Contents

Quantum Intermediate Representation (QIR)

This specification defines an LLVM-based intermediate representation for quantum programs.

Motivation

There are many languages and circuit-building packages that are used today to program quantum computers. These languages have some similarities, but vary significantly in their syntax and semantics.

Similarly, there are many different types of quantum computers available already, and more types being developed. Different computers have different instruction sets, different timings, and different classical capabilities.

It is very useful to have a single intermediate representation that can express programs from all of the different quantum languages and can be converted into code for any of the different quantum computers. This allows language developers to write one compiler and quantum computer developers to write one code generator, while still allowing all languages to run on all computers. This is a well-established pattern in classical compilation.

Model

We see compilation as having three high-level phases:

1. A language-specific phase that takes code written in some quantum language, performs language-specific transformations and optimizations, and compiles the result into this intermediate format.
2. A generic phase that performs transformations and analysis of the code in the intermediate format.
3. A target-specific phase that performs additional transformations and ultimately generates the instructions required by the execution platform in a target-specific format.

By defining our representation within the popular open-source LLVM framework, we enable users to easily write code analyzers and code transformers that operate at this level, before the final target-specific code generation.

It is neither required nor expected that any particular execution target actually implement every runtime function specified here. Rather, it is expected that the target-specific compiler will translate the functions defined here into the appropriate representation for the target, whether that be code, calls into target-specific libraries, metadata, or something else.

This applies to quantum functions as well as classical functions. We do not intend to specify a gate set that all targets must support, nor even that all targets use the gate model of computation. Rather, the quantum functions in this document specify the interface that language-specific compilers should meet. It is the role of the target-specific compiler to translate the quantum functions into an appropriate computation that meets the computing model and capabilities of the target platform.

Profiles

We know that many current targets will not support the full breadth of possible quantum programs that can be expressed in this representation. We define a sequence of specification *profiles* that define coherent subsets of functionality that a specific target can support.

Data Type Representation

QIR defines LLVM representations for a variety of classical and quantum data types that may be used as part of a compiled quantum program. For more information about classical memory management including reference and alias counting, see here.

There are several error conditions that are specified as causing a runtime failure. The `quantum__rt__fail` function is the mechanism to use to cause a runtime failure; it is documented in the Classical Runtime section.

Simple Types

The simple types are those whose values are fixed-size and do not contain pointers. They are represented as follows:

Type	LLVM Representation	Comments
`Int`	`i64`	A 64-bit signed integer. Targets should specify their behavior on integer overflow and division by zero.
`Double`	`double`	A 64-bit IEEE double-precision floating point number. Targets should specify their behavior on floating overflow and division by zero.
`Bool`	`i1`	0 is false, 1 is true.
`Pauli`	`%Pauli = i2`	0 is PauliI, 1 is PauliX, 3 is PauliY, and 2 is PauliZ.
`Range`	`%Range = {i64, i64, i64}`	In order, these are the start, step, and inclusive end of the range. When passed as a function argument or return value, ranges should be passed by value.

LLVM and QIR place some limits on integer values. Specifically, when raising an integer to a power, the exponent must fit into a 32-bit integer, `i32`. Also, bit shifts are limited to shift counts that are non-negative and less than 64.

A `%Range` is an expression that represents a sequence of integers. The first element of the sequence is the `start` of the range, the second element is `start+step`, the third element is `start+2*step`, and so forth. The `step` may be positive or negative, but not zero. An attempt to create a `%Range` with a zero step should cause a runtime failure.

The last element of the range may be `end`; that is, `end` is inclusive. A range is empty if `step` is positive and `end` is less than `start`, or if `step` is negative and `end` is greater than `start`. For example:

```
0..1..2 = {0, 1, 2}
0..2..4 = {0, 2, 4}
0..2..5 = {0, 2, 4}
4..-1..2 = {4, 3, 2}
5..-3..0 = {5, 2}
0..1..-1 = {}
0..-1..1 = {}
```

The following global constants are defined for use with `%Pauli` type:

```
@PauliI = constant i2 0
@PauliX = constant i2 1
@PauliY = constant i2 -1 ; The value 3 (binary 11) is displayed as a
                         ;  2-bit signed value of -1 (binary 11).
@PauliZ = constant i2 -2 ; The value 2 (binary 10) is displayed as a
                         ;  2-bit signed value of -2 (binary 10).
```

Measurement Results

Measurement results are represented as pointers to an opaque LLVM structure type, `%Result`. This allows each target implementation to provide a structure definition appropriate for that target. In particular, this makes it easier for implementations where measurement results might come back asynchronously.

The following utility functions are provided by the classical runtime for use with values of type `%Result*`:

Function	Signature	Description
__quantum__rt__result_get_zero	`%Result*()`	Returns a constant representing a measurement result zero.
__quantum__rt__result_get_one	`%Result*()`	Returns a constant representing a measurement result one.
__quantum__rt__result_equal	`i1(%Result*, %Result*)`	Returns true if the two results are the same, and false if they are different. If a `%Result*` parameter is null, a runtime failure should occur.
__quantum__rt__result_update_reference_count	`void(%Result*, i32)`	Adds the given integer value to the reference count for the result. Deallocates the result if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Result*` is a null pointer.

Qubits

Qubits are represented as pointers to an opaque LLVM structure type, `%Qubit`. This is done so that qubit values may be distinguished from other value types. It is not expected that qubit values actually be valid memory addresses, and neither user code nor runtime code should ever attempt to dereference a qubit value.

A qubit value should be thought of as an integer identifier that has been bit-cast into a special type so that it can be distinguished from normal integers. The only operation that may be performed on a qubit value is to pass it to a function.

Qubits may be managed either statically or dynamically. Static qubits have target-specific identifiers known at compile time, while dynamic qubits are managed by the quantum runtime.

A static qubit value may be created using the LLVM `inttoptr` instruction. For instance, to initialize a value that identifies device qubit 3, the following LLVM code would be used:

```
    %qubit3 = inttoptr i32 3 to %Qubit*
```

Dynamic qubits are managed using the quantum runtime functions.

Strings

Strings are represented as pointers to an opaque type.

Type	LLVM Representation
String	`%String*`

The following utility functions should be provided by the classical runtime to support strings:

Function	Signature	Description
__quantum__rt__string_create	`%String*(i8*)`	Creates a string from an array of UTF-8 bytes. The byte array is expected to be zero-terminated.
__quantum__rt__string_get_data	`i8*(%String*)`	Returns a pointer to the zero-terminated array of UTF-8 bytes.
__quantum__rt__string_get_length	`i32(%String*)`	Returns the length of the byte array that contains the string data.
__quantum__rt__string_update_reference_count	`void(%String*, i32)`	Adds the given integer value to the reference count for the string. Deallocates the string if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%String*` is a null pointer.
__quantum__rt__string_concatenate	`%String* (%String*, %String*)`	Creates a new string that is the concatenation of the two argument strings. If a `%String*` parameter is null, a runtime failure should occur.
__quantum__rt__string_equal	`i1(%String*, %String*)`	Returns true if the two strings are equal, false otherwise. If a `%String*` parameter is null, a runtime failure should occur.

The following utility functions support converting values of other types to strings. In every case, the returned string is allocated on the heap; the string cannot be allocated by the caller because the length of the string depends on the actual value.

