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	*elementType*`[]`
Tuples	`(`*type0*`, `*type1*`, ...)` e.g.: `(Int, Qubit)`
Functions	*input* `->` *output* e.g.: `ArcCos : (Double) -> Double`
Operations	*input* `=>` *output* `is` *variants* e.g.: `H : (Qubit => Unit is Adj)`

User-Defined Types

Declare UDT with anonymous items	`newtype` *Name* `= (Type, Type);` e.g.: `newtype` *Pair* `= (Int, Int);`
Define UDT literal	*Name*`(`*baseTupleLiteral*`)` e.g.: `let origin = Pair(0, 0);`
Unwrap operator ! (convert UDT to underlying type)	*VarName*`!` e.g.: `let originTuple = origin!;` (now `originTuple = (0, 0)`)

Declare UDT with named items	`newtype` *Name* `=`
	`(`*Name1*`: Type, `*Name2*`: Type);`
	e.g.: `newtype Complex =`
	`(Re : Double, Im : Double);`

Accessing named items of UDTs	*VarName*`::`*ItemName* e.g.: `complexVariable::Re`

Update-and-reassign for named UDT items	`set` *VarName* `w/=` *ItemName* `<-` *val*`;`
	e.g.: `mutable p = Complex(0., 0.);`
	`set p w/= Re <- 1.0;`

Symbols and Variables

Declare immutable symbol	`let` *varName* `=` *value*
Declare mutable symbol (variable)	`mutable` *varName* `=` *initialValue*
Update mutable symbol (variable)	`set` *varName* `=` *newValue*
Apply-and-reassign	`set` *varName operator*`=` *expression* e.g.: `set counter += 1;`

Functions and Operations

Define function (classical routine)	`function` *Name*`(`*in0* `:` *type0*`, ...)`
	`:` *returnType* `{`
	`//` *function body*
	`}`

Call function	*Name*`(`*parameters*`)` e.g.: `let two = Sqrt(4.0);`

Define operation (quantum routine) with explicitly specified body, controlled and adjoint variants	`operation` *Name*`(`*in0* `:` *type0*`, ...)`
	`:` *returnType* `{`
	`body { ... }`
	`adjoint { ... }`
	`controlled { ... }`
	`adjoint controlled { ... }`
	`}`

Define operation with automatically generated adjoint and controlled variants	`operation` *Name*`(`*in0* `:` *type0*`, ...)`
	`:` *returnType* `is Adj + Ctl {`
	`...`
	`}`

Call operation	*Name*`(`*parameters*`)` e.g.: `Ry(0.5 * PI(), q);`
Call adjoint operation	`Adjoint` *Name*`(`*parameters*`)` e.g.: `Adjoint Ry(0.5 * PI(), q);`

Call controlled operation	`Controlled` *Name*`(`*controlQubits*`,`
	parameters`)`
	e.g.: `Controlled Ry(controls,`
	`(0.5 * PI(), target));`

Control Flow

Iterate over a range of numbers	`for (`*index* `in` *range*`) {`
	`// Use integer` *index*
	`...`
	`}`
	e.g.: `for (i in 0..N-1) { ... }`

While loop (within functions)	`while (`*condition*`) {`
	`...`
	`}`

Iterate over an array	`for (`*val* `in` *array*`) {`
	`// Use value` *val*
	`...`
	`}`
	e.g.: `for (q in register) { ... }`

Repeat-until-success loop	`repeat { ... }`
	`until (`*condition*`)`
	`fixup { ... }`

Conditional statement	`if (`*cond1*`) { ... }`
	`elif (`*cond2*`) { ... }`
	`else { ... }`

Ternary operator	*condition* `?` *caseTrue* `|` *caseFalse*
Return a value	`return` *value*
Stop with an error	`fail "`*Error message*`"`
Conjugations ($ABA^\dagger$ pattern)	`within { ... }` `apply { ... }`

Arrays

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

Slicing (subarray)	*name*`[`*sliceRange*`]`
	e.g.: if `t = [1, 2, 3, 4, 5]`, then
	`t[1 .. 3]` is `[2, 3, 4]`
	`t[3 ...]` is `[4, 5]`
	`t[... 1]` is `[1, 2]`
	`t[0 .. 2 ...]` is `[1, 3, 5]`
	`t[...-1...]` is `[5, 4, 3, 2, 1]`

Debugging (classical)

Print a string	`Message("Hello Quantum!")`
Print an interpolated string	`Message($"Value = {`*val*`}")`

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	`using (reg = Qubit[N]) {`
	`    // Qubits in reg start in` $\|0\rangle$.
	`    ...`
	`    // Qubits must be returned to` $\|0\rangle$.
	`}`

Allocate one qubit	`using (one = Qubit()) { ... }`

Allocate a mix of qubit registers and individual qubits	`using ((x, y, ... ) =`
	`    (Qubit[N], Qubit(), ... )) {`
	`    ...`
	`}`

Debugging (quantum)

Print amplitudes of wave function	`DumpMachine("dump.txt")`
Assert that a qubit is in $\|0\rangle$ or $\|1\rangle$ state	`AssertQubit(Zero, zeroQubit)` `AssertQubit(One, oneQubit)`

Basic Quantum Gates

Pauli gates	`X(qubit)` : $\|0\rangle \mapsto \|1\rangle$, $\|1\rangle \mapsto \|0\rangle$ `Y(qubit)` : $\|0\rangle \mapsto i\,\|1\rangle$, $\|1\rangle \mapsto -i\,\|0\rangle$ `Z(qubit)` : $\|0\rangle \mapsto \|0\rangle$, $\|1\rangle \mapsto -\,\|1\rangle$
Hadamard	`H(qubit)` : $\|0\rangle \mapsto \|+\rangle = \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$, $\|1\rangle \mapsto \|-\rangle = \frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$
Controlled-NOT	`CNOT(controlQubit, targetQubit)` $\|00\rangle \mapsto \|00\rangle$, $\|01\rangle \mapsto \|01\rangle$, $\|10\rangle \mapsto \|11\rangle$, $\|11\rangle \mapsto \|10\rangle$
Apply several gates (Bell pair example)	`H(qubit1);` `CNOT(qubit1, qubit2);`

Measurements

Measure qubit in Pauli Z basis	`M(oneQubit)` yields a `Result` (`Zero` or `One`)
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 dirname`
Go to home	`cd ~`
Go up one directory	`cd ..`
Make new directory	`mkdir dirname`
Open current directory in VS Code	`code .`

Working with Q# Projects

Create new project	`dotnet new console -lang Q#` `--output project-dir`
Change directory to project directory	`cd project-dir`
Build project	`dotnet build`
Run all unit tests	`dotnet test`