netqasm

Release 0.13.2.dev22

QuTech

Oct 10, 2023

INSTALLATION

CHAPTER

ONE

INSTALLATION

1.1 Using pip

To install netqasm do

```
pip install netqasm
```

To simulate a set of application files using a simulator, additional packages are required. Currently supported simulators are:

- SquidASM: Requires `netsquid` and SquidASM installed. For how to install `netsquid`, see https://netsquid.org/#registration. SquidASM can be installed using pip while supplying your NetSquid account information:

  ```
  pip install squidasm --extra-index-url=https://{netsquid-user-name}:
  ↪{netsquid-password}@pypi.netsquid.org
  ```

- SimulaQron: Requires only `simulaqron` which can be installed using pip by

  ```
  pip install simulaqron
  ```

  SimulaQron itself needs a backend to be installed in order to run simulations. One of the supported backends is ProjectQ. To install ProjectQ, do:

  ```
  pip install projectq
  ```

Note: If you have trouble installing one of the packages above, it may be that you first need to install the *wheel* package, by `pip install wheel`.

1.2 From source

To install netqasm from source, clone this repo and run

```
make install
```

To verify the installation, do:

```
make verify
```

CHAPTER

TWO

GETTING STARTED

In this guide you will learn how to quickly get started using the NetQASM SDK and to run your applications on a simulator of a quantum network. Before continuing, make sure you have followed the *Installation*. We will be using the simulator SquidASM, (see *Installation*). To make sure that you're ready to go run

```
python3 -c "import squidasm; print(squidasm.__version__)"
```

In the next part, *Running your first app*, we will run our first app.

2.1 Running your first app

Before writing our own application we will see how we can execute a pre-defined one on the simulator SquidASM. Make sure you first followed the previous parts and have the relevant packages installed.

2.1.1 Creating an application folder

We'll first create an applicaton folder using a template. Do this by running:

```
netqasm new my-app
```

You can replace `my-app` with another name. What's important is that this directiory cannot exist in your current working directory, since it will be created. Go into the directory and see what files were created.

```
cd my-app
ls
```

You will see two files `app_sender.py` and `app_receiver.py`. These are the files defining the application itself. There are two parties in this application: `sender` and `receiver`, but you can have an arbitary number of parties. Each party is associated with a specific node in the network by the *roles.yaml* file. Furthermore, the network is specified in the file *network.yaml*. Lastly, the `sender.yaml` and `receiver.yaml` defines input to the application. For more details about these files and how to configure them, see *Application file structure*.

2.1.2 Running the app

Before describing what the application does and how you can change it, let's just run it and see what happens. To do this, type

```
netqasm simulate
```

which, if everything went well, will print some information about a `sender` and a `receiver`. What this application in fact does is to teleport a qubit from the `sender` to the `receiver`. We will go through how this works more in detail in *Using the SDK*. Amazing, you have just teleported a qubit over a simulated quantum network! What's perhaps even more exciting is that you will soon be able to execute this application without changing the files on real quantum hardware through the Quantum Network Explorer.

To see what options the CLI tool takes, do:

```
netqasm simulate --help
```

You can for example increase the amount of logging shown by `--log-level=INFO` (or `DEBUG`) or use a different qubit representation in the simulation by `--formalism=dm` to use the density matrix formalism in NetSquid. As described below you can also use a different simulator.

2.1.3 Inspecting the results

You may have seen that after running the application a new directory has appeared: `log`. This is were all the results of the simulation get stored. Each execution of the application creates a new directory in `log` using a timestamp. For convenience, a copy of this folder is also created with the name `LAST`. The files in the log-folder contain information what happened during the simulation: what quantum operations were applied, what classical messages were sent, what entangled pairs were created and what the outcome of the application was. This is further detailed in *Application file structure*.

