePrep())(arg)`. Functor applications also don't require parentheses around them in order to

invoke the corresponding specialization, nor do array or named item access expressions. Thus, the expression `arr2D[i][j]` is perfectly valid, as is `algorithms[0]::Register![i]`.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Copy-and-update expressions

Article • 12/20/2024

To reduce the need for mutable bindings, Q# supports copy-and-update expressions for arrays, which allow you to access items via an index or range of indices.

Copy-and-update expressions instantiate a new array with all items set to the corresponding value in the original array, except the certain specified items, which are set to the ones defined on the right-hand side of the expression. They are constructed using a ternary operator `w/ <-`; the syntax `w/` should be read as the commonly used short notation for "with":

Q#

```
    original w/ itemAccess <- modification
```

where `original` is an array expression, `itemAccess` is any expression that is valid for array slicing, and `modification` is the new value or values. Concretely, the `itemAccess` expression can be of type `Int` or `Range`. If `itemAccess` is a value of type `Int`, then the type of `modification` has to match the item type of the array. If `itemAccess` is a value of type `Range`, then the type of `modification` has to be the same as the array type.

For example, if `arr` contains an array `[0, 1, 2, 3]`, then

- `arr w/ 0 <- 10` is the array `[10, 1, 2, 3]`.
- `arr w/ 2 <- 10` is the array `[0, 1, 10, 3]`.
- `arr w/ 0..2..3 <- [10, 12]` is the array `[10, 1, 12, 3]`.

In terms of precedence, the copy-and-update operator is left-associative and has lowest precedence, and, in particular, lower precedence than the range operator (`..`) or the ternary conditional operator (`? |`). The chosen left associativity allows easy chaining of copy-and-update expressions:

Q#

```
    let model = ArrayConstructor()
        w/ 1 <- alpha
        w/ 3 <- gamma
        w/ 5 <- epsilon;
```

As for any operator that constructs an expression of the same type as the left-most expression involved, the corresponding evaluate-and-reassign statement is available.

The two following statements, for example, achieve the following: The first statement declares a mutable variable `arr` and binds it to the default value of an integer array. The second statement then builds a new array with the first item (with index 0) set to 10 and reassigns it to `arr`.

Q#

```
mutable arr = [0, size = 3]; // arr contains [0, 0, 0]
arr w/= 0 <- 10;             // arr contains [10, 0, 0]
```

The second statement is just short-hand for the more verbose syntax `arr = arr w/ 0 <- 10;`.

Copy-and-update expressions allow the efficient creation of new arrays based on existing ones. The implementation for copy-and-update expressions avoids copying the entire array by duplicating only the necessary parts to achieve the desired behavior and performs an in-place modification if possible. Hence, array initializations do not incur additional overhead due to immutability.

The `Std.Arrays` namespace provides an arsenal of convenient tools for array creation and manipulation.

Copy-and-update expressions are a convenient way to construct new arrays on the fly; the following expression, for example, evaluates to an array with all items set to `PauliI`, except the item at index `i`, which is set to `PauliZ`:

Q#

```
[PauliI, size = n] w/ i <- PauliZ
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Conditional expressions

Article • 02/29/2024

Conditional expressions consist of three sub-expressions, where the left-most sub-expression is of type `Bool` and determines which one of the two other sub-expressions is evaluated. They are of the form

```
Q#
cond ? ifTrue | ifFalse
```

Specifically, if `cond` evaluates to `true`, then the conditional expression evaluates to the `ifTrue` expression; otherwise, it evaluates to the `ifFalse` expression. The other expression (the `ifFalse` and `ifTrue` expression, respectively) is never evaluated, much like the branches in an `if` statement. For instance, in an expression `a == b ? C(qs) | D(qs)`, if `a` equals `b`, then the callable `C` is invoked. Otherwise, `D` is invoked.

The types of the `ifTrue` and the `ifFalse` expression have to have a common base type. Independent of which one ultimately yields the value to which the expression evaluates, its type always matches the determined base type.

For example, if

- `Op1` is of type `Qubit[] => Unit is Adj`
- `Op2` is of type `Qubit[] => Unit is Ctl`
- `Op3` is of type `Qubit[] => Unit is Adj + Ctl`

then

- `cond ? Op1 | Op2` is of type `Qubit[] => Unit`
- `cond ? Op1 | Op3` is of type `Qubit[] => Unit is Adj`
- `cond ? Op2 | Op3` is of type `Qubit[] => Unit is Ctl`

For more details, see subtyping.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Comparative expressions

Article • 02/21/2025

Equality comparisons

Equality comparisons (`==`) and *inequality comparisons* (`!=`) are currently limited to the following data types: `Int`, `BigInt`, `Double`, `String`, `Bool`, `Result`, `Pauli`, and `Qubit`. Equality comparisons of `struct` types and callables aren't currently supported.

Equality comparison for values of type `Qubit` evaluates whether the two expressions identify the same qubit. There's no notion of a quantum state in Q#; equality comparison, in particular, does *not* access, measure, or modify the quantum state of the qubits.

Equality comparisons for `Double` values may be misleading due to rounding effects. For instance, the following comparison evaluates to `false` due to rounding errors: `49.0 * (1.0/49.0) == 1.0`.

Equality comparison of arrays, and tuples are supported by comparisons of their items, and are only supported if all of their nested types support equality comparison.

Equality comparison of close-ended ranges is supported, and two ranges are considered equal if they produce the same sequence of integers. For example, the following two ranges

Q#

```
    let r1 = 0..2..5; // generates the sequence 0,2,4
    let r2 = 0..2..4; // generates the sequence 0,2,4
```

are considered equal. Equality comparison of open-ended ranges isn't supported.

Quantitative comparison

The operators *less-than* (`<`), *less-than-or-equal* (`<=`), *greater-than* (`>`), and *greater-than-or-equal* (`>=`) define quantitative comparisons. They can only be applied to data types that support such comparisons, that is, the same data types that can also support arithmetic expressions.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Logical expressions

Article • 02/29/2024

Logical operators are expressed as keywords. Q# supports the standard logical operators *AND* (`and`), *OR* (`or`), and *NOT* (`not`). Currently, there is not an operator for a logical *XOR*. All of these operators act on operands of type `Bool`, and result in an expression of type `Bool`. As is common in most languages, the evaluation of *AND* and *OR* short-circuits, meaning if the first expression of *OR* evaluates to `true`, the second expression is not evaluated, and the same holds if the first expression of *AND* evaluates to `false`. The behavior of conditional expressions in a sense is similar, in that only ever the condition and one of the two expressions is evaluated.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Bitwise expressions

Article • 02/21/2025

Bitwise operators are expressed as three non-letter characters. In addition to bitwise versions for *AND* (`&&&`), *OR* (`|||`), and *NOT* (`~~~`), a bitwise *XOR* (`^^^`) exists as well. They expect operands of type `Int` or `BigInt`, and for binary operators, the type of both operands has to match. The type of the entire expression equals the type of the operands.

Additionally, left- and right-shift operators (`<<<` and `>>>` respectively) exist, multiplying or dividing the given left-hand-side (lhs) expression by powers of two. The expression `lhs <<< 3` shifts the bit representation of `lhs` by three, meaning `lhs` is multiplied by `2^3`, provided that is still within the valid range for the data type of `lhs`. The lhs may be of type `Int` or `BigInt`. The right-hand-side expression always has to be of type `Int`. The resulting expression is of the same type as the lhs operand.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Arithmetic expressions

Article • 02/29/2024

Arithmetic operators are addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), negation (`-`), and exponentiation (`^`). They can be applied to operands of type `Int`, `BigInt`, or `Double`. Additionally, for integral types (`Int` and `BigInt`), an operator computing the modulus (`%`) is available.

For binary operators, the type of both operands must match, except for exponentiation; an exponent for a value of type `BigInt` must be of type `Int`. The type of the entire expression matches the type of the left operand. For exponentiation of `Int` and `BitInt`, the behavior is undefined if the exponent is negative or requires more than 32 bits to represent (that is, if it is larger than 2147483647).

Division and modulus for values of type `Int` and `BigInt` follow the following behavior for negative numbers:

Expand table

`A`	`B`	`A / B`	`A % B`
5	2	2	1
5	-2	-2	1
-5	2	-2	-1
-5	-2	2	-1

That is, `a % b` always has the same sign as `a`, and `b * (a / b) + a % b` always equals `a`.

Q# does not support automatic conversions between arithmetic data types or any other data types for that matter. This is of importance especially for the `Result` data type and facilitates restricting how runtime information can propagate. It has the benefit of avoiding accidental errors, such as ones related to precision loss.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Concatenation

Article • 02/21/2025

Concatenations are supported for values of type `String` and arrays. In both cases, they're expressed via the operator `+`. For instance, `"Hello " + "world!"` evaluates to `"Hello world!"`, and `[1, 2, 3] + [4, 5, 6]` evaluates to `[1, 2, 3, 4, 5, 6]`.

Concatenating two arrays requires that both arrays be of the same type. This requirement differs from constructing an array literal where the compiler determines a common base type for all array items. This difference is because arrays are treated as invariant. The type of the entire expression matches the type of the operands.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Functor application

Article • 04/19/2024

Functors are factories that allow you to access particular specialization implementations of a callable. Q# currently supports two functors; the `Adjoint` and the `Controlled`, both of which can be applied to operations that provide the necessary specializations.

The `Controlled` and `Adjoint` functors commute; if `ApplyUnitary` is an operation that supports both functors, then there is no difference between `Controlled Adjoint ApplyUnitary` and `Adjoint Controlled ApplyUnitary`. Both have the same type and, upon invocation, execute the implementation defined for the `controlled adjoint` specialization.

Adjoint functor

If the operation `ApplyUnitary` defines a unitary transformation *U* of the quantum state, `Adjoint ApplyUnitary` accesses the implementation of *U†*. The `Adjoint` functor is its own inverse, since *(U†)† = U* by definition. For example, `Adjoint Adjoint ApplyUnitary` is the same as `ApplyUnitary`.

The expression `Adjoint ApplyUnitary` is an operation of the same type as `ApplyUnitary`; it has the same argument and return type and supports the same functors. Like any operation, it can be invoked with an argument of suitable type. The following expression applies the adjoint specialization of `ApplyUnitary` to an argument `arg`:

Q#

```
Adjoint ApplyUnitary(arg)
```

Controlled functor

For an operation `ApplyUnitary` that defines a unitary transformation *U* of the quantum state, `Controlled ApplyUnitary` accesses the implementation that applies *U* conditional on all qubits in an array of control qubits being in the |1⟩ state.

The expression `Controlled ApplyUnitary` is an operation with the same return type and operation characteristics as `ApplyUnitary`, meaning it supports the same functors. It takes an argument of type `(Qubit[], <TIn>)`, where `<TIn>` should be replaced with the argument type of `ApplyUnitary`, taking singleton tuple equivalence into account.

Expand table

Operation	Argument Type	Controlled Argument Type
X	`Qubit`	`(Qubit[], Qubit)`
SWAP	`(Qubit, Qubit)`	`(Qubit[], (Qubit, Qubit))`

Concretely, if `cs` contains an array of qubits, `q1` and `q2` are two qubits, and the operation `SWAP` is as defined here, then the following expression exchanges the state of `q1` and `q2` if all qubits in `cs` are in the |1⟩ state:

Q#

```
Controlled SWAP(cs, (q1, q2))
```

> **Note**
>
> Conditionally applying an operation based on the control qubits being in a state other than the |1⟩ state may be achieved by applying the appropriate adjointable transformation to the control qubits before invocation, and applying the inverses after. Conditioning the transformation on all control qubits being in the |0⟩ state, for example, can be achieved by applying the `X` operation before and after. This can be conveniently expressed using a conjugation. Nonetheless, the verbosity of such a construct may merit additional support for a more compact syntax in the future.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Item access

Article • 08/20/2024

Q# supports item access for array items and for `struct` types. In both cases, the access is read-only; the value cannot be changed without creating a new instance using a copy-and-update expression.

Array item access and array slicing

Given an array expression and an expression of type `Int` or `Range`, a new expression may be formed using the array item access operator consisting of `[` and `]`.

If the expression inside the brackets is of type `Int`, then the new expression contains the array item at that index. For example, if `arr` is of type `Double[]` and contains five or more items, then `arr[4]` is an expression of type `Double`.

If the expression inside the brackets is of type `Range`, then the new expression contains an array of all the items indexed by the specified `Range`. If the `Range` is empty, then the resulting array is empty. For example,

Q#

```
let arr = [10, 11, 36, 49];
let ten = arr[0]; // contains the value 10
let odds = arr[1..2..4]; // contains the value [11, 49]
let reverse = arr[...-1...]; // contains the value [49, 36, 11, 10]
```

In the last line of the example, the start and end value of the range have been omitted for convenience. For more information, see Contextual expressions.

If the array expression is not a simple identifier, it must be enclosed in parentheses in order to extract an item or a slice. For instance, if `arr1` and `arr2` are both arrays of integers, an item from the concatenation would be expressed as `(arr1 + arr2)[13]`. For more information, see Precedence and associativity.

All arrays in Q# are zero-based, that is, the first element of an array `arr` is always `arr[0]`. An exception is thrown at runtime if the index or one of the indices used for slicing is outside the bounds of the array, for example, if it is less than zero or larger or equal to the length of the array.

Item access for struct types

The contained items in a `struct` can be accessed two ways:

- By reference, using a period (`.`)
- By deconstruction, using the `unwrap` (`!`) operator.

For example, given the struct `Complex`

Q#

```
struct Complex {
    Real : Double,
    Imaginary : Double,
}

// create an instance of type Complex
let complex = Complex(1., 0.);

// item access via deconstruction
let (re, _) = complex!;

// item access via name reference
let im = complex.Imaginary;
```

Access via deconstruction makes use of the unwrap operator (`!`). The unwrap operator returns a tuple of all contained items, where the shape of the tuple matches the one defined in the declaration, and a single item tuple is equivalent to the item itself (see this section).

The `!` operator has lower precedence than both item access operators, but higher precedence than any other operator. For a complete list of precedences, see Precedence and associativity.

For more information on deconstruction syntax, see Variable declarations and reassignments.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Call expressions

Article • 02/29/2024

Call expressions are an important part of any programming language. Operation and function calls, much like partial applications, can be used as an expression anywhere as long as the returned value is of a suitable type.

The usefulness of calling operations in this form primarily lies in debugging, and such operation calls are one of the most common constructs in any Q# program. At the same time, operations can only be called from within other operations and not from within functions. For more information, see also Qubits.

With callables being first-class values, call expressions are a generic way of supporting patterns that aren't common enough to merit their own dedicated language construct, or dedicated syntax has not (yet) been introduced for other reasons. Some examples of library methods that serve that purpose are `ApplyIf`, which invokes an operation conditional on a classical bit being set; `ApplyToEach`, which applies a given operation to each element in an array; and `ApplyWithInputTransformation`, as shown in the following sample.

Q#

```
operation ApplyWithInputTransformation<'TArg, 'TIn>(
    fn : 'TIn -> 'TArg,
    op : 'TArg => Unit,
    input : 'TIn
) : Unit {

    op(fn(input));
}
```

`ApplyWithInputTransformation` takes a function `fn`, an operation `op`, and an `input` value as arguments and then applies the given function to the input before invoking the given operation with the value returned from the function.

For the compiler to auto-generate the specializations to support particular functors, it usually requires that the called operations support those functors as well. The two exceptions are calls in outer blocks of conjugations, which always need to support the `Adjoint` functor but never need to support the `Controlled` functor, and self-adjoint operations, which support the `Adjoint` functor without imposing any additional requirements on the individual calls.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Returns and termination

Article • 02/21/2025

There are two expressions available that conclude the execution of the current subroutine or the program; the `return` and the `fail` expressions. Generally, callables may end their execution before executing all of their statements with a `return` or `fail` expression. A `return` expression just ends the execution of the current callable, while a `fail` ends the execution of the whole program and result in a runtime error.

Return expression

The `return` expression exits from the current callable and returns control to the callee. It changes the context of the execution by popping a stack frame.

The expression always returns a value to the context of the callee; it consists of the keyword `return`, followed by an expression of the appropriate type. The return value is evaluated before any terminating actions are performed and control is returned. Terminating actions include, for example, cleaning up and releasing qubits that are allocated within the context of the callable. When running on a simulator or validator, terminating actions often also include checks related to the state of those qubits. For example, they may check whether they are properly disentangled from all qubits that remain live.

The `return` expression at the end of a callable that returns a `Unit` value may be omitted. In that case, control is returned automatically when all statements complete and all terminating actions performed. Callables may contain multiple `return` expressions, albeit the adjoint implementation for operations containing multiple `return` expressions can't be automatically generated.

For example,

```
Q#
return 1;
```

or

```
Q#
return ();
```

Fail expression

The `fail` expression ends the computation entirely. It corresponds to a fatal error that aborts the program.

It consists of the keyword `fail`, followed by an expression of type `String`. The `String` should provide information about the encountered failure.

For example,

```
Q#
fail "Impossible state reached";
```

or, using an interpolated string,

```
Q#
fail $"Syndrome {syn} is incorrect";
```

In addition to the given `String`, a `fail` expression ideally collects and permits the retrieval of information about the program state. This facilitates diagnosing and remedying the source of the error, and requires support from the executing runtime and firmware, which may vary across different targets.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Iterations

Article • 02/21/2025

Loops that iterate over a sequence of values are expressed as `for` loops in Q#. A `for` loop in Q# doesn't break based on a condition but instead corresponds to an iteration, or what is often expressed as `foreach` or `iter` in other languages. There are currently two data types in Q# that support iteration: arrays and ranges.

The expression consists of the keyword `for`, followed by a symbol or symbol tuple, the keyword `in`, an expression of array or `Range` type, and a statement block.

The statement block (the body of the loop) is run repeatedly, with one or more loop variables bound to each value in the range or array. The same deconstruction rules apply to the defined loop variables as to any other variable assignment, such as bindings in `let`, `mutable`, `use`, and `borrow` statements. The loop variables themselves are immutably bound, can't be reassigned within the body of the loop, and go out of scope when the loop terminates. The expression over which the loop iterates is evaluated before entering the loop and doesn't change while the loop is running.

This scenario is illustrated in the following example. Suppose `qubits` is a value of type `Qubit[]`, then

Q#

```
for qubit in qubits {
    H(qubit);
}

mutable results : (Int, Result)[] = [];
for index in 0 .. Length(qubits) - 1 {
    results += [(index, M(qubits[index]))];
}

mutable accumulated = 0;
for (index, measured) in results {
    if measured == One {
        accumulated += 1 <<< index;
    }
}
```

Target-specific restrictions

Because there are no `break` or `continue` primitives in Q#, the length of the loop is known as soon as the iteration value is known. As such, `for` loops can be run on all quantum hardware.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Conditional loops

Article • 02/18/2025

Like most classical programming languages, Q# supports loops that break based on a condition: loops for which the number of iterations is unknown and may vary from run to run. Since the instruction sequence is unknown at compile-time, the compiler handles these conditional loops in a particular way in a quantum runtime.

> ⓘ **Important**
>
> **Quantum hardware restrictions**
>
> Loops that break based on a condition are challenging to process on quantum hardware *if the condition depends on measurement outcomes*, since the length of the instruction sequence to run isn't known ahead of time.
>
> Despite their common presence in particular classes of quantum algorithms, current hardware doesn't yet provide native support for these kinds of control flow constructs. Running these kinds of loops on quantum hardware can potentially be supported in the future by imposing a maximum number of iterations, or as other hardware support becomes available. Quantum simulators, however, run any loops based on measurements.

Compiling loops

As long as the condition doesn't depend on quantum measurements, conditional loops are processed with a just-in-time compilation before sending the instruction sequence to the quantum processor. In particular, using conditional loops within functions is unproblematic since code within functions can always run on conventional (non-quantum) hardware. Q#, therefore, supports the use of traditional `while` loops within functions.

Repeat expression

When you run programs on quantum simulators, Q# allows you to express control flow that depends on the results of quantum measurements. This capability enables probabilistic implementations that can significantly reduce computational costs. A common example is the *repeat-until-success* pattern, which repeats a computation until a certain condition - which usually depends on a measurement - is satisfied. Such

`repeat` loops are widely used in particular classes of quantum algorithms. Q# hence has a dedicated language construct to express them, despite that they still pose a challenge for execution on quantum hardware.

The `repeat` expression takes the following form

Q#

```
repeat {
    // ...
}
until condition
fixup {
    // ...
}
```

or alternatively

Q#

```
repeat {
    // ...
}
until condition;
```

where `condition` is an arbitrary expression of type `Bool`.

The `repeat` loop runs a block of statements before evaluating a condition. If the condition evaluates to true, the loop exits. If the condition evaluates to false, an additional block of statements defined as part of an optional `fixup` block, if present, is run before entering the next loop iteration.

While loop

A more familiar-looking loop for classical computations is the `while` loop, which consists of the keyword `while`, an expression of type `Bool`, and a statement block. For example, if `arr` is an array of positive integers,

Q#

```
mutable (item, index) = (-1, 0);
while index < Length(arr) && item < 0 {
    item = arr[index];
```

```
    index += 1;
}
```

The statement block is run as long as the condition evaluates to `true`.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Conditional branching

Article • 02/29/2024

Conditional branching is expressed in the form of `if` expressions. An `if` expression consists of an `if` clause, followed by zero or more `elif` clauses and optionally an else-block. Each clause follows the pattern

```
Q#

keyword condition {
    <statements>
}
```

where `keyword` is replaced with `if` or `elif` respectively, `condition` is an expression of type `Bool`, and `<statements>` is to be replaced with zero or more statements. The optional `else`-block consists of the keyword `else` followed by zero or more statements enclosed in braces, `{ }`.

The first block for which the `condition` evaluates to `true` will run. The `else` block, if present, runs if none of the conditions evaluate to `true`. The block is executed in its own scope, meaning any bindings made as part of the block are not visible after the block ends.

For example, suppose `qubits` is value of type `Qubit[]` and `r1` and `r2` are of type `Result`,

```
Q#

if r1 == One {
    let q = qubits[0];
    H(q);
}
elif r2 == One {
    let q = qubits[1];
    H(q);
}
else {
    H(qubits[2]);
}
```

You can also express simple branching in the form of a conditional expression.

Target-specific restrictions

The tight integration between control-flow constructs and quantum computations poses a challenge for current quantum hardware. Certain quantum processors do not support branching based on measurement outcomes. As such, comparison for values of type `Result` will always result in a compilation error for Q# programs that are targeted to run on such hardware.

Other quantum processors support specific kinds of branching based on measurement outcomes. The more general `if` expressions supported in Q# are compiled into suitable instructions that can be run on such processors. The imposed restrictions are that values of type `Result` may only be compared as part of the condition within `if` expressions in operations. Furthermore, the conditionally run blocks cannot contain any `return` expressions or update mutable variables that are declared outside that block.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Conjugations

Article • 02/21/2025

Conjugations are common in quantum computations. In mathematical terms, they are patterns of the form $U^\dagger V U$ for two unitary transformations U and V. That pattern is relevant due to the particularities of quantum memory: computations build up quantum correlations, or *entanglement*, to use the unique assets of quantum. However, that also means that once a subroutine no longer needs its qubits, those qubits cannot easily be reset and released since observing their state would impact the rest of the system. For that reason, the effect of a previous computation usually needs to be reversed before releasing and reusing quantum memory.

Q# hence has a dedicated construct for expressing computations that require such a cleanup. The expressions consist of two code blocks, one containing the implementation of U and one containing the implementation of V. The *uncomputation* (that is, the application of $U^\dagger$) is done automatically as part of the expression.

The expression takes the form

```
Q#

within {
    <statements>
}
apply {
    <statements>
}
```

where `<statements>` is replaced with any number of statements defining the implementation of U and V respectively. Both blocks may contain arbitrary classical computations, aside from the usual restrictions for automatically generating adjoint versions that apply to the `within` block. Mutably bound variables used as part of the `within` block may not be reassigned as part of the `apply` block.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Closures

Article • 07/27/2022

Closures are callables that capture variables from the enclosing environment. Both function and operation closures can be created. An operation closure can be created inside a function, but it can only be applied in an operation.

Q# has two mechanisms for creating closures: lambda expressions and partial application.

Lambda expressions

A lambda expression creates an anonymous function or operation. The basic syntax is a symbol tuple to bind the parameters, an arrow (`->` for a function and `=>` for an operation), and an expression to evaluate when applied.

Q#

```
// Function that captures 'x':
y -> x + y

// Operation that captures 'qubit':
deg => Rx(deg * PI() / 180.0, qubit)

// Function that captures nothing:
(x, y) -> x + y
```

Parameters

Parameters are bound using a symbol tuple that is identical to the left-hand side of a variable declaration statement. The type of the parameter tuple is implicit. Type annotations are not supported; if type inference fails, you may need to create a top-level callable declaration and use partial application instead.

Mutable capture variables

Mutable variables cannot be captured. If you only need to capture the value of a mutable variable at the instant the lambda expression is created, you can create an immutable copy:

Q#

```
// ERROR: 'variable' cannot be captured.
mutable variable = 1;
let f = () -> variable;

// OK.
let value = variable;
let g = () -> value;
```

Characteristics

The characteristics of an anonymous operation are inferred based on the body of the lambda expression. For example:

Q#

```
// Has type Unit => Unit is Adj + Ctl because X is known to be Adj + Ctl.
() => X(q)


// Has type Unit => Result because M is neither Adj nor Ctl.
() => M(q)
```

Because of limitations in characteristics inference, this is based only on type information known at the point where the lambda expression occurs. For example:

Q#

```
let foo = op => op(q);
foo(X);
```

`foo` is inferred to have the following type based on both the body of the lambda and the type of `X`:

Q#

```
(Qubit => Unit is Adj + Ctl) => Unit
```

But the most specific type that `foo` could have is:

Q#

```
(Qubit => Unit is Adj + Ctl) => Unit is Adj + Ctl
```

If you need different characteristics for an operation lambda than what was inferred, you will need to create a top-level operation declaration instead.

Partial application

Partial application is a convenient shorthand for applying some, but not all, of a callable's arguments. The syntax is the same as a call expression, but unapplied arguments are replaced with `_`. Conceptually, partial application is equivalent to a lambda expression that captures the applied arguments and takes in the unapplied arguments as parameters.

For example, given that `f` is a function and `o` is an operation, and the captured variable `x` is immutable:

Partial application	Lambda expression
`f(x, _)`	`a -> f(x, a)`
`o(x, _)`	`a => o(x, a)`
`f(_, (1, _))`	`(a, b) -> f(a, (1, b))`[^1]
`f((_, _, x), (1, _))`	`((a, b), c) -> f((a, b, x), (1, c))`

Mutable capture variables

Unlike lambda expressions, partial application can automatically capture a copy of the value of a mutable variable:

```
Q#

mutable variable = 1;
let f = Foo(variable, _);
```

This is equivalent to the following lambda expression:

```
Q#

mutable variable = 1;
let value = variable;
let f = x -> Foo(value, x);
```

[^1]: The parameter tuple is strictly written `(a, (b))`, but (b) is equivalent to b.

Contextual and omitted expressions

Article • 02/21/2025

Contextual expressions are expressions that are only valid in certain contexts, such as the use of item names in copy-and-update expressions without having to qualify them.

Expressions can be *omitted* when they can be inferred and automatically inserted by the compiler, such as evaluate-and-reassign statements.

Open-ended ranges are another example that applies to both contextual and omitted expressions. They are valid only within a certain context, and the compiler translates them into normal `Range` expressions during compilation by inferring suitable boundaries.

A value of type `Range` generates a sequence of integers, specified by a start value, a step value (optional), and an end value. For example, the `Range` literal expression `1..3` generates the sequence 1,2,3. Likewise, the expression `3..-1..1` generates the sequence 3,2,1. You can also use ranges to create a new array from an existing one by slicing, for example:

Q#

```
let arr = [1,2,3,4];
let slice1 = arr[1..2..4];  // contains [2,4]
let slice2 = arr[2..-1..0]; // contains [3,2,1]
```

You cannot define an infinite range in Q#; the start and end values always need to be specified. The only exception is when you use a `Range` to slice an array. In that case, the compiler can reasonably infer the start or end values of the range.

In the previous array slicing examples, it is reasonable for the compiler to assume that the intended range end should be the index of the last item in the array if the step size is positive. If the step size is negative, then the range end likely should be the index of the first item in the array, `0`. The converse holds for the start of the range.

To summarize, if you omit the range start value, the inferred start value

- is zero if no step is specified or the specified step is positive.
- is the length of the array minus one if the specified step is negative.

If you omit the range end value, the inferred end value

- is the length of the array minus one if no step is specified or the specified step is positive.
- is zero if the specified step is negative.

Q# hence allows the use of open-ended ranges within array slicing expressions, for example:

Q#

```
let arr = [1,2,3,4,5,6];
let slice1  = arr[3...];      // slice1 is [4,5,6];
let slice2  = arr[0..2...];   // slice2 is [1,3,5];
let slice3  = arr[...2];      // slice3 is [1,2,3];
let slice4  = arr[...2..3];   // slice4 is [1,3];
let slice5  = arr[...2...];   // slice5 is [1,3,5];
let slice7  = arr[4..-2...];  // slice7 is [5,3,1];
let slice8  = arr[...-1..3];  // slice8 is [6,5,4];
let slice9  = arr[...-1...];  // slice9 is [6,5,4,3,2,1];
let slice10 = arr[...];       // slice10 is [1,2,3,4,5,6];
```

Since the determination of whether the range step is positive or negative happens at runtime, the compiler inserts a suitable expression that is evaluated at runtime. For omitted end values, the inserted expression is `step < 0 ? 0 | Length(arr)-1`, and for omitted start values it is `step < 0 ? Length(arr)-1 | 0`, where `step` is the expression given for the range step, or `1` if no step is specified.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Literals

Article • 08/20/2024

Unit literal

The only existing literal for the Unit type is the value `()`.

The `Unit` value is commonly used as an argument to callables, either because no other arguments need to be passed or to delay execution. It is also used as return value when no other value needs to be returned, which is the case for unitary operations, that is, operations that support the `Adjoint` and/or the `Controlled` functor.

Int literals

Value literals for the Int type can be expressed in binary, octal, decimal, or hexadecimal representation. Literals expressed in binary are prefixed with `0b`, with `0o` for octal, and with `0x` for hexadecimal. There is no prefix for the commonly used decimal representation.

Expand table

Representation	Value Literal
Binary	`0b101010`
Octal	`0o52`
Decimal	`42`
Hexadecimal	`0x2a`

BigInt literals

Value literals for the BigInt type are always postfixed with `L` and can be expressed in binary, octal, decimal, or hexadecimal representation. Literals expressed in binary are prefixed with `0b`, with `0o` for octal, and with `0x` for hexadecimal. There is no prefix for the commonly used decimal representation.

Expand table

Representation	Value Literal
Binary	`0b101010L`
Octal	`0o52L`
Decimal	`42L`
Hexadecimal	`0x2aL`

Double literals

Value literals for the Double type can be expressed in standard or scientific notation.

Expand table

Representation	Value Literal
Standard	`0.1973269804`
Scientific	`1.973269804e-1`

If nothing follows after the decimal point, then the digit after the decimal point may be omitted. For example, `1.` is a valid `Double` literal and the same as `1.0`.

Bool literals

Existing literals for the Bool type are `true` and `false`.

String literals

A value literal for the String type is an arbitrary length sequence of Unicode characters enclosed in double quotes. Inside of a string, the back-slash character `\` may be used to escape a double quote character, and to insert a new-line as `\n`, a carriage return as `\r`, and a tab as `\t`.

The following are examples for valid string literals:

Q#

```
"This is a simple string."
"\"This is a more complex string.\", she said.\n"
```

Q# also supports *interpolated strings*. An interpolated string is a string literal that may contain any number of interpolation expressions. These expressions can be of arbitrary types. Upon construction, the expressions are evaluated and their `String` representation is inserted at the corresponding location within the defined literal. Interpolation is enabled by prepending the special character `$` directly before the initial quote, with no white space between them.

For instance, if `res` is an expression that evaluates to `1`, then the second sentence in the following `String` literal displays "The result was 1.":

```
Q#
```

```
$"This is an interpolated string. The result was {res}."
```

Qubit literals

No literals for the Qubit type exist, since quantum memory is managed by the runtime. Values of type `Qubit` can hence only be obtained via allocation.

Values of type `Qubit` represent an opaque identifier by which a quantum bit, or *qubit*, can be addressed. The only operator they support is equality comparison. For more information on the `Qubit` data type, See Qubits.

Result literals

Existing literals for the Result type are `Zero` and `One`.

Values of type `Result` represent the result of a binary quantum measurement. `Zero` indicates a projection onto the +1 eigenspace, `One` indicates a projection onto the -1 eigenspace.

Pauli literals

Existing literals for the Pauli type are `PauliI`, `PauliX`, `PauliY`, and `PauliZ`.

Values of type `Pauli` represent one of the four single-qubit Pauli matrices, with `PauliI` representing the identity. Values of type `Pauli` are commonly used to denote the axis for rotations and to specify with respect to which basis to measure.

Range literals

Value literals for the Range type are expressions of the form `start..step..stop`, where `start`, `step`, and `end` are expressions of type `Int`. If the step size is one, it may be omitted. For example, `start..stop` is a valid `Range` literal and the same as `start..1..stop`.

Values of type `Range` represent a sequence of integers, where the first element in the sequence is `start`, and subsequent elements are obtained by adding `step` to the previous one, until `stop` is passed. `Range` values are inclusive at both ends, that is, the last element of the range is `stop` if the difference between `start` and `stop` is a multiple of `step`. A range may be empty if, for instance, `step` is positive and `stop < start`.

The following are examples for valid `Range` literals:

- `1..3` is the range 1, 2, 3.
- `2..2..5` is the range 2, 4.
- `2..2..6` is the range 2, 4, 6.
- `6..-2..2` is the range 6, 4, 2.
- `2..-2..1` is the range 2.
- `2..1` is the empty range.

For more information, see Contextual expressions.

Array literals

An array literal is a sequence of zero or more expressions, separated by commas and enclosed in brackets `[` and `]`; for example, `[1,2,3]`. All expressions must have a common base type, which is the item type of the array. If an empty array is specified with `[]`, a type annotation may be needed for the compiler to determine the appropriate type of the expression.

Arrays of arbitrary length may be created using a sized-array expression. Such an expression is of the form `[expr, size = s]`, where `s` can be any expression of type `Int` and `expr` is evaluated to a value that will be the items of the array repeated `s` times. For example, `[1.2, size = 3]` creates the same array as `[1.2, 1.2, 1.2]`.

Tuple literals

A tuple literal is a sequence of one or more expressions of any type, separated by commas and enclosed in parentheses `(` and `)`. The type of the tuple includes the information about each item type.

Expand table

Value Literal	Type
`("Id", 0, 1.)`	`(String, Int, Double)`
`(PauliX,(3,1))`	`(Pauli, (Int, Int))`

Tuples containing a single item are treated as identical to the item itself, both in type and value, which is called singleton tuple equivalence.

Tuples are used to bundle values together into a single value, making it easier to pass them around. This makes it possible for every callable to take exactly one input and return exactly one output.

Literals for struct types

Values of a struct type are assigned when you create a new instance, using either the `new` statement or the constructor created by the compiler.

For example, if `IntPair` is a `struct` defined as