Function	Signature	Description
__quantum__rt__int_to_string	`%String*(i64)`	Returns a string representation of the integer.
__quantum__rt__double_to_string	`%String*(Double)`	Returns a string representation of the double.
__quantum__rt__bool_to_string	`%String*(i1)`	Returns a string representation of the Boolean.
__quantum__rt__result_to_string	`%String*(%Result*)`	Returns a string representation of the result.
__quantum__rt__pauli_to_string	`%String*(%Pauli)`	Returns a string representation of the Pauli.
__quantum__rt__qubit_to_string	`%String*(%Qubit*)`	Returns a string representation of the qubit.
__quantum__rt__range_to_string	`%String*(%Range)`	Returns a string representation of the range.
__quantum__rt__bigint_to_string	`%String*(%BigInt*)`	Returns a string representation of the big integer.

In all cases, if a pointer parameter is null, a runtime failure should occur.

Big Integers

Unlimited-precision integers, also known as "big integers", are represented as pointers to an opaque type.

Type	LLVM Representation
BigInt	`%BigInt*`

The following utility functions are provided by the classical runtime to support big integers.

Function	Signature	Description
__quantum__rt__bigint_create_i64	`%BigInt*(i64)`	Creates a big integer with the specified initial value.
__quantum__rt__bigint_create_array	`%BigInt*(i32, i8*)`	Creates a big integer with the value specified by the `i8` array. The 0-th element of the array is the highest-order byte, followed by the first element, etc.
__quantum__rt__bigint_get_data	`i8*(%BigInt*)`	Returns a pointer to the `i8` array containing the value of the big integer.
__quantum__rt__bigint_get_length	`i32(%BigInt*)`	Returns the length of the `i8` array that represents the big integer value.
__quantum__rt__bigint_update_reference_count	`void(%BigInt*, i32)`	Adds the given integer value to the reference count for the big integer. Deallocates the big integer if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%BigInt*` is a null pointer.
__quantum__rt__bigint_negate	`%BigInt* (%BigInt*)`	Returns the negative of the big integer.
__quantum__rt__bigint_add	`%BigInt* (%BigInt*, %BigInt*)`	Adds two big integers and returns their sum.
__quantum__rt__bigint_subtract	`%BigInt* (%BigInt*, %BigInt*)`	Subtracts the second big integer from the first and returns their difference.
__quantum__rt__bigint_multiply	`%BigInt* (%BigInt*, %BigInt*)`	Multiplies two big integers and returns their product.
__quantum__rt__bigint_divide	`%BigInt* (%BigInt*, %BigInt*)`	Divides the first big integer by the second and returns their quotient.
__quantum__rt__bigint_modulus	`%BigInt* (%BigInt*, %BigInt*)`	Returns the first big integer modulo the second.
__quantum__rt__bigint_power	`%BigInt* (%BigInt*, i32)`	Returns the big integer raised to the integer power. As with standard integers, the exponent must fit in 32 bits.
__quantum__rt__bigint_bitand	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-AND of two big integers.
__quantum__rt__bigint_bitor	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-OR of two big integers.
__quantum__rt__bigint_bitxor	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-XOR of two big integers.
__quantum__rt__bigint_bitnot	`%BigInt* (%BigInt*)`	Returns the bitwise complement of the big integer.

Function	Signature	Description
__quantum__rt__bigint_shiftleft	`%BigInt* (%BigInt*, i64)`	Returns the big integer arithmetically shifted left by the (positive) integer amount of bits.
__quantum__rt__bigint_shiftright	`%BigInt* (%BigInt*, i64)`	Returns the big integer arithmetically shifted right by the (positive) integer amount of bits.
__quantum__rt__bigint_equal	`i1(%BigInt*, %BigInt*)`	Returns true if the two big integers are equal, false otherwise.
__quantum__rt__bigint_greater	`i1(%BigInt*, %BigInt*)`	Returns true if the first big integer is greater than the second, false otherwise.
__quantum__rt__bigint_greater_eq	`i1(%BigInt*, %BigInt*)`	Returns true if the first big integer is greater than or equal to the second, false otherwise.

In all cases other than to `__quantum__rt__bigint_update_reference_count`, if a `%BigInt*` parameter is null, a runtime failure should occur.

Tuples and User-Defined Types

Tuple data, including values of user-defined types, is represented as the corresponding LLVM structure type. For instance, a tuple containing two integers, `(Int, Int)`, would be represented in LLVM as `type {i64, i64}`.

When invoking callable values using the `__quantum__rt__callable_invoke` runtime function, tuples are passed as a pointer to an opaque LLVM structure, `%Tuple`. The pointer is expected to point to the contained data such that it can be cast to the correct data structures by the receiving code. This permits the definition of runtime functions that are common for all tuples, such as the functions listed below.

Many languages provide immutable tuples, along with operators that allow a modified copy of an existing tuple to be created. QIR supports this by requiring the runtime to track and be able to access the following given a `%Tuple*`:

- The size of the tuple in bytes
- The alias count indicating how many handles to the tuple exist in the source code

The language specific compiler is responsible for injecting calls to increase and decrease the alias count as needed, as well as to accurately reflect when references to the LLVM structure representing a tuple are created and removed. See this section for further details on the distinction between alias and reference counting.

In the case where the source language treats tuples as immutable values, the language-specific compiler is expected to request the necessary copies prior to modifying the tuple in place. This is done by invoking the runtime function `__quantum__rt__tuple_copy` to create a byte-by-byte copy of a tuple. Unless the copying is forced via the second argument, the runtime may omit copying the value and instead simply return a pointer to the given argument if the alias count is 0 and it is therefore safe to modify the tuple in place.

The following utility functions are provided by the classical runtime to support tuples and user-defined types:

Function	Signature	Description
__quantum__rt__tuple_create	`%Tuple*(i64)`	Allocates space for a tuple requiring the given number of bytes, sets the reference count to 1 and the alias count to 0.
__quantum__rt__tuple_copy	`%Tuple* (%Tuple*, i1)`	Creates a shallow copy of the tuple if the alias count is larger than 0 or the second argument is `true`. Returns the given tuple pointer (the first parameter) otherwise, after increasing its reference count by 1. The reference count of the tuple elements remains unchanged. If the `%Tuple*` parameter is null, a runtime failure should occur.
__quantum__rt__tuple_update_reference_count	`void(%Tuple*, i32)`	Adds the given integer value to the reference count for the tuple. Deallocates the tuple if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Tuple*` is a null pointer.
__quantum__rt__tuple_update_alias_count	`void(%Tuple*, i32)`	Adds the given integer value to the alias count for the tuple. Fails if the count becomes negative. The call should be ignored if the given `%Tuple*` is a null pointer.

Unit

For source languages that include a unit type, the representation of this type in LLVM depends on its usage. If used as a return type for a callable, it should be translated into an LLVM `void` function. If it is used as a value, for instance as an element of a tuple, it should be represented as a null tuple pointer.

Arrays

Within QIR, arrays are represented and passed around as a pointer to an opaque LLVM structure, `%Array`. How array data is represented, i.e., what that pointer points to, is at the discretion of the runtime. All array manipulations, including item access, hence need to be performed by invoking the corresponding runtime function(s).

Because LLVM does not provide any mechanism for type-parameterized functions, runtime library routines that provide access to array elements return byte pointers that the calling code must `bitcast` to the appropriate type before using. When creating an array, the size of each element in bytes must be provided.

Many languages provide immutable arrays, along with operators that allow a modified copy of an existing array to be created. In QIR, this is implemented by creating a new copy of the existing array and then modifying the newly-created array in place. If the existing array is not used after the creation of the modified copy, it is possible to avoid the copy and modify the existing array in place instead. To achieve such a behavior, the language specific compiler should ensure that the alias count for arrays accurately reflects their use in the source language, and rely on the runtime function for copying to omit the copy when the alias count is 0.

In addition to creating modified copies of arrays, there are two other ways of constructing new arrays that permit for similar optimizations; array slicing and array projections.