2.1.4 Other pre-defined apps

You have just simulated an application which teleports a qubit. In *Using the SDK* you will learn how to write your own application. However, before that, there are also other pre-defined applications you can easily instantiate and execute using the `--template` flag to `netqasm new`. Type `netqasm new --help` for a complete list. To for example try out an application which performs anonymous transmission of a qubit, do:

```
netqasm new my-anonymous-app --template=anonymous_transmission
```

What actually happens under the hood here is that the source for the example applications here is copied into your newly created app. You can also just clone the *netqasm* repository and execute these examples, e.g. with

```
netqasm simulate --app-dir netqasm/examples/apps/bb84
```

2.1.5 Using other simulators

Above we have simulated an application which teleports a qubit over a quantum network using the simulator SquidASM. You can easily run the same application using another supported simulator. For example to use SimulaQron instead, simply do:

```
netqasm simulate --simulator=simulaqron
```

Note: For this to work you need simulaqron installed, otherwise the CLI will tell you that `ModuleNotFoundError: to use simulaqron as simulator, `simulaqron` needs to be installed`. SimulaQron can be installed using pip by

```
pip install simulaqron
```

2.2 Using the SDK

In *Running your first app* we have seen how to execute applications on a simulated quantum network. We will now learn how to write an application ourselves. Below are a few sample applications which we will go through, ranging from a very simple application on a single node to a more complicated examples further on.

2.2.1 Writing a first application

We will now create an very simple *hello-world*-application which consists of a single node that creates a qubit, performs a Hadamard gate and measures the qubit.

Let's first create a new folder `my-app`:

```
mkdir my-app
cd my-app
```

In this folder we will now create our application

```
touch app_alice.py
```

Open the file `app_alice.py` in your favorite editor/IDE and add the following code

Listing 1: app_alice.py

```
from netqasm.sdk.external import NetQASMConnection


def main(app_config=None):
    # Setup a connection to QNodeOS
    with NetQASMConnection("alice", log_config=app_config.log_config) as alice:
        print("Started NetQASM connection to QNodeOS")
```

Note: The reason `NetQASMConnection` is imported from `netqasm.sdk.external` is that the class used will depend on which simulator is specified. The information about which simulator is used is stored in the environement varible `NETQASM_SIMULATOR`, and makes `netqasm.sdk.external` load different classes depending on its

value. What this allows us to do is to simulate an application on different simulators, without changing anything, even imports.

What this code does is to setup a connection to the underlying (simulated) quantum node controller called `QNodeOS`, which can handle the NetQASM-instructions. You can already run this application by doing `netqasm simulate`, and if everything was correct you will see the message being printed.

In the context of the connection, we can now create a qubit

Listing 2: app_alice.py

```
from netqasm.sdk.external import NetQASMConnection
from netqasm.sdk import Qubit


def main(app_config=None):
    # Setup a connection to QNodeOS
    with NetQASMConnection("alice", log_config=app_config.log_config) as alice:
        # Create a qubit
        q = Qubit(alice)
```

Running the application now seems perhaps to not do anything. To make sure that a qubit is actually created we can set the log-level to `INFO`

```
netqasm simulate --log-level=DEBUG
```

You will then see that a subroutine is flushed and handled by the `SubroutineHandler`, which is a simplified version of a `QNodeOS` used in simulation.

Let's now perform a gate on the qubit and also measure it.

Listing 3: app_alice.py

```
from netqasm.sdk.external import NetQASMConnection
from netqasm.sdk import Qubit


def main(app_config=None):
    # Setup a connection to QNodeOS
    with NetQASMConnection("alice", log_config=app_config.log_config) as alice:
        # Create a qubit
        q = Qubit(alice)
        # Perform a Hadamard gate
        q.H()
        # Measure the qubit
        m = q.measure()
        # Print the outcome
        print(f"Outcome is: {m}")
```

Let's run this now (without setting the `--log-level`-flag) and see what the outcome is. Hmm, it doesn't print the outcome but rather says:

```
Outcome is: Future to be stored in array with address 0 at index 0.
To access the value, the subroutine must first be executed which can be done by␣
↪flushing.
```

The reason this happens is because the operations specified are in fact not directly executed on the (simulated) quantum hardware. Rather, they are buffered into a `NetQASM`-subroutine, until the subroutine is flushed and sent to `QNodeOS`.