```
Q#

struct IntPair {
    num1 : Int,
    num2 : Int,
}
```

then you create a new instance and assign values using

```
Q#

let MyPair = new IntPair { num1 = 5, num2 = 7 };
```

or

```
Q#

let MyPair = IntPair(5, 7);
```

Operation and function literals

Anonymous operations and functions can be created using a lambda expression.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Type System

Article • 02/21/2025

With the focus for quantum algorithm being more towards what should be achieved rather than on a problem representation in terms of data structures, taking a more functional perspective on language design is a natural choice. At the same time, the type system is a powerful mechanism that can be used for program analysis and other compile-time checks that facilitate formulating robust code.

Overall, the Q# type system is fairly minimalist, in the sense that there isn't an explicit notion of classes or interfaces as one might be used to from classical languages like C# or Java. We also take a somewhat pragmatic approach making incremental progress, such that certain constructs aren't yet fully integrated into the type system. An example of this are functors, which can be used within expressions but don't yet have a representation in the type system. Correspondingly, they can't currently be assigned or passed as arguments, similar as it's the case for type parametrized callables. We expect to make incremental progress in extending the type system to be more complete, and balance immediate needs with longer-term plans.

Available Types

All types in Q# are immutable.

Expand table

Type	Description
`Unit`	Represents a singleton type whose only value is `()`.
`Int`	Represents a 64-bit signed integer. Values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
`BigInt`	Represents signed integer values of any size.
`Double`	Represents a double-precision 64-bit floating-point number. Values range from -1.79769313486232e308 to 1.79769313486232e308, as well as NaN (not a number).
`Bool`	Represents Boolean values. Possible values are `true` or `false`.
`String`	Represents text as values that consist of a sequence of UTF-16 code units.
`Qubit`	Represents an opaque identifier by which virtual quantum memory can be addressed. Values of type `Qubit` are instantiated via allocation.

Type	Description
`Result`	Represents the result of a projective measurement onto the eigenspaces of a quantum operator with eigenvalues ±1. Possible values are `Zero` or `One`.
`Pauli`	Represents a single-qubit Pauli matrix. Possible values are `PauliI`, `PauliX`, `PauliY`, or `PauliZ`.
`Range`	Represents an ordered sequence of equally spaced `Int` values. Values may represent sequences in ascending or descending order.
Array	Represents values that each contain a sequence of values of the same type.
Tuple	Represents values that each contain a fixed number of items of different types. Tuples containing a single element are equivalent to the element they contain.
`struct`	Represents a user defined type consisting of named items of different types. Values are instantiated when declaring a new instance.
Operation	Represents a non-deterministic callable that takes one (possibly tuple-valued) input argument and returns one (possibly tuple-valued) output. Calls to operation values may have side effects and the output may vary for each call, even when invoked with the same argument.
Function	Represents a deterministic callable that takes one (possibly tuple-valued) input argument and returns one (possibly tuple-valued) output. Calls to function values don't have side effects and the output will always be the same given the same input.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Quantum-specific data types

Article • 02/21/2025

This topic describes the `Qubit` type, along with two other types that are somewhat specific to the quantum domain: `Pauli` and `Result`.

Qubit

Q# treats qubits as opaque items that can be passed to both functions and operations, but can only be interacted with by passing them to instructions that are native to the targeted quantum processor. Such instructions are always defined in the form of operations, since their intent is to modify the quantum state. The restriction that functions can't modify the quantum state, despite the fact that qubits can be passed as input arguments, is enforced by the requiring that functions can only call other functions, and can't call operations.

The Q# libraries are compiled against a standard set of intrinsic operations, meaning operations which have no definition for their implementation within the language. Upon targeting, the implementations that express them in terms of the instructions that are native to the execution target are linked in by the compiler. A Q# program thus combines these operations as defined by a target machine to create new, higher-level operations to express quantum computation. In this way, Q# makes it very easy to express the logic underlying quantum and hybrid quantum-classical algorithms, while also being very general with respect to the structure of a target machine and its realization of quantum state.

Within Q# itself, there is no type or construct in Q# that represents the quantum state. Instead, a qubit represents the smallest addressable physical unit in a quantum computer. As such, a qubit is a long-lived item, so Q# has no need for linear types. Hence, we don't explicitly refer to the state within Q#, but rather describe how the state is transformed by the program, for example, via the application of operations such as `X` and `H`. Similar to how a graphics shader program accumulates a description of transformations to each vertex, a quantum program in Q# accumulates transformations to quantum states, represented as entirely opaque references to the internal structure of a target machine.

A Q# program has no ability to introspect into the state of a qubit, and thus is entirely agnostic about what a quantum state is or on how it is realized. Rather, a program can call operations such as `Measure` to learn information about the quantum state of the computation.

Pauli

Values of type `Pauli` specify a single-qubit Pauli operator; the possibilities are `PauliI`, `PauliX`, `PauliY`, and `PauliZ`. `Pauli` values are used primarily to specify the basis for a quantum measurement.

Result

The `Result` type specifies the result of a quantum measurement. Q# mirrors most quantum hardware by providing measurements in products of single-qubit Pauli operators; a `Result` of `Zero` indicates that the +1 eigenvalue was measured, and a `Result` of `One` indicates that the -1 eigenvalue was measured. That is, Q# represents eigenvalues by the power to which -1 is raised. This convention is more common in the quantum algorithms community, as it maps more closely to classical bits.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Immutability

Article • 02/21/2025

All types in Q# are *value types*. Q# doesn't have a concept of a reference or pointer. Instead, it allows you to reassign a new value to a previously declared variable via an assignment expression. For example, there's no distinction in behavior between reassignments for variables of type `Int` or variables of type `Int[]`. Consider the following sequence of statements:

Q#

```
mutable arr1 = new Int[3];
let arr2 = arr1;
arr1 w/= 0 <- 3;
```

The first statement instantiates a new array of integers `[0,0,0]` and assigns it to `arr1`. The next statement assigns that value to a variable with name `arr2`. The last statement creates a new array instance based on `arr1` with the same values except for the value at index 0 which is set to 3. The newly created array is then assigned to the variable `arr1`. The last line makes use of the abbreviated syntax for evaluate-and-reassign statements, and could equivalently have been written as `arr1 = arr1 w/ 0 <- 1;`.
After running the three statements, `arr1` will contain the value `[3,0,0]` while `arr2` remains unchanged and contains the value `[0,0,0]`.

Q# clearly thus distinguishes the mutability of a handle and the behavior of a type. Mutability within Q# is a concept that applies to a *symbol* rather than a type or value; it applies to the handle that allows you to access a value rather than to the value itself. It is *not* represented in the type system, implicitly or explicitly.

Of course, this is merely a description of the formally defined behavior; under the hood, the actual implementation uses a reference counting scheme to avoid copying memory as much as possible. The modification is specifically done in place as long as there's only one currently valid handle that accesses a certain value.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Operations and functions

Article • 02/21/2025

As elaborated in more detail in the description of the qubit data type, quantum computations are executed in the form of side effects of operations that are natively supported on the targeted quantum processor. These are, in fact, the only side effects in Q#. Since all types are immutable, there are no side effects that impact a value that is explicitly represented in Q#. Hence, as long as an implementation of a certain callable doesn't directly or indirectly call any of these natively implemented operations, its execution always produces the same output, given the same input.

Q# allows you to explicitly split out such purely deterministic computations into *functions*. Since the set of natively supported instructions isn't fixed and built into the language itself, but rather fully configurable and expressed as a Q# library, determinism is guaranteed by requiring that functions can only call other functions and can't call any operations. Additionally, native instructions that aren't deterministic, that is, because they impact the quantum state, are represented as operations. With these two restrictions, functions can be evaluated as soon as their input value is known, and, in principle, never need to be evaluated more than once for the same input.

Q# therefore distinguishes between two types of callables: operations and functions. All callables take a single argument (potentially tuple-valued) as input and produce a single value (tuple) as output. Syntactically, the operation type is expressed as `<TIn> => <TOut> is <Char>`, where `<TIn>` is to be replaced by the argument type, `<TOut>` is to be replaced by the return type, and `<Char>` is to be replaced by the operation characteristics. If no characteristics need to be specified, the syntax simplifies to `<TIn> => <TOut>`. Similarly, function types are expressed as `<TIn> -> <TOut>`.

Aside from this determinism guarantee, there's little difference between operations and functions. Both are first-class values that can be passed around freely; they can be used as return values or arguments to other callables, as shown in the following example:

Q#

```
function Pow<'T>(op : 'T => Unit, pow : Int) : 'T => Unit {
    return PowImpl(op, pow, _);
}
```

Both can be instantiated based on a type-parametrized definition, for example, the type parametrized function `Pow` earlier, and they can be partially applied as done in the `return` statement in the example.

Operation characteristics

In addition to the information about input and output type, the operation type contains information about the characteristics of an operation. This information, for example, describes what functors are supported by the operation. Additionally, the internal representation also contains optimization-relevant information that is inferred by the compiler.

The characteristics of an operation are a set of predefined and built-in labels. They're expressed in the form of a special expression that is part of the type signature. The expression consists either of one of the predefined sets of labels, or of a combination of characteristics expressions via a supported binary operator.

There are two predefined sets, `Adj` and `Ctl`.

- `Adj` is the set that contains a single label indicating that an operation is adjointable, meaning it supports the Adjoint functor and the applied quantum transformation can be "undone," that is, it can be inverted.
- `Ctl` is the set that contains a single label indicating that an operation is controllable, meaning it supports the Controlled functor and its execution can be conditioned on the state of other qubits.

The two operators that are supported as part of characteristics expressions are the set union `+` and the set intersection `*`. In EBNF (Extended Backus–Naur form),

```
predefined = "Adj" | "Ctl";
characteristics = predefined
    | "(", characteristics, ")"
    | characteristics ("+"|"*") characteristics;
```

As one would expect, `*` has higher precedence than `+` and both are left-associative. The type of a unitary operation, for example, is expressed as `<TIn> => <TOut> is Adj + Ctl`, where `<TIn>` should be replaced with the type of the operation argument, and `<TOut>` replaced with the type of the returned value.

> **Note**
>
> Indicating the characteristics of an operation in this form has two major advantages; for one, new labels can be introduced without having exponentially many language keywords for all combinations of labels. More importantly, using

expressions to indicate the characteristics of an operation also supports parameterizations over operation characteristics in the future.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Singleton tuple equivalence

Article • 02/29/2024

To avoid any ambiguity between tuples and parentheses that group sub-expressions, a tuple with a single element is considered to be equivalent to the contained item, including its type. For example, the types `Int`, `(Int)`, and `((Int))` are treated as being identical. The same holds true for the values `5`, `(5)` and `(((5)))`, or for `(5, (6))` and `(5, 6)`. This equivalence applies for all purposes, including assignment. Since there is no dynamic dispatch or reflection in Q# and all types in Q# are resolvable at compile-time, singleton tuple equivalence can be readily implemented during compilation.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Subtyping and variance

Article • 08/20/2024

Q# supports only a few conversion mechanisms. Implicit conversions can happen only when applying binary operators, evaluating conditional expressions, or constructing an array literal. In these cases, a common supertype is determined and the necessary conversions are performed automatically. Aside from such implicit conversions, explicit conversions via function calls are possible and often necessary.

Currently, the only subtyping relation that exists applies to operations. Intuitively, it makes sense that one should be allowed to substitute an operation that supports more than the required set of functors. Concretely, for any two concrete types `TIn` and `TOut`, the subtyping relation is

```
(TIn => TOut) :>
(TIn => TOut is Adj), (TIn => TOut is Ctl) :>
(TIn => TOut is Adj + Ctl)
```

where `A :> B` indicates that `B` is a subtype of `A`. Phrased differently, `B` is more restrictive than `A` such that a value of type `B` can be used wherever a value of type `A` is required. If a callable relies on an argument (item) of being of type `A`, then an argument of type `B` can safely be substituted since if provides all the necessary capabilities.

This kind of polymorphism extends to tuples in that a tuple of type `B` is a subtype of a tuple type `A` if it contains the same number of items and the type of each item is a subtype of the corresponding item type in `A`. This is known as *depth subtyping*. There is currently no support for *width subtyping*, that is, there is no subtype relation between any two `struct` types or a `struct` type and any built-in type. The existence of the `unwrap` operator, which allows you to extract a tuple containing all named items, prevents this.

> **Note**
>
> In regards to callables, if a callable processes an argument of type `A`, then it is also capable of processing an argument of type `B`. If a callable is passed as an argument to another callable, then it has to be capable of processing anything that the type signature may require. This means that if the callable needs to be able to process an argument of type `B`, any callable that is capable of processing a more general

argument of type `A` can be passed safely. Conversely, we expect that if we require that the passed callable returns a value of type `A`, then the promise to return a value of type `B` is sufficient, since that value will provide all necessary capabilities.

The operation or function type is *contravariant* in its argument type and *covariant* in its return type. `A :> B` hence implies that for any concrete type `T1`,

```
(B → T1) :> (A → T1), and
(T1 → A) :> (T1 → B)
```

where `→` here can mean either a function or operation, and we omit any annotations for characteristics. Substituting `A` with `(B → T2)` and `(T2 → A)` respectively, and substituting `B` with `(A → T2)` and `(T2 → B)` respectively, leads to the conclusion that, for any concrete type `T2`,

```
((A → T2) → T1) :> ((B → T2) → T1), and
((T2 → B) → T1) :> ((T2 → A) → T1), and
(T1 → (B → T2)) :> (T1 → (A → T2)), and
(T1 → (T2 → A)) :> (T1 → (T2 → B))
```

By induction, it follows that every additional indirection reverses the variance of the argument type, and leaves the variance of the return type unchanged.

> ⓘ **Note**
>
> This also makes it clear what the variance behavior of arrays needs to be; retrieving items via an item access operator corresponds to invoking a function of type `(Int -> TItem)`, where `TItem` is the type of the elements in the array. Since this function is implicitly passed when passing an array, it follows that arrays need to be covariant in their item type. The same considerations also hold for tuples, which are immutable and thus covariant with respect to each item type. If arrays weren't immutable, the existence of a construct that would allow you to set items in an array, and thus take an argument of type `TItem`, would imply that arrays also need to be contravariant. The only option for data types that support getting and setting items is hence to be *invariant*, meaning there is no subtyping relation whatsoever; `B[]` is *not* a subtype of `A[]` even if `B` is a subtype of `A`. Despite the fact that arrays in Q# are **immutable**, they are invariant rather than covariant. This means, for

example, that a value of type `(Qubit => Unit is Adj)[]` cannot be passed to a callable that requires an argument of type `(Qubit => Unit)[]`. Keeping arrays invariant allows for more flexibility related to how arrays are handled and optimized in the runtime, but it may be possible to revise that in the future.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Type parameterizations

Article • 02/21/2025

Q# supports type-parameterized operations and functions. The Q# standard libraries make heavy use of type-parametrized callables to provide a host of useful abstractions, including functions like `Mapped` and `Fold` that are familiar from functional languages.

To motivate the concept of type parameterizations, consider the example of the function `Mapped`, which applies a given function to each value in an array and returns a new array with the computed values. This functionality can be perfectly described without specifying the item types of the input and output arrays. Since the exact types don't change the implementation of the function `Mapped`, it makes sense that it should be possible to define this implementation for arbitrary item types; we want to define a *factory* or *template* that, given the concrete types for the items in the input and output array, returns the corresponding function implementation. This notion is formalized in the form of type parameters.

Concretization

Any operation or function declaration may specify one or more type parameters that can be used as the types, or part of the types, of the callable's input or output, or both. The exceptions are entry points, which must be concrete and cannot be type-parametrized. Type parameter names start with a tick (') and may appear multiple times in the input and output types. All arguments that correspond to the same type parameter in the callable signature must be of the same type.

A type-parametrized callable needs to be concretized, that is, it must be provided with the necessary type arguments before it can be assigned or passed as argument, such that all type parameters can be replaced with concrete types. A type is considered concrete if it is one of the built-in types, a `struct` type, or if it is concrete within the current scope. The following example illustrates what it means for a type to be concrete within the current scope, and is explained in more detail below:

Q#

```
    function Mapped<'T1, 'T2> (
        mapper : 'T1 -> 'T2,
        array : 'T1[]
    ) : 'T2[] {

        mutable mapped = new 'T2[Length(array)];
        for (i in IndexRange(array)) {
```

```
            mapped w/= i <- mapper(array[i]);
        }
        return mapped;
    }

    function AllCControlled<'T3> (
        ops : ('T3 => Unit)[]
    ) : ((Bool,'T3) => Unit)[] {

        return Mapped(CControlled<'T3>, ops);
    }
```

The function `CControlled` is defined in the `Microsoft.Quantum.Canon` namespace. It takes an operation `op` of type `'TIn => Unit` as argument and returns a new operation of type `(Bool, 'TIn) => Unit` that applies the original operation, provided a classical bit (of type `Bool`) is set to *true*; this is often referred to as the classically controlled version of `op`.

The function `Mapped` takes an array of an arbitrary item type `'T1` as argument, applies the given `mapper` function to each item, and returns a new array of type `'T2[]` containing the mapped items. It is defined in the `Microsoft.Quantum.Array` namespace. For the purpose of the example, the type parameters are numbered to avoid making the discussion more confusing by giving the type parameters in both functions the same name. This isn't necessary; type parameters for different callables may have the same name, and the chosen name is only visible and relevant within the definition of that callable.

The function `AllCControlled` takes an array of operations and returns a new array containing the classically controlled versions of these operations. The call of `Mapped` resolves its type parameter `'T1` to `'T3 => Unit`, and its type parameter `'T2` to `(Bool,'T3) => Unit`. The resolving type arguments are inferred by the compiler based on the type of the given argument. We say that they are *implicitly* defined by the argument of the call expression. Type arguments can also be specified explicitly, as is done for `CControlled` in the same line. The explicit concretization `CControlled<'T3>` is necessary when the type arguments cannot be inferred.

The type `'T3` is concrete within the context of `AllCControlled`, since it is known for each *invocation* of `AllCControlled`. That means that as soon as the entry point of the program - which cannot be type-parametrized - is known, so is the concrete type `'T3` for each call to `AllCControlled`, such that a suitable implementation for that particular type resolution can be generated. Once the entry point to a program is known, all usages of type parameters can be eliminated at compile-time. We refer to this process as *monomorphization*.

Some restrictions are needed to ensure that this can indeed be done at compile-time as opposed to only at run time.

Restrictions

Consider the following example:

Q#

```
operation Foo<'TArg> (
    op : 'TArg => Unit,
    arg : 'TArg
) : Unit {

    let cbit = RandomInt(2) == 0;
    Foo(CControlled(op), (cbit, arg));
}
```

Ignoring that an invocation of `Foo` results in an infinite loop, it serves for the purpose of illustration. `Foo` invokes itself with the classically controlled version of the original operation `op` that has been passed in, as well as a tuple containing a random classical bit in addition to the original argument.

For each iteration in the recursion, the type parameter `'TArg` of the next call is resolved to `(Bool, 'TArg)`, where `'TArg` is the type parameter of the current call. Concretely, suppose `Foo` is invoked with the operation `H` and an argument `arg` of type `Qubit`. `Foo` then invokes itself with a type argument `(Bool, Qubit)`, which then invokes `Foo` with a type argument `(Bool, (Bool, Qubit))`, and so on. Clearly, in this case `Foo` can't be monomorphized at compile-time.

Additional restrictions apply to cycles in the call graph that involve only type-parametrized callables. Each callable needs to be invoked with the same set of type arguments after traversing the cycle.

> ⓘ **Note**
>
> It would be possible to be less restrictive and require that for each callable in the cycle, there is a finite number of cycles after which it is invoked with the original set of type arguments, such as the case for the following function:

Q#

```
function Bar<'T1,'T2,'T3>(a1:'T1, a2:'T2, a3:'T3) : Unit{
    Bar<'T2,'T3,'T1>(a2, a3, a1);
```

```
}
```

For simplicity, the more restrictive requirement is enforced. Note that for cycles that involve at least one concrete callable without any type parameter, such a callable ensures that the type-parametrized callables within that cycle are always called with a fixed set of type arguments.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Type inference

Article • 02/21/2025

Q#'s type inference algorithm is based on inference algorithms designed for the Hindley-Milner type system. While top-level callables must be declared with explicit type annotations, most types used within a callable can be inferred. For example, given these callables:

```
Q#
```

```
function Length<'a>(xs : 'a[]) : Int
function Mapped<'a, 'b>(f : 'a -> 'b, xs : 'a[]) : 'b[]
```

and this expression:

```
Q#
```

```
Mapped(Length, [[], ["a"], ["b", "c"]])
```

then the type argument to `Length` is inferred as `Length<String[]>`, and the type arguments to `Mapped` are inferred as `Mapped<String[], Int>`. It isn't required to write these types explicitly.

Ambiguous types

Sometimes there isn't a single principal type that can be inferred for a type variable. In these cases, type inference fails with an error referring to an ambiguous type. For example, change the previous example slightly:

```
Q#
```

```
Mapped(Length, [[]])
```

What is the type of `[[]]`? In some type systems, it's possible to give it the type `∀a. a[][]`, but this isn't supported in Q#. A concrete type is required, but there are an infinite number of types that work: `String[][]`, `(Int, Int)[][]`, `Double[][][]`, and so on. You must explicitly say which type you meant.

There are multiple ways to do this, depending on the situation. For example:

1. Call `Length` with a type argument.

```
Q#

Mapped(Length<String>, [[]])
```

2. Call `Mapped` with its first type argument. (The `_` for its second type argument means that it should still be inferred.)

```
Q#

Mapped<String[], _>(Length, [[]])
```

3. Replace the empty array literal with an explicitly-typed call to a library function.

```
Q#

Mapped<String[], _>(Length, [EmptyArray<String>()])
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Tutorial: Implement a quantum random number generator in Q#

In this tutorial, you learn to write a basic quantum program in Q# that leverages the nature of quantum mechanics to produce a random number.

In this tutorial, you will:

- ✓ Create a Q# program.
- ✓ Review the main components of a Q# program.
- ✓ Define the logic of a problem.
- ✓ Combine classical and quantum operations to solve a problem.
- ✓ Work with qubits and superposition to build a quantum random number generator.

Prerequisites

To develop and run the code sample in Visual Studio Code (VS Code) and Jupyter Notebook, install the following:

- The latest version of VS Code or open VS Code for the Web.
- The latest version of the Microsoft Quantum Development Kit (QDK) extension. For installation details, see Set up the QDK extension.
- The Python and Jupyter extensions for VS Code.
- The latest `qdk` Python library with the `jupyter` extra. To install these, open a terminal and run the following command:

  Bash

  ```
  pip install --upgrade "qdk[jupyter]"
  ```

Define the problem

Classical computers don't produce random numbers, but rather pseudorandom numbers. A pseudorandom number generator generates a deterministic sequence of numbers based on

some initial value, called a seed. To better approximate random values, this seed is often the current time from the CPU's clock.

Quantum computers, on the other hand, can generate truly random numbers. This is because the measurement of a qubit in superposition is a probabilistic process. The result of the measurement is random, and there's no way to predict the outcome. This is the basic principle of quantum random number generators.

A qubit is a unit of quantum information that can be in superposition. When measured, a qubit can only be either in the 0 state or in the 1 state. However, before measurement, the state of the qubit represents the probability of reading either a 0 or a 1 with a measurement.

You start by taking a qubit in a basis state, for example zero. The first step of the random number generator is to use a Hadamard operation to put the qubit into an equal superposition. The measurement of this state results in a zero or a one with 50% probability of each outcome, a truly random bit.

There's no way of knowing what you will get after the measurement of the qubit in superposition, and the result is a different value each time the code is invoked. But how can you use this behavior to generate larger random numbers?

Let's say you repeat the process four times, generating this sequence of binary digits:

$$0, 1, 1, 0$$

If you concatenate, or combine, these bits into a bit string, you can form a larger number. In this example, the bit sequence 0110 is equivalent to six in decimal.

$$0110_{\,binary} \equiv 6_{\,decimal}$$

If you repeat this process many times, you can combine multiple bits to form any large number. Using this method, you can create a number to use as a secure password, since you can be sure that no hacker could determine the results of the sequence of measurements.

Define the random number generator logic

Let's outline what the logic of a random number generator should be:

1. Define `max` as the maximum number you want to generate.
2. Define the number of random bits that you need to generate. This is done by calculating how many bits, `nBits`, you need to express integers up to `max`.
3. Generate a random bit string that's `nBits` in length.

4. If the bit string represents a number greater than `max`, go back to step three.
5. Otherwise, the process is complete. Return the generated number as an integer.

As an example, let's set `max` to 12. That is, 12 is the largest number you want to use as a password.

You need $\lfloor ln(12)/ln(2) + 1 \rfloor$, or 4 bits to represent a number between 0 and 12. We can use the built-in function `BitSizeI`, which takes any integer and returns the number of bits required to represent it.

Let's say you generate the bit string 1101_{binary}, which is equivalent to $13_{decimal}$. Because 13 is greater than 12, you repeat the process.

Next, you generate the bit string 0110_{binary}, which is equivalent to $6_{decimal}$. Because 6 is less than 12, the process is complete.

The quantum random number generator will return number 6 as your password. In practice, set a larger number as the maximum because lower numbers are easy to crack by just trying all possible passwords. In fact, to increase the difficulty of guessing or cracking your password, you could use ASCII code to convert binary to text and to generate a password by using numbers, symbols, and mixed-case letters.

Write a random bit generator

The first step is to write a Q# operation that generates a random bit. This operation will be one of the building blocks of the random number generator.

Q#

```
operation GenerateRandomBit() : Result {
    // Allocate a qubit.
    use q = Qubit();

    // Set the qubit into superposition of 0 and 1 using the Hadamard
    H(q);

    // At this point the qubit `q` has 50% chance of being measured in the
    // |0⟩ state and 50% chance of being measured in the |1⟩ state.
    // Measure the qubit value using the `M` operation, and store the
    // measurement value in the `result` variable.
    let result = M(q);

    // Reset qubit to the |0⟩ state.
    // Qubits must be in the |0⟩ state by the time they are released.
    Reset(q);
```

```
    // Return the result of the measurement.
    return result;
}
```

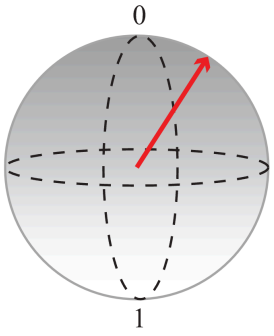
Now take a look at new code.

- You define the `GenerateRandomBit` operation, which takes no input and produces a value of type `Result`. The `Result` type represents the result of a measurement and can have two possible values: `Zero` or `One`.
- You allocate a single qubit with the `use` keyword. When it gets allocated, a qubit is always in the |0⟩ state.
- You use the `H` operation to place the qubit in an equal superposition.
- You use the `M` operation to measure the qubit, return the measured value (`Zero` or `One`).
- You use the `Reset` operation to reset the qubit to the |0⟩ state.

By putting the qubit in superposition with the `H` operation and measuring it with the `M` operation, the result is a different value each time the code is invoked.

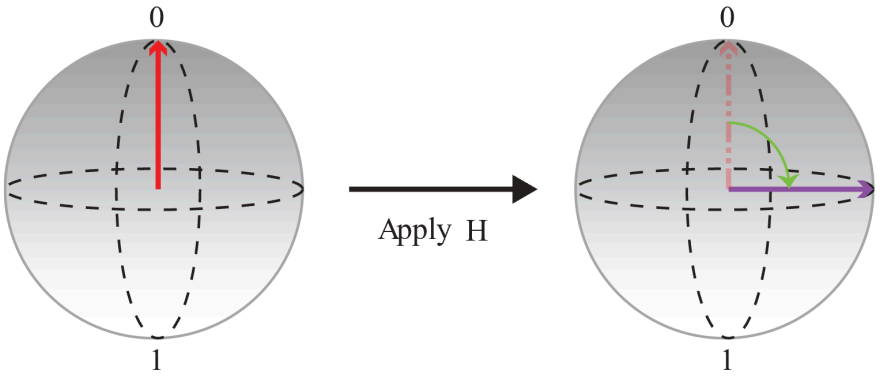
Visualize the Q# code with the Bloch sphere

In the Bloch sphere, the north pole represents the classical value 0 and the south pole represents the classical value 1. Any superposition can be represented by a point on the sphere (represented by an arrow). The closer the end of the arrow to a pole the higher the probability the qubit collapses into the classical value assigned to that pole when measured. For example, the qubit state represented by the arrow in the following figure has a higher probability of giving the value 0 if you measure it.

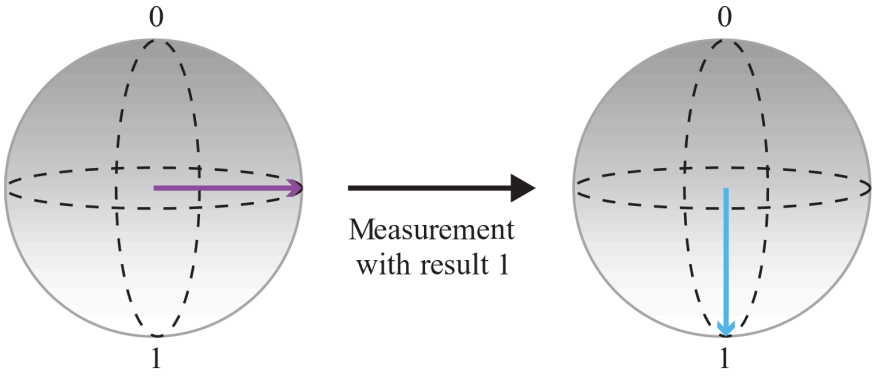


You can use this representation to visualize what the code is doing:

1. First, start with a qubit initialized in the |0⟩ state and apply an `H` operation to create an equal superposition in which the probabilities for 0 and 1 are the same.



2. Then measure the qubit and save the output:



Since the outcome of the measurement is random and the probabilities of measuring 0 and 1 are the same, you have obtained a completely random bit. You can call this operation several times to create integers. For example, if you call the operation three times to obtain three random bits, you can build random 3-bit numbers (that is, a random number between 0 and 7).

Write a complete random number generator

1. First, you need to import the required namespaces from the Q# standard library to the program. The Q# compiler loads many common functions and operations automatically, however for the complete random number generator, you need some additional functions and operations from two Q# namespaces: `Std.Math` and `Std.Convert`.

   Q#

   ```
   import Std.Convert.*;
   import Std.Math.*;
   ```

2. Next, you define the `GenerateRandomNumberInRange` operation. This operation repeatedly calls the `GenerateRandomBit` operation to build a string of bits.

   Q#

   ```
   /// Generates a random number between 0 and `max`.
   operation GenerateRandomNumberInRange(max : Int) : Int {
   ```

```
    // Determine the number of bits needed to represent `max` and store it
    // in the `nBits` variable. Then generate `nBits` random bits which will
    // represent the generated random number.
    mutable bits = [];
    let nBits = BitSizeI(max);
    for idxBit in 1..nBits {
        bits += [GenerateRandomBit()];
    }
    let sample = ResultArrayAsInt(bits);

    // Return random number if it is within the requested range.
    // Generate it again if it is outside the range.
    return sample > max ? GenerateRandomNumberInRange(max) | sample;
}
```

Let's take a moment to review the new code.

- You need to calculate the number of bits needed to express integers up to `max`. The `BitSizeI` function from the `Std.Math` namespace converts an integer to the number of bits needed to represent it.
- The `SampleRandomNumberInRange` operation uses a `for` loop to generate random numbers until it generates one that's equal to or less than `max`. The `for` loop works exactly the same as a `for` loop in other programming languages.
- The variable `bits` is a mutable variable. A mutable variable is one that can change during the computation. You use the `set` directive to change a mutable variable's value.
- The `ResultArrayAsInt` function, from the default `Std.Convert` namespace, converts the bit string to a positive integer.

3. Finally, you add an entry point to the program. By default, the Q# compiler looks for a `Main` operation and starts processing there. It calls the `GenerateRandomNumberInRange` operation to generate a random number between 0 and 100.

Q#

```
operation Main() : Int {
    let max = 100;
    Message($"Sampling a random number between 0 and {max}: ");

    // Generate random number in the 0..max range.
    return GenerateRandomNumberInRange(max);
}
```

The `let` directive declares variables that don't change during the computation. Here you define the maximum value as 100.

For more information about the `Main` operation, see Entry Points.

