

Q# 0.10 Language Quick Reference

Primitive Types	
64-bit integers	Int
Double-precision floats	Double
Booleans	Bool e.g.: true or false
Qubits	Qubit
Pauli basis	Pauli e.g.: PauliI, PauliX, PauliY, or PauliZ
Measurement results	Result e.g.: Zero or One
Sequences of integers	Range e.g.: 1..10 or 5..-1..0
Strings	String e.g.: "Hello Quantum!"
"Return no information" type	Unit e.g.: ()

Derived Types	
Arrays	<i>elementType</i> []
Tuples	(<i>type0</i> , <i>type1</i> , ...) e.g.: (Int, Qubit)
Functions	<i>input</i> -> <i>output</i> e.g.: ArcCos : (Double) -> Double
Operations	<i>input</i> => <i>output</i> is <i>variants</i> e.g.: H : (Qubit => Unit is Adj)

User-Defined Types	
Declare UDT with anonymous items	<code>newtype Name = (Type, Type);</code> e.g.: <code>newtype Pair = (Int, Int);</code>
Define UDT literal	<code>Name(baseTupleLiteral)</code> e.g.: <code>let origin = Pair(0, 0);</code>
Unwrap operator ! (convert UDT to underlying type)	<code>VarName!</code> e.g.: <code>let originTuple = origin!;</code> (now <code>originTuple = (0, 0)</code>)
Declare UDT with named items	<code>newtype Name = (Name1: Type, Name2: Type);</code> e.g.: <code>newtype Complex = (Re : Double, Im : Double);</code>
Accessing named items of UDTs	<code>VarName::ItemName</code> e.g.: <code>complexVariable::Re</code>
Update-and-reassign for named UDT items	<code>set VarName w/= ItemName <- val;</code> e.g.: <code>mutable p = Complex(0., 0.);</code> <code>set p w/= Re <- 1.0;</code>

Symbols and Variables	
Declare immutable symbol	<code>let varName = value</code>
Declare mutable symbol (variable)	<code>mutable varName = initialValue</code>
Update mutable symbol (variable)	<code>set varName = newValue</code>
Apply-and-reassign	<code>set varName operator= expression</code> e.g.: <code>set counter += 1;</code>

Functions and Operations	
Define function (classical routine)	<code>function Name(in0 : type0, ...)</code> : <i>returnType</i> { // function body }
Call function	<code>Name(parameters)</code> e.g.: <code>let two = Sqrt(4.0);</code>
Define operation (quantum routine) with explicitly specified body, controlled and adjoint variants	<code>operation Name(in0 : type0, ...)</code> : <i>returnType</i> { body { ... } adjoint { ... } controlled { ... } adjoint controlled { ... } }
Define operation with automatically generated adjoint and controlled variants	<code>operation Name(in0 : type0, ...)</code> : <i>returnType</i> is Adj + Ctl { ... }
Call operation	<code>Name(parameters)</code> e.g.: <code>Ry(0.5 * PI(), q);</code>
Call adjoint operation	<code>Adjoint Name(parameters)</code> e.g.: <code>Adjoint Ry(0.5 * PI(), q);</code>
Call controlled operation	<code>Controlled Name(controlQubits, parameters)</code> e.g.: <code>Controlled Ry(controls, (0.5 * PI(), target));</code>

Control Flow	
Iterate over a range of numbers	<code>for (index in range) {</code> // Use integer <i>index</i> ... }
While loop (within functions)	<code>while (condition) {</code> ... }
Iterate over an array	<code>for (val in array) {</code> // Use value <i>val</i> ... }
Repeat-until-success loop	<code>repeat { ... }</code> until (<i>condition</i>) fixup { ... }
Conditional statement	<code>if (cond1) { ... }</code> <code>elif (cond2) { ... }</code> <code>else { ... }</code>
Ternary operator	<code>condition ? caseTrue caseFalse</code>
Return a value	<code>return value</code>
Stop with an error	<code>fail "Error message"</code>
Conjugations (ABA^\dagger pattern)	<code>within { ... }</code> <code>apply { ... }</code>