Let's fix our code by adding an explicit `flush` before the `print`.

Listing 4: app_alice.py

```
from netqasm.sdk.external import NetQASMConnection
from netqasm.sdk import Qubit


def main(app_config=None):
    # Setup a connection to QNodeOS
    with NetQASMConnection("alice", log_config=app_config.log_config) as alice:
        # Create a qubit
        q = Qubit(alice)
        # Perform a Hadamard gate
        q.H()
        # Measure the qubit
        m = q.measure()
        # Flush the current subroutine
        alice.flush()
        # Print the outcome
        print(f"Outcome is: {m}")
```

Running the application again will now either print `Outcome is: 0` or `Outcome is: 1`. Run it a few times to see the different outcomes.

Note: A connection is automatically flushed whenever it goes out of scope. So in the above example we could have just as well done:

Listing 5: app_alice.py

```
from netqasm.sdk.external import NetQASMConnection
from netqasm.sdk import Qubit


def main(app_config=None):
    # Setup a connection to QNodeOS
    with NetQASMConnection("alice", log_config=app_config.log_config) as alice:
        # Create a qubit
        q = Qubit(alice)
        # Perform a Hadamard gate
        q.H()
        # Measure the qubit
        m = q.measure()
    # Print the outcome
    print(f"Outcome is: {m}")
```

Tip: It is important to understand how the execution happens when running the application. Try adding some print statements in the application, turn on `INFO`-logging and see if you can understand why the print-statements and logging-statements are in the order you see.

2.2.2 Creating entanglement between nodes

Let's now extend our application by adding another node `bob` and have the two nodes create entanglement with each other.

To do this we will need to setup an EPR socket. We do this by instanciating an object of *`EPRSocket`* and give this to the `NetQASMConnection`. Consider the following code-example for the node with role `alice`:

Listing 6: app_alice.py

```
from netqasm.sdk.external import NetQASMConnection
from netqasm.sdk import EPRSocket


def main(app_config=None):
    # Specify an EPR socket to bob
    epr_socket = EPRSocket("bob")

    alice = NetQASMConnection(
        "alice",
        log_config=app_config.log_config,
        epr_sockets=[epr_socket],
    )
    with alice:
        # Create an entangled pair using the EPR socket to bob
        q_ent = epr_socket.create()[0]
        # Measure the qubit
        m = q_ent.measure()
    # Print the outcome
    print(f"alice's outcome is: {m}")
```

The code for `bob` will be very similar, with the only difference being that `bob` *receives* an entangled pair by calling `recv` on the EPR socket object. Create a new file `app_bob.py` in the same directory as `app_alice.py`:

Listing 7: app_bob.py

```
from netqasm.sdk.external import NetQASMConnection
from netqasm.sdk import EPRSocket


def main(app_config=None):
    # Specify an EPR socket to bob
    epr_socket = EPRSocket("alice")

    bob = NetQASMConnection(
        "bob",
        log_config=app_config.log_config,
        epr_sockets=[epr_socket],
    )
    with bob:
        # Receive an entangled pair using the EPR socket to alice
        q_ent = epr_socket.recv()[0]
        # Measure the qubit
        m = q_ent.measure()
    # Print the outcome
    print(f"bob's outcome is: {m}")
```

Running this application files using `netqasm simulate` prints the outcomes of the two nodes. Since by default no

noise is used, their outcomes will always be equal.

Tip: Check out the documentation of *EPRSocket* to see what arguments *create()* and *recv()* can take. For example you will see that a number of pairs can be specified, which is why these methods return a list of *Qubit*-objects. Also check out the methods *create_context()* and *recv_context()*, which allows to specify what to do whenever a pair is generated, using a context.