4. The complete code for the random number generator is as follows:

Q#

```
import Std.Convert.*;
import Std.Math.*;

operation Main() : Int {
    let max = 100;
    Message($"Sampling a random number between 0 and {max}: ");

    // Generate random number in the 0..max range.
    return GenerateRandomNumberInRange(max);
}

/// Generates a random number between 0 and `max`.
operation GenerateRandomNumberInRange(max : Int) : Int {
    // Determine the number of bits needed to represent `max` and store it
    // in the `nBits` variable. Then generate `nBits` random bits which will
    // represent the generated random number.
    mutable bits = [];
    let nBits = BitSizeI(max);
    for idxBit in 1..nBits {
        bits += [GenerateRandomBit()];
    }
    let sample = ResultArrayAsInt(bits);

    // Return random number if it is within the requested range.
    // Generate it again if it is outside the range.
    return sample > max ? GenerateRandomNumberInRange(max) | sample;
}

operation GenerateRandomBit() : Result {
    // Allocate a qubit.
    use q = Qubit();

    // Set the qubit into superposition of 0 and 1 using a Hadamard operation
    H(q);

    // At this point the qubit `q` has 50% chance of being measured in the
    // |0⟩ state and 50% chance of being measured in the |1⟩ state.
    // Measure the qubit value using the `M` operation, and store the
    // measurement value in the `result` variable.
    let result = M(q);

    // Reset qubit to the |0⟩ state.
    // Qubits must be in the |0⟩ state by the time they are released.
    Reset(q);
```

```
    // Return the result of the measurement.
    return result;
}
```

Run the random number generator program

Choose your preferred development environment, either the QDK extension in VS Code or the QDK Python library in Jupyter Notebook.

Q# file in VS Code

1. In VS Code, open the **File** menu and choose **New Text File** to create a new file.
2. Save the file as `RandomNumberGenerator.qs`. This file will contain the Q# code for your program.
3. Copy the following code in the `RandomNumberGenerator.qs` file.

Q#

```
import Std.Convert.*;
import Std.Math.*;

operation Main() : Int {
    let max = 100;
    Message($"Sampling a random number between 0 and {max}: ");

    // Generate random number in the 0..max range.
    return GenerateRandomNumberInRange(max);
}

/// # Summary
/// Generates a random number between 0 and `max`.
operation GenerateRandomNumberInRange(max : Int) : Int {
    // Determine the number of bits needed to represent `max` and store it
    // in the `nBits` variable. Then generate `nBits` random bits which will
    // represent the generated random number.
    mutable bits = [];
    let nBits = BitSizeI(max);
    for idxBit in 1..nBits {
        bits += [GenerateRandomBit()];
    }
    let sample = ResultArrayAsInt(bits);

    // Return random number if it is within the requested range.
    // Generate it again if it is outside the range.
    return sample > max ? GenerateRandomNumberInRange(max) | sample;
```

```
}

/// # Summary
/// Generates a random bit.
operation GenerateRandomBit() : Result {
    // Allocate a qubit.
    use q = Qubit();

    // Set the qubit into superposition of 0 and 1 using the Hadamard
    // operation `H`.
    H(q);

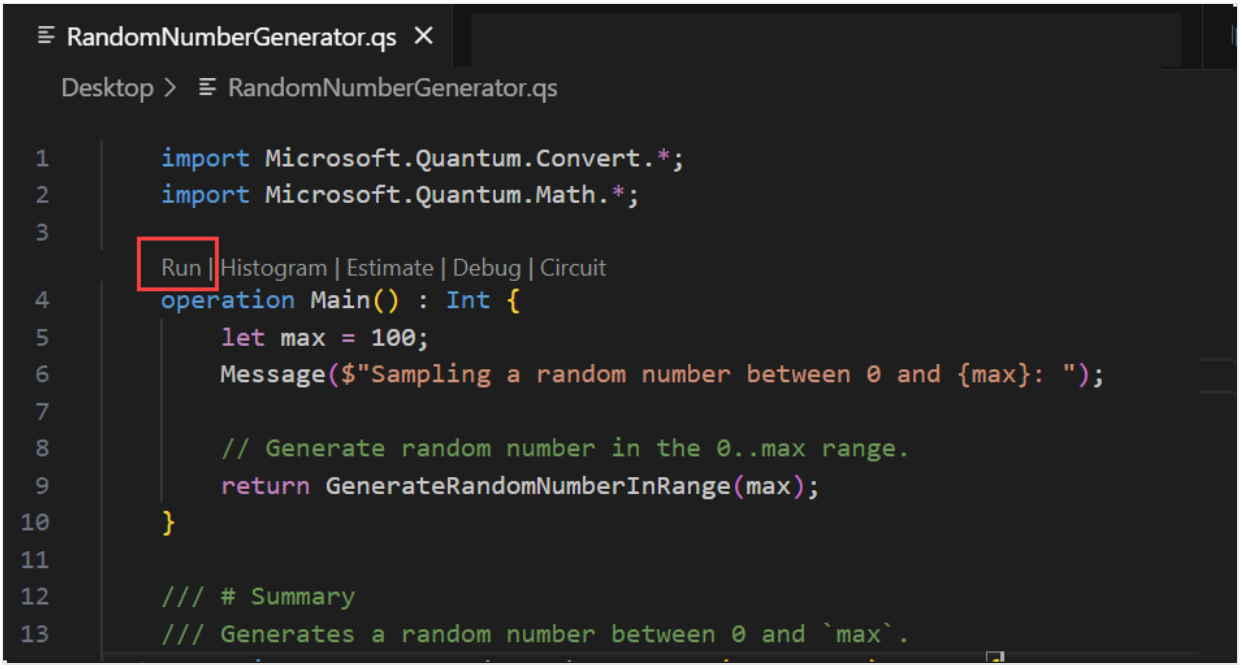
    // At this point the qubit `q` has 50% chance of being measured in the
    // |0⟩ state and 50% chance of being measured in the |1⟩ state.
    // Measure the qubit value using the `M` operation, and store the
    // measurement value in the `result` variable.
    let result = M(q);

    // Reset qubit to the |0⟩ state.
    // Qubits must be in the |0⟩ state by the time they are released.
    Reset(q);

    // Return the result of the measurement.
    return result;

    // Note that Qubit `q` is automatically released at the end of the
block.
}
```

4. To run your program, choose the **Run** from the list of commands preceding `Main()`, or press **Ctrl** + **F5**. The program runs the `Main()` operation on the default simulator.



5. Your output will appear in the debug console.

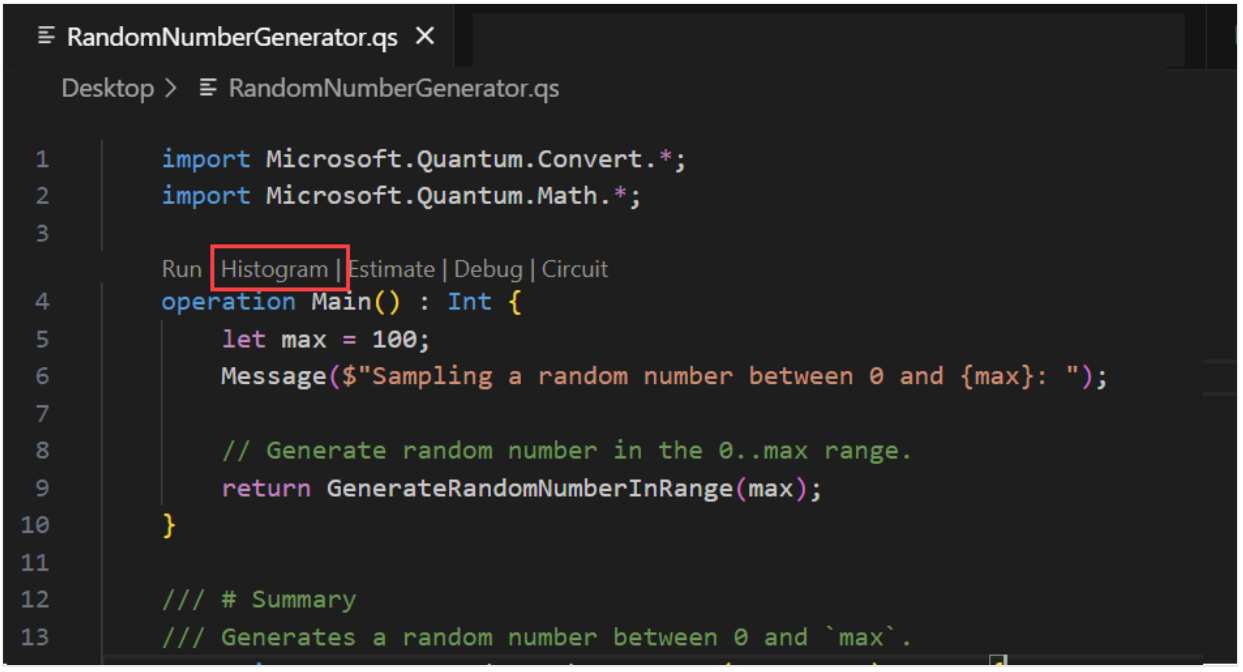
6. Run the program again to see a different result.

> ⓘ Note
>
> If the QIR target profile isn't set to **Unrestricted**, then you get an error when you run the program. For this program, the compiler automatically sets the target profile to **Unrestricted** unless you set the profile yourself.

Plot the frequency histogram

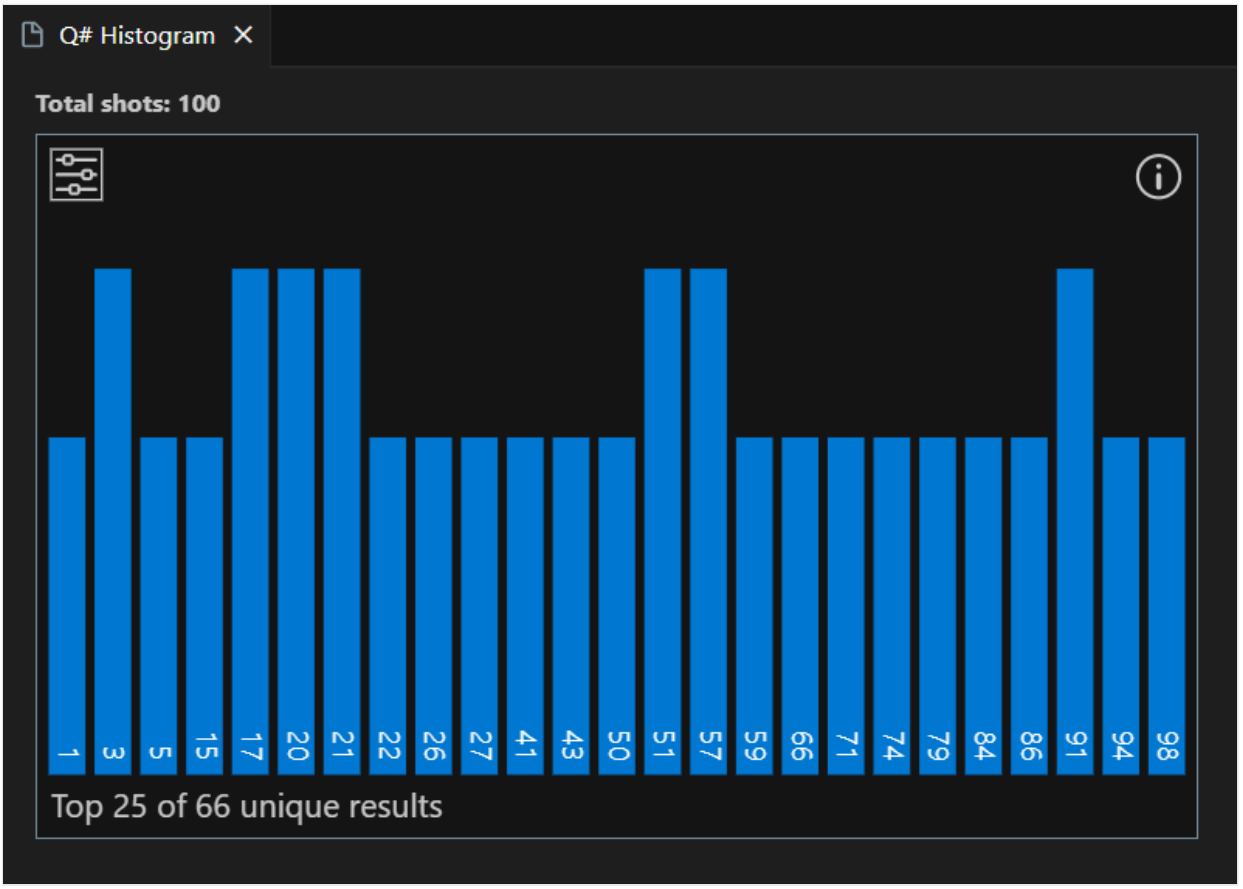
Let's visualize the distribution of results obtained from running the quantum program multiple times. The frequency histogram helps visualize the probability distribution of these outcomes.

1. Select **View -> Command Palette** and type **histogram**, which brings up the **QDK: Run file and show histogram** command. You can also select **Histogram** from the list of commands preceding `Main()`. Select this option to open the Q# histogram window.



2. Enter a number of **shots** to execute the program, for example, 100 shots, and press **Enter**. The histogram will display in the Q# histogram window.

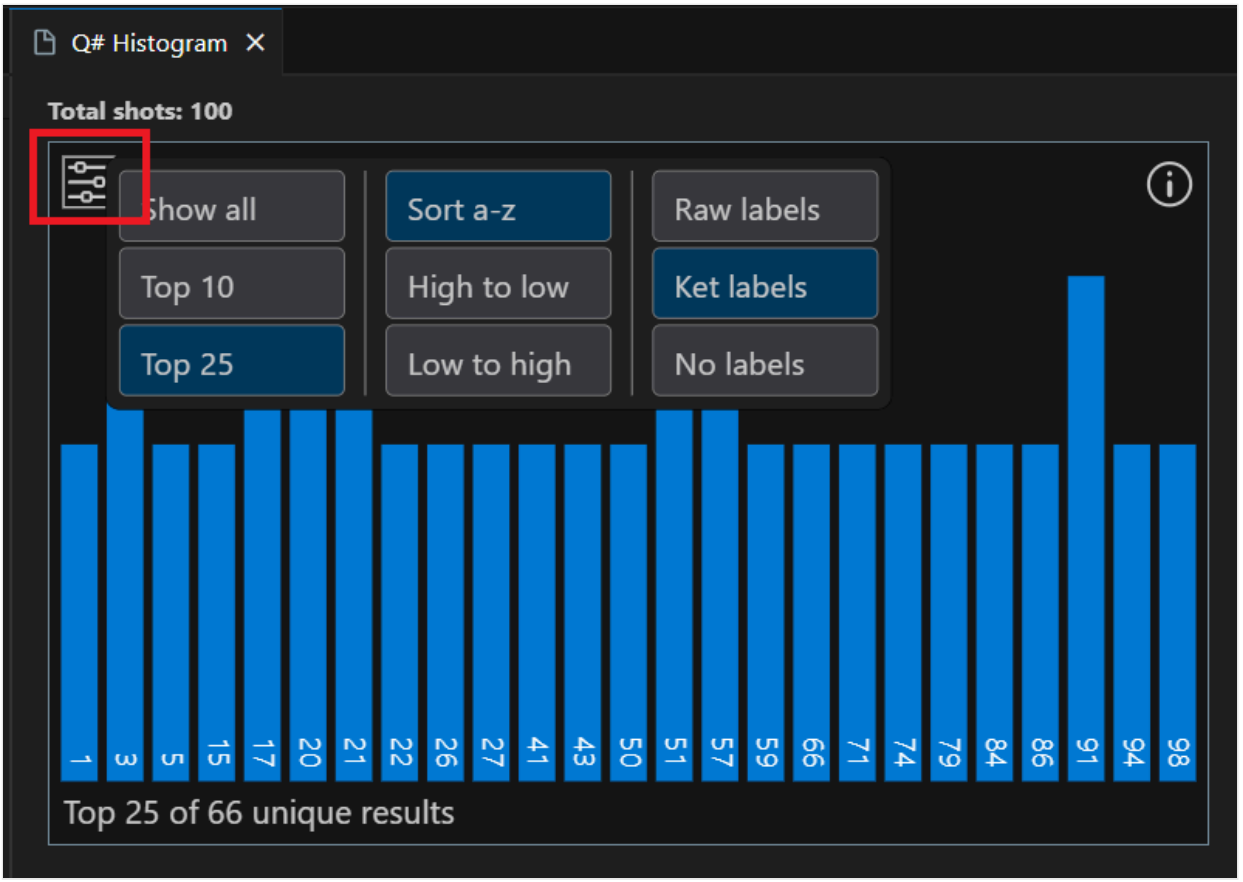
3. Each bar in the histogram corresponds to a possible outcome, and its height represents the number of times that outcome is observed. The number of different results may differ each time you run the histogram.



> **Tip**
>
> You can zoom the histogram using the mouse scroll wheel or a trackpad gesture. When zoomed in, you can pan the chart by pressing 'Alt' while scrolling.

4. Select a bar to display the **percentage** of that outcome.

5. Select the top-left **settings icon** to display options. You can display top 10 results, top 25 results, or all results. You can also sort the results from high to low, or low to high.



> [!NOTE]
> This code snippet does not currently run on any available Azure Quantum hardware targets, as the callable `ResultArrayAsInt` requires a QPU with a full computation profile.

Related content

Explore other Q# tutorials:

- Quantum entanglement shows how to write a Q# program that manipulates and measures qubits and demonstrates the effects of superposition and entanglement.
- Grover's search algorithm shows how to write a Q# program that uses Grover's search algorithm.
- Quantum Fourier Transforms explores how to write a Q# program that directly addresses specific qubits.
- The Quantum Katas are self-paced tutorials and programming exercises aimed at teaching the elements of quantum computing and Q# programming at the same time.

Last updated on 05/15/2026

Tutorial: Explore quantum entanglement with Q#

In this tutorial, you write a Q# program that prepares two qubits in a specific quantum state, operates on the qubits to entangle them with each other, and performs measurements to demonstrate the effects of superposition and entanglement. You build your Q# program piece-by-piece to introduce qubit states, quantum operations, and measurements.

Before you begin, review the following quantum computing concepts:

- Classical bits hold a single binary value such as a 0 or 1, but qubits can be in a superposition of the two states, 0 and 1. Each possible qubit state is described by a set of probability amplitudes.
- When you measure a qubit's state, you always get either 0 or 1. The probability of each result is determined by the probability amplitudes that define the superposition state when you make a measurement.
- Multiple qubits can be entangled such that you can't describe them independently from each other. When you measure one qubit in an entangled pair, you also get information about the other qubit without measuring it.

In this tutorial, you learn how to:

- ✓ Create Q# operations to initialize a qubit states.
- ✓ Put a qubit into a superposition state.
- ✓ Entangle a pair of qubits.
- ✓ Measure a qubit and view the results.

Prerequisites

To develop and run the code sample in your local development environment, install the following tools:

- The latest version of Visual Studio Code (VS Code), or open VS Code for the Web.
- The latest version of the Microsoft Quantum Development Kit (QDK) extension. For installation details, see Set up the QDK.

Create a new Q# file

1. In VS Code, open the **File** menu and choose **New Text File** to create a new file.
2. Save the file as `CreateBellStates.qs`. This file is where you write the Q# code for your program.

Initialize a qubit to a known state

The first step is to define a Q# operation that initializes a qubit to a desired classical state, either 0 or 1. This operation measures a qubit in a general quantum state, which returns a Q# `Result` type value of either `Zero` or `One`. If the measurement result is different from the desired state, then the operation flips the state so that the operation returns the desired state 100% of the time.

Open `CreateBellStates.qs` and copy the following code:

Q#

```
operation SetQubitState(desired : Result, target : Qubit) : Unit {
    if desired != M(target) {
        X(target);
    }
}
```

The code example introduces two standard Q# operations, `M` and `X`, that transform the state of a qubit.

Here's a detailed description of how the `SetQubitState` operation works:

1. Takes two parameters: a Result type parameter named `desired` that represents the desired state for the qubit to be in (`Zero` or `One`), and a Qubit type parameter.
2. Performs a measurement operation, `M`, which measures the state of the qubit (`Zero` or `One`) and compares the result to the value that you pass for `desired`.
3. If the measurement result doesn't match the value for `desired`, then an `X` operation is applied to the qubit. This operation flips the state of the qubit so that the measurement probabilities for `Zero` and `One` are reversed.

Write a test operation to test the Bell state

To call the `SetQubitState` operation in your Q# program, create another operation named `Main`. This operation allocates two qubits, calls `SetQubitState` to set the first qubit to a known state, and then measures the qubits to see the results.

Add the following operation to your `CreateBellStates.qs` file after the `SetQubitState` operation:

Q#

```
operation Main() : (Int, Int, Int, Int) {
    mutable numOnesQ1 = 0;
    mutable numOnesQ2 = 0;
    let count = 1000;
    let initial = One;

    // allocate the qubits
    use (q1, q2) = (Qubit(), Qubit());
    for test in 1..count {
        SetQubitState(initial, q1);
        SetQubitState(Zero, q2);

        // measure each qubit
        let resultQ1 = M(q1);
        let resultQ2 = M(q2);

        // Count the number of 'Ones' returned:
        if resultQ1 == One {
            numOnesQ1 += 1;
        }
        if resultQ2 == One {
            numOnesQ2 += 1;
        }
    }

    // reset the qubits
    SetQubitState(Zero, q1);
    SetQubitState(Zero, q2);


    // Display the times that |0> is returned, and times that |1> is returned
    Message($"Q1 - Zeros: {count - numOnesQ1}");
    Message($"Q1 - Ones: {numOnesQ1}");
    Message($"Q2 - Zeros: {count - numOnesQ2}");
    Message($"Q2 - Ones: {numOnesQ2}");
    return (count - numOnesQ1, numOnesQ1, count - numOnesQ2, numOnesQ2 );
}
```

In the code, the `count` and `initial` variables are set to `1000` and `One` respectively. This initializes the first qubit to `One` and measures each qubit 1000 times.

The `Main` operation does the following:

1. Sets variables for the number of shots (`count`) and the initial qubit state (`One`).
2. Calls the `use` statement to initialize two qubits.
3. Loops over the experiment `count` times.

4. In the loop, calls `SetQubitState` to set the specified `initial` value on the first qubit, and then calls `SetQubitState` again to set the second qubit to the `Zero` state.
5. In the loop, applies the `M` operation to measure each qubit, and then stores the number of measurements for each qubit that return `One`.
6. After the loop completes, calls `SetQubitState` again to reset the qubits to a known state (`Zero`). You must reset qubits that you allocate with the `use` statement.
7. Calls the `Message` function to print your results in the console.

Run the code

Before you write code for superposition and entanglement, test your current program to see the initialization and measurement of the qubits.

To run the code as a standalone program, the Q# compiler needs to know where to start the program. Because you didn't specify a namespace, the compiler recognizes the default entry point as the `Main` operation. For more information, see Projects and implicit namespaces.

Your `CreateBellStates.qs` file now looks like this:

Q#

```
operation SetQubitState(desired : Result, target : Qubit) : Unit {
    if desired != M(target) {
        X(target);
    }
}

operation Main() : (Int, Int, Int, Int) {
    mutable numOnesQ1 = 0;
    mutable numOnesQ2 = 0;
    let count = 1000;
    let initial = One;

    // allocate the qubits
    use (q1, q2) = (Qubit(), Qubit());
    for test in 1..count {
        SetQubitState(initial, q1);
        SetQubitState(Zero, q2);

        // measure each qubit
        let resultQ1 = M(q1);
        let resultQ2 = M(q2);

        // Count the number of 'Ones' returned:
        if resultQ1 == One {
            numOnesQ1 += 1;
        }
```

```
        if resultQ2 == One {
            numOnesQ2 += 1;
        }
    }

    // reset the qubits
    SetQubitState(Zero, q1);
    SetQubitState(Zero, q2);


    // Display the times that |0> is returned, and times that |1> is returned
    Message($"Q1 - Zeros: {count - numOnesQ1}");
    Message($"Q1 - Ones: {numOnesQ1}");
    Message($"Q2 - Zeros: {count - numOnesQ2}");
    Message($"Q2 - Ones: {numOnesQ2}");
    return (count - numOnesQ1, numOnesQ1, count - numOnesQ2, numOnesQ2 );
}
```

To run the program, choose the **Run** command from the code lens that precedes the `Main` operation, or enter **Ctrl** + **F5**. The program runs the `Main` operation on the default simulator.

Your output appears in the debug console.

Output

```
Q1 - Zeros: 0
Q1 - Ones: 1000
Q2 - Zeros: 1000
Q2 - Ones: 0
```

Your program doesn't modify the qubit states yet, so measurement of the first qubit always returns `One`, and the second qubit always returns `Zero`.

If you change the value of `initial` to `Zero` and run the program again, then the first qubit also always returns `Zero`.

Output

```
Q1 - Zeros: 1000
Q1 - Ones: 0
Q2 - Zeros: 1000
Q2 - Ones: 0
```

Put a qubit into a superposition state

Currently, the qubits in your program are in a classical state, either 1 or 0, just like bits on a regular computer. To entangle the qubits, you must first put one of the qubits into an equal

superposition state. Measurement of a qubit in an equal superposition state has a 50% to return `Zero` and a 50% chance to return `One`.

To put a qubit into a superposition state, use the Q# `H`, or Hadamard, operation. The `H` operation converts a qubit that's in a pure `Zero` or `One` state into a sort of halfway state between `Zero` and `One`.

Modify your code in the `Main` operation. Reset the initial value to `One` and insert a line for the `H` operation:

Q#

```
for test in 1..count {
    use (q1, q2) = (Qubit(), Qubit());
    for test in 1..count {
        SetQubitState(initial, q1);
        SetQubitState(Zero, q2);

        H(q1);  // Add the H operation after initialization and before measurement

        // measure each qubit
        let resultQ1 = M(q1);
        let resultQ2 = M(q2);
        ...
```

Run your program again. Because the first qubit is in an equal superposition when you measure it, you get close to a 50/50 result for `Zero` and `One`. For example, your output looks similar to this:

Output

```
Q1 - Zeros: 523
Q1 - Ones: 477
Q2 - Zeros: 1000
Q2 - Ones: 0
```

Every time you run the program, the results for the first qubit vary slightly, but are close to 50% `One` and 50% `Zero`, while the results for the second qubit are still always `Zero`.

Initialize the first qubit to `Zero` instead of `One` and run the program again. You get similar results because the `H` operation turns both a pure `Zero` state and a pure `One` state into an equal superposition state.

Entangle two qubits

Entangled qubits are correlated such that they can't be described independently from each other. When you measure the state of one entangled qubit, you also know the state of the other qubit without measuring it. This tutorial uses an example with two entangled qubits, but you can entangle three or more qubits too.

To create an entangled state, use the Q# `CNOT`, or Controlled-NOT, operation. When you apply `CNOT` to two qubits, one qubit is the control qubit and the other is the target qubit. If the state of the control qubit is `One`, then the `CNOT` operation flips the state of the target qubit. Otherwise, `CNOT` does nothing to the qubits.

Add the `CNOT` operation to your program immediately after the `H` operation. Your full program looks like this:

Q#

```
operation SetQubitState(desired : Result, target : Qubit) : Unit {
    if desired != M(target) {
        X(target);
    }
}

operation Main() : (Int, Int, Int, Int) {
    mutable numOnesQ1 = 0;
    mutable numOnesQ2 = 0;
    let count = 1000;
    let initial = Zero;

    // allocate the qubits
    use (q1, q2) = (Qubit(), Qubit());
    for test in 1..count {
        SetQubitState(initial, q1);
        SetQubitState(Zero, q2);

        H(q1);
        CNOT(q1, q2);      // Add the CNOT operation after the H operation

        // measure each qubit
        let resultQ1 = M(q1);
        let resultQ2 = M(q2);

        // Count the number of 'Ones' returned:
        if resultQ1 == One {
            numOnesQ1 += 1;
        }
        if resultQ2 == One {
            numOnesQ2 += 1;
        }
    }

    // reset the qubits
```

```
    SetQubitState(Zero, q1);
    SetQubitState(Zero, q2);


    // Display the times that |0> is returned, and times that |1> is returned
    Message($"Q1 - Zeros: {count - numOnesQ1}");
    Message($"Q1 - Ones: {numOnesQ1}");
    Message($"Q2 - Zeros: {count - numOnesQ2}");
    Message($"Q2 - Ones: {numOnesQ2}");
    return (count - numOnesQ1, numOnesQ1, count - numOnesQ2, numOnesQ2 );
}
```

Run the program and view the output. Your results very slightly each time you run the program.

Output

```
Q1 - Zeros: 502
Q1 - Ones: 498
Q2 - Zeros: 502
Q2 - Ones: 498
```

The statistics for the first qubit still show about a 50% chance to measure both `One` and `Zero`, but the measurement results for the second qubit aren't always `Zero` now. Each qubit has the same number of `Zero` results and `One` results. The measurement result for the second qubit is always the same as the result for the first qubit because the two qubits are entangled. If the first qubit is measured to be `Zero`, then the entangled qubit must also be `Zero`. If the first qubit is measured to be `One`, then the entangled qubit must also be `One`.

Plot the frequency histogram

To plot a frequency histogram that shows the distribution of results when you run your program multiple times, complete the following steps:

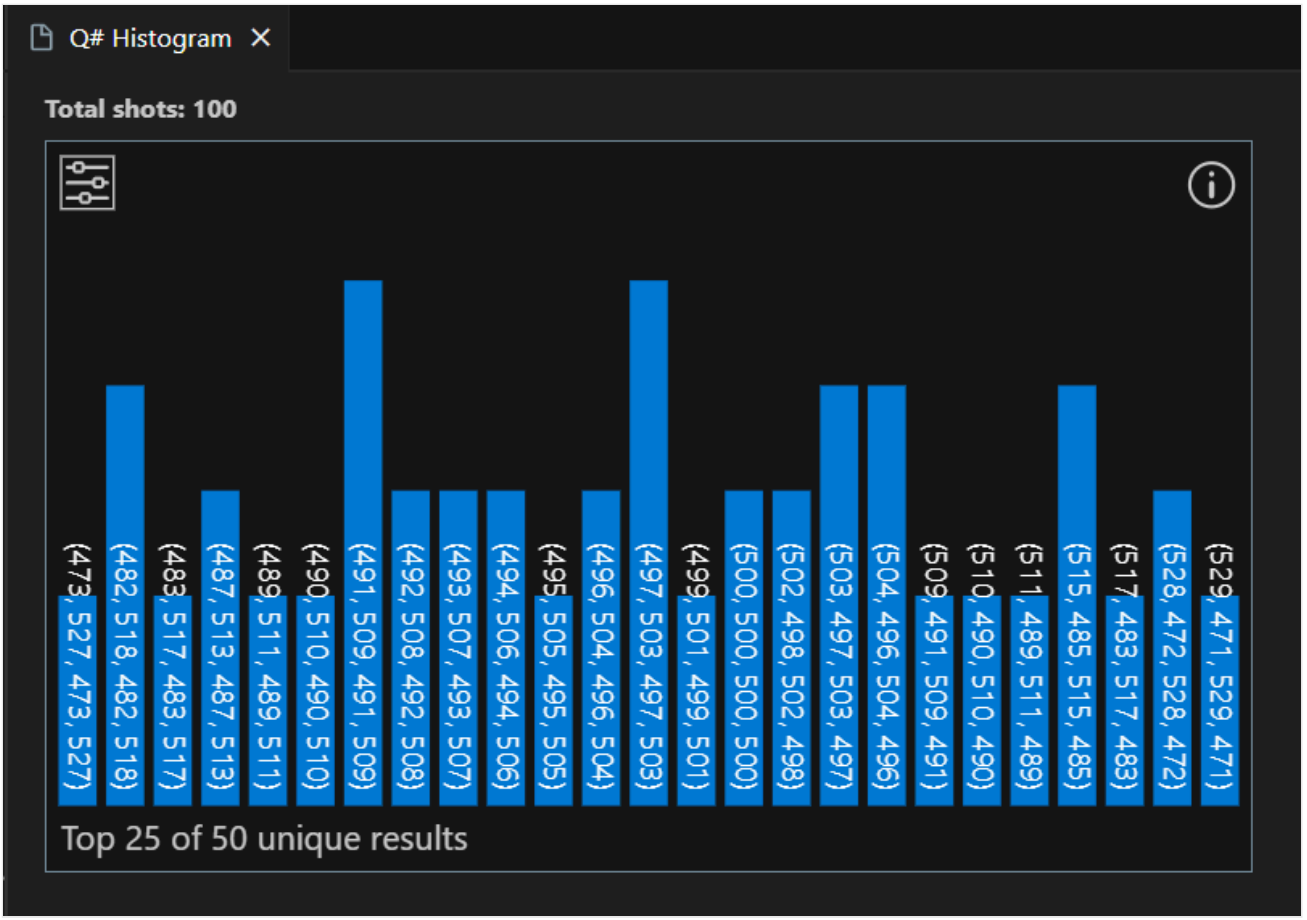
1. Open your `CreateBellStates.qs` file in VS Code.

2. Open the **View** menu and choose **Command Palette**.

3. Enter **histogram** to bring up the **QDK: Run file and show histogram** option. Or, choose the **Histogram** command from the code lens option that precedes the `Main` operation. Then, enter a number of shots (for example, 100). The Q# histogram opens in a new tab.

   Each bar in the histogram corresponds to a possible outcome when the entanglement circuit runs 1000 times. The height of a bar represents the number of times that outcome occurs. For example, the following histogram shows a distribution with 50 unique results.