- An array *slice* is specified by providing a dimension to slice on and a `%Range` to slice with. The resulting array has the same number of dimensions as the original array, but only those elements in the sliced dimension whose original indices were part of the resolution of the `%Range`. Those elements get new indices in the resulting array based on their appearance order in the `%Range`. In particular, if the step of the `%Range` is negative, the elements in the sliced dimension will be in the reverse order than they were in the original array. If the `%Range` is empty, the resulting array will be empty.

  Array slices can be created using the `__quantum__rt__array_slice_1d` or `__quantum__rt__array_slice` runtime functions.

- An array *projection* is specified by providing a dimension to project along and an `i64` index value to project to. The resulting array has one fewer dimension than the original array, and is the segment of the original array with the projected dimension fixed to the given index value. Projection is the array access analog to partial application; effectively it creates a new array that has the same elements as the original array, but one of the indices is fixed at a constant value. Array projections can be created using the `__quantum__rt__array_project` runtime function.

Attempting to access an index or dimension outside the bounds of an array should cause an immediate runtime failure. This applies to slicing and projection operations as well as to element access. When validating indices for slicing, only indices that are actually part of the resolved range should be considered.

The following utility functions are provided by the classical runtime to support arrays:

Function	Signature	Description
__quantum__rt__array_create_1d	`%Array* void(i32, i64)`	Creates a new 1-dimensional array. The `i32` is the size of each element in bytes. The `i64` is the length of the array. The bytes of the new array should be set to zero. If the length is zero, the result should be an empty 1-dimensional array.
__quantum__rt__array_copy	`%Array* (%Array*, i1)`	Creates a shallow copy of the array if the alias count is larger than 0 or the second argument is `true`. Returns the given array pointer (the first parameter) otherwise, after increasing its reference count by 1. The reference count of the array elements remains unchanged.
__quantum__rt__array_concatenate	`%Array* (%Array*, %Array*)`	Returns a new array which is the concatenation of the two passed-in one-dimensional arrays. If either array is not one-dimensional or if the array element sizes are not the same, then a runtime failure should occur.
__quantum__rt__array_slice_1d	`%Array* (%Array*, %Range, i1)`	Creates and returns an array that is a slice of an existing 1-dimensional array. The slice may be accessing the same memory as the given array unless its alias count is larger than 0 or the last argument is `true`. The `%Range` specifies the indices that should be the elements of the returned array. The reference count of the elements remains unchanged.

Function	Signature	Description
__quantum__rt__array_get_size_1d	`i64(%Array*)`	Returns the length of a 1-dimensional array.
__quantum__rt__array_get_element_ptr_1d	`i8*(%Array*, i64)`	Returns a pointer to the element of the array at the zero-based index given by the `i64`.
__quantum__rt__array_update_reference_count	`void(%Array*, i32)`	Adds the given integer value to the reference count for the array. Deallocates the array if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Array*` is a null pointer.
__quantum__rt__array_update_alias_count	`void(%Array*, i32)`	Adds the given integer value to the alias count for the array. Fails if either count becomes negative. The call should be ignored if the given `%Array*` is a null pointer.

For all of these functions other than `__quantum__rt__array_update_reference_count` or `__quantum__rt__array_update_alias_count`, if an `%Array*` pointer is null, a runtime failure should result.

The following utility functions are provided if multidimensional array support is enabled:

Function	Signature	Description
__quantum__rt__array_create	`%Array* void(i32, i32, i64*)`	Creates a new array. The first `i32` is the size of each element in bytes. The second `i32` is the dimension count. The `i64*` should point to an array of `i64`s contains the length of each dimension. The bytes of the new array should be set to zero. If any length is zero, the result should be an empty array with the given number of dimensions.
__quantum__rt__array_get_dim	`i32(%Array*)`	Returns the number of dimensions in the array.
__quantum__rt__array_get_size	`i64(%Array*, i32)`	Returns the length of a dimension of the array. The `i32` is the zero-based dimension to return the length of; it must be smaller than the number of dimensions in the array.
__quantum__rt__array_get_element_ptr	`i8*(%Array*, i64*)`	Returns a pointer to the indicated element of the array. The `i64*` should point to an array of `i64`s that are the indices for each dimension.

Function	Signature	Description
__quantum__rt__array_slice	`%Array* (%Array*, i32, %Range, i1)`	Creates and returns an array that is a slice of an existing array. The slice may be accessing the same memory as the given array unless its alias count is larger than 0 or the last argument is `true`. The `i32` indicates which dimension the slice is on, and must be smaller than the number of dimensions in the array. The `%Range` specifies the indices in that dimension that should be the elements of the returned array. The reference count of the elements remains unchanged.
__quantum__rt__array_project	`%Array* (%Array*, i32, i64, i1)`	Creates and returns an array that is a projection of an existing array. The projection may be accessing the same memory as the given array unless its alias count is larger than 0 or the last argument is `true`. The `i32` indicates which dimension the projection is on, and the `i64` specifies the index in that dimension to project. The reference count of all array elements remains unchanged. If the existing array is one-dimensional then a runtime failure should occur.

There are special runtime functions defined for allocating or releasing an array of qubits. See here for these functions.

For all of these functions, if an `%Array*` pointer is null, a runtime failure should occur.

Callables

We use the term *callable* to mean a subroutine in the source language. Different source languages use different names for this concept.

> *Note:* The QIR specification permits the usage of subroutines as first class values, and includes the necessary expressiveness to e.g. provide runtime support for functor application. This introduces the need to define a common structure to represent callable values and their arguments. If the source language does not make use of such features and there is no need to represent callable values in the compilation, then the corresponding sections in this document do not apply.

Runtime Failure

There are several error conditions that are specified as causing a runtime failure. The `__quantum__rt__fail` function is the mechanism to use to cause a runtime failure; it is documented in the Classical Runtime section.

Basics

Callables are represented by up to four different LLVM functions to handle different combinations of functors: one for the base version and one each for the adjoint, controlled, and controlled-adjoint specializations. A callable that does not have a specific specialization would not have the corresponding LLVM function.

The names of the functions should be the namespace-qualified name of the callable with each period replaced by two underscores, '__', then another two underscores, followed by `body` for the base version, `adj`

for the adjoint specialization, `ctl` for the controlled specialization, and `ctladj` for the controlled adjoint specialization.

The signatures of the callables should match the source language signature. For instance, the base version of a callable that takes a floating-point number "theta" and a qubit "qb" and returns a result would generate:

```
define %Result *Some__Namespace__Symbol__body (double theta, %Qubit *qb)
{
  ; code goes here
}
```

Direct invocations of callables should be generated into normal LLVM function calls.

Callable Values

In order to support lambda captures and partial application, as well as applying functors to callable values (function pointers), such values need to be represented by a more complex data structure than simply a standard function pointer. These values are represented by a pointer to an opaque LLVM type, `%Callable`.

Wrapper Functions

Because LLVM does not support generics, the LLVM function pointers used to initialize a `%Callable` have to be of a single type. To accomplish this, we create a new "wrapper" function for each of the callable's specializations. All such wrapper functions have the same signature and so are of the same LLVM type. The name of a wrapper function should be the same as the name of the function it is wrapping, with "__wrapper" appended.

The four specializations of a callable are:

- The "body", which is the normal, unmodified callable.
- The "adj", which implements the adjoint of the quantum operation defined by the callable.
- The "ctl", which implements the controlled version of the quantum operation defined by the callable.
- The "ctladj", which implements the adjoint of the controlled version.

A callable may define a "ctladj" specialization if and only if it defines both "adj" and "ctl" specializations.