2.2.3 Adding classical communication

Applications generally also need to communicate classicaly between nodes, to for example communicate measurement outcomes. We will extend our example by having `alice` communicate her outcome to `bob`. `bob` will use this outcome to possible apply a correction in order to make his qubit be in the state $|0\rangle$ in both cases. Consider the following code-snippets for `alice` and `bob`:

Listing 8: app_alice.py

```
from netqasm.sdk.external import NetQASMConnection, Socket
from netqasm.sdk import EPRSocket


def main(app_config=None):
    # Setup a classical socket to bob
    socket = Socket("alice", "bob", log_config=app_config.log_config)

    # Specify an EPR socket to bob
    epr_socket = EPRSocket("bob")

    alice = NetQASMConnection(
        "alice",
        log_config=app_config.log_config,
        epr_sockets=[epr_socket],
    )
    with alice:
        # Create an entangled pair using the EPR socket to bob
        q_ent = epr_socket.create()[0]
        # Measure the qubit
        m = q_ent.measure()
    # Print the outcome
    print(f"alice's outcome is: {m}")

    # Send the outcome to bob
    socket.send(str(m))
```

Listing 9: app_bob.py

```
from netqasm.sdk.external import NetQASMConnection, Socket
from netqasm.sdk import EPRSocket


def main(app_config=None):
    # Setup a classical socket to alice
    socket = Socket("bob", "alice", log_config=app_config.log_config)

    # Specify an EPR socket to bob
```

(continues on next page)

(continued from previous page)

```
    epr_socket = EPRSocket("alice")

    bob = NetQASMConnection(
        "bob",
        log_config=app_config.log_config,
        epr_sockets=[epr_socket],
    )
    with bob:
        # Receive an entangled pair using the EPR socket to alice
        q_ent = epr_socket.recv()[0]

        # Receive the outcome from alice
        m = int(socket.recv())

        # Apply correction depending on outcome
        if m == 1:
            q_ent.X()

        # Measure the qubit
        m = q_ent.measure()

    # Print the outcome
    print(f"bob's outcome is: {m}")
```

Running the above example we can see that the outcome of `bob` is always 0, independently of the outcome of `alice`.

2.2.4 A more complex example

We will now look at a more complicated example, where we will use quantum error correction to protect an entangled qubit from errors. In this example we will use the most simple quantum error-correction code, namely the repition code on three qubits. Before implementing the actual quantum error-correction code, let's define how we want the main functions to look like. On `alice`'s side we will

1. Create encode the qubit
2. Randomly apply a bit-flip
3. Correct any error
4. Decode the qubit again.
5. Measure the qubit and print the outcome

Listing 10: app_alice.py

```
def main(app_config=None):
    # Specify an EPR socket to bob
    epr_socket = EPRSocket("bob")

    alice = NetQASMConnection(
        "alice",
        log_config=app_config.log_config,
        epr_sockets=[epr_socket],
    )
    with alice:
        # Create an entangled pair using the EPR socket to bob
        q_ent = epr_socket.create()[0]
```

(continues on next page)

(continued from previous page)

```
        # Encode into repitition code
        logical_qubit = encode(q_ent)

        # Randomly introduce a bit-flip
        if random.randint(0, 1):
            i = random.choice(range(3))
            print(f"applying bit flip on qubit {i}")
            # q = random.choice(logical_qubit)
            q = logical_qubit[i]
            q.X()

        # Correct a possible bit-flip
        correct(logical_qubit)

        # Decode back
        decode(logical_qubit)

        # Measure the qubit
        m = logical_qubit[0].measure()

    # Print the outcome
    print(f"alice's outcome is: {m}")
```

`bob` on the other hand will simple measure his entangled qubit and print the outcome.