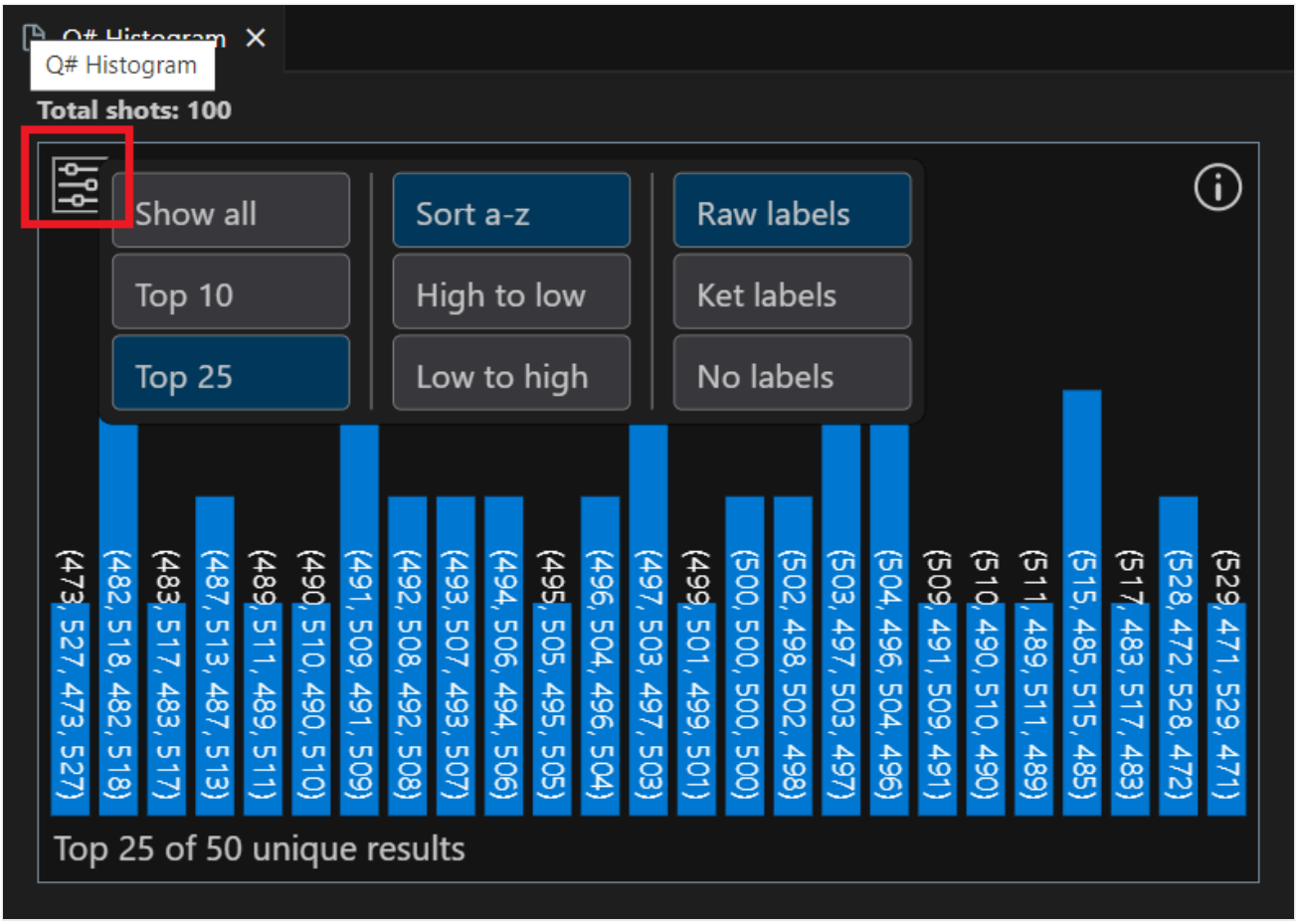
Note that for each outcome, the measurement results for the first and the second qubit are always the same.



> **Tip**
>
> To zoom in on the histogram, use the mouse scroll wheel or a trackpad gesture. To pan the chart when you're zoomed in, hold **Alt** while you scroll.

4. Choose a bar to display the percentage of total shots that produced that outcome.

5. Choose the **settings icon** in the top-left to display visualization options.



6. Run the code again, but this time with 1000 shots. As the number of shots increases, the distribution of results approaches a normal distribution.

Related content

Explore other Q# tutorials:

- Grover's search algorithm shows how to write a Q# program that uses Grover's search algorithm.
- Quantum Fourier Transform explores how to write a Q# program that directly addresses specific qubits.
- The Quantum Katas are self-paced tutorials and programming exercises that teach the elements of quantum computing and Q# programming at the same time.

Last updated on 05/15/2026

Tutorial: Implement Grover's search algorithm in Q#

In this tutorial, you implement Grover's algorithm in Q# to solve search-based problems. For an in-depth explanation of the theory behind Grover's algorithm, see the Theory of Grover's algorithm.

In this tutorial, you:

- ✓ Define Grover's algorithm for a search problem
- ✓ Implement Grover's algorithm in Q#

Prerequisites

To develop and run the code sample in Visual Studio Code (VS Code), you need to install the following tools:

- The latest version of VS Code, or open VS Code for the Web.
- The latest version of the Microsoft Quantum Development Kit (QDK) extension. For installation details, see Set up the QDK.

Define the problem

Grover's algorithm is one of the most famous algorithms in quantum computing. The type of problem it solves is often referred to as "searching a database", but it's more accurate to think of it in terms of the search problem.

Any search problem can be mathematically formulated with an abstract function $f(x)$ that accepts search items x. If the item x is a solution to the search problem, then $f(x) = 1$. If the item x isn't a solution, then $f(x) = 0$. The search problem consists of finding any item x_0 such that $f(x_0) = 1$.

Thus, you can formulate any search problem as: given a classical function $f(x) : \{0, 1\}^n \to \{0, 1\}$, where n is the bit-size of the search space, find an input x_0 for which $f(x_0) = 1$.

To implement Grover's algorithm to solve a search problem, you need to:

1. Transform the problem to the form of a **Grover's task**. For example, suppose you want to find the factors of an integer M using Grover's algorithm. You can transform the integer factorization problem to a Grover's task by creating a function

   $$f_M(x) = 1[r],$$

   where $1[r] = 1$ if $r = 0$ and $1[r] = 0$ if $r \neq 0$ and r is the remainder of M/x. This way, the integers x_i that make $f_M(x_i) = 1$ are the factors of M and you have transformed the problem to a Grover's task.
2. Implement the function of the Grover's task as a quantum oracle. To implement Grover's algorithm, you need to implement the function $f(x)$ of your Grover's task as a quantum oracle.
3. Use Grover's algorithm with your oracle to solve the task. Once you have a quantum oracle, you can plug it into your Grover's algorithm implementation to solve the problem and interpret the output.

Grover's algorithm

Suppose there are $N = 2^n$ eligible items for the search problem and they are indexed by assigning each item an integer from 0 to $N - 1$. The steps of the algorithm are:

1. Start with a register of n qubits initialized in the state $|0\rangle$.
2. Prepare the register into a uniform superposition by applying H to each qubit in the register:

   $$|\psi\rangle = \frac{1}{N^{1/2}} \sum_{x=0}^{N-1} |x\rangle$$

3. Apply the following operations to the register N_{optimal} times:
   a. The phase oracle O_f that applies a conditional phase shift of -1 for the solution items.
   b. Apply H to each qubit in the register.
   c. Apply $-O_0$, a conditional phase shift of -1 to every computational basis state except $|0\rangle$.
   d. Apply H to each qubit in the register.
4. Measure the register to obtain the index of an item that's a solution with very high probability.
5. Check the item to see if it's a valid solution. If not, start again.

Write the code for Grover's algorithm in Q#

This section discusses how to implement the algorithm in Q#. There are a few things to consider when implementing Grover's algorithm. You need to define what is your marked state, how to reflect about it, and how many iterations to run the algorithm for. You also need to define the oracle that implements the function of the Grover's task.

Define the marked state

First, you define what input you are trying to find in the search. To do so, write an operation that applies the steps **b**, **c** and **d** from the Grover's algorithm.

Together, these steps are also known as the **Grover'S diffusion operator** $-H^{\otimes n}O_0H^{\otimes n}$.

Q#

```
operation ReflectAboutMarked(inputQubits : Qubit[]) : Unit {
    Message("Reflecting about marked state...");
    use outputQubit = Qubit();
    within {
        // We initialize the outputQubit to (|0⟩ - |1⟩) / √2, so that
        // toggling it results in a (-1) phase.
        X(outputQubit);
        H(outputQubit);
        // Flip the outputQubit for marked states.
        // Here, we get the state with alternating 0s and 1s by using the X
        // operation on every other qubit.
        for q in inputQubits[...2...] {
            X(q);
        }
    } apply {
        Controlled X(inputQubits, outputQubit);
    }
}
```

The `ReflectAboutMarked` operation reflects about the basis state marked by alternating zeros and ones. It does so by applying the Grover's diffusion operator to the input qubits. The operation uses an auxiliary qubit, `outputQubit`, which is initialized in the state $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ by applying the X and H gates. The operation then applies the X gate to every other qubit in the register, which flips the state of the qubit. Finally, it applies the controlled X gate to the auxiliary qubit and the input qubits. This operation flips the auxiliary qubit if and only if all the input qubits are in the state $|1\rangle$, which is the marked state.

Define the number of optimal iterations

Grover's search has an optimal number of iterations that yields the highest probability of measuring a valid output. If the problem has $N = 2^n$ possible eligible items, and M of them are

solutions to the problem, the optimal number of iterations is:

$$N_{\text{optimal}} \approx \frac{\pi}{4}\sqrt{\frac{N}{M}}$$

Continuing to iterate past the optimal number of iterations starts reducing that probability until you reach nearly-zero success probability on iteration $2N_{\text{optimal}}$. After that, the probability grows again until $3N_{\text{optimal}}$, and so on.

In practical applications, you don't usually know how many solutions your problem has before you solve it. An efficient strategy to handle this issue is to "guess" the number of solutions M by progressively increasing the guess in powers of two (i.e. $1, 2, 4, 8, 16, \ldots, 2^n$). One of these guesses will be close enough that the algorithm will still find the solution with an average number of iterations around $\sqrt{\frac{N}{M}}$.

The following Q# function calculates the optimal number of iterations for a given number of qubits in a register.

Q#

```
function CalculateOptimalIterations(nQubits : Int) : Int {
    if nQubits > 63 {
        fail "This sample supports at most 63 qubits.";
    }
    let nItems = 1 <<< nQubits; // 2^nQubits
    let angle = ArcSin(1. / Sqrt(IntAsDouble(nItems)));
    let iterations = Round(0.25 * PI() / angle - 0.5);
    return iterations;
}
```

The `CalculateOptimalIterations` function uses the formula shown earlier to calculate the number of iterations, and then rounds it to the nearest integer.

Define the Grover's operation

The Q# operation for Grover's search algorithm has three inputs:

- The number of qubits, `nQubits : Int`, in the qubit register. This register encodes the tentative solution to the search problem, and is measured after the operation.
- The number of optimal iterations, `iterations : Int`.
- An operation, `phaseOracle : Qubit[] => Unit) : Result[]`, that represents the phase oracle for the Grover's task. This operation applies a unitary transformation over a generic qubit register.

Q#

```
operation GroverSearch( nQubits : Int, iterations : Int, phaseOracle : Qubit[] =>
Unit) : Result[] {

    use qubits = Qubit[nQubits];
    PrepareUniform(qubits);

    for _ in 1..iterations {
        phaseOracle(qubits);
        ReflectAboutUniform(qubits);
    }

    // Measure and return the answer.
    return MResetEachZ(qubits);
}
```

The `GroverSearch` operation initializes a register of n qubits in the state $|0\rangle$, prepares the register into a uniform superposition, and then applies Grover's algorithm for the specified number of iterations. The search itself consists of repeatedly reflecting about the marked state and the start state, which you can write out in Q# as a for loop. Finally, it measures the register and returns the result.

The code makes use of three helper operations: `PrepareUniform`, `ReflectAboutUniform`, and `ReflectAboutAllOnes`.

Given a register in the all-zeros state, the `PrepareUniform` operation prepares a uniform superposition over all basis states.

Q#

```
operation PrepareUniform(inputQubits : Qubit[]) : Unit is Adj + Ctl {
    for q in inputQubits {
        H(q);
    }
}
```

The `ReflectAboutAllOnes` operation reflects about the all-ones state.

Q#

```
operation ReflectAboutAllOnes(inputQubits : Qubit[]) : Unit {
    Controlled Z(Most(inputQubits), Tail(inputQubits));
}
```

The operation `ReflectAboutUniform` reflects about the uniform superposition state. First, it transforms the uniform superposition to all-zero. Then, it transforms the all-zero state to all-ones. Finally, it reflects about the all-ones state. The operation is called `ReflectAboutUniform` because it can be geometrically interpreted as a reflection in the vector space about the uniform superposition state.

Q#

```
operation ReflectAboutUniform(inputQubits : Qubit[]) : Unit {
    within {
        Adjoint PrepareUniform(inputQubits);
        // Transform the all-zero state to all-ones
        for q in inputQubits {
            X(q);
        }
    } apply {
        ReflectAboutAllOnes(inputQubits);
    }
}
```

Run the final code

Now you have all the ingredients to implement a particular instance of Grover's search algorithm and solve the factoring problem. To finish, the `Main` operation sets up the problem by specifying the number of qubits and the number of iterations

Q#

```
operation Main() : Result[] {
    let nQubits = 5;
    let iterations = CalculateOptimalIterations(nQubits);
    Message($"Number of iterations: {iterations}");

    // Use Grover's algorithm to find a particular marked state.
    let results = GroverSearch(nQubits, iterations, ReflectAboutMarked);
    return results;
}
```

Run the program

1. In VS Code, open the **File** menu and choose **New Text File** to create a new file.

2. Save the file as `GroversAlgorithm.qs`. This file contains the Q# code for your program.

3. Copy the following code into the `GroversAlgorithm.qs` file.

Q#

```
import Std.Convert.*;
import Std.Math.*;
import Std.Arrays.*;
import Std.Measurement.*;
import Std.Diagnostics.*;


operation Main() : Result[] {
    let nQubits = 5;
    let iterations = CalculateOptimalIterations(nQubits);
    Message($"Number of iterations: {iterations}");

    // Use Grover's algorithm to find a particular marked state.
    let results = GroverSearch(nQubits, iterations, ReflectAboutMarked);
    return results;
}

operation GroverSearch(
    nQubits : Int,
    iterations : Int,
    phaseOracle : Qubit[] => Unit) : Result[] {

    use qubits = Qubit[nQubits];

    PrepareUniform(qubits);

    for _ in 1..iterations {
        phaseOracle(qubits);
        ReflectAboutUniform(qubits);
    }

    // Measure and return the answer.
    return MResetEachZ(qubits);
}

function CalculateOptimalIterations(nQubits : Int) : Int {
    if nQubits > 63 {
        fail "This sample supports at most 63 qubits.";
    }
    let nItems = 1 <<< nQubits; // 2^nQubits
    let angle = ArcSin(1. / Sqrt(IntAsDouble(nItems)));
    let iterations = Round(0.25 * PI() / angle - 0.5);
    return iterations;
}

operation ReflectAboutMarked(inputQubits : Qubit[]) : Unit {
    Message("Reflecting about marked state...");
    use outputQubit = Qubit();
    within {
        // We initialize the outputQubit to (|0⟩ - |1⟩) / √2, so that
        // toggling it results in a (-1) phase.
```

```
        X(outputQubit);
        H(outputQubit);
        // Flip the outputQubit for marked states.
        // Here, we get the state with alternating 0s and 1s by using the X
        // operation on every other qubit.
        for q in inputQubits[...2...] {
            X(q);
        }
    } apply {
        Controlled X(inputQubits, outputQubit);
    }
}

operation PrepareUniform(inputQubits : Qubit[]) : Unit is Adj + Ctl {
    for q in inputQubits {
        H(q);
    }
}

operation ReflectAboutAllOnes(inputQubits : Qubit[]) : Unit {
    Controlled Z(Most(inputQubits), Tail(inputQubits));
}

operation ReflectAboutUniform(inputQubits : Qubit[]) : Unit {
    within {
        // Transform the uniform superposition to all-zero.
        Adjoint PrepareUniform(inputQubits);
        // Transform the all-zero state to all-ones
        for q in inputQubits {
            X(q);
        }
    } apply {
        // Now that we've transformed the uniform superposition to the
        // all-ones state, reflect about the all-ones state, then let the
        // within/apply block transform us back.
        ReflectAboutAllOnes(inputQubits);
    }
}
```

4. To run your program, choose **Run** from the menu for the `Main` operation, or press **Ctrl** + **F5**. By default, the compiler runs the `Main` operation or function on the default simulator.

5. Your output displays in the debug console in the terminal.

> **Note**
>
> If the QIR target profile isn't set to **Unrestricted**, then you get an error when you run the program. For this program, the compiler automatically sets the target profile to **Unrestricted** unless you set the profile yourself.

Related content

Explore other Q# tutorials:

- Quantum entanglement shows how to write a Q# program that manipulates and measures qubits and demonstrates the effects of superposition and entanglement.
- Quantum random number generator shows how to write a Q# program that generates random numbers out of qubits in superposition.
- Quantum Fourier Transform explores how to write a Q# program that directly addresses specific qubits.
- The Quantum Katas are self-paced tutorials and programming exercises aimed at teaching the elements of quantum computing and Q# programming at the same time.

Last updated on 05/15/2026

Tutorial: Implement the Quantum Fourier Transform in Q#

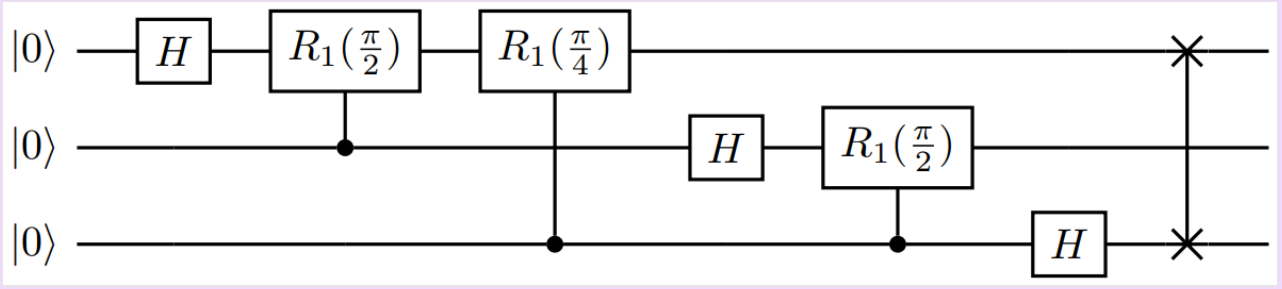
Summarize this article for me

This tutorial shows how to write and simulate a basic quantum program that operates on individual qubits.

Although Q# was primarily created as a high-level programming language for large-scale quantum programs, it can also be used to explore the lower level of quantum programming, that is, directly addressing specific qubits. Specifically, this tutorial takes a closer look at the Quantum Fourier Transform (QFT), a subroutine that is integral to many larger quantum algorithms.

In this tutorial, you learn how to:

- ✓ Define quantum operations in Q#.
- ✓ Write the Quantum Fourier Transform circuit.
- ✓ Simulate a quantum operation from qubit allocation to measurement output.
- ✓ Observe how the quantum system's simulated wavefunction evolves throughout the operation.

> **Note**
>
> This lower level view of quantum information processing is often described in terms of quantum circuits, which represent the sequential application of gates, or *operations*, to specific qubits of a system. Thus, the single- and multi-qubit operations you sequentially apply can be readily represented in circuit diagrams. For example, the full three-qubit quantum Fourier transform used in this tutorial has the following representation as a circuit:
>
> 
>

> **Tip**
>
> If you want to accelerate your quantum computing journey, check out Code with Microsoft Quantum, a unique feature of the Microsoft Quantum website. Here, you

can run built-in Q# samples or your own Q# programs, generate new Q# code from your prompts, open and run your code in VS Code for the Web with one click, and ask Copilot questions about quantum computing.

Prerequisites

- The latest version of Visual Studio Code (VS Code) or open VS Code for the Web.
- The latest version of the Microsoft Quantum Development Kit (QDK) extension. For installation details, see Set up the QDK extension.
- To use Jupyter Notebooks, install the Python and Jupyter extensions in VS Code.
- The latest `qdk` Python package with the `jupyter` extra. To install these, open a terminal and run the following command:

  Bash

  ```
  pip install --upgrade "qdk[jupyter]"
  ```

Create a new Q# file

1. In VS Code, open the **File** menu and choose **New Text File**.
2. Save the file as **QFTcircuit.qs**. This file contains the Q# code for your program.
3. Open **QFTcircuit.qs**.

Write a QFT circuit in Q#

The first part of this tutorial consists of defining the Q# operation `Main`, which performs the quantum Fourier transform on three qubits. The `DumpMachine` function is used to observe how the simulated wavefunction of the three-qubit system evolves across the operation. In the second part of the tutorial, you add measurement functionality and compare the pre- and post-measurement states of the qubits.

You build the operation step by step. Copy and paste the code in the following sections into the **QFTcircuit.qs** file.

You can view the full Q# code for this section as reference.

Import required Q# libraries

Inside your Q# file, import the relevant `Std.*` namespaces.

Q#

```
import Std.Diagnostics.*;
import Std.Math.*;
import Std.Arrays.*;

// operations go here
```

Define operations with arguments and returns

Next, define the `Main` operation:

Q#

```
operation Main() : Unit {
    // do stuff
}
```

The `Main` operation never takes arguments, and for now returns a `Unit` object, which is analogous to returning `void` in C# or an empty tuple, `Tuple[()]`, in Python. Later, you modify the operation to return an array of measurement results.

Allocate qubits

Within the Q# operation, allocate a register of three qubits with the `use` keyword. With `use`, the qubits are automatically allocated in the state.

Q#

```
use qs = Qubit[3]; // allocate three qubits

Message("Initial state |000>:");
DumpMachine();
```

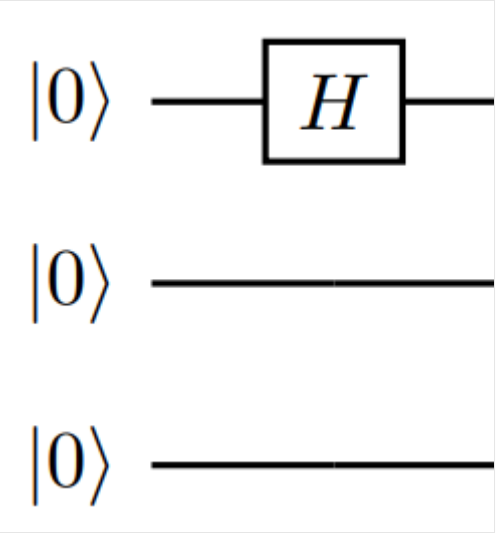
As in real quantum computations, Q# doesn't allow you to directly access qubit states. However, the `DumpMachine` operation prints the target machine's current state, so it can provide valuable insight for debugging and learning when used together with the full state simulator.

Apply single-qubit and controlled operations

Next, apply the operations that comprise the `Main` operation itself. Q# already contains many of these, and other basic quantum operations, in the `Std.Intrinsic` namespace.

> **Note**
>
> `Std.Intrinsic` wasn't imported in the earlier code snippet with the other namespaces, as it is loaded automatically by the compiler for all Q# programs.

The first operation applied is the `H` (Hadamard) operation to the first qubit:



To apply an operation to a specific qubit from a register (for example, a single `Qubit` from an array `Qubit[]`), use standard index notation. So, applying the `H` operation to the first qubit of the register `qs` takes the form:

Q#

```
H(qs[0]);
```

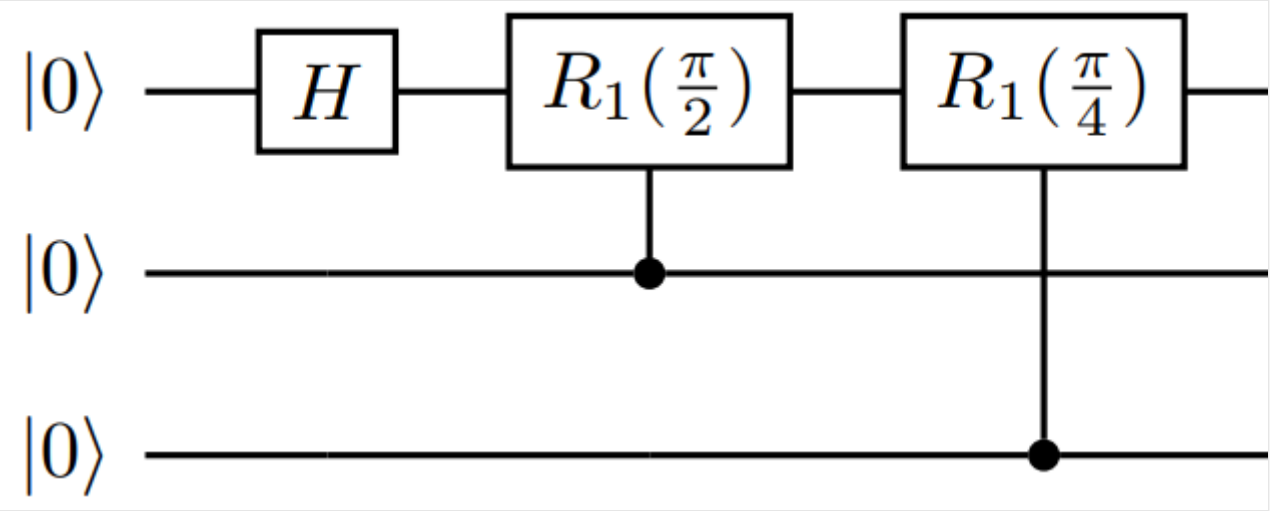
Besides applying the `H` operation to individual qubits, the QFT circuit consists primarily of controlled `R1` rotations. A `R1(θ, <qubit>)` operation in general leaves the component of the qubit unchanged while applying a rotation of $e^{i\theta}$ to the component.

Q# makes it easy to condition the run of an operation upon one, or multiple, control qubits. In general, the call is prefaced with `Controlled`, and the operation arguments change as follows:

`Op(<normal args>)` `Controlled Op([<control qubits>], (<normal args>))`

Note that the control qubit argument must be an array, even if it is for a single qubit.

The controlled operations in the QFT are the `R1` operations that act on the first qubit (and controlled by the second and third qubits):



In your Q# file, call these operations with these statements:

Q#

```
Controlled R1([qs[1]], (PI()/2.0, qs[0]));
Controlled R1([qs[2]], (PI()/4.0, qs[0]));
```

The `PI()` function is used to define the rotations in terms of pi radians.

Apply SWAP operation

After applying the relevant `H` operations and controlled rotations to the second and third qubits, the circuit looks like this:

Q#

```
//second qubit:
H(qs[1]);
Controlled R1([qs[2]], (PI()/2.0, qs[1]));

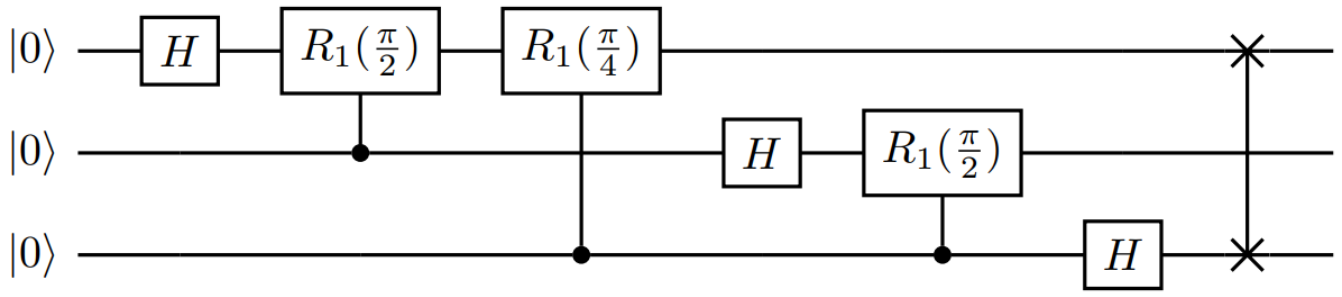
//third qubit:
H(qs[2]);
```

Finally, you apply a `SWAP` operation to the first and third qubits to complete the circuit. This operation is necessary because the quantum Fourier transform outputs the qubits in reverse order, so the swaps allow for seamless integration of the subroutine into larger algorithms.

Q#

```
SWAP(qs[2], qs[0]);
```

The Q# operation now includes the qubit-level operations of the quantum Fourier transform:



Deallocate qubits

The last step is to call `DumpMachine()` again to see the post-operation state, and to deallocate the qubits. The qubits were in state when you allocated them and need to be reset to their initial state using the `ResetAll` operation.

Requiring that all qubits be explicitly reset to is a basic feature of Q#, as it allows other operations to know their state precisely when they begin using those same qubits (a scarce resource). Additionally, resetting them assures that they aren't entangled with any other qubits in the system. If the reset isn't performed at the end of a `use` allocation block, a runtime error might be thrown.

Add the following lines to your Q# file:

Q#

```
Message("After:");
DumpMachine();

ResetAll(qs); // deallocate qubits
```

The complete QFT operation

The Q# program is complete. Your **QFTcircuit.qs** file should now look like this:

Q#

```
import Std.Diagnostics.*;
import Std.Math.*;
import Std.Arrays.*;

operation Main() : Unit {
```

```
    use qs = Qubit[3]; // allocate three qubits

    Message("Initial state |000>:");
    DumpMachine();

    //QFT:
    //first qubit:
    H(qs[0]);
    Controlled R1([qs[1]], (PI()/2.0, qs[0]));
    Controlled R1([qs[2]], (PI()/4.0, qs[0]));

    //second qubit:
    H(qs[1]);
    Controlled R1([qs[2]], (PI()/2.0, qs[1]));

    //third qubit:
    H(qs[2]);

    SWAP(qs[2], qs[0]);

    Message("After:");
    DumpMachine();

    ResetAll(qs); // deallocate qubits

}
```

Run the QFT circuit

For now, the `Main` operation doesn't return any value - the operation returns `Unit` value. Later, you modify the operation to return an array of measurement results (`Result[]`).

Q# standalone program

1. To run your program, choose **Run Q# File** from the menu that precedes `Main`, or press **Ctrl** + **F5**. The program runs the `Main` operation on the default simulator.
2. The `Message` and `DumpMachine` outputs display in the debug console.

If you're curious about how other input states are affected, experiment with applying other qubit operations before the transform.

Add measurements to the QFT circuit

The display from the `DumpMachine` function showed the results of the operation, but unfortunately, a cornerstone of quantum mechanics states that a real quantum system can't

have such a `DumpMachine` function. Instead, the information is extracted through measurements, which in general not only fail to provide information on the full quantum state, but can also drastically alter the system itself.

There are many sorts of quantum measurements, but the example here focuses on the most basic: projective measurements on single qubits. Upon measurement in a given basis (for example, the computational basis), the qubit state is projected onto whichever basis state was measured, hence destroying any superposition between the two.

Modify the QFT operation

To implement measurements within a Q# program, use the `M` operation, which returns a `Result` type.

First, modify the `Main` operation to return an array of measurement results, `Result[]`, instead of `Unit`.

Q#

```
operation Main() : Result[] {
```

Define and initialize `Result[]` array

Before allocating qubits, declare and bind a three-element array (one `Result` for each qubit):

Q#

```
mutable resultArray = [Zero, size = 3];
```

The `mutable` keyword prefacing `resultArray` allows the variable to be modified later in the code, for example, when adding measurement results.

Perform measurements in a `for` loop and add results to array

After the QFT transform operations, insert the following code:

Q#

```
for i in IndexRange(qs) {
    resultArray w/= i <- M(qs[i]);
}
```

The `IndexRange` function called on an array (for example, the array of qubits, `qs`) returns a range over the indices of the array. Here, it's used in the `for` loop to sequentially measure each qubit using the `M(qs[i])` statement. Each measured `Result` type (either `Zero` or `One`) is then added to the corresponding index position in `resultArray` with an update-and-reassign statement.

> **Note**
>
> The syntax of this statement is unique to Q#, but corresponds to the similar variable reassignment `resultArray[i] <- M(qs[i])` seen in other languages such as F# and R.

The keyword `set` is always used to reassign variables bound using `mutable`.

Return `resultArray`

With all three qubits measured and the results added to `resultArray`, you're safe to reset and deallocate the qubits as before. To return the measurements, insert:

Q#

```
return resultArray;
```

Run the QFT circuit with the measurements

Now change the placement of the `DumpMachine` functions to output the state before and after the measurements. Your final Q# code should look like this:

Q#

```
import Std.Diagnostics.*;
import Std.Math.*;
import Std.Arrays.*;

operation Main() : Result[] {

    mutable resultArray = [Zero, size = 3];

    use qs = Qubit[3];

    //QFT:
    //first qubit:
    H(qs[0]);
    Controlled R1([qs[1]], (PI()/2.0, qs[0]));
    Controlled R1([qs[2]], (PI()/4.0, qs[0]));
```

```
    //second qubit:
    H(qs[1]);
    Controlled R1([qs[2]], (PI()/2.0, qs[1]));

    //third qubit:
    H(qs[2]);

    SWAP(qs[2], qs[0]);

    Message("Before measurement: ");
    DumpMachine();

    for i in IndexRange(qs) {
        resultArray w/= i <- M(qs[i]);
    }

    Message("After measurement: ");
    DumpMachine();

    ResetAll(qs);
    Message("Post-QFT measurement results [qubit0, qubit1, qubit2]: ");
    return resultArray;

}
```

> **Tip**
>
> Remember to save your file every time you introduce a change to the code before running it again.

Q# standalone program

1. To run your program, choose **Run Q#** from the menu that precedes `Main`, or press **Ctrl + F5**. The program runs the `Main` operation on the default simulator.
2. The `Message` and `DumpMachine` outputs display in the debug console.