There is no need to create wrappers for callables that are never pointed to. That is, **a callable that is never turned into a callable value does not need wrapper functions**.

Each wrapper is an LLVM function that takes three tuple header pointers as input and returns no output; that is, `void(%Tuple*, %Tuple*, %Tuple*)`. The first input is the capture tuple, which is used for closures. The second input is the argument tuple. The third input points to the result tuple, which will be allocated by the caller. If the callable has `Unit` result, then the result tuple pointer will be null.

Each wrapper function should start with a prologue that decomposes the argument and capture tuples. Depending on the result type, it should end with an epilogue that fills in the result tuple.

We use a caller-allocates strategy for the result tuple because this allows us to avoid a heap allocation in many cases. If the callee allocates space for the result tuple, that space has to be on the heap because a stack-based allocation would be released when the callee returns. The caller can usually allocate the result tuple on the stack, or reuse the result tuple pointer it received for tail calls.

For instance, for a callable named `Some.Namespace.Symbol` with all four specializations, the compiler should generate the following in LLVM:

```
define void Some__Namespace__Symbol__body__wrapper (%Tuple* capture,
    %Tuple* args, %Tuple* result)
{
  ; code to get arguments out of the args and capture tuples goes here
  ; call to Some__Namespace__Symbol__body() goes here
```

```
  ; code to fill in the result tuple goes here
  ret void;
}

define void Some__Namespace__Symbol__adj__wrapper (%Tuple* capture,
    %Tuple* args, %Tuple* result)
{
  ; code to get arguments out of the args and capture tuples goes here
  ; call to Some__Namespace__Symbol__adj() goes here
  ; code to fill in the result tuple goes here
  ret void;
}

define void Some__Namespace__Symbol__ctl__wrapper (%Tuple* capture,
    %Tuple* args, %Tuple* result)
{
  ; code to get arguments out of the args and capture tuples goes here
  ; call to Some__Namespace__Symbol__ctl() goes here
  ; code to fill in the result tuple goes here
  ret void;
}

define void Some__Namespace__Symbol__ctladj__wrapper (%Tuple* capture,
    %Tuple* args, %Tuple* result)
{
  ; code to get arguments out of the args and capture tuples goes here
  ; call to Some__Namespace__Symbol__ctladj() goes here
  ; code to fill in the result tuple goes here
  ret void;
}
```

Implementation Table

For each callable that is used to create a callable value, a table is created with pointers to the four wrapper functions; specializations that do not exist for a specific callable have a null pointer in that place. The table is defined as a global constant whose name is the namespace-qualified name of the callable with periods replaced by double underscores, "__".

For the example above, the following would be generated:

```
@Some__Namespace__Symbol = constant
  [void (%Tuple*, %Tuple*, %Tuple*)*]
  [
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Some__Namespace__Symbol__body__wrapper,
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Some__Namespace__Symbol__adj__wrapper,
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Some__Namespace__Symbol__ctl__wrapper,
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Some__Namespace__Symbol__ctladj__wrapper
  ]
```

There is no need to create an implementation table for callables that are never pointed to. That is, **a callable that is never turned into a callable value does not need an implementation table**.

Creating Callable Values

As mentioned above, callable values are represented by a pointer to an opaque LLVM structure , `%Callable`. These values are created using the `__quantum__rt__callable_create` or `__quantum__rt__callable_copy` runtime functions.

The `__quantum__rt__callable_create` function takes an implementation table, a memory management table, and a capture tuple and returns a pointer to a new `%Callable`. The capture tuple in the `%Callable` is passed as the first argument to the wrapper function when the callable value is invoked. It is intended to hold values captured by a lambda or provided values in a partial application. If there are no captured values, a null pointer should be passed.

The `__quantum__rt__callable_copy` function creates a copy of an existing `%Callable`. It is most often used before using the `__quantum__rt__callable_make_adjoint` or `__quantum__rt__callable_make_controlled` functions to apply a functor to a callable value.

Invoking a Callable Value

As mentioned above, direct invocations of callables should be generated into normal LLVM function calls.

To invoke a callable value represented as a `%Callable*`, the `__quantum__rt__callable_invoke` runtime function should be used. This function uses the information in the callable value to invoke the proper implementation with the appropriate parameters. To satisfy LLVM's strong typing, this function requires the arguments to be assembled into a tuple and passed to the runtime function as a tuple pointer.

Applying Functors to Callable Values

The Adjoint and Controlled functors are important for expressing quantum algorithms. They are implemented by the `__quantum__rt__callable_make_adjoint` and `__quantum__rt__callable_make_control` runtime functions, which update a `%Callable` in place by applying the `Adjoint` or `Controlled` functors respectively.

To support cases where the original, unmodified `%Callable` is still needed after functor application, the `__quantum__rt__callable_copy` routine may be used to create a new copy of the original `%Callable`; the functor may then be applied to the new `%Callable`. For instance, to implement the following:

```
let f = someOp;
let g = Adjoint f;
// ... code that uses both f and g ...
```

The following snippet of LLVM code could be generated:

```
%f = call %Callable* @__quantum__rt__callable_create(
  [4 x void (%Tuple*, %Tuple*, %Tuple*)*]* @someOp,
  [2 x void (%Tuple*, i32)*]* null,
  %Tuple* null)
%g = call %__quantum__rt__callable_copy(%f)
call %__quantum__rt__callable_make_adjoint(%g)
```

The actual implementation of the `%Callable` needs to support the following behavior:

- For `__quantum__rt__callable_make_adjoint`, the parity of the number of times this function has been applied should be tracked. The `Adjoint` functor is its own inverse, so applying it twice is the same as not applying it at all. Applying this function to a `%Callable` whose implementation table has a null "adj" entry should cause a runtime failure.
- For `__quantum__rt__callable_make_control`, the count of the number of times the function has been applied must be tracked. The `Controlled` functor is not its own inverse. Applying this function to a `%Callable` whose implementation table has a null "ctl" entry should cause a runtime failure.

- The order of applying these two functions does not need to be tracked. The `Adjoint` and `Controlled` functors commute.

When `__quantum__rt__callable_invoke` is called, the entry in the callable's implementation table to be used is selected as follows:

- If the adjoint parity is even and the controlled count is zero, the "body" entry (index 0) should be used.
- If the adjoint parity is odd and the controlled count is zero, the "adj" entry (index 1) should be used.
- If the adjoint parity is even and the controlled count is greater than zero, the "ctl" entry (index 2) should be used.
- If the adjoint parity is odd and the controlled count is greater than zero, the "ctladj" entry (index 3) should be used.

If the controlled count is greater than one, then `__quantum__rt__callable_invoke` also needs to do some manipulation of the input tuple. Each application of the `Controlled` functor modifies the signature of the specialization by replacing the current argument tuple with a two-tuple containing the array of control qubits as the first element and a tuple of the remaining arguments as the second tuple.

For instance, if the base callable expects an argument tuple `{ i64, %Qubit* }`, then the `Controlled` version expects `{ %Array*, { i64, %Qubit* }* }`, and the twice-`Controlled` version expects `{ %Array*, { %Array*, { i64, %Qubit* }* }* }`. The "ctl" implementation function always expects `{ %Array*, { i64, %Qubit* }* }`. Thus, if the controlled count is greater than 1, `__quantum__rt__callable_invoke` needs to disassemble the argument tuple, concatenate the control qubit arrays, and form the expected argument tuple.

One additional complexity is that the above is modified slightly if the base callable expects a single argument. In this case, the `Controlled` version expects a two-element tuple as above, where the second element is the base argument. For instance, if the base callable expects `%Qubit*`, the singly-`Controlled` version expects `{ %Array*, %Qubit* }` rather than `{ %Array*, { %Qubit* }* }`. This means that the second element of the singly-`Controlled` argument tuple is not always a pointer to a struct, and in particular may have variable length up to the size of a `%Range`.