Listing 11: app_bob.py

```
def main(app_config=None):
    # Specify an EPR socket to bob
    epr_socket = EPRSocket("alice")

    bob = NetQASMConnection(
        "bob",
        log_config=app_config.log_config,
        epr_sockets=[epr_socket],
    )
    with bob:
        # Receive an entangled pair using the EPR socket to alice
        q_ent = epr_socket.recv()[0]

        # Measure the qubit
        m = q_ent.measure()

    # Print the outcome
    print(f"bob's outcome is: {m}")
```

Let's now implement the functions: `encode`, `correct` and `decode`:

Listing 12: app_alice.py

```
import random

from netqasm.sdk.external import NetQASMConnection, Socket
from netqasm.sdk import EPRSocket, Qubit, parity_meas, t_inverse, toffoli_gate

def encode(qubit):
```

(continues on next page)

(continued from previous page)

```
    """Encodes a qubit into a repitition code by intializing two more

    Parameters
    ----------
    qubit : :class:`~.Qubit`
        Qubit to be encoded

    Returns
    -------
    list : list of encoded qubits
    """
    conn = qubit.connection
    logical_qubit = [qubit, Qubit(conn), Qubit(conn)]
    for q in logical_qubit[1:]:
        logical_qubit[0].cnot(q)
    return logical_qubit
```

Listing 13: app_alice.py

```
def correct(logical_qubit):
    """Tries to correct a bit flip

    Parameters
    ----------
    logical_qubit : list of :class:`~.Qubit`
    """
    # Check code syndromes
    s1 = parity_meas(logical_qubit, 'ZZI')
    s2 = parity_meas(logical_qubit, 'IZZ')

    print(f"syndrome is ({s1}, {s2})")
    if (s1, s2) == (0, 0):  # No error
        pass
    elif (s1, s2) == (0, 1):  # Error on third
        logical_qubit[2].X()
    elif (s1, s2) == (1, 0):  # Error on first qubit
        logical_qubit[0].X()
    else:  # Error on second qubit
        logical_qubit[1].X()
```

Listing 14: app_alice.py

```
def decode(logical_qubit):
    """Decodes the repitition code on three qubits
    After the first qubit in the list will be the decode qubit.

    Parameters
    ----------
    logical_qubit : list of :class:`~.Qubit
    """
    for q in logical_qubit[1:]:
        logical_qubit[0].cnot(q)
        # Toffoli with first qubit as target
        toffoli_gate(*reversed(logical_qubit))
```

Let's now run our application and see what happens. Hmm, we get an error.

Tip: Try to figure out what goes wrong before reading the solution below.

Our mistake is that we are trying to use the outcomes `s1` and `s2`, in the `correct`-function, before the subroutine is flushed. One way to solve this is to add a `flush`-statement as follows:

Listing 15: app_alice.py

```
def correct(logical_qubit):
    """Tries to correct a bit flip

    Parameters
    ----------
    logical_qubit : list of :class:`~.Qubit`
    """
    # Check code syndromes
    s1 = parity_meas(logical_qubit, 'ZZI')
    s2 = parity_meas(logical_qubit, 'IZZ')

    conn = logical_qubit[0].connection
    conn.flush()

    print(f"syndrome is ({s1}, {s2})")
    if (s1, s2) == (0, 0):  # No error
        pass
    elif (s1, s2) == (0, 1):  # Error on third
        logical_qubit[2].X()
    elif (s1, s2) == (1, 0):  # Error on first qubit
        logical_qubit[0].X()
    else:  # Error on second qubit
        logical_qubit[1].X()
```

The application now works :) You can see that independently on which qubit the bit-flip might occur, `alice` and `bob` always receive the same outcome, meaning that our error-correction code is working.