Your output should look similar to this:

Output

```
Before measurement:

 Basis | Amplitude      | Probability | Phase
 ------------------------------------------------
 |000⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000
 |001⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000
 |010⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000
```

```
 |011⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000
 |100⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000
 |101⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000
 |110⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000
 |111⟩ |  0.3536+0.0000𝑖 |    12.5000% |   0.0000

After measurement:

 Basis | Amplitude      | Probability | Phase
 -----------------------------------------------
 |010⟩ |  1.0000+0.0000𝑖 |   100.0000% |   0.0000

Post-QFT measurement results [qubit0, qubit1, qubit2]:

[Zero, One, Zero]
```

This output illustrates a few different things:

1. When comparing the returned result to the pre-measurement `DumpMachine`, it clearly does *not* illustrate the post-QFT superposition over basis states. A measurement only returns a single basis state, with a probability determined by the amplitude of that state in the system's wavefunction.
2. From the post-measurement `DumpMachine`, measurement *changes* the state itself, projecting it from the initial superposition over basis states to the single basis state that corresponds to the measured value.

If you repeat this operation many times, the result statistics begin to illustrate the equally weighted superposition of the post-QFT state that gives rise to a random result on each shot. *However*, besides being inefficient and still imperfect, this would nevertheless only reproduce the relative amplitudes of the basis states, not the relative phases between them. The latter isn't an issue in this example, but relative phases display if given a more complex input to the QFT than .

Use the Q# operations to simplify the QFT circuit

As mentioned in the introduction, much of Q#'s power rests in the fact that it allows you to abstract-away the worries of dealing with individual qubits. Indeed, if you want to develop full-scale, applicable quantum programs, worrying about whether an `H` operation goes before or after a particular rotation would only slow you down. Azure Quantum provides the `ApplyQFT` operation, which you can use and apply for any number of qubits.

1. Replace everything from the first `H` operation to the `SWAP` operation, inclusive, with:

Q#

```
ApplyQFT(qs);
```

2. Your code should now look like this

Q#

```
import Std.Diagnostics.*;
import Std.Math.*;
import Std.Arrays.*;

operation Main() : Result[] {

    mutable resultArray = [Zero, size = 3];

    use qs = Qubit[3];

    //QFT:
    //first qubit:

    ApplyQFT(qs);

    Message("Before measurement: ");
    DumpMachine();

    for i in IndexRange(qs) {
        resultArray w/= i <- M(qs[i]);
    }

    Message("After measurement: ");
    DumpMachine();

    ResetAll(qs);
    Message("Post-QFT measurement results [qubit0, qubit1, qubit2]: ");
    return resultArray;

}
```

3. Run the Q# program again and notice that the output is the same as before.

4. To see the real benefit of using Q# operations, change the number of qubits to something other than `3`:

Q#

```
mutable resultArray = [Zero, size = 4];

use qs = Qubit[4];
```

```
//...
```

You can thus apply the proper QFT for any given number of qubits, without having to worry about adding new `H` operations and rotations on each qubit.

Related content

Explore other Q# tutorials:

- Quantum random number generator shows how to write a Q# program that generates random numbers out of qubits in superposition.
- Grover's search algorithm shows how to write a Q# program that uses Grover's search algorithm.
- Quantum entanglement shows how to write a Q# program that manipulates and measures qubits and demonstrates the effects of superposition and entanglement.
- The Quantum Katas are self-paced tutorials and programming exercises aimed at teaching the elements of quantum computing and Q# programming at the same time.

Last updated on 02/25/2026

Vectors and matrices in quantum computing

Article • 01/16/2025

Some familiarity with linear algebra is essential to understand quantum computing. This article introduces the basic concepts of linear algebra and how to work with vectors and matrices in quantum computing.

Vectors

A column vector, or vector for short, v of dimension (or size) n is a collection of n complex numbers $(v_1, v_2, \ldots, v_n)$ arranged as a column:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

The norm of a vector v is defined as $\sqrt{\sum_i |v_i|^2}$. A vector is called a *unit vector* if its norm is 1.

The *adjoint* of a column vector v is a row vector denoted as $v^\dagger$ and is defined as the conjugate transpose of v. For a column vector v of dimension n, the adjoint is a row vector of dimension $1 \times n$:

$$\begin{bmatrix} v_1 \vdots v_n \end{bmatrix}^\dagger = \begin{bmatrix} v_1^* & \cdots & v_n^* \end{bmatrix}$$

where v_i^* denotes the complex conjugate of v_i.

Using linear algebra, the state of a qubit $\psi = a\,|0\rangle + b\,|1\rangle$ is described as a **quantum state vector** $[a\ b]$, where $|a|^2 + |b|^2 = 1$. For more information, see The qubit.

Scalar product

Two vectors can be multiplied together through the *scalar product*, also known as *dot product* or *inner product*. As the name implies, the result of the scalar product of two vectors is a scalar. The scalar product gives the projection of one vector onto another and is used to express one vector as a sum of other simpler vectors. The scalar product between two column vectors u and v is denoted as $\langle u, v \rangle = u^\dagger v$ and is defined as

$$\langle u, v \rangle = u^\dagger v = \begin{bmatrix} u_1^* & \cdots & u_n^* \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = u_1^* v_1 + \cdots + u_n^* v_n.$$

With the scalar product, the norm of a vector v can be written as $\sqrt{\langle v, v \rangle}$.

You can multiply a vector with a number a to form a new vector whose entries are multiplied by a. You can also add two vectors u and v to form a new vector whose entries are the sum of the entries of u and v. These operations are the following:

$$au + bv = \begin{bmatrix} au_1 + bv_1 \\ au_2 + bv_2 \\ \vdots \\ au_n + bv_n \end{bmatrix}$$

Matrices

A *matrix* of size $m \times n$ is a collection of $m \cdot n$ complex numbers arranged in m rows and n columns as shown below:

$$M = \begin{bmatrix} M_{11} M_{12} \cdots M_{1n} \\ M_{21} M_{22} \cdots M_{2n} \\ \ddots \\ M_{m1} M_{m2} \cdots M_{mn} \end{bmatrix}$$

> **Note**
>
> Note that a vector of dimension n is simply a matrix of size $n \times 1$.

Quantum operations are represented by *squared matrices*, that is, the number of rows and columns are equal. For example, single-qubit operations are represented by 2×2 matrices, like the Pauli X operation

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

> **Tip**
>
> In Q#, the Pauli X operation is represented by the `X` operation.

As with vectors, you can multiply a matrix with a number c to obtain a new matrix where every entry is multiplied with c, and two matrices of the same size can be added to produce a new matrix whose entries are the sum of the respective entries of the two matrices.

Matrix multiplication

You can also multiply a matrix M of dimension $m \times n$ and a matrix N of dimension $n \times p$ to get a new matrix P of dimension $m \times p$ as follows:

$$\begin{bmatrix} M_{11}M_{12}\cdots M_{1n} \\ M_{21}M_{22}\cdots M_{2n} \\ \ddots \\ M_{m1}M_{m2}\cdots M_{mn} \end{bmatrix} \begin{bmatrix} N_{11}N_{12}\cdots N_{1p} \\ N_{21}N_{22}\cdots N_{2p} \\ \ddots \\ N_{n1}N_{n2}\cdots N_{np} \end{bmatrix} = \begin{bmatrix} P_{11}P_{12}\cdots P_{1p} \\ P_{21}P_{22}\cdots P_{2p} \\ \ddots \\ P_{m1}P_{m2}\cdots P_{mp} \end{bmatrix}$$

where the entries of P are $P_{ik} = \sum_j M_{ij}N_{jk}$. For example, the entry P_{11} is the scalar product of the first row of M with the first column of N. Note that since a vector is simply a special case of a matrix, this definition extends to matrix-vector multiplication.

Special types of matrices

One special square matrix is the *identity matrix*, denoted $\mathbb{I}$, which has all its diagonal elements equal to 1 and the remaining elements equal to 0:

$$\mathbb{I} = \begin{bmatrix} 10\cdots 0 \\ 01\cdots 0 \\ \ddots \\ 00\cdots 1 \end{bmatrix}.$$

For a square matrix A, a matrix B is its *inverse* if $AB = BA = \mathbb{I}$. If a matrix A has an inverse, the inverse matrix is unique and is written as A^{-1}.

For any matrix M, the adjoint or conjugate transpose of M is a matrix N such that $N_{ij} = M^*_{ji}$. The adjoint of M is denoted $M^\dagger$.

A matrix U is *unitary* if $UU^\dagger = U^\dagger U = \mathbb{I}$ or equivalently, $U^{-1} = U^\dagger$. One important property of unitary matrices is that they preserve the norm of a vector. This happens because

$$\langle v, v\rangle = v^\dagger v = v^\dagger U^{-1}Uv = v^\dagger U^\dagger Uv = \langle Uv, Uv\rangle.$$

> **Note**
>
> Quantum operations are represented by unitary matrices, which are squared matrices whose adjoint is equal to their inverse.

A matrix M is called *Hermitian* if $M = M^\dagger$.

In quantum computing, there are essentially only two matrices that you encounter: Hermitian and unitary.

Tensor product

Another important operation is the *tensor product*, also called the *matrix direct product* or *Kronecker product*.

Consider the two vectors $v = \begin{bmatrix} a \\ b \end{bmatrix}$ and $u = \begin{bmatrix} c \\ d \end{bmatrix}$. Their tensor product is denoted as $v \otimes u$ and results in a block matrix.

$$\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} a \begin{bmatrix} c \\ d \end{bmatrix} b \begin{bmatrix} c \\ d \end{bmatrix} \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}$$

> **Note**
>
> Note that the tensor product is distinguished from matrix multiplication, which is an entirely different operation.

The tensor product is used to represent the combined state of multiple qubits. The real power of quantum computing comes from leveraging multiple qubits to perform computations. For more, see Operations on multiple qubits.

Related content

- The qubit
- Multiple qubits
- Dirac notation
- Pauli measurements

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

The qubit in quantum computing

Article • 02/21/2025

Just as bits are the fundamental object of information in classical computing, *qubits* (quantum bits) are the fundamental object of information in quantum computing. To understand this correspondence, this article looks at the simplest example: a single qubit.

Representing a qubit

While a bit, or binary digit, can have a value either 0 or 1, a qubit can have a value that is either 0, 1 or a quantum superposition of 0 and 1.

The state of a single qubit can be described by a two-dimensional column vector of unit norm, that is, the magnitude squared of its entries must sum to 1. This vector, called the quantum state vector, holds all the information needed to describe the one-qubit quantum system just as a single bit holds all of the information needed to describe the state of a binary variable. For the basics of vectors and matrices in quantum computing, see Vector and matrices.

Any two-dimensional column vector of real or complex numbers with norm 1 represents a possible quantum state held by a qubit. Thus $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ represents a qubit state if α and β are complex numbers satisfying $|\alpha|^2 + |\beta|^2 = 1$.

Some examples of valid quantum state vectors representing qubits are $\begin{bmatrix}1\\ 0\end{bmatrix}$ and $\begin{bmatrix}0\\ 1\end{bmatrix}$. These two vectors form a basis for the vector space that describes the qubit's state. This means that any quantum state vector can be written as a sum of these basis vectors. Specifically, the vector $\begin{bmatrix}x\\ y\end{bmatrix}$ can be written as $x\begin{bmatrix}1\\ 0\end{bmatrix} + y\begin{bmatrix}0\\ 1\end{bmatrix}$. While any rotation of these vectors would serve as a perfectly valid basis for the qubit, this particular one is chosen, by calling it the *computational basis*.

These two quantum states are taken to correspond to the two states of a classical bit, namely 0 and 1. The standard convention is to choose

$$0 \equiv \begin{bmatrix}1\\ 0\end{bmatrix}$$

$$1 \equiv \begin{bmatrix}0\\ 1\end{bmatrix},$$

although the opposite choice could equally well be taken. Thus, out of the infinite number of possible single-qubit quantum state vectors, only two correspond to states of classical bits; all other quantum states don't.

Measuring a qubit

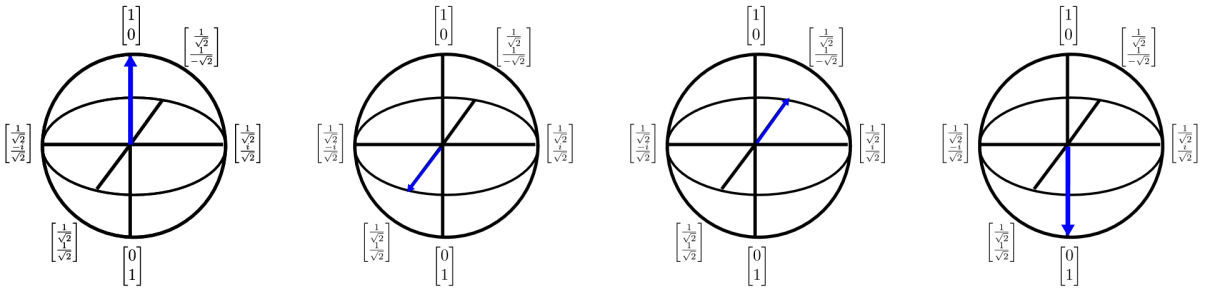
How to represent a qubit being explained, one can gain some intuition for what these states represent by discussing the concept of *measurement*. A measurement corresponds to the informal idea of "looking" at a qubit, which immediately collapses the quantum state to one of the two classical states $\begin{bmatrix}1\\0\end{bmatrix}$ or $\begin{bmatrix}0\\1\end{bmatrix}$. When a qubit given by the quantum state vector $\begin{bmatrix}\alpha\\\beta\end{bmatrix}$ is measured, the outcome 0 is obtained with probability $|\alpha|^2$ and the outcome 1 with probability $|\beta|^2$. On outcome 0, the qubit's new state is $\begin{bmatrix}1\\0\end{bmatrix}$; on outcome 1 its state is $\begin{bmatrix}0\\1\end{bmatrix}$. Note that these probabilities sum up to 1 because of the normalization condition $|\alpha|^2 + |\beta|^2 = 1$.

The properties of measurement also mean that the overall sign of the quantum state vector is irrelevant. Negating a vector is equivalent to $\alpha \to -\alpha$ and $\beta \to -\beta$. Because the probability of measuring 0 and 1 depends on the magnitude squared of the terms, inserting such signs does not change the probabilities whatsoever. Such phases are commonly called "*global phases*" and more generally can be of the form $e^{i\phi}$ rather than just ± 1.

A final important property of measurement is that it doesn't necessarily damage all quantum state vectors. If one starts with a qubit in the state $\begin{bmatrix}1\\0\end{bmatrix}$, which corresponds to the classical state 0, measuring this state always yields the outcome 0 and leave the quantum state unchanged. In this sense, if there are only classical bits (for example, qubits that are either $\begin{bmatrix}1\\0\end{bmatrix}$ or $\begin{bmatrix}0\\1\end{bmatrix}$) then measurement doesn't damage the system. This means that one can replicate classical data and manipulate it on a quantum computer just as one could do on a classical computer. The ability, however, to store information in both states at once is what elevates quantum computing beyond what is possible classically and further robs quantum computers of the ability to copy quantum data indiscriminately, see also the no-cloning theorem.

Visualizing qubits and transformations using the Bloch sphere

Qubits may also be pictured in 3D using the *Bloch sphere* representation. The Bloch sphere gives a way of describing a single-qubit quantum state (which is a two-dimensional complex vector) as a three-dimensional real-valued vector. This is important because it allows us to visualize single-qubit states and thereby develop reasoning that can be invaluable in understanding multi-qubit states (where sadly the Bloch sphere representation breaks down). The Bloch sphere can be visualized as follows:



The arrows in this diagram show the direction in which the quantum state vector is pointing and each transformation of the arrow can be thought of as a rotation about one of the cardinal axes. While thinking about a quantum computation as a sequence of rotations is a powerful intuition, it is challenging to use this intuition to design and describe algorithms. Q# alleviates this issue by providing a language for describing such rotations.

Single-qubit operations

Quantum computers process data by applying a universal set of quantum gates that can emulate any rotation of the quantum state vector. This notion of universality is akin to the notion of universality for traditional (for example, classical) computing where a gate set is considered to be universal if every transformation of the input bits can be performed using a finite length circuit. In quantum computing, the valid transformations that we are allowed to perform on a qubit are unitary transformations and measurement. The *adjoint operation* or the complex conjugate transpose is of crucial importance to quantum computing because it is needed to invert quantum transformations.

Single-qubit operations, or single-qubit quantum gates can be classified into two categories: Clifford gates and the non-Clifford gates. Non-Clifford gates consist only of T-gate (also known as the $\pi/8$ gate).

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

The standard set of single-qubit Clifford gates, included by default in Q#, include

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \qquad S = \begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix} = T^2, \qquad X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} = HT^4H,$$

$$Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} = T^2HT^4HT^6, \qquad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = T^4.$$

Here the operations X, Y and Z are used especially frequently and are named *Pauli operators* after their creator Wolfgang Pauli. Together with the non-Clifford gate (the T-gate), these operations can be composed to approximate any unitary transformation on a single qubit.

While the previous constitute the most popular primitive gates for describing operations on the logical level of the stack (think of the logical level as the level of the quantum algorithm), it is often convenient to consider less basic operations at the algorithmic level, for example operations closer to a function description level. Fortunately, Q# also has methods available for implementing higher-level unitaries, which in turn allow high-level algorithms to be implemented without explicitly decomposing everything down to Clifford and T-gates.

The simplest such primitive is the single qubit-rotation. Three single-qubit rotations are typically considered: R_x, R_y and R_z. To visualize the action of the rotation $R_x(\theta)$, for example, imagine pointing your right thumb along the direction of the x-axis of the Bloch sphere and rotating the vector with your hand through an angle of $\theta/2$ radians. This confusing factor of 2 arises from the fact that orthogonal vectors are 180° apart when plotted on the Bloch sphere, yet are actually 90° degrees apart geometrically. The corresponding unitary matrices are:

$$\begin{aligned}
R_z(\theta) &= e^{-i\theta Z/2} = \begin{bmatrix}e^{-i\theta/2} & 0\\ 0 & e^{i\theta/2}\end{bmatrix},\\
R_x(\theta) &= e^{-i\theta X/2} = HR_z(\theta)H = \begin{bmatrix}\cos(\theta/2) & -i\sin(\theta/2)\\ -i\sin(\theta/2) & \cos(\theta/2)\end{bmatrix},\\
R_y(\theta) &= e^{-i\theta Y/2} = SHR_z(\theta)HS^\dagger = \begin{bmatrix}\cos(\theta/2) & -\sin(\theta/2)\\ \sin(\theta/2) & \cos(\theta/2)\end{bmatrix}.
\end{aligned}$$

Just as any three rotations can be combined together to perform an arbitrary rotation in three dimensions, it can be seen from the Bloch sphere representation that any unitary matrix can be written as a sequence of three rotations as well. Specifically, for every unitary matrix U there exists $\alpha, \beta, \gamma, \delta$ such that $U = e^{i\alpha}R_x(\beta)R_z(\gamma)R_x(\delta)$. Thus $R_z(\theta)$ and H also form a universal gate set although it isn't a discrete set because θ can take any value. For this reason, and due to applications in quantum simulation, such continuous gates are crucial for quantum computation, especially at the quantum algorithm design level. To achieve fault-tolerant hardware implementation, they will ultimately be compiled into discrete gate sequences that closely approximate these rotations.

Related content

- Multiple qubits
- Dirac notation
- Pauli measurements
- T gates and T factories
- Quantum circuits

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Operations on multiple qubits

This article reviews the rules used to build multi-qubit states out of single-qubit states and discusses the gate operations needed to include in a gate set to form a universal many-qubit quantum computer. These tools are necessary to understand the gate sets that are commonly used in Q# code. They're also important to gain intuition about why quantum effects such as entanglement or interference render quantum computing more powerful than classical computing.

Single-qubit and multi-qubit gates

The true power of quantum computing only becomes evident as you increase the number of qubits. Single qubits possess some counter-intuitive features, such as the ability to be in more than one state at a given time. However, if you only had single-qubit gates in a quantum computer, then a calculator and certainly a classical supercomputer would dwarf its computational power.

Quantum computing power arises, in part, because the dimension of the vector space of quantum state vectors grows exponentially with the number of qubits. This means that while a single qubit can be trivially modeled, simulating a fifty-qubit quantum computation would arguably push the limits of existing supercomputers. Increasing the size of the computation by only one extra qubit doubles the memory required to store the state and roughly doubles the computational time. This rapid doubling of computational power is why a quantum computer with a relatively small number of qubits can far surpass the most powerful supercomputers of today, tomorrow, and beyond for some computational tasks.

Two-qubit states

Suppose you have two separate qubits in the following states:

$$\psi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

$$\phi = \begin{bmatrix} \gamma \\ \delta \end{bmatrix}$$

The corresponding two-qubit state is the result of the tensor product, or Kronecker product, of vectors, which is defined as follows:

$$\psi \otimes \phi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \otimes \begin{bmatrix} \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \begin{bmatrix} \gamma \\ \delta \end{bmatrix} \\ \beta \begin{bmatrix} \gamma \\ \delta \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{bmatrix}.$$

Therefore, given two single-qubit states ψ and ϕ, each of dimension 2, the corresponding two-qubit state $\psi \otimes \phi$ is 4-dimensional. The vector

$$\begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix}$$

represents a quantum state on two qubits if

$$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1.$$

More generally, you can see that the unit vector $v_1 \otimes v_2 \otimes \cdots \otimes v_n$ of dimension $2 \cdot 2 \cdot 2 \cdots = 2^n$ represents the quantum state of n qubits using this construction. As with single qubits, the quantum state vector of multiple qubits holds all the information needed to describe the system's behavior. For more information about vectors and tensor products, see Vectors and Matrices in Quantum Computing.

The computational basis for two-qubit states is formed by the tensor products of one-qubit basis states. For a two-qubit system, there are the following four states:

$$00 \equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$01 \equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$10 \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$11 \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

While you can always take the tensor product of two single-qubit states to form a two-qubit state, not all two-qubit quantum states can be written as the tensor product of two single-qubit states. For example, there are no states ψ and ϕ such that their tensor product is the state

$$\psi \otimes \phi = \begin{bmatrix} 1/\sqrt{2} \\ 0 \\ 0 \\ 1/\sqrt{2} \end{bmatrix}.$$

Such a two-qubit state, which can't be written as the tensor product of single-qubit states, is called an entangled state; the two qubits are said to be entangled. Loosely speaking, because the quantum state can't be thought of as a tensor product of single qubit states, the information that the state holds isn't confined to either of the qubits individually. Rather, the information is stored non-locally in the correlations between the two states. This non-locality of information is one of the major distinguishing features of quantum computing over classical computing, and is essential for many quantum protocols, including quantum error correction.

Measuring two-qubit states

Measuring two-qubit states is similar to single-qubit measurements. Measuring the state

$$\begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix}$$

yields 00 with probability $|\alpha_{00}|^2$, 01 with probability $|\alpha_{01}|^2$, 10 with probability $|\alpha_{10}|^2$, and 11 with probability $|\alpha_{11}|^2$. The variables α_{00}, α_{01}, α_{10}, and α_{11} were deliberately named to make this connection clear. After the measurement, if the outcome is 00 then the quantum state of the two-qubit system collapses and is now

$$00 \equiv \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

It's also possible to measure just one qubit of a two-qubit quantum state. When you measure only one qubit of a two-qubit state, the impact of measurement is subtly different than measuring two qubits. It's different because the entire state isn't collapsed to a computational basis state, rather it collapses to only one subsystem. In other words, measuring one qubit of a two-qubit state only collapses the related subsystem to a computational basis state.

To see this, consider measuring the first qubit of the following state, which is formed by applying the Hadamard transform H on two qubits initially set to the 0 state:

$$H^{\otimes 2}\left(\begin{bmatrix}1\\0\end{bmatrix}\otimes\begin{bmatrix}1\\0\end{bmatrix}\right)=\frac{1}{2}\begin{bmatrix}1&1&1&1\\1&-1&1&-1\\1&1&-1&-1\\1&-1&-1&1\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix}=\frac{1}{2}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\mapsto\begin{cases}\text{outcome}=0 & \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\\0\\0\end{bmatrix}\\ \text{outcome}=1 & \frac{1}{\sqrt{2}}\begin{bmatrix}0\\0\\1\\1\end{bmatrix}\end{cases}.$$

Both outcomes have a 50% probability of occurring. That outcome can be intuited from the fact that the quantum state before measurement doesn't change if 0 is swapped with 1 on the first qubit.

The mathematical rule for measuring the first or second qubit is simple. Let e_k be the k^{th} computational basis vector and S be the set of all e_k such that the qubit in question takes the value 1 for that value of k. For example, if you're interested in measuring the first qubit then S would consist of $e_1 \equiv 10$ and $e_3 \equiv 11$. Similarly, if you're interested in the second qubit S would consist of $e_2 \equiv 01$ and $e_3 \equiv 11$. Then the probability of measuring the chosen qubit to be 1 is for state vector ψ

$$P(\text{outcome}=1)=\sum_{e_k \text{ in the set } S}\psi^\dagger e_k e_k^\dagger \psi.$$

> **Note**
>
> This article uses the little-endian format to label the computational basis. In little-endian format, the least significant bits come first. For example, in little-endian format the string of bits 001 represents the number four.

Since each qubit measurement can only yield 0 or 1, the probability of measuring 0 is $1-P(\text{outcome}=1)$, which is why you only need a formula for the probability of measuring 1.

The action that such a measurement has on the state can be expressed mathematically as

$$\psi\mapsto\frac{\sum_{e_k \text{ in the set } S} e_k e_k^\dagger \psi}{\sqrt{P(\text{outcome}=1)}}.$$

The cautious reader may worry about what happens if the denominator is zero. While such state is undefined, you don't need to worry about such eventualities because the probability is

zero.

If you take ψ to be the uniform state vector given previously and are interested in measuring the first qubit, then

$$P(\text{measurement of first qubit} = 1) = (\psi^\dagger e_1)(e_1^\dagger \psi) + (\psi^\dagger e_3)(e_3^\dagger \psi) = |e_1^\dagger \psi|^2 + |e_3^\dagger \psi|^2.$$

This is just the sum of the two probabilities that would be expected for measuring the results 10 and 11. For our example, this evaluates to

$$\frac{1}{4}\left|\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\right|^2 + \frac{1}{4}\left|\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\right|^2 = \frac{1}{2}.$$

which perfectly matches our intuition. Similarly, the state after the first qubit is measured as 1 can be written as

$$\frac{\frac{e_1}{2} + \frac{e_3}{2}}{\sqrt{\frac{1}{2}}} = \frac{1}{\sqrt{2}}\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$$

again in accordance with our intuition.

Two-qubit operations

As in the single-qubit case, any unitary transformation is a valid operation on qubits. In general, a unitary transformation on n qubits is a matrix U of size $2^n \times 2^n$ (so that it acts on vectors of size 2^n), such that $U^{-1} = U^\dagger$. For example, the CNOT (controlled-NOT) gate is a commonly used two-qubit gate and is represented by the following unitary matrix:

$$\text{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

We can also form two-qubit gates by applying single-qubit gates on both qubits. For example, if you apply the gates

$$\begin{bmatrix}a & b\\c & d\end{bmatrix}$$

and

$$\begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

to the first and second qubits, respectively, this is equivalent to applying the two-qubit unitary given by their tensor product:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae & af & be & bf \\ ag & ah & bg & bh \\ ce & cf & de & df \\ cg & ch & dg & dh \end{bmatrix}.$$

Thus, you can form two-qubit gates by taking the tensor product of some known single-qubit gates. Some examples of two-qubit gates include $H \otimes H$, $X \otimes \mathbf{1}$, and $X \otimes Z$.

Note that while any two single-qubit gates define a two-qubit gate by taking their tensor product, the converse isn't true. Not all two-qubit gates can be written as the tensor product of single-qubit gates. Such a gate is called an *entangling* gate. One example of an entangling gate is the CNOT gate.

The intuition behind a controlled-not gate can be generalized to arbitrary gates. A controlled gate in general is a gate that acts as identity unless a specific qubit is 1. You denote a controlled unitary, controlled in this case on the qubit labeled x, with a $\Lambda_x(U)$. As an example $\Lambda_0(U)e_1 \otimes \psi = e_1 \otimes U\psi$ and $\Lambda_0(U)e_0 \otimes \psi = e_0 \otimes \psi$, where e_0 and e_1 are the computational basis vectors for a single qubit corresponding to the values 0 and 1. For example, consider the following controlled-Z gate then you can express this as

$$\Lambda_0(Z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = (\mathbf{1} \otimes H)\,\mathrm{CNOT}(\mathbf{1} \otimes H).$$

Building controlled unitaries in an efficient manner is a major challenge. The simplest way to implement controlled unitaries requires forming a database of controlled versions of fundamental gates and replacing every fundamental gate in the original unitary operation with its controlled counterpart. This method is often wasteful, and insight often can be used to just replace a few gates with controlled versions to achieve the same impact. For this reason, the framework provides the ability to perform either the naive method of controlling or allow the user to define a controlled version of the unitary if an optimized hand-tuned version is known.

Gates can also be controlled using classical information. A classically controlled not-gate, for example, is just an ordinary not-gate that is only applied if a classical bit is 1 as opposed to a quantum bit. In this sense, a classically controlled gate can be thought of as an if statement in the quantum code wherein the gate is applied only in one branch of the code.

As in the single-qubit case, a two-qubit gate set is universal if any 4×4 unitary matrix can be approximated by a product of gates from this set to arbitrary precision. One example of a universal gate set is the Hadamard gate, the T gate, and the CNOT gate. By taking products of these gates, you can approximate any unitary matrix on two qubits.

Many-qubit systems

We follow exactly the same patterns explored in the two-qubit case to build many-qubit quantum states from smaller systems. Such states are built by forming tensor products of smaller states. For example, consider encoding the bit string 1011001 in a quantum computer. You can encode this as

$$1011001 \equiv \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix}.$$

Quantum gates work in exactly the same way. For example, if you wish to apply the X gate to the first qubit and then perform a CNOT between the second and third qubits you can express this transformation as

$$(X \otimes \mathrm{CNOT}_{12} \otimes \mathbf{1} \otimes \mathbf{1} \otimes \mathbf{1} \otimes \mathbf{1}) \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \equiv 0011001.$$

In many qubit systems, there's often a need to allocate and deallocate qubits that serve as temporary memory for the quantum computer. Such a qubit is said to be auxiliary. By default, you can assume the qubit state is initialized to e_0 upon allocation. You can further assume that the gate returns again to e_0 before deallocation. This assumption is important because if an auxiliary qubit becomes entangled with another qubit register when it becomes deallocated, then the process of deallocation damages the auxiliary qubit. For this reason, you always assume that such qubits are reverted to their initial state before being released.

Finally, although new gates needed to be added to our gate set to achieve universal quantum computing for two qubit quantum computers, no new gates need to be introduced in the multi-qubit case. The gates H, T, and CNOT form a universal gate set on many qubits because any general unitary transformation can be broken into a series of two qubit rotations. You then can use the theory developed for the two-qubit case and use it again here when you have many qubits.

> **Note**
>
> While the linear algebraic notation that used to this point can certainly be used to describe multi-qubit states, it becomes increasingly cumbersome as you increase the size of the states. The resulting column-vector for a 7-bit string, for example, is 128

dimensional, which makes it cumbersome to express it using the notation described earlier. Instead, Dirac notation, a symbolic shorthand that simplifies the representation of quantum states, is used. For more information, see **Dirac notation**.

Related content

- Dirac notation
- Pauli measurements
- T gates and T factories
- Quantum circuits
- Quantum oracles

Last updated on 01/05/2026

Dirac notation in quantum computing

Article • 02/13/2025

Dirac notation is a concise and powerful way to describe quantum states and operations. It was named after the physicist Paul Dirac, who developed the notation in the 1930s. Dirac notation is used in quantum computing to describe quantum states, quantum operations, and quantum measurements.

This article introduces you to Dirac notation and show you how to use it to describe quantum states and operations.

Vectors in Dirac notation

There are two types of vectors in Dirac notation: the *bra* vector, corresponding to a row vector, and the *ket* vector, corresponding to a column vector.

If ψ is a column vector, then you can write it in Dirac notation as $|\psi\rangle$, where the $|\cdot\rangle$ denotes that it's a *ket* vector.

Similarly, the row vector $\psi^\dagger$ is expressed as $\langle\psi|$, which is a *bra* vector. In other words, $\psi^\dagger$ is obtained by applying entry-wise complex conjugation to the elements of the transpose of ψ. The bra-ket notation directly implies that $\langle\psi|\psi\rangle$ is the inner product of vector ψ with itself, which is by definition 1.

More generally, if ψ and ϕ are quantum state vectors, then their inner product is $\langle\phi|\psi\rangle$. This inner product implies that the probability of measuring the state $|\psi\rangle$ to be $|\phi\rangle$ is $|\langle\phi|\psi\rangle|^2$.

The computational basis states 0 and 1 are represented as $[1\ 0] = |0\rangle$ and $[0\ 1] = |1\rangle$, respectively.

Example: Represent the Hadamard operation with Dirac notation

Let's apply the Hadamard gate H to the quantum states $|0\rangle$ and $|1\rangle$ using Dirac notation:

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} = H|0\rangle = |+\rangle$$

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix} = H|1\rangle = |-\rangle$$

The resulting states correspond to the unit vectors in the $+x$ and $-x$ directions on the Bloch sphere. These states can also be expanded using Dirac notation as sums of $|0\rangle$ and $|1\rangle$:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Computational basis vectors

Every quantum state can always be expressed as sums of computational basis vectors and such sums are easily expressed using Dirac notation. The converse is also true in that the states $|+\rangle$ and $|-\rangle$ also form a basis for quantum states. You can see this basis from the fact that

$$|0\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle)$$

$$|1\rangle = \frac{1}{\sqrt{2}}(|+\rangle - |-\rangle)$$

As an example of Dirac notation, consider the braket $\langle 0|1\rangle$, which is the inner product between 0 and 1. It can be written as

$$\langle 0|1\rangle = \begin{bmatrix}1 & 0\end{bmatrix} \begin{bmatrix}0\\1\end{bmatrix} = 0.$$

This example says that $|0\rangle$ and $|1\rangle$ are orthogonal vectors, meaning that $\langle 0|1\rangle = \langle 1|0\rangle = 0$. Also, by definition $\langle 0|0\rangle = \langle 1|1\rangle = 1$, which means that the two computational basis vectors can also be called *orthonormal*.