To resolve this, `__quantum__rt__callable_invoke` needs to have access to the length of the inner argument tuple once it has unwrapped down to that point. This could be stored in the `%Tuple` by `__quantum__rt__tuple_create`, or it could be provided by the classical runtime from the length originally provided for the heap allocation.

The "ctl" implementation function cannot do this manipulation itself because it does not have access to the controlled count and so cannot tell what the actual argument tuple's structure is. Similarly, while the calling code knows the exact signature, it also does not have access to the controlled count, and so cannot unambiguously determine the expected argument tuple; specifically, it cannot tell if an inner tuple is the result of an application of `Controlled` or just part of the base signature.

Implementing Lambdas, Partial Application, and Currying

The language-specific compiler should generate a new callable at the global scope of the appropriate type with implementation provided by the anonymous body of the lambda; this is known as "lifting" the lambda. A unique name should be generated for this callable. Lifted callables can support functors, just like any other callable. The language-specific compiler is responsible for determining the set of functors that should be supported by the lifted callable and generating code for them accordingly.

At the point where a lambda is created as a value in the code, a new callable data structure should be created with the appropriate contents. Any values referenced inside the lambda that are defined in a scope external to the lambda should be added to the lambda's capture tuple. The language-specific compiler is responsible for having references within the lambda to the captured values refer to the capture tuple.

Partial application and currying are alternative forms of closures; that is, both create a lambda values, although the source syntax is different from a lambda expression.

Partial applications and curried functions should be rewritten into lambdas by the language-specific compiler. The lambda body may need to include additional code that performs argument tuple construction before calling the underlying callable.

Memory Management Table

Since any captured values need to remain alive as long as the callable value exists, they also need to be unreferenced when the callable value is released. While sufficient type information for the captured values is known upon creation of the value, the information is no longer available at the time when it is released. Upon creation, a table with two function pointers for modifying reference and alias counts for captured values is hence associated with a callable value.

Like the implementation table, the table is defined as a global constant with a unique name. It contains two pointers of type `void(%Tuple*, i32)*`; the first one points to the function for modifying the reference counts of captured values, the second points to the one for modifying the alias counts. Either of those pointers may be null, and if no managed values were captured, a null pointer should be passed instead of a table upon callable creation.

As for tuple and array elements, the responsibility of managing the reference and alias count for captured values lays with the compiler. The two functions can be invoked using the runtime function `__quantum__rt__capture_update_reference_count` and `__quantum__rt__capture_update_alias_count` respectively, see the description below.

External Callables

Callables may be specified as external; that is, they are declared in the quantum source, but are defined in an external component that is statically or dynamically linked with the compiled quantum code. Such callables are also sometimes referred to as ”intrinsic”.

In particular, the quantum instruction set supported by a particular target should be represented as a set of external callables.

The source compiler should generate appropriate LLVM declarations for any external callables referred to by the generated code. Declarations for other external callables can be included if desired.

Generating the proper linkage is the responsibility of the target-specific compilation phase.

Generics

QIR does not provide support for generic or type-parameterized callables. It relies on the language-specific compiler to generate a new, uniquely-named callable for each combination of concrete type parameters. The LLVM representation treats these generated callables as the actual callables to generate code for; the original callables with open type parameters are not represented in LLVM.

Runtime Functions

The following functions are provided by the classical runtime to support callable values:

Function	Signature	Description
__quantum__rt__callable_create	`%Callable*([4 x void (%Tuple*, %Tuple*, %Tuple*)*]*, [2 x void(%Tuple*, i32)]*, %Tuple*)`	Initializes the callable with the provided function table, memory management table, and capture tuple. The memory management table pointer and the capture tuple pointer should be null if there is no capture.

Function	Signature	Description
__quantum__rt__callable_copy	`%Callable* (%Callable*, i1)`	Creates a shallow copy of the callable if the alias count is larger than 0 or the second argument is `true`. Returns the given callable pointer (the first parameter) otherwise, after increasing its reference count by 1. The reference count of the capture tuple remains unchanged. If the `%Callable*` parameter is null, a runtime failure should occur.
__quantum__rt__callable_invoke	`void(%Callable*, %Tuple*, %Tuple*)`	Invokes the callable with the provided argument tuple and fills in the result tuple. The `%Tuple*` parameters may be null if the callable either takes no arguments or returns `Unit`. If the `%Callable*` parameter is null, a runtime failure should occur.
__quantum__rt__callable_make_adjoint	`void(%Callable*)`	Updates the callable by applying the Adjoint functor. If the `%Callable*` parameter is null or if the corresponding entry in the callable's function table is null, a runtime failure should occur.
__quantum__rt__callable_make_controlled	`void(%Callable*)`	Updates the callable by applying the Controlled functor. If the `%Callable*` parameter is null or if the corresponding entry in the callable's function table is null, a runtime failure should occur.
__quantum__rt__callable_update_reference_count	`void(%Callable*, i32)`	Adds the given integer value to the reference count for the callable. Deallocates the callable if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Callable*` is a null pointer.
__quantum__rt__callable_update_alias_count	`void(%Callable*, i32)`	Adds the given integer value to the alias count for the callable. Fails if the count becomes negative. The call should be ignored if the given `%Callable*` is a null pointer.

Function	Signature	Description
__quantum__rt__capture_update_reference_count	`void(%Callable*, i32)`	Invokes the function at index 0 in the memory management table of the callable with the capture tuple and the given 32-bit integer. Does nothing if if the memory management table pointer or the function pointer at that index is null, or if the given `%Callable*` is a null pointer.
__quantum__rt__capture_update_alias_count	`void(%Callable*, i32)`	Invokes the function at index 1 in the memory management table of the callable with the capture tuple and the given 32-bit integer. Does nothing if the memory management table pointer or the function pointer at that index is null, or if the given `%Callable*` is a null pointer.

Classical Runtime

Most functions in the classical runtime are defined in the Data Types specification.

Additionally, QIR requires the following functions to be present for the purpose of logging and program termination:

Function	Signature	Description
__quantum__rt__message	`void(%String*)`	Include the given message in the computation's execution log or equivalent.
__quantum__rt__fail	`void(%String*)`	Fail the computation with the given error message.

Memory Management

QIR does not require the runtime to provide garbage collection. Instead, it specifies a set of runtime functions that can be used by the language specific compiler to implement a reference counting scheme if needed, if the source language requires automatic memory management.

Reference and Alias Counting

QIR specifies a set of runtime functions for types that are represented as pointers that may be used by the language-specific compiler to expose them as immutable types in the language. The exception is the `%Qubit*` type, for which no such functions exist since the management of quantum memory is distinct from classical memory management, see here for more detail.

To ensure that unnecessary copying of data can be avoided, QIR distinguishes two kinds of counts that can be tracked: reference counts and alias counts.

Reference counts track the number of handles that allow access to a certain value *in LLVM*. They hence determine when the value can be released by the runtime; values are allocated with a reference count of 1, and will be released when their reference count reaches 0.

Alias counts, on the other hand, track how many handles to a value exist *in the source language*. They determine when the runtime needs to copy data; when copy functions are invoked, the copy is executed only if

the alias count is larger than 0, or the copy is explicitly forced. Alias counts are useful for optimizing the handling of data types that are represented as pointers in QIR, but are value types, i.e. immutable, within the source language.