However, there is something we can still improve. Namely, we can avoid the call to `flush` and instead make use of classical logic in `NetQASM` and let this be handled by `QNodeOS`. The reason we would want to do this is that whenever a `flush` happens, extra communication between the `application layer` and `QNodeOS` is needed, see our paper for more details. We can see this happen if we increase the logging. Run the above example by `netqasm simulate --log-level=INFO`, which will produce a lot of logging. Importantly, you can see that

`alice` submits two subroutines to `QNodeOS` and not only one.

In the next part we look at how to use simple built-in classical logic in `NetQASM` to minimize the communication needed between the `application layer` and `QNodeOS`.

2.2.5 Simple classical logic

Let's now improve our `correct`-function above by avoiding the call to flush and use the simple logic built-in to `NetQASM`. We can rewrite the function to instead do:

Listing 16: app_alice.py

```
def correct2(logical_qubit):
    """Tries to correct a bit flip

    Parameters
    ----------
    logical_qubit : list of :class:`~.Qubit`
    """
    # Check code syndromes
    s1 = parity_meas(logical_qubit, 'ZZI')
    s2 = parity_meas(logical_qubit, 'IZZ')

    with s1.if_eq(0):
        with s2.if_eq(1):  # outcomes (0, 1) error on third qubit
            logical_qubit[2].X()
    with s1.if_eq(1):
        with s2.if_eq(0):  # outcomes (1, 0) error on first qubit
            logical_qubit[0].X()
        with s2.if_eq(1):  # outcomes (1, 1) error on second qubit
            logical_qubit[1].X()
```

If you now run the application with the updated function and with `INFO` logging, you will see that `alice` only uses one subroutine. What happens under the hood, is that these if-statements are compiled into branching instructions in the `NetQASM`-language.

Note: The current syntax, e.g. `with s1.if_eq(0):` might change. Ideally, we would be able to write plain Python-if-statements in the future.

Tip: Check out the documentation for *`Future`*. This is what's returned when measuring a qubit and on which one can apply simple logical statements such as *`if_eq()`*. You can also for example use the methods *`if_ez()`* and *`if_nz()`*.

As a next step, you can read more about how to configure the simulation of the application, what network to use, noise etc, in the section *Application file structure*. In *SDK objects* the main functions and classes to be used are documented. For the full API of the package, refer to the API Reference. Enjoy programming applications for a quantum internet!

2.3 Application file structure

Applications are organized into directories: all source code entry points and configuration files relating to an application must be in the same directory.

Quantum network applications typically involve multiple parties, each on a separate node in the network. We refer to these parties as *roles*. For example, a teleportation application might have two roles: a *sender* and a *receiver*.

An application directory consists of the following files:

- **Application source code**. These are Python source code files with names starting with `app_`, e.g. `app_alice.py`. There should be one file for each 'role' involved in the networked application.
- **Application input**. Each role can have a YAML file specifying the inputs to the application. Each role has its own local inputs. Names correspond to the roles in the application. For example, the role 'sender' has an input file `sender.yaml`.
- **Network configuration**. A single YAML file `network.yaml` specifying characteristics of the simulated network. See below for a detailed description of the expected format.
- **Role-network mapping**. Roles are logical concepts related to the application, and not directly tied to a physical (simulated) node. In the `roles.yaml` file you can specify which role (i.e. which `app_<role>.py` file) runs on which network node.

Tip: The `netqasm new` and `netqasm init` commands automatically create the above file types.

2.3.1 File content format

Application source code

Each `app_<role>.py` file should have a `main` function. See for more details *Using the SDK*.

Application input

A `<role>.yaml` file should contain a simply dictionary of inputs. For example:

```
theta: 3.1415
x: 1
```

Network configuration

See the *NetworkConfig* dataclass.

Role-network mapping

The `roles.yaml` file should contain a simple list of key-value pairs. Each key should be a role name and each value a node name occurring in the network configuration.

2.3.2 Results and logs

Running an application produces two kinds of output: **application results** and **log files**.