These orthonormal properties are used in the following example. If you have a state $|\psi\rangle = \frac{3}{5}|1\rangle + \frac{4}{5}|0\rangle$, then because $\langle 1|0\rangle = 0$ the probability of measuring 1 is

$$\left|\langle 1|\psi\rangle\right|^2 = \left|\frac{3}{5}\langle 1|1\rangle + \frac{4}{5}\langle 1|0\rangle\right|^2 = \frac{9}{25}.$$

Tensor product notation

Dirac notation is useful to express the tensor product. Tensor product is important in quantum computing because the state vector described by two uncorrelated quantum registers is the tensor products of the two state vectors.

The tensor product $\psi \otimes \phi$ for any two quantum state vectors ϕ and ψ is written in Dirac notation as $|\psi\rangle \otimes |\phi\rangle$. By convention, you can also write the tensor product as $|\psi\rangle\,|\phi\rangle = |\psi\phi\rangle$.

For example, the state with two qubits initialized to the zero state is $|0\rangle \otimes |0\rangle = |0\rangle\,|0\rangle = |00\rangle$.

Example: Describe superposition with Dirac notation

As another example of how you can use Dirac notation to describe a quantum state, consider the following equivalent ways of writing a quantum state that is an equal superposition over every possible bit string of length n

$$H^{\otimes n}\,|0\rangle = \frac{1}{2^{n/2}} \sum_{j=0}^{2^n-1} |j\rangle = |+\rangle^{\otimes n}.$$

Here you may wonder why the sum goes from 0 to $2^n - 1$ for n bits. First, there are 2^n different configurations that n bits can take. You can see this configuration by noting that one bit can take 2 values but two bits can take 4 values, and so forth. In general, this means that there are 2^n different possible bit strings but the largest value encoded in any of them $1 \cdots 1 = 2^n - 1$ and hence it's the upper limit for the sum. Also, in this example you didn't use $|+\rangle^{\otimes n} = |+\rangle$ in analogy to $|0\rangle^{\otimes n} = |0\rangle$. This notational convention is reserved for the computational basis state with every qubit initialized to zero.

Express linearity with Dirac notation

Another feature of Dirac notation is the fact that it's linear. For example, for two complex numbers α and β, you can write

$$|\psi\rangle \otimes (\alpha\,|\phi\rangle + \beta\,|\chi\rangle) = \alpha\,|\psi\rangle\,|\phi\rangle + \beta\,|\psi\rangle\,|\chi\rangle.$$

You can distribute the tensor product notation in Dirac notation so that taking tensor products between state vectors ends up looking just like ordinary multiplication.

Bra vectors follow a similar convention to ket vectors. For example, the vector $\langle\psi|\,\langle\phi|$ is equivalent to the state vector $\psi^\dagger \otimes \phi^\dagger = (\psi \otimes \phi)^\dagger$. If the ket vector $|\psi\rangle$ is $\alpha\,|0\rangle + \beta\,|1\rangle$, then the bra vector version of the vector is $\langle\psi| = |\psi\rangle^\dagger = (\langle 0|\alpha^* + \langle 1|\beta^*)$.

As an example, imagine that you wish to calculate the probability of measuring the state $|\psi\rangle = \frac{3}{5}|1\rangle + \frac{4}{5}|0\rangle$ using a quantum program for measuring states to be either $|+\rangle$ or $|-\rangle$. Then the probability that the device would output that the state is $|-\rangle$ is

$$|\langle -|\psi\rangle|^2 = \left|\frac{1}{\sqrt{2}}(\langle 0| - \langle 1|)(\frac{3}{5}|1\rangle + \frac{4}{5}|0\rangle)\right|^2 = \left|-\frac{3}{5\sqrt{2}} + \frac{4}{5\sqrt{2}}\right|^2 = \frac{1}{50}.$$

The fact that the negative sign appears in the calculation of the probability is a manifestation of quantum interference, which is one of the mechanisms by which quantum computing gains advantages over classical computing.

ketbra or outer product

The final item worth discussing in Dirac notation is the *ketbra* or outer product. The outer product is represented within Dirac notations as $|\psi\rangle\langle\phi|$. The outer product is defined via matrix multiplication as $|\psi\rangle\langle\phi| = \psi\phi^\dagger$ for quantum state vectors ψ and ϕ. The simplest, and arguably most common example of this notation, is

$$|0\rangle\langle 0| = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1 & 0\end{bmatrix} = \begin{bmatrix}1 & 0\\0 & 0\end{bmatrix} \qquad |1\rangle\langle 1| = \begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0 & 1\end{bmatrix} = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}.$$

Ketbras are often called projectors because they project a quantum state onto a fixed value. Since these operations aren't unitary (and do not even preserve the norm of a vector), a quantum computer can't deterministically apply a projector. However projectors do a beautiful job of describing the action that measurement has on a quantum state. For example, if you measure a state $|\psi\rangle$ to be 0, then the resulting transformation that the state experiences as a result of the measurement is

$$|\psi\rangle \to \frac{(|0\rangle\langle 0|)|\psi\rangle}{|\langle 0|\psi\rangle|} = |0\rangle,$$

as you would expect if you measured the state and found it to be $|0\rangle$. To reiterate, such projectors can't be applied on a state in a quantum computer deterministically. Instead, they can at best be applied randomly with the result $|0\rangle$ appearing with some fixed probability. The probability of such a measurement succeeding can be written as the expectation value of the quantum projector in the state

$$\langle\psi|(|0\rangle\langle 0|)|\psi\rangle = |\langle\psi|0\rangle|^2,$$

which illustrates that projectors give a new way of expressing the measurement process.

If instead you consider measuring the first qubit of a multi-qubit state to be 1, then you can also describe this process conveniently using projectors and Dirac notation:

$$P(\text{first qubit} = 1) = \langle\psi|\left(|1\rangle\langle 1| \otimes \mathbf{1}^{\otimes n-1}\right)|\psi\rangle.$$

Here the identity matrix can be conveniently written in Dirac notation as

$$\mathbb{I} = |0\rangle\langle 0| + |1\rangle\langle 1| = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

For the case where there are two-qubits, the projector can be expanded as

$$|1\rangle\langle 1| \otimes \mathbb{I} = |1\rangle\langle 1| \otimes (|0\rangle\langle 0| + |1\rangle\langle 1|) = |10\rangle\langle 10| + |11\rangle\langle 11|.$$

and you can see that this projector is consistent with the discussion about measurement likelihoods for multiqubit states using column-vector notation:

$$P(\text{first qubit} = 1) = \psi^\dagger (e_{10}e_{10}^\dagger + e_{11}e_{11}^\dagger)\psi = |e_{10}^\dagger \psi|^2 + |e_{11}^\dagger \psi|^2,$$

which matches the multi-qubit measurement discussion. The generalization of this result to the multi-qubit case, however, is slightly more straightforward to express using Dirac notation than column-vector notation, and is entirely equivalent to the previous treatment.

Density operators

Another useful operator to express using Dirac notation is the *density operator*, sometimes also known as a *state operator*. As the quantum state vector, the density operator describes the quantum state of a system. While quantum state vectors can only represent *pure states*, density operators can also represent *mixed states*.

More generally, a given matrix ρ is a valid density operator if the following conditions are fulfilled:

- ρ is a matrix of complex numbers
- $\rho = \rho^\dagger$ (that is, ρ is Hermitian)
- Every eigenvalue p of ρ is non-negative
- All the eigenvalues of ρ sum to 1

Together, these conditions guarantee that ρ can be thought of as an ensemble. A density operator for a quantum state vector $|\psi\rangle$ takes the form $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$ is an eigenvalue decomposition of ρ, then ρ describes the ensemble $\rho = |\psi_i\rangle \text{with probability} p_i$.

Pure quantum states are characterized by a single ket vector or wavefunction, and can't be written as a statistical mixture, or *convex combination*, of other quantum states. A mixed quantum state is a statistical ensemble of pure states.

A Bloch sphere represents pure states by a point on the surface of the sphere, and mixed states by an interior point on the sphere. The center of the sphere represents a

mixed state of a single qubit, by symmetry. The purity of a state can be visualized as the degree in which it's close to the surface of the sphere.

This concept of representing the state as a matrix, rather than a vector, is often convenient because it gives a convenient way of representing probability calculations. It also allows you to describe both statistical uncertainty and quantum uncertainty within the same formalism.

A density operator ρ represents a pure state if and only if:

- ρ can be written as an outer product of a state vector, $\rho = |\psi\rangle \langle\psi|$
- $\rho = \rho^2$
- $tr(\rho^2) = 1$

To tell how close a given density operator ρ is to being pure, you can look at the trace - the sum of the diagonal elements - of ρ^2. A density operator represents a pure state if and only if $tr(\rho^2) = 1$.

Q# gate sequences equivalent to quantum states

A final point worth raising about quantum notation and the Q# programming language: earlier, this article mentioned that the quantum state is the fundamental object of information in quantum computing. It may then come as a surprise that in Q# there's no notion of a quantum state. Instead, Q# describes all states only by the operations used to prepare them. The previous example is an excellent illustration of this definition. Rather than expressing a uniform superposition over every quantum bit string in a register, you can represent the result as $H^{\otimes n} |0\rangle$. This exponentially shorter description of the state has the advantage that you can classically reason about it. It also concisely defines the operations it needs to be propagated through the software stack to implement the algorithm. For this reason, Q# is designed to emit gate sequences rather than quantum states; however, at a theoretical level the two perspectives are equivalent.

Related topics

- Pauli measurements
- T gates and T factories
- Quantum circuits
- Quantum oracles

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Single-qubit and multi-qubit Pauli measurements

Article • 12/18/2024

As you work with Q#, you find that *Pauli measurements* are a common type of measurement. Pauli measurements generalize computational basis measurements to include measurements in other bases and of parity between different qubits. In such cases, it's common to discuss measuring a Pauli operator, which is an operator such as X, Y, Z or $Z \otimes Z, X \otimes X, X \otimes Y$, and so forth. For the basics of quantum measurement, see The qubit and Multiple qubits.

Discussing measurement in terms of Pauli operators is common in the subfield of quantum error correction.
Q# guide follows a similar convention; this article explains this alternative view of measurements.

> **Tip**
>
> In Q#, multi-qubit Pauli operators are generally represented by arrays of type `Pauli[]`. For example, to represent $X \otimes Z \otimes Y$, you can use the array `[PauliX, PauliZ, PauliY]`.

Before delving into the details of how to think of a Pauli measurement, it's useful to think about what measuring a single qubit inside a quantum computer does to the quantum state. Imagine a n-qubit quantum state; then measuring one qubit immediately rules out half of the 2^n possibilities that state could be in. In other words, the measurement projects the quantum state onto one of two half-spaces. You can generalize the way you think about measurement to reflect this intuition.

In order to concisely identify these subspaces, one needs a language for describing them. One way to describe the two subspaces is by specifying them through a matrix that just has two unique eigenvalues, taken by convention to be ± 1. For a simple example of describing subspaces in this way, consider Z:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

When you read the diagonal elements of the Pauli-Z matrix, you can see that Z has two eigenvectors, $|0\rangle$ and $|1\rangle$, with corresponding eigenvalues ± 1. Thus, if a measurement of the qubit results in `Zero` (corresponding to the state $|0\rangle$), it's known that the state of

the qubit is a $+1$ eigenstate of the Z operator. Similarly, if the result is `One`, it's known that the state of the qubit is a -1 eigenstate of Z. This process is referred to in the language of Pauli measurements as "measuring Pauli Z," and is entirely equivalent to performing a computational basis measurement.

Any 2×2 matrix that is a unitary transformation of Z also satisfies this criteria. That is, you could also use a matrix $A = U^\dagger Z U$, where U is any other unitary matrix, to give a matrix that defines the two outcomes of a measurement in its ± 1 eigenvectors. The notation of Pauli measurements references this unitary equivalence by identifying X, Y, Z measurements as equivalent measurements that you could do to gain information from a qubit. These measurements are given here for convenience.

Expand table

Pauli Measurement	Unitary transformation
Z	$\mathbf{1}$
X	H
Y	$HS^\dagger$

That is, using this language, "measure Y" is equivalent to applying $HS^\dagger$ and then measuring in the computational basis, where `S` is an intrinsic quantum operation sometimes called the "phase gate," and can be simulated using the unitary matrix

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

It's also equivalent to applying $HS^\dagger$ to the quantum state vector and then measuring Z, such that the following operation is equivalent to `Measure([PauliY], [q])`:

Q#

```
operation MeasureY(qubit : Qubit) : Result {
    mutable result = Zero;
    within {
        Adjoint S(q);
        H(q);
    } apply {
        result = M(q);
    }
    return result;
}
```

The correct state is then found by transforming back to the computational basis, which amounts to applying SH to the quantum state vector; in the code snippet, the transformation back to the computational basis is handled automatically with the use of the `within … apply` block.

In Q#, the outcome---that is, the classical information extracted from interacting with the state---is given using a `Result` value $j \in \{\texttt{Zero}, \texttt{One}\}$ indicating if the result is in the $(-1)^j$ eigenspace of the Pauli operator measured.

Multiple-qubit measurements

Measurements of multi-qubit Pauli operators are defined similarly, as seen from:

$$Z \otimes Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Thus the tensor products of two Pauli-Z operators forms a matrix composed of two spaces consisting of $+1$ and -1 eigenvalues. As with the single-qubit case, both constitute a *half-space*, meaning that half of the accessible vector space belongs to the $+1$ eigenspace and the remaining half to the -1 eigenspace. In general, it's easy to see from the definition of the tensor product that any tensor product of Pauli-Z operators and the identity also obeys this. For example,

$$Z \otimes \mathbf{1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

As before, any unitary transformation of such matrices also describes two half-spaces labeled with ± 1 eigenvalues. For example, $X \otimes X = H \otimes H (Z \otimes Z) H \otimes H$ from the identity that $Z = HXH$. Similar to the one-qubit case, all two-qubit Pauli-measurements can be written as $U^\dagger (Z \otimes 1) U$ for 4×4 unitary matrices U. The transformations are enumerated in the following table.

> **Note**
>
> In this table, SWAP is used to indicate the matrix

$$\mathrm{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

used to simulate the intrinsic operation `SWAP`.

Expand table

Pauli Measurement	Unitary transformation
$Z \otimes \mathbf{1}$	$\mathbf{1} \otimes \mathbf{1}$
$X \otimes \mathbf{1}$	$H \otimes \mathbf{1}$
$Y \otimes \mathbf{1}$	$HS^\dagger \otimes \mathbf{1}$
$\mathbf{1} \otimes Z$	SWAP
$\mathbf{1} \otimes X$	$(H \otimes \mathbf{1})\,\mathrm{SWAP}$
$\mathbf{1} \otimes Y$	$(HS^\dagger \otimes \mathbf{1})\,\mathrm{SWAP}$
$Z \otimes Z$	CNOT_{10}
$X \otimes Z$	$\mathrm{CNOT}_{10}(H \otimes \mathbf{1})$
$Y \otimes Z$	$\mathrm{CNOT}_{10}(HS^\dagger \otimes \mathbf{1})$
$Z \otimes X$	$\mathrm{CNOT}_{10}(\mathbf{1} \otimes H)$
$X \otimes X$	$\mathrm{CNOT}_{10}(H \otimes H)$
$Y \otimes X$	$\mathrm{CNOT}_{10}(HS^\dagger \otimes H)$
$Z \otimes Y$	$\mathrm{CNOT}_{10}(\mathbf{1} \otimes HS^\dagger)$
$X \otimes Y$	$\mathrm{CNOT}_{10}(H \otimes HS^\dagger)$
$Y \otimes Y$	$\mathrm{CNOT}_{10}(HS^\dagger \otimes HS^\dagger)$

Here, the `CNOT` operation appears for the following reason. Each Pauli measurement that doesn't include the $\mathbf{1}$ matrix is equivalent up to a unitary to $Z \otimes Z$ by the earlier reasoning. The eigenvalues of $Z \otimes Z$ only depend on the parity of the qubits that comprise each computational basis vector, and the controlled-not operations serve to compute this parity and store it in the first bit. Then once the first bit is measured, you can recover the identity of the resultant half-space, which is equivalent to measuring the Pauli operator.

Also, while it can be tempting to assume that measuring $Z \otimes Z$ is the same as sequentially measuring $Z \otimes \mathbb{1}$ and then $\mathbb{1} \otimes Z$, this assumption would be false. The reason is that measuring $Z \otimes Z$ projects the quantum state into either the $+1$ or -1 eigenstate of these operators. Measuring $Z \otimes \mathbb{1}$ and then $\mathbb{1} \otimes Z$ projects the quantum state vector first onto a half space of $Z \otimes \mathbb{1}$ and then onto a half space of $\mathbb{1} \otimes Z$. As there are four computational basis vectors, performing both measurements reduces the state to a quarter-space and hence reduces it to a single computational basis vector.

Correlations between qubits

Another way of looking at measuring tensor products of Pauli matrices, such as $X \otimes X$ or $Z \otimes Z$, is that these measurements let you look at information stored in the correlations between the two qubits. Measuring $X \otimes 1$ lets you look at information that is locally stored in the first qubit. While both types of measurements are equally valuable in quantum computing, the former illuminates the power of quantum computing. It reveals that in quantum computing, often the information you wish to learn isn't stored in any single qubit but rather stored non-locally in all the qubits at once, and therefore only by looking at it through a joint measurement (e.g. $Z \otimes Z$) does this information become manifest.

Arbitrary Pauli operators such as $X \otimes Y \otimes Z \otimes \mathbf{1}$ can also be measured. All such tensor products of Pauli operators have only two eigenvalues ± 1 and both eigenspaces constitute half-spaces of the entire vector space. Thus they coincide with the requirements stated earlier.

In Q#, such measurements return j if the measurement yields a result in the eigenspace of sign $(-1)^j$. Having Pauli measurements as a built-in feature in Q# is helpful because measuring such operators requires long chains of controlled-NOT gates and basis transformations to describe the diagonalizing U gate needed to express the operation as a tensor product of Z and 1. By being able to specify that you wish to do one of these pre-defined measurements, you don't need to worry about how to transform your basis such that a computational basis measurement provides the necessary information. Q# handles all the necessary basis transformations for you automatically.

The No-Cloning Theorem

Quantum information is powerful. It enables you to do amazing things such as factor numbers exponentially faster than the best known classical algorithms, or efficiently simulate correlated electron systems that classically require exponential cost to simulate accurately. However, there are limitations to the power of quantum computing. One such limitation is given by the *No-Cloning Theorem*.

The No-Cloning Theorem is aptly named. It disallows cloning of generic quantum states by a quantum computer. The proof of the theorem is remarkably straightforward. While a full proof of the no-cloning theorem is too technical for this article, the proof in the case of no additional auxiliary qubits is within the scope.

For such a quantum computer, the cloning operation must be described with a unitary matrix. Quantum measurement is disallowed, since it would corrupt the quantum state to be cloned. To simulate the cloning operation, the unitary matrix used needs to have the property that

$$U\left|\psi\right\rangle\left|0\right\rangle = \left|\psi\right\rangle\left|\psi\right\rangle$$

for any state $\left|\psi\right\rangle$. The linearity property of matrix multiplication then implies that for any second quantum state $\left|\phi\right\rangle$,

$$\begin{aligned} U\left[\frac{1}{\sqrt{2}}(\left|\phi\right\rangle + \left|\psi\right\rangle)\right]\left|0\right\rangle &= \frac{1}{\sqrt{2}}U\left|\phi\right\rangle\left|0\right\rangle + \frac{1}{\sqrt{2}}U\left|\psi\right\rangle\left|0\right\rangle \\ &= \frac{1}{\sqrt{2}}(\left|\phi\right\rangle\left|\phi\right\rangle + \left|\psi\right\rangle\left|\psi\right\rangle) \\ &\neq \left(\frac{1}{\sqrt{2}}(\left|\phi\right\rangle + \left|\psi\right\rangle)\right) \otimes \left(\frac{1}{\sqrt{2}}(\left|\phi\right\rangle + \left|\psi\right\rangle)\right). \end{aligned}$$

This provides the fundamental intuition behind the No-Cloning Theorem: any device that copies an unknown quantum state must induce errors on at least some of the states it copies. While the key assumption that the cloner acts linearly on the input state can be violated through the addition and measurement of auxiliary qubits, such interactions also leak information about the system through the measurement statistics and prevent exact cloning in such cases as well.

The No-Cloning Theorem is important for qualitative understanding of quantum computing because if you could clone quantum states inexpensively then you would be granted a near-magical ability to learn from quantum states. Indeed, you could violate Heisenberg's vaunted uncertainty principle. Alternatively, you could use an optimal cloner to take a single sample from a complex quantum distribution and learn everything you could possibly learn about that distribution from just one sample. This would be like you flipping a coin and observing heads and then upon telling a friend about the result having them respond "Ah the distribution of that coin must be Bernoulli with $p = 0.512643\ldots$!" Such a statement would be nonsensical because one bit of information (the heads outcome) simply can't provide the many bits of information needed to encode the distribution without substantial prior information. Similarly, without prior information you can't perfectly clone a quantum state just as you can't prepare an ensemble of such coins without knowing p.

Information isn't free in quantum computing. Each qubit measured gives a single bit of information, and the No-Cloning Theorem shows that there's no backdoor that can be exploited to get around the fundamental tradeoff between information gained about the system and the disturbance invoked upon it.

Related content

- T gates and T factories
- Quantum circuits
- Quantum oracles
- Grover's algorithm

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Theory of Grover's search algorithm

Article • 01/16/2025

In this article you'll find a detailed theoretical explanation of the mathematical principles that make Grover's algorithm work.

For a practical implementation of Grover's algorithm to solve mathematical problems, see Implement Grover's search algorithm.

Statement of the problem

Grover's algorithm speeds up the solution to unstructured data searches (or *search problem*), running the search in fewer steps than any classical algorithm could. Any search task can be expressed with an abstract function $f(x)$ that accepts search items x. If the item x is a solution to the search task, then $f(x) = 1$. If the item x isn't a solution, then $f(x) = 0$. The *search problem* consists of finding any item x_0 such that $f(x_0) = 1$.

Any problem that allows you to check whether a given value x is a valid solution (a "yes or no problem") can be formulated in terms of the search problem. The following are some examples:

- Boolean satisfiability problem: Given a set of Boolean values, does the set satisfy a given Boolean formula?
- Traveling salesman problem: What is the shortest possible loop that connects all cities?
- Database search problem: Does a database table contain the record x?
- Integer factorization problem: Is the number N divisible by the number x?

The task that Grover's algorithm aims to solve can be expressed as follows: given a classical function $f(x) : \{0, 1\}^n \to \{0, 1\}$, where n is the bit-size of the search space, find an input x_0 for which $f(x_0) = 1$. The complexity of the algorithm is measured by the number of uses of the function $f(x)$. Classically, in the worst-case scenario, $f(x)$ has to be evaluated a total of $N - 1$ times, where $N = 2^n$, trying out all the possibilities. After $N - 1$ elements, it must be the last element. Grover's quantum algorithm can solve this problem much faster, providing a quadratic speed up. Quadratic here implies that only about $\sqrt{N}$ evaluations would be required, compared to N.

Outline of the algorithm

Suppose there are $N = 2^n$ eligible items for the search task and they are index by assigning each item an integer from 0 to $N - 1$. Further, suppose that there are M different valid inputs, meaning that there are M inputs for which $f(x) = 1$. The steps of the algorithm are then as follows:

1. Start with a register of n qubits initialized in the state $|0\rangle$.
2. Prepare the register into a uniform superposition by applying H to each qubit of the register:

$$|\text{register}\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$$

3. Apply the following operations to the register N_{optimal} number of times:
   a. The phase oracle O_f that applies a conditional phase shift of -1 for the solution items.
   b. Apply H to each qubit in the register.
   c. A conditional phase shift of -1 to every computational basis state except $|0\rangle$. This can be represented by the unitary operation $-O_0$, as O_0 represents the conditional phase shift to $|0\rangle$ only.
   d. Apply H to each qubit in the register.
4. Measure the register to obtain the index of a item that's a solution with very high probability.
5. Check if it's a valid solution. If not, start again.

$N_{\text{optimal}} = \left\lfloor \frac{\pi}{4} \sqrt{\frac{N}{M}} - \frac{1}{2} \right\rfloor$ is the optimal number of iterations that maximizes the likelihood of obtaining the correct item by measuring the register.

> ⓘ **Note**
>
> The joint application of the steps 3.b, 3.c, and 3.d is usually known as *Grover's diffusion operator*.

The overall unitary operation applied to the register is:

$$(-H^{\otimes n} O_0 H^{\otimes n} O_f)^{N_{\text{optimal}}} H^{\otimes n}$$

Following the register's state step by step

To illustrate the process, let's follow the mathematical transformations of the state of the register for a simple case in which there are only two qubits and the valid element is $|01\rangle$.

1. The register starts in the state:

$$|\text{register}\rangle = |00\rangle$$

2. After applying H to each qubit the register's state transforms to:

$$|\text{register}\rangle = \frac{1}{\sqrt{4}} \sum_{i \in \{0,1\}}^{2} |i\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$$

3. Then, the phase oracle is applied to get:

$$|\text{register}\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle + |11\rangle)$$

4. Then H acts on each qubit again to give:

$$|\text{register}\rangle = \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle + |11\rangle)$$

5. Now the conditional phase shift is applied on every state except $|00\rangle$:

$$|\text{register}\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$$

6. Finally, the first Grover iteration ends by applying H again to get:

$$|\text{register}\rangle = |01\rangle$$

By following the above steps, the valid item is found in a single iteration. As you will see later, this is because for N=4 and a single valid item, the optimal number of times is $N_{\text{optimal}} = 1$.

Geometrical explanation

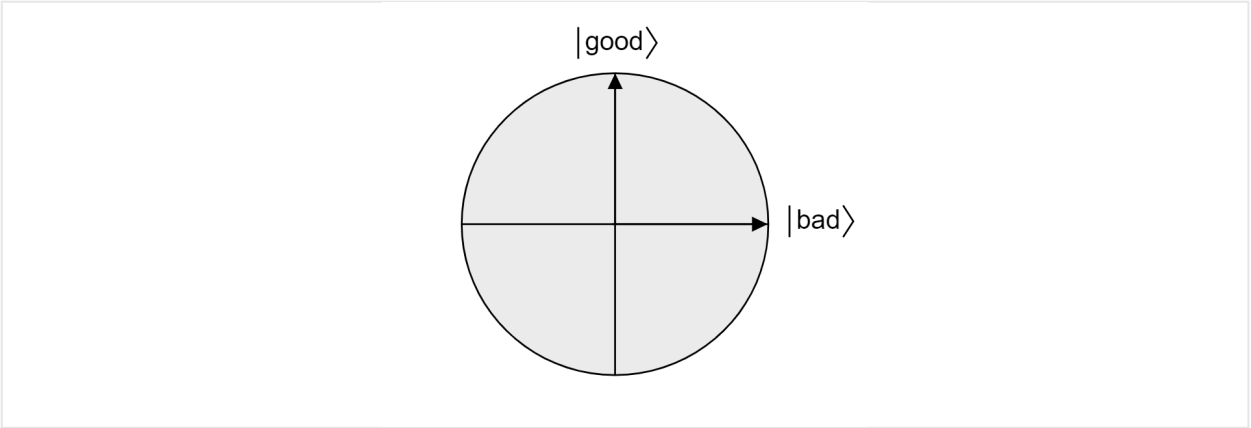
To see why Grover's algorithm works, let's study the algorithm from a geometrical perspective. Supposing there are M valid solutions, the superposition of all quantum states that *aren't* a solution to the search problem:

$$|\text{bad}\rangle = \frac{1}{\sqrt{N-M}} \sum_{x:f(x)=0} |x\rangle$$

The superposition of all states that *are* a solution to the search problem:

$$|\text{good}\rangle = \frac{1}{\sqrt{M}} \sum_{x:f(x)=1} |x\rangle$$

Since *good* and *bad* are mutually exclusive sets because an item can't be valid and not valid, the states are orthogonal. Both states form the orthogonal basis of a plane in the vector space. One can use this plane to visualize the algorithm.



Now, suppose $|\psi\rangle$ is an arbitrary state that lives in the plane spanned by $|\mathrm{good}\rangle$ and $|\mathrm{bad}\rangle$. Any state living in that plane can be expressed as:

$$|\psi\rangle = \alpha\,|\mathrm{good}\rangle + \beta\,|\mathrm{bad}\rangle$$

where α and β are real numbers. Now, let's introduce the reflection operator $R_{|\psi\rangle}$, where $|\psi\rangle$ is any qubit state living in the plane. The operator is defined as:

$$R_{|\psi\rangle} = 2\,|\psi\rangle\,\langle\psi| - \mathcal{I}$$

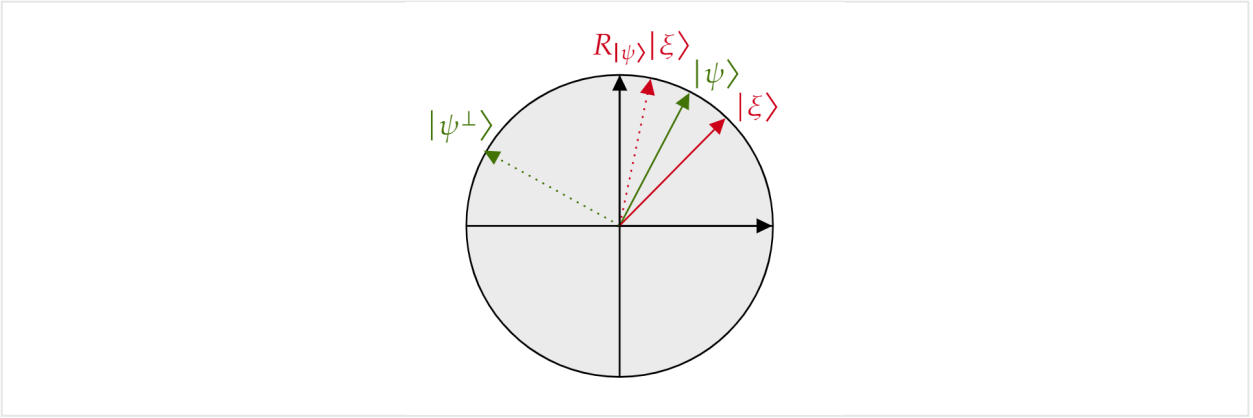
It is called the reflection operator about $|\psi\rangle$ because it can be geometrically interpreted as reflection about the direction of $|\psi\rangle$. To see it, take the orthogonal basis of the plane formed by $|\psi\rangle$ and its orthogonal complement $\left|\psi^{\perp}\right\rangle$. Any state $|\xi\rangle$ of the plane can be decomposed in such basis:

$$|\xi\rangle = \mu\,|\psi\rangle + \nu\left|\psi^{\perp}\right\rangle$$

If one applies the operator $R_{|\psi\rangle}$ to $|\xi\rangle$:

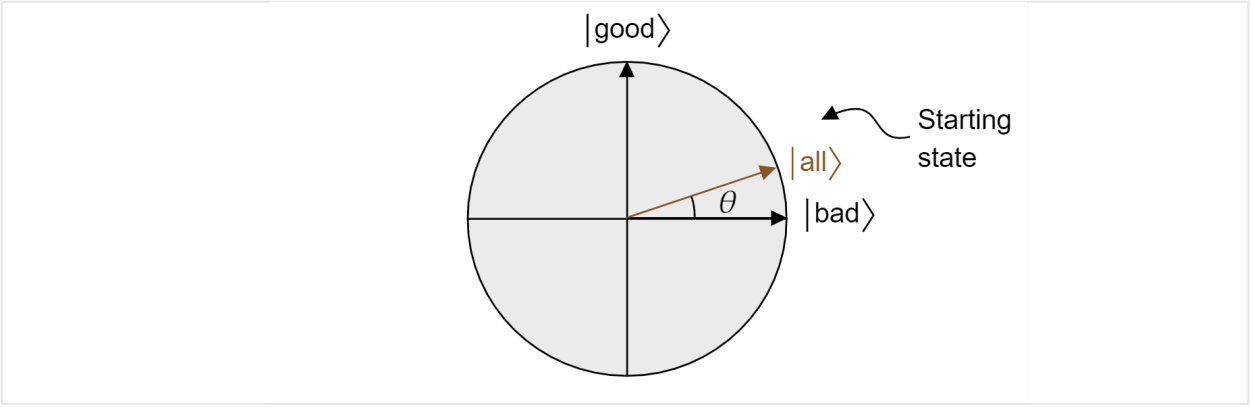
$$R_{|\psi\rangle}\,|\xi\rangle = \mu\,|\psi\rangle - \nu\left|\psi^{\perp}\right\rangle$$

The operator $R_{|\psi\rangle}$ inverts the component orthogonal to $|\psi\rangle$ but leaves the $|\psi\rangle$ component unchanged. Therefore, $R_{|\psi\rangle}$ is a reflection about $|\psi\rangle$.



Grover's algorithm, after the first application of H to every qubit, starts with an uniform superposition of all states. This can be written as:

$$|\text{all}\rangle = \sqrt{\frac{M}{N}}\,|\text{good}\rangle + \sqrt{\frac{N-M}{N}}\,|\text{bad}\rangle$$



And thus the state lives in the plane. Note that the probability of obtaining a correct result when measuring from the equal superposition is just $|\langle\text{good}|\,|\text{all}\rangle|^2 = M/N$, which is what you would expect from a random guess.