The compiler is responsible for generating code that tracks both counts correctly by injecting the corresponding calls to modify them. A call to modify such counts will only ever modify the count for the given instance itself and not for any inner items such as the elements of a tuple or an array, or a value captured by a callable; the compiler is responsible for injecting calls to update counts for inner items as needed. A runtime implementation is free to provide another mechanism for garbage collection and to treat calls to modify reference counts as hints or as simple no-ops.

- Runtime routines that create a new instance always initialize the instance with a reference count of 1, and an alias count of 0.
- For each pointer type, with the exception of `%Qubit*`, a runtime function ending in `_update_reference_count` exists that can be used to modify the reference count of an instance as needed. If the reference count reaches 0, the instance may be released. Decreasing the reference count below 0 or accessing a value after its reference count has reached 0 results in undefined behavior.
- For all data types that support a runtime function to create a shallow copy, a runtime function ending in `_update_alias_count` exists that can be used to modify the alias count of an instance as needed. These functions exist for `%Tuple*`, `%Array*`, and `%Callable*` types. The alias count can never be negative; decreasing the alias count below 0 results in a runtime failure.
- The functions that modify reference and alias count should accept a null instance pointer and simply ignore the call if the pointer is null.

Quantum Instruction Set and Runtime

Runtime Failure

There are several error conditions that are specified as causing a runtime failure. The `quantum__rt__fail` function is the mechanism to use to cause a runtime failure; it is documented in the Classical Runtime section.

Standard Operations (Gates)

As recommended by the LLVM documentation, we do not define new LLVM instructions for quantum operations. Instead, we expect each target to define a set of quantum operations as LLVM functions that may be used by language-specific compilers.

Qubits

We define the following functions for managing qubits:

Function	Signature	Description
__quantum__rt__qubit_allocate	`%Qubit*()`	Allocates a single qubit.
__quantum__rt__qubit_allocate_array	`%Array*(i64)`	Creates an array of the given size and populates it with newly-allocated qubits.
__quantum__rt__qubit_release	`void(%Qubit*)`	Releases a single qubit. Passing a null pointer as argument should cause a runtime failure.
__quantum__rt__qubit_release_array	`void(%Array*)`	Releases an array of qubits; each qubit in the array is released, and the array itself is unreferenced. Passing a null pointer as argument should cause a runtime failure.

The language-specific compiler may assume that qubits are always allocated in a zero-state. Since individual

targets may give different guarantees regarding the qubit state upon allocation, the target-specific compilation phase should insert the code required to ensure that the state of qubits is set appropriately.

Any measurements or resets applied upon release are at the discretion of the target as well; for the quantum algorithm to be correct, the qubits that are to be released hence should be unentangled from qubits that remain live prior to invoking the release function.

Metadata

Here we use "metadata" to signify both LLVM metadata and attributes. While metadata is more flexible, in some cases attributes may be preferred either because passes are required to keep them or because there are existing LLVM attributes with the required semantics.

Representing Source-Language Attributes

Many languages allow attributes to be placed on callable and type definitions. For instance, in Q# attributes are compile-time constant values of specific user-defined types that themselves have the `Microsoft.Quantum.Core.Attribute` attribute.

The language compiler should represent these attributes as LLVM metadata associated with the callable or type. For callables, the metadata representing the attribute should be attached to the LLVM global symbol that defines the implementation table for the callable. The identifier of the metadata node should be "!quantum.", where "!" is the LLVM standard prefix for a metadata value, followed by the namespace-qualified name of the attribute. For example, a callable `Your.Op`with two attributes, `My.Attribute(6, "hello")` and `Their.Attribute(2.1)`, applied to it would be represented in LLVM as follows:

```
@Your.Op = constant
  [void (%Tuple*, %Tuple*, %Tuple*)*]
  [
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Your.Op-body,
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Your.Op-adj,
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Your.Op-ctl,
    void (%Tuple*, %Tuple*, %Tuple*)*
        @Your.Op-ctladj
  ], !quantum.My.Attribute {i64 6, !"hello\00"},
     !quantum.Their.Attribute {double 2.1}
```

LLVM does not allow metadata to be associated with structure definitions, so there is no direct way to represent attributes attached to user-defined types. Thus, attributes on types are represented as named (module-level) metadata, where the metadata node's name is "quantum." followed by the namespace-qualified name of the type. The metadata itself is the same as for callables, but wrapped in one more level of LLVM structure in order to handle multiple attributes on the same structure. For example, a type `Your.Type`with two attributes, `My.Attribute(6, "hello")` and `Their.Attribute(2.1)`, applied to it would be represented in LLVM as follows:

```
!quantum.Your.Type = !{ !{!"quantum.My.Attribute\00", i64 6, !"hello\00"},
                        !{ !"quantum.Their.Attribute\00", double 2.1} }
```

Standard LLVM Metadata

Debugging Information

Compilers are strongly urged to follow the recommendations in Source Level Debugging with LLVM.

Branch Prediction

Compilers are strongly urged to follow the recommendations in LLVM Branch Weight Metadata.

Other Compiler-Generated Metadata

If the QIR includes a function that is the quantum entry point, it should be marked with an LLVM "Entry-Point" attribute.

Discussion

> It is likely that there is other useful information that could be represented as LLVM metadata in QIR. We anticipate that this will become clearer through use.

Executable Code Generation

There are several areas where a code generator may want to significantly deviate from a simple rewrite of basic intrinsics to target machine code:

- The intermediate representation assumes that the runtime does not perform garbage collection, and thus carefully tracks stack versus heap allocation and reference counting for heap-allocated structures. A runtime that provides full garbage collection may wish to remove the reference count field from several intermediate representation structures and elide calls to the various `unreference` functions.
- Many types are defined as pointers to opaque structures. The code generator will need to either provide a concrete realization of the structure or replace the pointer type with some other representation entirely.
- Depending on the characteristics of the target architecture, the code generator may prefer to use different representations for the various types given concrete types here. For instance, on some architectures it will make more sense to represent small types as bytes rather than as single or double bits.
- The primitive quantum operations provided by a particular target architecture may differ significantly from the intrinsics defined in this specification. It is expected that code generators will significantly rewrite sequences of quantum intrinsics into sequences that are optimal for the specific target.

Quantum Intermediate Representation: Profiles

A *profile* of a specification is a subset of the specification that defines a coherent set of functionalities and capabilities that might be offered by a system. Defining profiles allows a specification to be forward-looking and expansive while implementations are not yet capable of meeting the full specification. Profiles also provide a roadmap for implementors by specifying stages they can progress through that will be useful to consumers of the specification.

This document drafts two initial profiles of the QIR specification. We expect a more comprehensive specification for these and additional profiles to be added in the future.

Profile A: Basic Quantum Functionality

The *Basic Quantum Functionality* profile defines a minimal subset of the QIR that includes quantum operations but explicitly rules out any decision making or "fast feedback" based on measurement results.

In terms of the intermediate representation, this translates into the following restrictions:

- Values of type `%Result` may be stored in memory, stored as part of a tuple or as an element of an array, or returned from an operation or function. No other actions may be performed with them. In particular, they may not be compared against other `%Result` values or converted into values of any other type. Note that this implies that control flow cannot be based on the result of a measurement.
- Once a qubit is measured, nothing further will be done with it other than releasing it.

- No arithmetic or other calculations may be performed with classical values. Any such computations in the original source code are performed in the service before passing the QIR to the target and the results folded in as constants in the QIR.
- The only LLVM primitives allowed are: `call`, `bitcast`, `getelementptr`, `load`, `store`, `ret`, and `extractvalue`.
- The only QIR runtime functions allowed are: [**to be determined**].
- LLVM functions will always be passed null pointers for the capture tuple.
- The only classical value types allowed are `%Int`, `%Double`, `%Result`, `%Pauli`, and tuples of these values.
- The argument tuple passed to an operation will be a tuple of the above types.
- Outside of the argument tuple, values of types other than `%Result` will only appear as literals.