Arrays	
Allocate array	<code>mutable name = new Type[Length]</code> e.g.: <code>mutable b = new Bool[2];</code>
Get array length	<code>Length(name)</code>
Access k-th element	<code>name[k]</code> NB: indices are 0-based
Assign k-th element (copy-and-update)	<code>set name w/= k <- value</code> e.g.: <code>set b w/= 0 <- true;</code>
Array literal	<code>[value0, value1, ...]</code> e.g.: <code>let b = [true, false, true];</code>
Array concatenation	<code>array1 + array2</code> e.g.: <code>let t = [1, 2, 3] + [4, 5];</code>
Slicing (subarray)	<code>name[sliceRange]</code> e.g.: <code>if t = [1, 2, 3, 4, 5], then</code> <code>t[1 .. 3]</code> is <code>[2, 3, 4]</code> <code>t[3 ...]</code> is <code>[4, 5]</code> <code>t[... 1]</code> is <code>[1, 2]</code> <code>t[0 .. 2 ...]</code> is <code>[1, 3, 5]</code> <code>t[...-1...]</code> is <code>[5, 4, 3, 2, 1]</code>

Debugging (classical)	
Print a string	<code>Message("Hello Quantum!")</code>
Print an interpolated string	<code>Message(\$"Value = {val}")</code>

Resources

Documentation	
Quantum Development Kit	https://docs.microsoft.com/quantum
Q# Language Reference	https://docs.microsoft.com/quantum/language
Q# Libraries Reference	https://docs.microsoft.com/qsharp/api

Q# Code Repositories	
QDK Samples	https://github.com/microsoft/quantum
QDK Libraries	https://github.com/microsoft/QuantumLibraries
Quantum Katas (tutorials)	https://github.com/microsoft/QuantumKatas
Q# compiler and extensions	https://github.com/microsoft/qsharp-compiler
Simulation framework	https://github.com/microsoft/qsharp-runtime
Jupyter kernel and Python host	https://github.com/microsoft/iqsharp
Source code for the documentation	https://github.com/MicrosoftDocs/quantum-docs-pr

Qubit Allocation	
Allocate a register of N qubits	<pre>using (reg = Qubit[N]) { // Qubits in <i>reg</i> start in $0\rangle$. ... // Qubits must be returned to $0\rangle$. }</pre>
Allocate one qubit	<pre>using (one = Qubit()) { ... }</pre>
Allocate a mix of qubit registers and individual qubits	<pre>using ((x, y, ...) = (Qubit[N], Qubit(), ...)) { ... }</pre>

Debugging (quantum)	
Print amplitudes of wave function	DumpMachine("dump.txt")
Assert that a qubit is in $ 0\rangle$ or $ 1\rangle$ state	<pre>AssertQubit(Zero, zeroQubit) AssertQubit(One, oneQubit)</pre>

Basic Quantum Gates	
Pauli gates	<pre>X(qubit) : 0> ↦ 1>, 1> ↦ 0> Y(qubit) : 0> ↦ i 1>, 1> ↦ -i 0> Z(qubit) : 0> ↦ 0>, 1> ↦ - 1></pre>
Hadamard	<pre>H(qubit) : 0> ↦ +> = $\frac{1}{\sqrt{2}}(0\rangle + 1\rangle)$, 1> ↦ -> = $\frac{1}{\sqrt{2}}(0\rangle - 1\rangle)$</pre>
Controlled-NOT	<pre>CNOT(controlQubit, targetQubit) 00> ↦ 00>, 01> ↦ 01>, 10> ↦ 11>, 11> ↦ 10></pre>
Apply several gates (Bell pair example)	<pre>H(qubit1); CNOT(qubit1, qubit2);</pre>

Measurements	
Measure qubit in Pauli Z basis	<pre>M(oneQubit) yields a Result (Zero or One)</pre>
Reset qubit to $ 0\rangle$	Reset(oneQubit)
Reset an array of qubits to $ 0..0\rangle$	ResetAll(register)

Working with Q# from command line

Command Line Basics	
Change directory	cd <i>dirname</i>
Go to home	cd ~
Go up one directory	cd ..
Make new directory	mkdir <i>dirname</i>
Open current directory in VS Code	code .

Working with Q# Projects	
Create new project	dotnet new console -lang Q# --output <i>project-dir</i>
Change directory to project directory	cd <i>project-dir</i>
Build project	dotnet build
Run all unit tests	dotnet test