The oracle O_f adds a negative phase to any solution to the search problem. Therefore, it can be written as a reflection about the $|\text{bad}\rangle$ axis:

$$O_f = R_{|\text{bad}\rangle} = 2\,|\text{bad}\rangle\,\langle\text{bad}| - \mathbb{I}$$

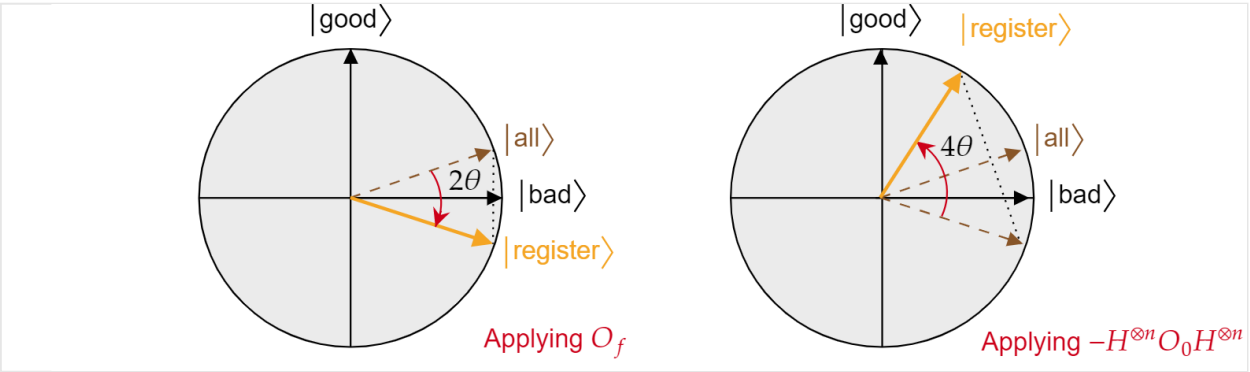
Analogously, the conditional phase shift O_0 is just an inverted reflection about the state $|0\rangle$:

$$O_0 = R_{|0\rangle} = -2\,|0\rangle\,\langle 0| + \mathbb{I}$$

Knowing this fact, it's easy to check that the Grover diffusion operation $-H^{\otimes n}O_0H^{\otimes n}$ is also a reflection about the state $|all\rangle$. Just do:

$$-H^{\otimes n}O_0H^{\otimes n} = 2H^{\otimes n}\,|0\rangle\,\langle 0|H^{\otimes n} - H^{\otimes n}\mathbb{I}H^{\otimes n} = 2\,|\text{all}\rangle\,\langle\text{all}| - \mathbb{I} = R_{|\text{all}\rangle}$$

It has been demonstrated that each iteration of Grover's algorithm is a composition of two reflections $R_{|\text{bad}\rangle}$ and $R_{|\text{all}\rangle}$.



The combined effect of each Grover iteration is a counterclockwise rotation of an angle 2θ. Fortunately, the angle θ is easy to find. Since θ is just the angle between $|\text{all}\rangle$ and $|\text{bad}\rangle$, one can use the scalar product to find the angle. It is known that $\cos\theta = \langle\text{all}|\text{bad}\rangle$, so one needs to calculate $\langle\text{all}|\text{bad}\rangle$. From the decomposition of $|\text{all}\rangle$ in terms of $|\text{bad}\rangle$ and $|\text{good}\rangle$, it follows:

$$\theta = \arccos\left(\langle\text{all}|\text{bad}\rangle\right) = \arccos\left(\sqrt{\frac{N-M}{N}}\right)$$

The angle between the state of the register and the $|\text{good}\rangle$ state decreases with each iteration, resulting in a higher probability of measuring a valid result. To calculate this probability, you just need to calculate $|\langle\text{good}|\text{register}\rangle|^2$. The angle between $|\text{good}\rangle$ and $|\text{register}\rangle$ is denoted as $\gamma(k)$, where k is the iteration count:

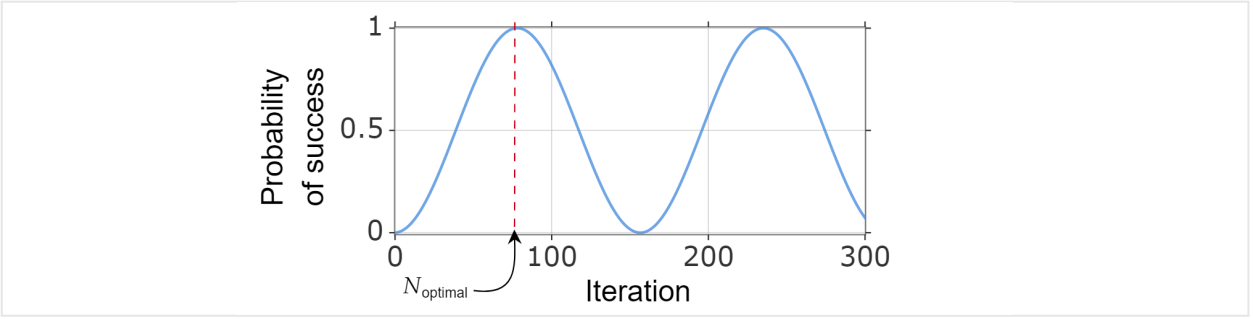
$$\gamma = \frac{\pi}{2} - \theta - 2k\theta = \frac{\pi}{2} - (2k+1)\theta$$

Therefore, the probability of success is:

$$P(\text{success}) = \cos^2(\gamma(k)) = \sin^2\left[(2k+1)\arccos\left(\sqrt{\frac{N-M}{N}}\right)\right]$$

Optimal number of iterations

As the probability of success can be written as a function of the number of iterations, the optimal number of iterations N_{optimal} can be found by computing the smallest positive integer that (approximately) maximizes the success probability function.



It is known that $\sin^2 x$ reaches its first maximum for $x = \frac{\pi}{2}$, so:

$$\frac{\pi}{2} = (2k_{\text{optimal}} + 1) \arccos\left(\sqrt{\frac{N-M}{N}}\right)$$

This gives:

$$k_{\text{optimal}} = \frac{\pi}{4 \arccos\left(\sqrt{1 - M/N}\right)} - 1/2 = \frac{\pi}{4}\sqrt{\frac{N}{M}} - \frac{1}{2} - O\left(\sqrt{\frac{M}{N}}\right)$$

Where in the last step, $\arccos\sqrt{1-x} = \sqrt{x} + O(x^{3/2})$.

Therefore, you can pick $N_{\text{optimal}} = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} - \frac{1}{2} \right\rfloor$.

Complexity analysis

From the previous analysis, $O\left(\sqrt{\frac{N}{M}}\right)$ queries of the oracle O_f are needed to find a valid item. However, can the algorithm be implemented efficiently in terms of time complexity? O_0 is based on computing Boolean operations on n bits and is known to be implementable using $O(n)$ gates. There are also two layers of n Hadamard gates. Both of these components thus require only $O(n)$ gates per iteration. Because $N = 2^n$, it follows that $O(n) = O(log(N))$. Therefore, if $O\left(\sqrt{\frac{N}{M}}\right)$ iterations and $O(log(N))$ gates per iteration are needed, the total time complexity (without taking into account the oracle implementation) is $O\left(\sqrt{\frac{N}{M}} log(N)\right)$.

The overall complexity of the algorithm ultimately depends on the complexity of the implementation of the oracle O_f. If a function evaluation is much more complicated on a quantum computer than on a classical one, the overall algorithm runtime will be longer in the quantum case, even though technically, it uses fewer queries.

References

If you want to continue learning about Grover's algorithm, you can check any of the following sources:

- Original paper by Lov K. Grover
- Quantum search algorithms section of Nielsen, M. A. & Chuang, I. L. (2010). Quantum Computation and Quantum Information.
- Grover's algorithm on Arxiv.org

Related content

- Entanglement in quantum computing
- Vectors and matrices
- T gates and T factories
- Multiple qubits

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Quantum circuit diagram conventions

Article • 01/16/2025

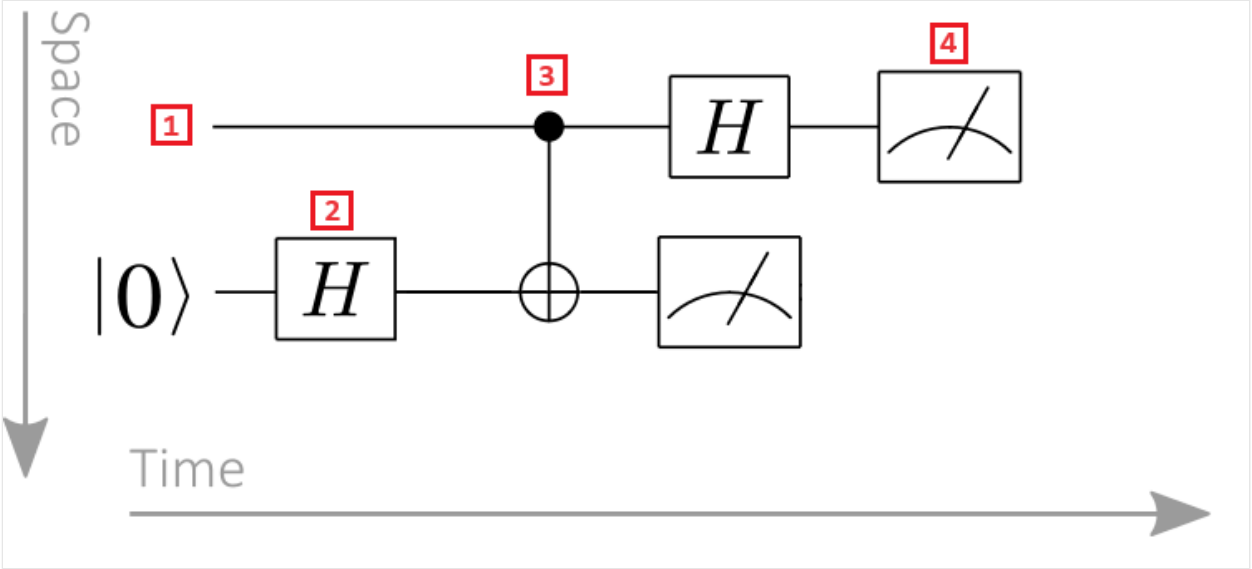
Sometimes quantum algorithms are easier to understand in a circuit diagram than in the equivalent written matrix representation. This article explains how to read quantum circuit diagrams and their conventions.

For more information, see How to visualize quantum circuits diagrams.

Reading quantum circuit diagrams

In a quantum circuit, time flows from left to right. Quantum gates are ordered in chronological order with the left-most gate as the gate first applied to the qubits.

Take the following quantum circuit diagram as an example:



1. **Qubit register**: Qubit registers are displayed as horizontal lines, with each line representing a qubit. The top line is qubit register labeled 0, the second line is qubit register labeled 1, and so on.
2. **Quantum gate**: Quantum operations are represented by *quantum gates*. The term quantum gate is analogous to classical logic gates. Gates acting on one or more qubit registers are denoted as a box. In this example, the symbol represents a Hadamard operation.
3. **Controlled gate**: Controlled gates act on two or more qubits. In this example, the symbol represent a CNOT gate. The black circle represents the control qubit, and the cross within a circle represents the target qubit.
4. **Measurement operation**: The meter symbol represents a measurement operation. The measurement operation takes a qubit register as input and outputs classical

information.

Applying quantum gates

Because time flows from left to right, the left-most gate is applied first For example, the action of the following quantum circuit is the unitary matrix CBA.



> **Note**
>
> Matrix multiplication obeys the opposite convention: the right-most matrix is applied first. In quantum circuit diagrams, however, the left-most gate is applied first. This difference can at times lead to confusion, so it is important to note this significant difference between the linear algebraic notation and quantum circuit diagrams.

Inputs and outputs of quantum circuits

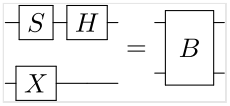
In a quantum circuit diagram, the wires entering a quantum gate represent the qubits that are input to the quantum gate, and the wires exiting the quantum gate represent the qubits that are output from the quantum gate.

The number of inputs of a quantum gate are equal to the number of outputs of a quantum gate. This is because quantum operations are unitary and hence reversible. If a quantum gate had more outputs than inputs, it would not be reversible and hence not unitary, which is a contradiction.

For this reason any box drawn in a circuit diagram must have precisely the same number of wires entering it as exiting it.

Multi-qubit operations

Multi-qubit circuit diagrams follow similar conventions to single-qubit ones. For example, a two-qubit unitary operation B can be defined to be $(HS \otimes X)$, so the equivalent quantum circuit is the following:



You can also view B as having an action on a single two-qubit register rather than two one-qubit registers depending on the context in which the circuit is used.

Perhaps the most useful property of such abstract circuit diagrams is that they allow complicated quantum algorithms to be described at a high level without having to compile them down to fundamental gates. This means that you can get an intuition about the data flow for a large quantum algorithm without needing to understand all the details of how each of the subroutines within the algorithm work.

Controlled gates

Quantum controlled gates are two-qubit gates that apply a single-qubit gate to a target qubit if a control qubit is in a specific state.

For example, consider a quantum controlled gate, denoted $\Lambda(G)$, where a single qubit's value controls the application of the G operation. The controlled gate $\Lambda(G)$ can be understood by looking at the following example of a product state input:

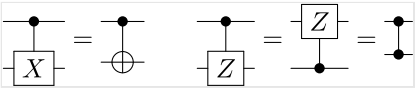
$$\Lambda(G)(\alpha\left|0\right\rangle + \beta\left|1\right\rangle)\left|\psi\right\rangle = \alpha\left|0\right\rangle\left|\psi\right\rangle + \beta\left|1\right\rangle G\left|\psi\right\rangle$$

That is to say, the controlled gate applies G to the register containing ψ if and only if the control qubit takes the value 1. In general, such controlled operations are described in circuit diagrams by the following symbol:



Here the black circle denotes the quantum bit on which the gate is controlled and a vertical wire denotes the unitary that is applied when the control qubit takes the value 1.

For the special cases where $G = X$ and $G = Z$, the following notation is used to describe the controlled version of the gates (note that the controlled-X gate is the CNOT gate):



Q# provides methods to automatically generate the controlled version of an operation, which saves the programmer from having to hand code these operations. An example of this is shown below:

Q#

```
operation PrepareSuperposition(qubit : Qubit) : Unit
is Ctl { // Auto-generate the controlled specialization of the operation
```

```
    H(qubit);
}
```

Classically controlled gates

Quantum gates can also be applied after a measurement, where the result of the measurement acts as a classical control bit.

The following symbol represents a classically controlled gate, where G is applied conditioned on the classical control bit being value 1:



Measurement operator

Measurement operations take a qubit register, measures it, and outputs the result as classical information.

A measurement operation is denoted by a meter symbol and always takes as input a qubit register (denoted by a solid line) and outputs classical information (denoted by a double line). Specifically, the symbol of the measurement operation looks like:



In Q#, the `Measure` operator implements the measurement operation.

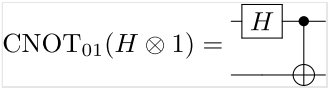
Example: Unitary transformation

Consider the unitary transformation $\mathrm{CNOT}_{01}(H \otimes 1)$. This gate sequence is of fundamental significance to quantum computing because it creates a maximally entangled two-qubit state:

$$\mathrm{CNOT}_{01}(H \otimes 1)\,|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Operations with this or greater complexity are ubiquitous in quantum algorithms and quantum error correction.

The circuit diagram for preparing this maximally entangled quantum state is:



The symbol behind the Hadamard gate represents a CNOT gate, where the black circle indicates the control qubit and the cross within a circle indicates the target qubit. This quantum circuit is depicted as acting on two qubits (or equivalently two registers consisting of one qubit).

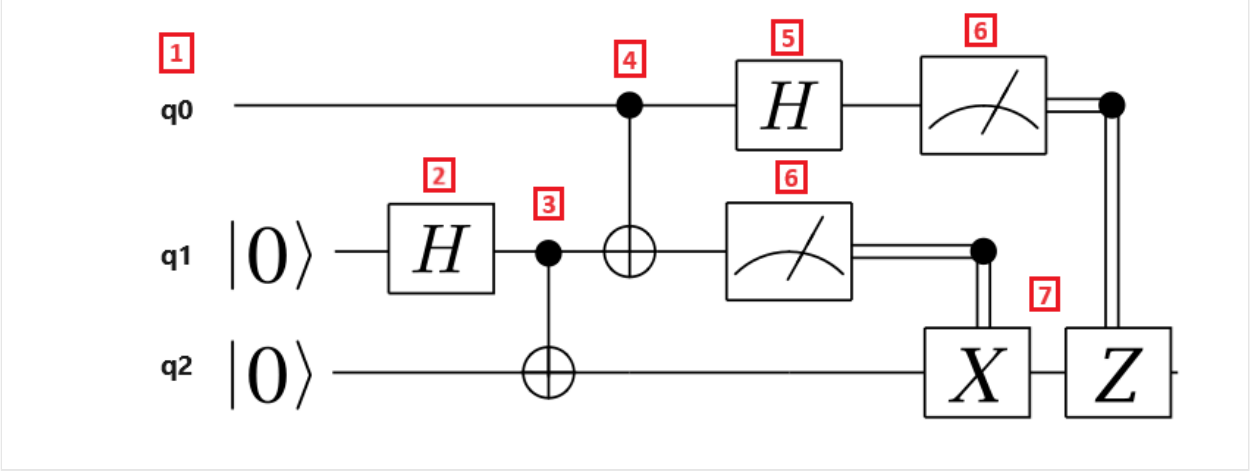
Example: Teleportation circuit diagram

Quantum teleportation is one of the best quantum algorithm for illustrating circuit components.

Quantum teleportation is a protocol that allows a quantum state to be transmitted from one qubit to another, with the help of a shared entangled state between the sender and receiver, and classical communication between them.

For learning purposes, the sender is called Alice, the receiver is called Bob, and the qubit to be teleported is called the message qubit. Alice and Bob hold one qubit each, and Alice has an extra qubit which is the message qubit.

The following circuit diagram illustrates the teleportation protocol:



Let's break down the steps of the teleportation protocol:

1. Qubit register **q0** is the message qubit, qubit register **q1** is Alice's qubit, and qubit register **q2** is Bob's qubit. The message qubit is in an unknown state, and Alice and Bob's qubits are in the $|0\rangle$ state.
2. A **Hadamard gate** is applied to **Alice's qubit**. Because the qubit is in the $|0\rangle$ state, the resulting state is $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.
3. A **CNOT gate** is applied to **Alice and Bob's qubits**. Alice's qubit is the control qubit, and Bob's qubit is the target qubit. The resulting state is $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Alice and Bob now share an entangled state.
4. A **CNOT gate** is applied to the **message qubit and Alice's qubit**. Since Alice's qubit is also entangled with Bob's qubit, the resulting state is a three-qubit entangled

state.

5. A **Hadamard gate** is applied to the **message qubit**.
6. Alice **measures** her two qubits and tells the measurement results to Bob, which isn't reflected in the circuit. The measurement results are two classical bits, which can take the values 00, 01, 10, or 11.
7. Two classically controlled **Pauli gates** X and Z are applied to **Bob's qubit**, depending on the result bit being value 1. The resulting state is the original message qubit state.

Related content

- Quantum entanglement
- Grover's algorithm
- Quantum Intermediate Representation
- Vectors and matrices

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Entanglement and correlations

Article • 09/16/2024

Entanglement is a fundamental concept in quantum mechanics that describes a quantum correlation between quantum systems. When two or more qubits are entangled, the state of one qubit is dependent on the state of the other qubit, even if they're far apart. This quantum correlation is a unique feature of quantum systems that doesn't have a classical counterpart.

This article provides an overview of entanglement, correlations, and explains how to create entanglement using quantum gates.

What is entanglement?

Imagine that you have two qubits, A and B. The qubits are independent from each other, which means that the information about the state of qubit A, whatever it is, belongs only to qubit A. Similarly, the information about the state of qubit B belongs to qubit B. In this case, the qubits aren't entangled, because they aren't sharing any information about their states.

Now imagine that you entangle the qubits. If qubits A and B are entangled, the information about the state of qubit A isn't independent of the state of qubit B. When entangled, information is shared between both qubits, and there's no way to know the state of qubit A or qubit B. You can only describe the state of the global system, not the state of the individual qubits.

Entanglement is a **quantum correlation** between two or more particles. If two particles are entangled, they can't be described independently, but only as a whole system.

Two or more particles can be entangled even if they are separated by large distances. This correlation is stronger than any classical correlation, and it's a key resource for quantum information processing tasks such as quantum teleportation, quantum cryptography, and quantum computing. If you want to learn how to teleport a qubit using entanglement, check out this module in the Azure Quantum training path.

> **Note**
>
> Entanglement is a property of multi-qubit systems, not of single qubits. That is, a single qubit can't be entangled.

Defining entanglement in quantum systems

Imagine two qubits A and B such that the state of the global system $|\phi\rangle$ is:

$$|\phi\rangle = \frac{1}{\sqrt{2}}(|0_A 0_B\rangle + |1_A 1_B\rangle)$$

> **Note**
>
> In Dirac notation, $|0_A 0_B\rangle = |0\rangle_A |0\rangle_B$. The first position corresponds to the first qubit, and the second position corresponds to the second qubit.

The global system $|\phi\rangle$ is in a superposition of the states $|00\rangle$ and $|11\rangle$. But what is the individual state of qubit A? And of qubit B? If you try to describe the state of qubit A without considering the state of qubit B, you fail. Subsystems A and B are entangled and can't be described independently.

If you measure both qubits, only two outcomes are possible: $|00\rangle$ and $|11\rangle$, each with the same probability of $\frac{1}{2}$. The probability of obtaining the states $|01\rangle$ and $|10\rangle$ is zero.

But what happens if you measure only one qubit? When two particles are entangled the measurement outcomes are also correlated. That is, whatever operation happens to the state of one qubit in an entangled pair, also affects to the state of the other qubit.

If you measure only the qubit A and you get the $|0\rangle$ state, this means that the global system collapses to the state $|00\rangle$. This is the only possible outcome, since the probability of measuring $|01\rangle$ is zero. So, without measuring the qubit B you can be sure that the second qubit is also in $|0\rangle$ state. The measurement outcomes are correlated because the qubits are entangled.

The quantum state $|\phi\rangle$ is called a **Bell state**. There are four Bell states:

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

$$|\phi^-\rangle = \frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|11\rangle$$

$$|\psi^+\rangle = \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

$$|\psi^-\rangle = \frac{1}{\sqrt{2}}|01\rangle - \frac{1}{\sqrt{2}}|10\rangle$$

> **Note**
>
> This example uses two qubits, but quantum entanglement isn't limited to two qubits. In general it's possible that multiple-qubit systems share entanglement.

Creating entanglement with quantum operations

You can use quantum operations to create quantum entanglement. One of the most common ways to create entanglement to two qubits in the state $|00\rangle$ is by applying the Hadamard operation H and the controlled-NOT operation $CNOT$ to transform them into the Bell state $|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

The $CNOT$ operation takes two qubits as input, one acts as control qubit and the other is the target qubit. The `CNOT` operation flips the state of the target qubit if, and only if, the state of the control qubit is $|1\rangle$.

Expand table

Input	Output
$\|00\rangle$	$\|00\rangle$
$\|01\rangle$	$\|01\rangle$
$\|10\rangle$	$\|11\rangle$
$\|11\rangle$	$\|10\rangle$

Here's how it works:

1. Take two qubits in the state $|00\rangle$. The first qubit is the control qubit and the second qubit is the target qubit.

2. Create a superposition state only in the control qubit by applying H.

$$H|0_c\rangle = \frac{1}{\sqrt{2}}(|0_c\rangle + |1_c\rangle)$$

> **Note**
>
> The subscripts $_c$ and $_t$ specify the control and target qubits.

3. Apply the $CNOT$ operator to the control qubit, which is in a superposition state, and the target qubit, which is in the state $|0_t\rangle$.

$$CNOT\frac{1}{\sqrt{2}}(|0_c\rangle + |1_c\rangle)|0\rangle_t = CNOT\frac{1}{\sqrt{2}}(|0_c0_t\rangle + ||1_c0_t\rangle) =$$

$$= \frac{1}{\sqrt{2}}(CNOT\,|0_c0_t\rangle + CNOT\,|1_c0_t\rangle) =$$

$$= \frac{1}{\sqrt{2}}(|0_c0_t\rangle + |1_c1_t\rangle)$$

> **Tip**
>
> To learn how to entangle two qubits with Q#, see **Quickstart: Create your first Q# program**.

Separability and quantum entanglement

Entanglement can be seen as the lack of separability: a state is entangled when it isn't separable.

A quantum state is separable if it can be written as a product state of the subsystems. That is, a state $\ket{\phi}_{\text{AB}}$ $is separable if it can be written as a combination of product states of the subsystems, that is \ket{\phi}_{\text{AB}} = \ket{a}_A \otimes \ket{b}_B$.

Entanglement in pure states

A pure quantum state is a single ket vector, such as the state $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

Pure states can't be written as a statistical mixture (or *convex combination*) of other quantum states.

On the Bloch sphere, pure states are represented by a point on the surface of the sphere, whereas mixed states are represented by an interior point.

A **pure state** $\ket{\phi}_{AB}$ $is entangled if it can't be written as a combination of product states of the subsystems, that is \ket{\phi}_{\text{AB}} = \ket{a}_A \otimes \ket{b}_B$.

For example, consider the state

$$|\psi\rangle_{AB} = \frac{1}{2}(|00\rangle + |10\rangle + |01\rangle + |11\rangle)$$

At first, the state $|\psi\rangle_{AB}$ doesn't look like a product state, but if we rewrite the state as

$$|\psi\rangle_{AB} = \frac{1}{\sqrt{2}}(|0\rangle_A + |1\rangle_A) \otimes \frac{1}{\sqrt{2}}(|0\rangle_B + |1\rangle_B) = |+\rangle_A|+\rangle_B$$

the state $|\psi\rangle_{\text{AB}}$ is a product state, therefore it's not entangled.

Entanglement in mixed states

Mixed quantum states are a statistical ensemble of pure states. To describe mixed states is easier to use their density matrix ρ rather than the ket notation.

A mixed state ρ is separable if it can be written as a *convex combination* of product states of the subsystems, such as

$$\rho = \sum_j p_j \rho_j^A \otimes \rho_j^B$$

where $p_j \geq 0, \sum p_j = 1$ and $\rho_j^A \geq 0, \rho_j^B \geq 0$.

For more information, see Density matrices.

A mixed state ρ is entangled if it's not separable, that is, it can't be written as a convex combination of product states.

> **Note**
>
> - If an entangled state ρ is pure, then it contains only quantum correlations.
> - If an entangled state ρ is mixed, then it contains both classical and quantum correlations.

Understanding classical correlations

Classical correlations are due to the lack of knowledge of the state of the system. That is, there's some randomness associated to classical correlation, but it can be eliminated by gaining knowledge.

For example, consider two boxes, each containing one ball. You know that both balls are the same color, either blue or red. If you open one box and find out that the ball inside is blue, then we know that the other ball is blue too. Therefore, they are correlated. However, the uncertainty that we have when opening the box is due to our lack of knowledge, it's not fundamental. The ball was blue before we opened the box. Thus, this is a classical correlation, not a quantum correlation.

The mixed quantum state of the system formed by the two boxes ρ_{boxes} can be written as

$$\rho_{boxes} = \frac{1}{2}(|red\rangle\langle red|_A \otimes |red\rangle\langle red|_B) + \frac{1}{2}(|blue\rangle\langle blue|_A \otimes |blue\rangle\langle blue|_B)$$

Notice that the state ρ_{boxes} is separable, where $p_1 = p_2 = \frac{1}{2}$ then it contains only classical correlations. Another example of a mixed separable state is

$$\rho = \frac{1}{2}(|0\rangle\langle 0|_A \otimes |0\rangle\langle 0|_B) + \frac{1}{2}(|1\rangle\langle 1|_A \otimes |1\rangle\langle 1|_B)$$

Now, consider the following state:

$$\rho = \frac{1}{4}(|00\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 00| + |11\rangle\langle 11|) = |\phi^+\rangle\langle\phi^+|$$

In this case, our knowledge of the state is perfect, we know with maximal certainty that the system AB is in the Bell state $|\phi^+\rangle$ and ρ is a pure state. Therefore, there aren't classical correlations. But if we measure an observable on subsystem A, we obtain a random result which gives us information about the state of the subsystem B. This randomness is fundamental, namely these are quantum correlations.

An example of a quantum state that contains both classical and quantum correlations is

$$\rho = \frac{1}{2}(|\phi^+\rangle\langle\phi^+| + |\phi^-\rangle\langle\phi^-|)$$

> **Note**
>
> A separable state contains only classical correlations.

Related content

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Introduction to quantum error correction

Article • 09/16/2024

This article explains the basics of quantum error correction, the types of quantum errors, and some common quantum error correction codes. It also provides an example of how to correct errors using the three-qubit code.

What is quantum error correction?

Quantum error correction (QEC) is a technique that allows us to protect quantum information from errors. Error correction is especially important in quantum computers, because efficient quantum algorithms make use of large-scale quantum computers, which are sensitive to noise.

The basic principle behind quantum error correction is that the number of bits used to encode a given amount of information is increased. This redundancy allows the code to detect and correct errors.

The error rates for quantum computers are typically higher than classical computer's errors due to the challenges associated with building and operating quantum systems. Noise, decoherence, and imperfections in quantum gates can cause errors in quantum computations. Current quantum computers have error rates in the range of 1% to 0.1%. In other words, this means that on average one out of every 100 to 1000 quantum gate operations results in an error.

Types of quantum errors

There are two fundamental types of quantum errors: bit flips and phase flips.

Bit flip errors occur when a qubit changes from $|0\rangle$ to $|1\rangle$ or vice versa. Bit flip errors are also known as σ_x-errors, because they map the qubit states $\sigma_x |0\rangle = |1\rangle$ and $\sigma_x |1\rangle = |0\rangle$. This error is analogous to a classical bit flip error.

Phase flip errors occur when a qubit changes its phase. They are also known as σ_z-errors, because they map the qubit states $\sigma_z |0\rangle = |0\rangle$ and $\sigma_z |1\rangle = -|1\rangle$. This type of error has no classical analog.

In quantum computing, quantum errors can manifest as bit flips, phase flips, or a combination of both.

How does quantum error correction work?

Quantum error correction codes work by encoding the quantum information into a larger set of qubits, called the *physical qubits*. The joint state of the physical qubits represents a *logical qubit*.

The physical qubits are subject to errors due to decoherence and imperfections in quantum gates. The code is designed so that errors can be detected and corrected by measuring some of the qubits in the code.

For example, imagine you want to send the single-qubit message $|0\rangle$. You could use three physical qubits to encode the message, sending $|000\rangle$, which is known as a *codeword*. This error-correcting code is a *repetition code*, because the message is repeated three times.

Now, imagine that a single bit-flip error occurs during transmission so that what the recipient receives is the state $|010\rangle$. In this scenario, the recipient may be able to infer that the intended message is $|000\rangle$. However, if the message is subject to two bit-flip errors, the recipient may infer an incorrect message. Finally, if all three bits are flipped so that the original message $|000\rangle$ becomes $|111\rangle$, the recipient has no way of knowing an error occurred.

The code distance of a QEC code is the minimum number of errors that change one codeword into another, that is, the number of errors that can't be detected. The code distance d can be defined as

$$d = 2t + 1$$

where t is the number of errors the code can correct. For example, the three-bit code can detect and correct one bit-flip error, so $t = 1$, and thus the code distance is $d = 3$.

Note that repetition codes, such as the three-bit code used in this example, can only correct bit-flip errors, and not phase flip errors. To correct both types of errors, more sophisticated quantum error correction codes are needed.

Types of QEC codes

There are many different types of QEC codes, each with its own properties and advantages. Some common QEC codes are:

- **Repetition code:** The simplest quantum error correction code, where a single qubit is encoded into multiple qubits by repeating it multiple times. The repetition code can correct bit flip errors, but not phase flip errors.

- **Shor code:** The first quantum error correction code, developed by Peter Shor. It encodes one logical qubit into nine physical qubits. Shor code can correct one-bit flip error or one phase flip error, but it can't correct both types of errors at the same time.
- **Steane code:** This is a seven-qubit code that can correct both bit flip and phase flip errors. It has the advantage of being fault-tolerant, meaning that the error correction process itself doesn't introduce extra errors.
- **Surface code:** This is a topological error correction code that uses a two-dimensional lattice of qubits to encode logical qubits. It has a high error correction threshold and is considered one of the most promising techniques for large-scale, fault-tolerant quantum computing. The surface code is used by the Azure Quantum Resource Estimator.
- **Hastings-Haah code:** This quantum error correction code offers better space-time costs than surface codes on Majorana qubits in many regimes. For gate-based instruction sets, the overhead is larger, which makes this approach less efficient than the surface code.

Example: The three-qubit code

The three-qubit error correction code is a simple repetition code that can detect and correct one bit flip error. It encodes a single logical qubit into three physical qubits by repeating the qubit three times.

> **Tip**
>
> Check out the **Q# code sample for the three-qubit code**.

Imagine you want to send an arbitrary single qubit $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$. To avoid errors, you encode the basis states $|0\rangle$ and $|1\rangle$ into a joint state of three qubits. The two logical basis states are $|0_L\rangle = |000\rangle$ and $|1_L\rangle = |111\rangle$.

Therefore, the single qubit $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ is encoded as:

$$|\phi_L\rangle = \alpha|000\rangle + \beta|111\rangle = \alpha|0_L\rangle + \beta|1_L\rangle$$

Let's break down the steps of the three-qubit code.

Preparing the qubits

First, you encode your single qubit $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ into a joint state of three qubits.

Next, you prepare two further qubits in the state $|0\rangle$. So, the global state of all three qubits is $(\alpha|0\rangle + \beta|1\rangle)|0\rangle|0\rangle = \alpha|000\rangle + \beta|100\rangle$.

Lastly, you encode the single qubit into a joint state of three qubits, by applying two CNOT operations. The first CNOT uses the first qubit as control and acts on the second qubit, producing $\alpha|000\rangle + \beta|110\rangle$. The second CNOT uses the first qubit as control and acts on the third qubit. The state of the three qubits is now $\alpha|000\rangle + \beta|111\rangle$.