If it doesn't exist already, a `log` directory is automatically created in the application directory. In it, for each run of the application, a directory is created with a name indicating the time it was generated, e.g. `20201117-104744`. For convenience, the last directory is always copied into a directory called `LAST`, so in most cases you can just look into `log/LAST` for the relevant output.

The output files are the following:

- **Application results**. Application-specific results are written to `results.yaml`.
- **Instruction logs**. The simulator logs which NetQASM instructions it executes on each simulated node to a YAML file. For each `<node>`, a file `<node>_instrs.yaml` is generated.
- **Network log**. Events related to the network and that are not local to one specific node are logged to `network_log.yaml`. These event currently only include entanglement generation events.
- **Classical communciation logs**. For each `<role>`, a file `<role>_class_comm.yaml` is generated with all messages that are sent or received to/from that role.
- **Applicaton logs**. Applications can also log custom information to a file. An 'application log' from a `<role>` ends up in `<role>_app_log.yaml`.

2.3.3 Output file contents

Application results

The `results.yaml` file contains the Python directories that are `return` ed at the end of the `main` function in each role's source code.

Instruction logs

A `<role>_instr.yaml` contains a list of all NetQASM instructions that were executed by `<role>`, in chronological order.

Each log statement includes the instruction type, and the time at which it was executed. It also includes the states of the qubits in the network (across all nodes) and which of these are entangled or not. Entangled qubits appear in the same *qubit group*.

This information is about the states directly *after* the instruction is executed.

See the *InstrLogEntry* dataclass for the format of each log entry.

Network log

The `network_log.yaml` contains a list of all entanglement events that happened in the simulated network, in chronological order. As in the instruction log, the qubit states and groups are given as they are immediately *after* the event.

Two events (called *stages*) exist: `START` (entanglement generation has started) and `FINISH` (entanglement has successfully been generated).

Furtermore, there are two *types* of entanglement generation:

- `MD` (Measure Directly): upon successful generation, immediately measure the two (one in each node) qubits. So, directly after a `FINISH` event of type `MD`, the corresponding qubits do not appear in the qubit information.
- `CK` (Create and Keep): upon successful generation, keep the qubits alive. The corresponding qubits appear in the qubit group information, and are (obviously) entangled.

`START` events do *not* give information about the *type*. This is only given at `FINISH` events.

`MD` events (at the `FINISH` stage) have additional information `BAS` and `MSR`. These are the bases (one for each node) used for the (immediate) mesaurement, and the mesaurement outcomes themselves.

See the *NetworkLogEntry* dataclass for the format of each log entry.

Classical communciation logs

Each `<role>_class_comm.yaml` contains a list of all messages that were sent or received by `<role>`, in chronological order.

See *ClassCommLogEntry* dataclass for the format of each log entry.

Application logs

Each `<role>_app_log.yaml` contains a list of custom log statements coming from `app_<role>.py`.

2.4 SDK objects

Described below are the user-exposed components of the NetQASM SDK.

- *BaseNetQASMConnection*
- `Qubit`
- *EPRSocket*
- Futures:
  - *Future*
  - *Array*
  - *RegFuture*
- *Socket*

Note that *BaseNetQASMConnection* and *Socket* are abstract base-classes. These are implemented specifically for a runtime, e.g. a simulator or hardware runtime. However, when using the SDK this is handled automatically if these are imported from `netqasm.sdk.external`. For example one might write an application as follows

```
from netqasm.sdk.external import NetQASMConnection, Socket

# Setup a classical socket
bob_socket = Socket("alice", "bob")

with NetQASMConnection('alice'):
    # Main application...
```

The classes `NetQASMConnection` and *Socket* will be different subclasses of the abstract classes, depending on which runtime is used. If for example the application is simulated using **`SquidASM`_** and NetSquid, then the `NetQASMConnection` will in fact be the class *squidasm.sdk.NetSquidConnection*. These different classes expose the same set of functionalities. They only differ in the way they communicate with the underlying simulator. For more details see *Running your first app*.