Profile B: Basic Measurement Feedback

The *Basic Measurement Feedback* profile expands on the Basic Quantum Functionality profile to allow limited capabilities to control the execution of quantum operations based on prior measurement results. These capabilities correspond roughly to what are commonly known as "binary controlled gates".

In terms of the intermediate representation, this translates into the following restrictions:

- Comparison of `%Result` values is allowed, but only to compute the input to a conditional branch.
- Boolean computations are not allowed. Boolean expressions on `%Result` comparisons must be represented by a sequence of simple comparisons and branches. Effectively, complex "if" clauses must be translated into embedded simple "if"s.
- Any basic blocks whose execution depends on the result of a `%Result` comparison may only include calls to quantum operations, comparisons of `%Result`s, and branches. In particular, classical arithmetic and other purely classical operations may not be performed inside of such a basic block.
- Basic blocks that depend on the result of a `%Result` comparison may not form a cycle (loop).
- No arithmetic or other calculations may be performed with classical values. Any such computations in the original source code are performed in the service before passing the QIR to the target and the results folded in as constants in the QIR.
- The only LLVM primitives allowed are: `call`, `bitcast`, `getelementptr`, `load`, `store`, `ret`, `extractvalue`, `icmp`, `alloca`, and `br`.
- The only QIR runtime functions allowed are: [**to be determined**].
- LLVM functions will always be passed a null pointer for the capture tuple.
- LLVM functions may fill in the result tuple. If they do, the result tuple may only contain `%Result` values or tuples of `%Result` values.
- The only classical value types allowed are `%Int`, `%Double`, `%Result`, `%Pauli`, `%Unit`, and tuples of these values.
- The argument tuple passed to an operation will be a tuple of the above types.
- Outside of the argument tuple, values of types other than `%Result` will only appear as literals.

Appendix: Library Reference

This table lists all of the runtime functions specified by QIR:

Function	Signature	Description
__quantum__rt__array_concatenate	`%Array* (%Array*, %Array*)`	Returns a new array which is the concatenation of the two passed-in one-dimensional arrays. If either array is not one-dimensional or if the array element sizes are not the same, then a runtime failure should occur.

Function	Signature	Description
__quantum__rt__array_copy	`%Array* (%Array*, i1)`	Creates a shallow copy of the array if the alias count is larger than 0 or the second argument is `true`. Returns the given array pointer (the first parameter) otherwise, after increasing its reference count by 1. The reference count of the array elements remains unchanged.
__quantum__rt__array_create	`%Array* void(i32, i32, i64*)`	Creates a new array. The first `i32` is the size of each element in bytes. The second `i32` is the dimension count. The `i64*` should point to an array of `i64`s contains the length of each dimension. The bytes of the new array should be set to zero. If any length is zero, the result should be an empty array with the given number of dimensions.
__quantum__rt__array_create_1d	`%Array* void(i32, i64)`	Creates a new 1-dimensional array. The `i32` is the size of each element in bytes. The `i64` is the length of the array. The bytes of the new array should be set to zero. If the length is zero, the result should be an empty 1-dimensional array.
__quantum__rt__array_get_dim	`i32(%Array*)`	Returns the number of dimensions in the array.
__quantum__rt__array_get_element_ptr	`i8*(%Array*, i64*)`	Returns a pointer to the indicated element of the array. The `i64*` should point to an array of `i64`s that are the indices for each dimension.
__quantum__rt__array_get_element_ptr_1d	`i8*(%Array*, i64)`	Returns a pointer to the element of the array at the zero-based index given by the `i64`.
__quantum__rt__array_get_size	`i64(%Array*, i32)`	Returns the length of a dimension of the array. The `i32` is the zero-based dimension to return the length of; it must be smaller than the number of dimensions in the array.
__quantum__rt__array_get_size_1d	`i64(%Array*)`	Returns the length of a 1-dimensional array.

Function	Signature	Description
__quantum__rt__array_project	`%Array* (%Array*, i32, i64, i1)`	Creates and returns an array that is a projection of an existing array. The projection may be accessing the same memory as the given array unless its alias count is larger than 0 or the last argument is `true`. The `i32` indicates which dimension the projection is on, and the `i64` specifies the index in that dimension to project. The reference count of all array elements remains unchanged. If the existing array is one-dimensional then a runtime failure should occur.
__quantum__rt__array_slice	`%Array* (%Array*, i32, %Range, i1)`	Creates and returns an array that is a slice of an existing array. The slice may be accessing the same memory as the given array unless its alias count is larger than 0 or the last argument is `true`. The `i32` indicates which dimension the slice is on, and must be smaller than the number of dimensions in the array. The `%Range` specifies the indices in that dimension that should be the elements of the returned array. The reference count of the elements remains unchanged.
__quantum__rt__array_slice_1d	`%Array* (%Array*, %Range, i1)`	Creates and returns an array that is a slice of an existing 1-dimensional array. The slice may be accessing the same memory as the given array unless its alias count is larger than 0 or the last argument is `true`. The `%Range` specifies the indices that should be the elements of the returned array. The reference count of the elements remains unchanged.
__quantum__rt__array_update_alias_count	`void(%Array*, i32)`	Adds the given integer value to the alias count for the array. Fails if either count becomes negative. The call should be ignored if the given `%Array*` is a null pointer.

Function	Signature	Description
__quantum__rt__array_update_reference_count	`void(%Array*, i32)`	Adds the given integer value to the reference count for the array. Deallocates the array if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Array*` is a null pointer.
__quantum__rt__bigint_add	`%BigInt* (%BigInt*, %BigInt*)`	Adds two big integers and returns their sum.
__quantum__rt__bigint_bitand	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-AND of two big integers.
__quantum__rt__bigint_bitnot	`%BigInt* (%BigInt*)`	Returns the bitwise complement of the big integer.
__quantum__rt__bigint_bitor	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-OR of two big integers.
__quantum__rt__bigint_bitxor	`%BigInt* (%BigInt*, %BigInt*)`	Returns the bitwise-XOR of two big integers.
__quantum__rt__bigint_create_array	`%BigInt*(i32, i8*)`	Creates a big integer with the value specified by the `i8` array. The 0-th element of the array is the highest-order byte, followed by the first element, etc.
__quantum__rt__bigint_create_i64	`%BigInt*(i64)`	Creates a big integer with the specified initial value.
__quantum__rt__bigint_divide	`%BigInt* (%BigInt*, %BigInt*)`	Divides the first big integer by the second and returns their quotient.
__quantum__rt__bigint_equal	`i1(%BigInt*, %BigInt*)`	Returns true if the two big integers are equal, false otherwise.
__quantum__rt__bigint_get_data	`i8*(%BigInt*)`	Returns a pointer to the `i8` array containing the value of the big integer.
__quantum__rt__bigint_get_length	`i32(%BigInt*)`	Returns the length of the `i8` array that represents the big integer value.
__quantum__rt__bigint_greater	`i1(%BigInt*, %BigInt*)`	Returns true if the first big integer is greater than the second, false otherwise.
__quantum__rt__bigint_greater_eq	`i1(%BigInt*, %BigInt*)`	Returns true if the first big integer is greater than or equal to the second, false otherwise.
__quantum__rt__bigint_modulus	`%BigInt* (%BigInt*, %BigInt*)`	Returns the first big integer modulo the second.
__quantum__rt__bigint_multiply	`%BigInt* (%BigInt*, %BigInt*)`	Multiplies two big integers and returns their product.