Sending the qubits

You send all three qubits. Assuming only one-bit flip errors can occur, the received qubits are in one of the following states:

Expand table

State	Error
$\alpha\|000\rangle + \beta\|111\rangle$	No error
$\alpha\|100\rangle + \beta\|011\rangle$	Qubit 1
$\alpha\|010\rangle + \beta\|101\rangle$	Qubit 2
$\alpha\|001\rangle + \beta\|110\rangle$	Qubit 3

Adding auxiliary qubits

First, you introduce two more qubits, prepared in the state $|00\rangle$. This auxiliary pair of qubits are used to extract information of the error without directly measuring or obtaining information about the logical state.

Next, you carry out four CNOT operations: the first two operations use the first and second received qubits as control and act on the first auxiliary qubit, and last two operations use the first and third received qubits as control and act on the second auxiliary bit. The total state of all five qubits is now:

Expand table

State	Error
$(\alpha\|000\rangle + \beta\|111\rangle)\|00\rangle$	No error
$(\alpha\|100\rangle + \beta\|011\rangle)\|11\rangle$	Qubit 1

State	Error
$(\alpha \vert 010\rangle + \beta \vert 101\rangle) \vert 10\rangle$	Qubit 2
$(\alpha \vert 001\rangle + \beta \vert 110\rangle) \vert 01\rangle$	Qubit 3

Retrieving the error syndrome

To retrieve the error information, you measure the two auxiliary qubits in the computational basis states $\vert 0\rangle$ and $\vert 1\rangle$. By doing this, you recover the joint state, which is called the *error syndrome* because it helps diagnose the errors in the received qubits.

Now you know which of the four possible states the three received qubits are in. You can correct the error by applying the correction operation. In this case you're dealing with bit flip errors, so the correction is a σ_x operation applied to one (or none) of the qubits.

For example, if the error syndrome is $\vert 00\rangle$, then the received qubits are in the state $\alpha \vert 000\rangle + \beta \vert 111\rangle$, which is the state you originally sent. If the error syndrome is $\vert 11\rangle$, then the received qubits are in the state $\alpha \vert 100\rangle + b \vert 011\rangle$. There's a bit flip error on the first qubit, which you can correct by applying a σ_x operation to the first qubit.

Expand table

Error syndrome	Collapse state	Correction
$\vert 00\rangle$	$\alpha \vert 000\rangle + \beta \vert 111\rangle$	Do nothing
$\vert 01\rangle$	$\alpha \vert 100\rangle + \beta \vert 011\rangle$	Apply σ_x to qubit 3
$\vert 10\rangle$	$\alpha \vert 010\rangle + \beta \vert 101\rangle$	Apply σ_x to qubit 2
$\vert 11\rangle$	$\alpha \vert 001\rangle + \beta \vert 110\rangle$	Apply σ_x to qubit 1

Extracting the original qubit

Finally, to extract the single qubit you wanted to transmit originally, you apply two CNOT operations: one uses the first qubit as control and acts on the second qubit, and other uses the first qubit as control and acts on the third one.

The state of the first qubit is now $\alpha \vert 0\rangle + \beta \vert 1\rangle$, which is the original qubit you wanted to transmit.

Important

The QEC code doesn't gain any information regarding the coefficients α and β, hence superpositions of the computational state remain intact during correction.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Quantum intermediate representation

Article • 02/14/2025

Quantum Intermediate Representation (QIR) is an intermediate representation which serves as a common interface between quantum programming languages/frameworks and targeted quantum computation platforms. QIR specifies a set of rules for representing quantum programs using a language and hardware agnostic format within the LLVM IR. The QIR is a project developed by the QIR Alliance of which Microsoft is one of its members.

What is an intermediate representation?

A common pattern in classical compilers is to start by compiling the source language into an intermediate representation. An intermediate representation is – as its name indicates – an intermediary step in the conversion of instructions from source code to machine language.

An intermediate representation acts as an abstract representation of a program. All programs, irrespective of the language they are written in, are translated into this intermediate representation by a so-called front-end compiler, while a back-end component is responsible for translating that intermediate representation into a machine representation. The intermediate representation allows thus to decouple source languages from hardware platforms and makes it possible to build a compiler in a modular way, where each new language only requires a new front end to be supported on all platforms for which a back end is available.

The intermediate representation is typically designed to allow many different source languages to be represented. Moreover, at this intermediate level it is also possible to perform some optimization and circuit rearrangement that makes the final implementation more efficient. Once the final target execution platform is known, the intermediate representation can be compiled to actual executable code.

This approach allows many source languages to share a common set of optimizers and executable generators. It also makes it easy to compile a single source language for many different targets. The intermediate representation provides a common platform that can be shared across many sources and targets and allows a great deal of reuse in compiler machinery.

What is Quantum Intermediate Representation?

QIR is an intermediate representation for quantum programs developed by the QIR Alliance, to which Microsoft belongs. It provides a common interface that supports many languages and target platforms for quantum computation. You can think of QIR as a universal mid-layer language that enables communication between high-level languages and machines. While Q# compiles to QIR, QIR isn't specific to Q#: any quantum programming framework can leverage QIR to represent a quantum program. It is hardware-agnostic, which means that it doesn't specify a quantum instruction or gate set, leaving that to the target computing environment.

QIR is based on the popular open-source LLVM classical compiler. LLVM is a collection of modular and reusable compiler and toolchain technologies that has been adapted by a wide set of languages. QIR specifies a set of rules for representing quantum constructs in LLVM, however it doesn't require any extensions or modifications to LLVM.

The fact that LLVM is the underlying toolchain means that QIR is naturally able to process both classical and quantum logic. This feature is essential for hybrid quantum–classical algorithms, which have become increasingly important for applications of quantum computing. In addition, it allows you to leverage compilation and optimization tools from the classical computing industry, and, therefore, to reduce the cost of writing translations.

Many leading quantum computing industries have adopted QIR already. For example, NVIDIA, Oak Ridge National Laboratory, Quantinuum, Quantum Circuits Inc., and Rigetti Computing are building toolchains that leverage QIR.

For more information, see the QIR Specification. If you are interested in compiler tools and projects that use QIR, please take a look at these QIR repositories.

What is the QIR Alliance?

The QIR Alliance is a joint effort to develop a forward-looking quantum intermediate representation with the goal to enable full interoperability within the quantum ecosystem, reduce development effort from all parties, and provide a representation suitable for current and future heterogeneous quantum processors.

Quantum SDKs and languages appear and evolve at a fast pace, along with new quantum processors with unique and distinct capabilities from each other. To provide interoperability between new languages and new hardware capabilities it is imperative for the ecosystem to develop and share an intermediate representation that works with present and future quantum hardware.

With their collective work and partnership, the QIR Alliance aims to:

- Reduce the required development effort for all parties by promoting interoperability between different frameworks and languages.
- Enable the development of shared libraries both for quantum application development, and for quantum compiler development.
- Build on state-of-the-art compiler technology and leverage existing tools, libraries and learnings from high-performance computing.
- Allow for incremental and progressive evolution in how classical and quantum computations can interact at the hardware level.
- Provide the flexibility to easily connect emerging technologies in a way that permits experimentation with distinct and differentiated hardware capabilities.

The QIR Alliance is part of the Linux Foundation's Joint Development Foundation work on open standards. Founding members include Microsoft, as well as Quantinuum, Oak Ridge National Laboratory, Quantum Circuits Inc. and Rigetti Computing.

What does Quantum Intermediate Representation look like?

Since QIR is based on LLVM, QIR looks like LLVM.

For example, consider the following Q# code to generate a Bell pair:

Q#

```
operation CreateBellPair(q1 : Qubit, q2 : Qubit) : Unit {
    H(q1);
    CNOT(q1, q2);
}
```

When compiled to QIR, this becomes:

```
define void @CreateBellPair__body(%Qubit* %q1, %Qubit* %q2) {
entry:
  call void @__quantum__qis__h(%Qubit* %q1)
  call void @__quantum__qis__cnot(%Qubit* %q1, %Qubit* %q2)
  ret void
}
```

In this snippet, you can see a few QIR features:

- Operations in Q# (or any other quantum programming language) are represented by LLVM functions.

- Qubits are represented as pointers to a named opaque structure type called `%Qubit`.

While the QIR for the `CreateBellPair` operation is very simple, QIR inherits all of the capabilities of LLVM to express loops, conditionals, and other complex control flow. QIR also inherits LLVM's ability to express arbitrary classical computation.

For more information, watch Microsoft's developer session from the 2021 Q2B event.

Why is Quantum Intermediate Representation important?

QIR is an essential tool when running quantum algorithms on real hardware. But intermediate representations can play an important role even if you just want to develop algorithms at a more theoretical level.

For example, one application enabled by QIR is to use the Clang compiler, a C language front end for LLVM, to compile QIR into executable machine code for a classical target. This provides an easy path to building a simulator in C or C++ by implementing the quantum instructions, which could simplify the creation of quantum simulators.

Moreover, you could use the intermediate representation to generate code that is later provided as input into a quantum simulator – instead of a real device – which could potentially use a different language than the source code. In this way, you can easily compare and benchmark different languages or simulators using a common framework.

In terms of code optimization, there are optimization steps that can be performed at the intermediate level that can make the overall algorithm implementation more efficient. Investigating this optimization of your input code can help you get a better understanding of where to make algorithms more efficient and how to improve the quantum programming languages.

Another application is to use the standard LLVM "pass" infrastructure to create quantum code optimizers that operate on QIR. The language- and hardware-independent approach of QIR enables reusing those optimizers for different computation languages and computing platforms with almost no effort.

Related content

- Introduction to Q#

- Introduction to resource estimation
- Rigetti provider

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Understanding quantum oracles

Article • 09/16/2024

An oracle, O, is an unexposed operation that is used as input to another algorithm. Often, such operations are defined using a classical function $f : \{0,1\}^n \to \{0,1\}^m$, which takes an n-bit binary input and produces an m-bit binary output. To do so, consider a particular binary input $x = (x_0, x_1, \ldots, x_{n-1})$. You can label qubit states as $\left|\vec{x}\right\rangle = |x_0\rangle \otimes |x_1\rangle \otimes \cdots \otimes |x_{n-1}\rangle$.

You might first attempt to define O so that $O|x\rangle = |f(x)\rangle$, but this method has a couple of problems. First, f might have a different size of input and output ($n \neq m$), such that applying O would change the number of qubits in the register. Second, even if $n = m$, the function might not be invertible: if $f(x) = f(y)$ for some $x \neq y$, then $O|x\rangle = O|y\rangle$ but $O^\dagger O|x\rangle \neq O^\dagger O|y\rangle$. This means you can't construct the adjoint operation $O^\dagger$, and oracles have to have an adjoint defined for them.

Define an oracle by its effect on computational basis states

You can deal with both of these problems by introducing a second register of m qubits to hold the answer. Then, define the effect of the oracle on all computational basis states: for all $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$,

$$O(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus f(x)\rangle.$$

Now $O = O^\dagger$ by construction and you resolved both of the earlier problems.

> **Tip**
>
> To see that $O = O^\dagger$, note that $O^2 = \mathbf{1}$ since $a \oplus b \oplus b = a$ for all $a, b \in 0, 1$. As a result, $O|x\rangle|y \oplus f(x)\rangle = |x\rangle|y \oplus f(x) \oplus f(x)\rangle = |x\rangle|y\rangle$.

Importantly, defining an oracle this way for each computational basis state $|x\rangle|y\rangle$ also defines how O acts for any other state. This behavior follows immediately from the fact that O, like all quantum operations, is linear in the state that it acts on. Consider the Hadamard operation, for example, which is defined $H|0\rangle = |+\rangle$ and $H|1\rangle = |-\rangle$. If you wish to know how H acts on $|+\rangle$, you can use that H is linear,

$$H|+\rangle = \frac{1}{\sqrt{2}} H(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}(H|0\rangle + H|1\rangle)$$
$$= \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle) = \frac{1}{2}(|0\rangle + |1\rangle + |0\rangle - |1\rangle) = |0\rangle.$$

When defining the oracle O, you can similarly use that any state $|\psi\rangle$ on $n+m$ qubits can be written as

$$|\psi\rangle = \sum_{x\in\{0,1\}^n, y\in\{0,1\}^m} \alpha(x,y)|x\rangle|y\rangle.$$

Here, $\alpha : \{0,1\}^n \times \{0,1\}^m \to \mathbb{C}$ represents the coefficients of the state $|\psi\rangle$. Thus,

$$O|\psi\rangle = O \sum_{x\in\{0,1\}^n, y\in\{0,1\}^m} \alpha(x,y)|x\rangle|y\rangle$$
$$= \sum_{x\in\{0,1\}^n, y\in\{0,1\}^m} \alpha(x,y) O|x\rangle|y\rangle$$
$$= \sum_{x\in\{0,1\}^n, y\in\{0,1\}^m} \alpha(x,y)|x\rangle|y \oplus f(x)\rangle.$$

Phase oracles

Alternatively, you can encode f into an oracle O by applying a *phase* based on the input to O. For example, you might define O such that

$$O|x\rangle = (-1)^{f(x)}|x\rangle.$$

If a phase oracle acts on a register initially in a computational basis state $|x\rangle$, then this phase is a global phase and hence not observable. But such an oracle can be a powerful resource if applied to a superposition or as a controlled operation. For example, consider a phase oracle O_f for a single-qubit function f. Then,

$$O_f|+\rangle = O_f(|0\rangle + |1\rangle)/\sqrt{2}$$
$$= ((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle)/\sqrt{2}$$
$$= (-1)^{f(0)}(|0\rangle + (-1)^{f(1)-f(0)}|1\rangle)/\sqrt{2}$$
$$= (-1)^{f(0)} Z^{f(0)-f(1)}|+\rangle.$$

> **Note**
>
> Note that $Z^{-1} = Z^\dagger = Z$ and therefore $Z^{f(0)-f(1)} = Z^{f(1)-f(0)}$.

More generally, both views of oracles can be broadened to represent classical functions, which return real numbers instead of only a single bit.

Choosing the best way to implement an oracle depends heavily on how this oracle is to be used within a given algorithm. For example, Deutsch-Jozsa algorithm relies on the oracle implemented in the first way, while Grover's algorithm relies on the oracle implemented in the second way.

For more information, see the discussion in Gilyén 1711.00465.

Related content

- Grover's algorithm
- Quantum Intermediate Representation
- Vectors and matrices
- T gates and T factories

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

The role of T gates and T factories in quantum computing

This article describes the role of T gates and T factories in fault tolerant quantum computing. Giving a quantum algorithm, the estimation of required resources for running the T gates and T factories becomes crucial to determine the feasibility of the algorithm. The Microsoft Quantum resource estimator calculates the number of T states needed to run the algorithm, the number of physical qubits for a single T factory, and the runtime of the T factory.

Universal set of quantum gates

According to DiVincenzo's criteria a scalable quantum computer must be able to implement a **universal set of quantum gates**. A *universal set* contains all the gates needed to perform any quantum computation, that is, any computation must decompose back into a finite sequence of universal gates. At a minimum, a quantum computer must be able to move single qubits to any position on the Bloch Sphere (using single-qubit gates), as well as introduce entanglement in the system, which requires a multi-qubit gate.

There are only four functions that map one bit to one bit on a classical computer. In contrast, there are an infinite number of unitary transformations on a single qubit on a quantum computer. Therefore, no finite set of primitive quantum operations or gates, can exactly replicate the infinite set of unitary transformations allowed in quantum computing. This means, unlike classical computing, it is impossible for a quantum computer to implement every possible quantum program exactly using a finite number of gates. Thus quantum computers cannot be universal in the same sense of classical computers. As a result, when we say that a set of gates is *universal* for quantum computing we actually mean something slightly weaker than we mean with classical computing.

For universality, it's required that a quantum computer only *approximates* every unitary matrix within a finite error using a finite length gate sequence.

In other words, a set of gates is a universal gate set if any unitary transformation can be approximately written as a product of gates from this set. It's required that for any prescribed error bound, there exist gates $G_1, G_2, \ldots, G_N$ from the gate set such that

$$G_N G_{N-1} \cdots G_2 G_1 \approx U.$$

Because the convention for matrix multiplication is to multiply from right to left the first gate operation in this sequence, G_N, is actually the last one applied to the quantum state vector. More formally, such a gate set is said to be universal if for every error tolerance $\epsilon > 0$ there exists $G_1, \ldots, G_N$ such that the distance between $G_N \ldots G_1$ and U is at most ϵ. Ideally the value of N needed to reach this distance of ϵ should scale poly-logarithmically with $1/\epsilon$.

For example, the set formed by Hadamard, CNOT and T gates is a universal set, from which any quantum computation (on any number of qubits) can be generated. The Hadamard and the T gate set generates any single-qubit gate:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \qquad T = \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}.$$

In a quantum computer, quantum gates can be classified into two categories: Clifford gates and non-Clifford gates, in this case, the T gate. Quantum programs made from only Clifford gates can be simulated efficiently using a classical computer, and therefore, non-Clifford gates are required to obtain quantum advantage. In many quantum error correction (QEC) schemes the so-called Clifford gates are easy to implement, that is they require very few resources in terms of operations and qubits to implement fault tolerantly, whereas non-Clifford gates are quite costly when requiring fault tolerance. In a universal quantum gate set, the T gate is commonly used as the non-Clifford gate.

The standard set of single-qubit Clifford gates, included by default in Q#, include

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \qquad S = \begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix} = T^2, \qquad X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} = HT^4H,$$

$$Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} = T^2HT^4HT^6, \qquad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = T^4.$$

Together with the non-Clifford gate (the T gate), these operations can be composed to approximate any unitary transformation on a single qubit.

T factories in the Microsoft Quantum resource estimator

The non-Clifford T gate preparation is crucial because the other quantum gates are not sufficient for universal quantum computation. To implement non-Clifford operations for practical-scale algorithms, low error rate T gates (or T states) is required. However, they can be difficult to directly implement on logical qubits, and can also be difficult for some physical qubits.

In a fault tolerant quantum computer, the required low error rate T states are produced using a T state distillation factory, or T factory for short. These T factories typically involve a sequence of rounds of distillation, where each round takes in many noisy T states encoded in a smaller distance code, processes them using a distillation unit, and outputs fewer less noisy T states encoded in a larger distance code, with the number of rounds, distillation units, and distances all being parameters which can be varied. This procedure is iterated, where the output T states of one round are fed into the next round as inputs.

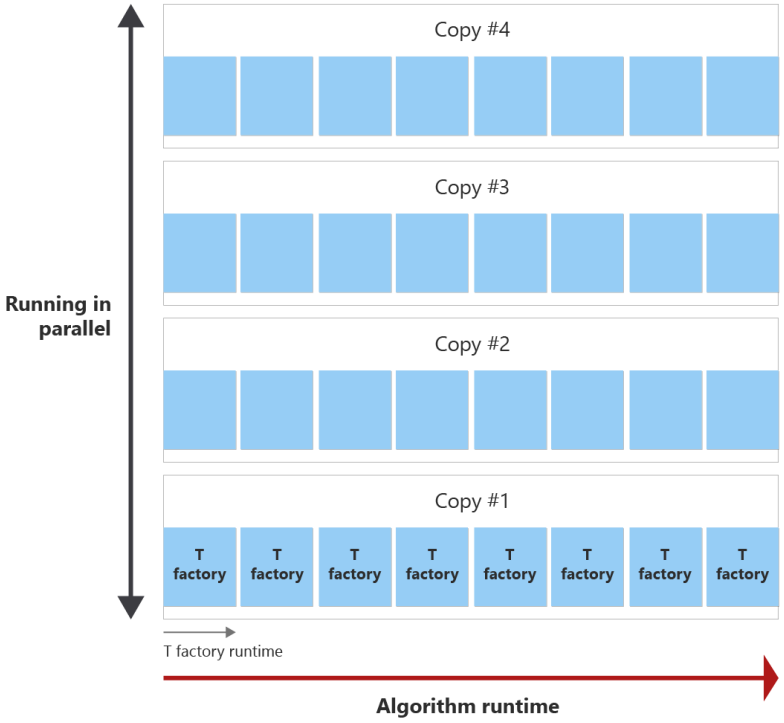
Based on the duration of the T factory, the Microsoft Quantum resource estimator determines how often a T factory can be invoked before it exceeds the total runtime of the algorithm, and thus how many T states can be produced during the algorithm runtime. Usually, more T states are required than what can be produced within the invocations of a single T factory during the algorithm runtime. In order to produce more T states, the resource estimator uses copies of the T factories.

T factory physical estimation

The resource estimator calculates the total number of T states needed to run the algorithm, and the number of physical qubits for a single T factory and its runtime.

The goal is to produce all T states within the algorithm runtime with as few T factory copies as possible. The following diagram shows an example of the runtime of the algorithm and the runtime of one T factory. You can see that the runtime of the T factory is shorter than the runtime of the algorithm. In this example, one T factory can distill one T state. Two questions arise:

- How often can the T factory be invoked before the end of the algorithm?
- How many copies of the T factory distillation round are necessary to create the number of T states required during the algorithm's runtime?



Before the end of the algorithm, the T factory can be invoked eight times, which is called a distillation round. For example, if you need 30 T states, a single T factory is invoked eight times during runtime of the algorithm and thus, it creates eight T states. Then you need four copies of the T factory distillation round running in parallel to distill the 30 T states needed.

> **Note**
>
> Note that T factory copies and T factory invocations aren't the same.

The T state distillation factories are implemented in a sequence of rounds, where each round consists of a set of copies of distillation units running in parallel. The resource estimator calculates how many physical qubits are needed to run one T factory and for how long the T factory runs, among other required parameters.

You can only do full invocations of a T factory. Therefore, there may be situations in which the accumulated runtime of all T factory invocations is less than the algorithm runtime. Since qubits are reused by different rounds, the number of physical qubits for one T factory is the maximum number of physical qubits used for one round. The runtime of the T factory is the sum of the runtime in all rounds.

> **Note**
>
> If the physical T gate error rate is lower than the required logical T state error rate, the resource estimator cannot perform a good resource estimation. When you submit a resource estimation job, you may encounter that the T factory cannot be found because the required logical T state error rate is either too low or too high.

For more information, see Appendix C of Assessing requirements to scale to practical quantum advantage.

Related content

- Quantum circuits
- Quantum oracles
- Grover's algorithm
- Quantum Intermediate Representation

Last updated on 01/29/2026

Hybrid quantum computing concepts

Article • 09/16/2024

All quantum computing is hybrid in some sense. The earliest quantum systems used classical processes to send instructions to the quantum computer, and receive and process the resulting data. The most advanced hybrid architectures bring a tighter and richer integration between classical and quantum computation, speeding up the execution time and opening the door to new generation of algorithms. For example, classical computing is often faster and more efficient than quantum computing for certain tasks, such as data processing and analysis. However, quantum computing performs better for certain types of optimization and simulation problems.

Classical and quantum registers

In quantum computing, even though the classical and quantum processors are tightly integrated, they are still separate physical entities, and hybrid quantum computing programs can take advantage of each of their capabilities.

A classical register uses the familiar silicon chip-based architecture and is best suited for operations such as sending instructions to the quantum processor, capturing measurement results, and using those results to determine the next set of instructions.

A quantum register is a system comprised of multiple qubits. Quantum computers excel at performing complex calculations by manipulating their qubits within a quantum register.

Mid-circuit measurement

Mid-circuit measurement is the process of performing quantum state measurements at various points during the execution of the program, rather than only at the end. These measurements are used to obtain information about the intermediate states of the system and are used by the classical code to make real-time decisions about the flow of the program. Mid-circuit measurements also act as error correction "sanity checks" by verifying the state of the circuit at a certain point before moving on, and are closely related to qubit reuse.

Qubit reuse

Today's quantum computers are able to support an increasing number of qubits. However, we're still far from being able to support the millions of qubits needed to run fully fault-tolerant computations. For more information about the scale of quantum computers needed, see the Introduction to resource estimation. Additionally, it's desirable to use as few qubits as possible.

Qubit reuse is the practice of designing circuits to use the same qubit multiple times in a quantum computation to minimize the total qubits needed to run your program. For example, after performing a mid-circuit measurement and processing the result, that qubit can be reset and reused for another computation instead of allocating a new qubit. There are various techniques for re-using qubits in quantum computing, such as quantum teleportation, quantum error correction, and measurement-based quantum computing.

Error mitigation

To make current quantum hardware more robust against error and noise, *logical qubits* can be used. Logical qubits are created using multiple physical qubits to encode and protect quantum information. However, because multiple qubits must be used to create one logical qubit, the overall number of qubits that can be used for computations is reduced. As the ability of hardware to support more physical qubits increases, so will the fault tolerance capabilities.

In addition to using logical qubits, errors in quantum computations can be mitigated by using techniques such as multiple measurements, designing algorithms that reduce the number of operations needed, or adjusting parameters on quantum gates to reduce the impact of noise.

Error correction and fault tolerance

Error correction and fault tolerance are critical aspects of quantum computing, as qubits are more prone to errors than classical bits due to the delicate nature of quantum states, and highly fault tolerant systems are needed to achieve the full benefits of distributed hybrid quantum computing. In classical computing, errors can be corrected by adding redundancy to the computation and using error-correction codes. However, traditional error-correction techniques aren't directly applicable to quantum computing, as they rely on being able to repeat the computation multiple times, which isn't possible in quantum computing due to the no-cloning theorem.

Floquet codes are a new class of error correction codes that respond to noise and errors dynamically, instead of traditional correction codes that protect against static errors. For

more information, see Error correction with Floquet codes.

Hybrid algorithms

- **Variational Quantum Eigensolver (VQE)** - a quantum algorithm for quantum chemistry, quantum simulations and optimization problems and is used to find the ground state of a given physical system. The classical computer is used to define quantum circuits with certain parameters. After the quantum state is measured, the classical computer evaluates how to improve the parameters and then resubmits the circuits. VQEs are usually long-running programs that can benefit from the tighter integration of hybrid quantum computing.
- **Quantum Approximate Optimization Algorithm (QAOA)** - also a variational quantum algorithm, it is used for finding approximate solutions to combinatorial optimization problems - problems where the number of possible solutions grows extremely large with the size of the problem. It is an active field of research to identify its applicability for applications such as air traffic control, shipping or delivery routes, or financial optimizations.
- **Iterative phase estimation** - is another method of performing *quantum phase estimation*, which estimates the phase of a unitary operator, and is used in many other quantum algorithms. Both methods use a series of rotation gates to determine the phase, however iterative phase estimation utilizes the classical register to store information and calculations about gate measurements. This reduces the number of qubits required and minimizes noise and errors.

For more information about quantum algorithms, visit the Quantum Algorithm Zoo.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Contributing to the Microsoft Quantum Development Kit

Thanks for being a part of the Microsoft Quantum community, we're excited for your contributions!

- If you'd like to contribute code to the Microsoft Quantum Development Kit or report a bug, see Contributing to Q#.
- If you'd like to contribute documentation to the Microsoft Quantum Development Kit, see How to contribute documentation.

Last updated on 01/29/2026

Further reading: Quantum computing learning resources

Article • 02/21/2025

This article compiles some of the most popular resources that you may find useful when learning quantum computing.

Microsoft quantum computing resources

Learn how to develop and apply quantum computing solutions with the Quantum Development Kit (QDK) and Azure Quantum services.

- Azure Quantum training path: an interactive, free, hands-on learning path. In these modules, you learn about quantum computing and how to develop quantum solutions using Q# and the Azure Quantum Development Kit.
- Quantum Katas: a collection of self-paced Q# quantum programming tutorials.
- Azure Quantum videos: a playlist with videos of Azure Quantum announcements, demos, and discussions from *Quantum Innovator Series*.
- Q# code samples: start building your first quantum solution with this collection of ready-to-use code samples.
- Q# blog: a blog written by developers for developers. You can read about the latest QDK and Q# insights, and find out about quantum challenges and hackathons announcements.
- Research publications: read about the latest advancement in quantum hardware and algorithms developed by Microsoft researchers.

These and more quantum computing resources can be found in the Microsoft's quantum learning page.

Q# community made content

The following resources are created and developed by the quantum community who is excited about quantum programming.

Books created by the community

- *Introduction to Quantum Computing with Q# and QDK*, Filip Wojcieszyn, Springer, 2022.

- *Q# Pocket Guide*, Mariia Mykhailova, O'Reilly, 2021. Q# samples updated are here.
- *Introducing Microsoft Quantum Computing for Developers*, Johnny Hooyberghs, Springer, 2021.
- *Learn Quantum Computing with Python and Q# - A hands-on approach*, S. Kaiser and C. Granade. Manning Publications, 2021.

Blogs created by the community

- Awesome qsharp: an open source list of Q# code and resources.
- Q# Community: a GitHub space for community-driven projects.

Forums and communities for quantum developers

Quantum computing StackExchange: an online community for quantum developers to learn, and share their knowledge.

Quantum computing courses

Check out the following quantum computing learning courses.

- Quantum Computing with Microsoft QDK: a series of liveProjects that help you to learn quantum software development by creating end-to-end projects. You explore the full potential of quantum for cryptography, data transmission, data reconstruction, and more.

- Quantum Computing - Brilliant Course: learn to build quantum algorithms from the ground up with a quantum computer simulated in your browser in this course, created in collaboration with quantum researchers and practitioners from Microsoft, X, and Caltech's Institute for Quantum Information and Matter (IQIM).
- Quantum Computing through Comics - HackadayU classes: learn about quantum computing concepts and algorithm programming through classroom discussion and intuitive comics.

Bibliography

The following bibliography is a collection of publications that cover a wide range of quantum computing topics.

Quantum computing for beginners

If you're a quantum enthusiast and want to start learning the theory behind quantum computing, the following resources offer topics such as quantum physics, computer science, and linear algebra.

- Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information. Quantum Computation and Quantum Information.* UK: Cambridge University Press, 2010.
- Kaye, P., Laflamme, R., & Mosca, M. *An introduction to quantum computing.* Oxford University Press, 2007.
- Rieffel, E. G., & Polak, W. H. *Quantum computing: A gentle introduction.* MIT Press, 2011.

Different types of qubits

- Sergey Bravyi, Oliver Dial, Jay M. Gambetta, Dario Gil, and Zaira Nazario. *The future of quantum computing with superconducting qubits*, 2022.
- Microsoft Quantum. *InAs-Al Hybrid Devices Passing the Topological Gap Protocol*, arXiv:2207.02472 [cond-mat.mes-hall], (2022).
- M Saffman. *Quantum computing with atomic qubits and rydberg interactions: progress and challenges*, Journal of Physics B: Atomic, Molecular, and Optical Physics, 49(20):202001, (2016).
- J. I. Cirac and P. Zoller. *Quantum computations with cold trapped ions*, Phys. Rev. Lett., 74:4091–4094 (1995).

Quantum Error Correction

- Michael Beverland, Vadym Kliuchnikov, and Eddie Schoute. *Surface code compilation via edge-disjoint paths*, PRX Quantum, 3:020342, (2022) .
- Adam Paetznick, Christina Knapp, Nicolas Delfosse, Bela Bauer, Jeongwan Haah, Matthew B. Hastings, and Marcus P. da Silva. *Performance of planar floquet codes with majorana-based qubits*, 2022.
- Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. *Surface codes: Towards practical large-scale quantum computation*, Phys. Rev. A, 86:032324, (2012).
- Daniel Gottesman. *An introduction to quantum error correction and fault-tolerant quantum computation*. In Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics, volume 68, pages 13–58, (2010).

Resource Estimation

- M. E. Beverland, P. Murali,1 M. Troyer, K. M. Svore, T. Hoefler, V. Kliuchnikov, G. H. Low, M. Soeken, A. Sundaram, and A. Vaschillo. *Assessing requirements to scale to practical quantum advantage*, arXiv:2211.07629v1, 2022.
- Isaac H. Kim, Ye-Hua Liu, Sam Pallister, William Pol, Sam Roberts, and Eunseok Lee. Fault-tolerant resource estimate for quantum chemical simulations: Case study on li-ion battery electrolyte molecules. Phys. Rev. Research, 4:023019, Apr 2022.
- Giulia Meuli, Mathias Soeken, Martin Roetteler, and Thomas H¨aner. *Enabling accuracy-aware quantum compilers using symbolic resource estimation*, Proc. ACM Program. Lang., 4(OOPSLA), 2020.

Fault-tolerant quantum computing

- Hector Bombin, Chris Dawson, Ryan V. Mishmash, Naomi Nickerson, Fernando Pastawski, and Sam Roberts. *Logical blocks for fault-tolerant topological quantum computation*, 2021.
- Antonio D. C´orcoles, Abhinav Kandala, Ali Javadi-Abhari, Douglas T. McClure, Andrew W. Cross, Kristan Temme, Paul D. Nation, Matthias Steffen, and Jay M. Gambetta. *Challenges and opportunities of near-term quantum computing systems*, Proceedings of the IEEE, 108(8):1338–1352 (2020).
- Michael Edward Beverland. *Toward realizable quantum computers*, PhD thesis, California Institute of Technology, 2016.
- Peter W Shor. *Fault-tolerant quantum computation*. In Proceedings of 37th conference on foundations of computer science, pages 56–65. IEEE (1996).

Quantum chemistry

- J. Tilly, Hongxiang Chen, Shuxiang Cao, D. Picozzi, K. Setia, Ying Li, E. Grant, L. Wossnig, I. Rungger, G. Booth, J. Tennyson. *The Variational Quantum Eigensolver: a review of methods and best practices*, arXiv:2111.05176v3 [quant-ph], 2022.
- V. von Burg, Guang Hao Low, T. Häner, D.S. Steiger, M. Reiher, M. Roetteler, and M. Troyer. *Quantum computing enhanced computational catalysis*. Phys. Rev. Research 3, 033055 (2021).
- Bela Bauer, Sergey Bravyi, Mario Motta, and Garnet Kin-Lic Chan. *Quantum algorithms for quantum chemistry and quantum materials science*, Chemical Reviews, 120(22):12685–12717 (2020).

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A