Function	Signature	Description
__quantum__rt__bigint_negate	`%BigInt* (%BigInt*)`	Returns the negative of the big integer.
__quantum__rt__bigint_power	`%BigInt* (%BigInt*, i32)`	Returns the big integer raised to the integer power. As with standard integers, the exponent must fit in 32 bits.
__quantum__rt__bigint_shiftleft	`%BigInt* (%BigInt*, i64)`	Returns the big integer arithmetically shifted left by the (positive) integer amount of bits.
__quantum__rt__bigint_shiftright	`%BigInt* (%BigInt*, i64)`	Returns the big integer arithmetically shifted right by the (positive) integer amount of bits.
__quantum__rt__bigint_subtract	`%BigInt* (%BigInt*, %BigInt*)`	Subtracts the second big integer from the first and returns their difference.
__quantum__rt__bigint_to_string	`%String* (%BigInt*)`	Returns a string representation of the big integer.
__quantum__rt__bigint_update_reference_count	`void(%BigInt*, i32)`	Adds the given integer value to the reference count for the big integer. Deallocates the big integer if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%BigInt*` is a null pointer.
__quantum__rt__bool_to_string	`%String*(i1)`	Returns a string representation of the Boolean.
__quantum__rt__callable_copy	`%Callable* (%Callable*, i1)`	Creates a shallow copy of the callable if the alias count is larger than 0 or the second argument is `true`. Returns the given callable pointer (the first parameter) otherwise, after increasing its reference count by 1. The reference count of the capture tuple remains unchanged. If the `%Callable*` parameter is null, a runtime failure should occur.
__quantum__rt__callable_create	`%Callable*([4 x void (%Tuple*, %Tuple*, %Tuple*)*]*, [2 x void(%Tuple*, i32)]*, %Tuple*)`	Initializes the callable with the provided function table, memory management table, and capture tuple. The memory management table pointer and the capture tuple pointer should be null if there is no capture.
__quantum__rt__callable_invoke	`void(%Callable*, %Tuple*, %Tuple*)`	Invokes the callable with the provided argument tuple and fills in the result tuple. The `%Tuple*` parameters may be null if the callable either takes no arguments or returns `Unit`. If the `%Callable*` parameter is null, a runtime failure should occur.

Function	Signature	Description
__quantum__rt__callable_make_adjoint	`void(%Callable*)`	Updates the callable by applying the Adjoint functor. If the `%Callable*` parameter is null or if the corresponding entry in the callable's function table is null, a runtime failure should occur.
__quantum__rt__callable_make_controlled	`void(%Callable*)`	Updates the callable by applying the Controlled functor. If the `%Callable*` parameter is null or if the corresponding entry in the callable's function table is null, a runtime failure should occur.
__quantum__rt__callable_update_alias_count	`void(%Callable*, i32)`	Adds the given integer value to the alias count for the callable. Fails if the count becomes negative. The call should be ignored if the given `%Callable*` is a null pointer.
__quantum__rt__callable_update_reference_count	`void(%Callable*, i32)`	Adds the given integer value to the reference count for the callable. Deallocates the callable if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Callable*` is a null pointer.
__quantum__rt__capture_update_alias_count	`void(%Callable*, i32)`	Invokes the function at index 1 in the memory management table of the callable with the capture tuple and the given 32-bit integer. Does nothing if the memory management table pointer or the function pointer at that index is null, or if the given `%Callable*` is a null pointer.
__quantum__rt__capture_update_reference_count	`void(%Callable*, i32)`	Invokes the function at index 0 in the memory management table of the callable with the capture tuple and the given 32-bit integer. Does nothing if if the memory management table pointer or the function pointer at that index is null, or if the given `%Callable*` is a null pointer.
__quantum__rt__double_to_string	`%String*(Double)`	Returns a string representation of the double.
__quantum__rt__fail	`void(%String*)`	Fail the computation with the given error message.
__quantum__rt__int_to_string	`%String*(i64)`	Returns a string representation of the integer.

Function	Signature	Description
__quantum__rt__message	`void(%String*)`	Include the given message in the computation's execution log or equivalent.
__quantum__rt__pauli_to_string	`%String*(%Pauli)`	Returns a string representation of the Pauli.
__quantum__rt__qubit_allocate	`%Qubit*()`	Allocates a single qubit.
__quantum__rt__qubit_allocate_array	`%Array*(i64)`	Creates an array of the given size and populates it with newly-allocated qubits.
__quantum__rt__qubit_release	`void(%Qubit*)`	Releases a single qubit. Passing a null pointer as argument should cause a runtime failure.
__quantum__rt__qubit_release_array	`void(%Array*)`	Releases an array of qubits; each qubit in the array is released, and the array itself is unreferenced. Passing a null pointer as argument should cause a runtime failure.
__quantum__rt__qubit_to_string	`%String*(%Qubit*)`	Returns a string representation of the qubit.
__quantum__rt__range_to_string	`%String*(%Range)`	Returns a string representation of the range.
__quantum__rt__result_equal	`i1(%Result*, %Result*)`	Returns true if the two results are the same, and false if they are different. If a `%Result*` parameter is null, a runtime failure should occur.
__quantum__rt__result_get_one	`%Result*()`	Returns a constant representing a measurement result one.
__quantum__rt__result_get_zero	`%Result*()`	Returns a constant representing a measurement result zero.
__quantum__rt__result_to_string	`%String* (%Result*)`	Returns a string representation of the result.
__quantum__rt__result_update_reference_count	`void(%Result*, i32)`	Adds the given integer value to the reference count for the result. Deallocates the result if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Result*` is a null pointer.
__quantum__rt__string_concatenate	`%String* (%String*, %String*)`	Creates a new string that is the concatenation of the two argument strings. If a `%String*` parameter is null, a runtime failure should occur.
__quantum__rt__string_create	`%String*(i8*)`	Creates a string from an array of UTF-8 bytes. The byte array is expected to be zero-terminated.
__quantum__rt__string_equal	`i1(%String*, %String*)`	Returns true if the two strings are equal, false otherwise. If a `%String*` parameter is null, a runtime failure should occur.

Function	Signature	Description
__quantum__rt__string_get_data	`i8*(%String*)`	Returns a pointer to the zero-terminated array of UTF-8 bytes.
__quantum__rt__string_get_length	`i32(%String*)`	Returns the length of the byte array that contains the string data.
__quantum__rt__string_update_reference_count	`void(%String*, i32)`	Adds the given integer value to the reference count for the string. Deallocates the string if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%String*` is a null pointer.
__quantum__rt__tuple_copy	`%Tuple* (%Tuple*, i1)`	Creates a shallow copy of the tuple if the alias count is larger than 0 or the second argument is `true`. Returns the given tuple pointer (the first parameter) otherwise, after increasing its reference count by 1. The reference count of the tuple elements remains unchanged. If the `%Tuple*` parameter is null, a runtime failure should occur.
__quantum__rt__tuple_create	`%Tuple*(i64)`	Allocates space for a tuple requiring the given number of bytes, sets the reference count to 1 and the alias count to 0.
__quantum__rt__tuple_update_alias_count	`void(%Tuple*, i32)`	Adds the given integer value to the alias count for the tuple. Fails if the count becomes negative. The call should be ignored if the given `%Tuple*` is a null pointer.
__quantum__rt__tuple_update_reference_count	`void(%Tuple*, i32)`	Adds the given integer value to the reference count for the tuple. Deallocates the tuple if the reference count becomes 0. The behavior is undefined if the reference count becomes negative. The call should be ignored if the given `%Tuple*` is a null pointer.