2.4.1 NetQASM connection

Interface to quantum node controllers.

This module provides the *BaseNetQASMConnection* class which represents the connection with a quantum node controller.

class netqasm.sdk.connection.**BaseNetQASMConnection**(*app_name*, *node_name=None*, *app_id=None*, *max_qubits=5*, *hardware_config=None*, *log_config=None*, *epr_sockets=None*, *compiler=None*, *return_arrays=True*, *_init_app=True*, *_setup_epr_sockets=True*)

Base class for representing connections to a quantum node controller.

A BaseNetQASMConnection instance provides an interface for Host programs to interact with a quantum node controller like QNodeOS, which controls the quantum hardware.

The interaction with the quantum node controller includes registering applications, opening EPR sockets, sending NetQASM subroutines, and getting execution results,

A BaseNetQASMConnection instance also provides a 'context' for the Host to run its code in. Code within this context is compiled into NetQASM subroutines and then sent to the quantum node controller.

Parameters

- **app_name** (*str*) –
- **node_name** (*Optional[str]*) –
- **app_id** (*Optional[int]*) –
- **max_qubits** (*int*) –
- **hardware_config** (*Optional[HardwareConfig]*) –
- **log_config** (*Optional[LogConfig]*) –
- **epr_sockets** (*Optional[List[esck.EPRSocket]]*) –
- **compiler** (*Optional[Type[SubroutineTranspiler]]*) –
- **return_arrays** (*bool*) –
- **_init_app** (*bool*) –

- **_setup_epr_sockets** (*bool*) –

__init__(*app_name*, *node_name=None*, *app_id=None*, *max_qubits=5*, *hardware_config=None*, *log_config=None*, *epr_sockets=None*, *compiler=None*, *return_arrays=True*, *_init_app=True*, *_setup_epr_sockets=True*)

BaseNetQASMConnection constructor.

In most cases, you will want to instantiate a subclass of this.

Parameters

- **app_name** (*str*) – Name of the application. Specifically, this is the name of the program that runs on this particular node. So, *app_name* can often be seen as the name of the "role" within the multi-party application or protocol. For example, in a Blind Computation protocol, the two roles may be "client" and "server"; the *app_name* of a particular *BaseNetQASMConnection* instance may then e.g. be "client".
- **node_name** (*Optional[str]*) – name of the Node that is controlled by the quantum node controller that we connect to. The Node name may be different from the *app_name*, and e.g. reflect its geographical location, like a city name. If None, the Node name is obtained by querying the global *NetworkInfo* by using the *app_name*.
- **app_id** (*Optional[int]*) – ID of this application. An application registered in the quantum node controller using this connection will use this app ID. If None, a unique ID will be chosen (unique among possible other applications that were registered through other BaseNetQASMConnection instances).
- **max_qubits** (*int*) – maximum number of qubits that can be in use at the same time by applications registered through this connection. Defaults to 5.
- **hardware_config** (*Optional[HardwareConfig]*) – configuration object with info about the underlying hardware. Used by the Builder of this Connection. When None, a generic configuration object is created with the default qubit count of 5.
- **log_config** (*Optional[LogConfig]*) – configuration object specifying what to log.
- **epr_sockets** (*Optional[List[esck.EPRSocket]]*) – list of EPR sockets. If *_setup_epr_sockets* is True, these EPR sockets are automatically opened upon entering this connection's context.
- **compiler** (*Optional[Type[SubroutineTranspiler]]*) – the class that is used to instantiate the compiler. If None, a compiler is used that can compile for the hardware of the node this connection is to.
- **return_arrays** (*bool*) – whether to add "return array"-instructions at the end of subroutines. A reason to set this to False could be that a quantum node controller does not support returning arrays back to the Host.
- **_init_app** (*bool*) – whether to immediately send a "register application" message to the quantum node controller upon construction of this connection.
- **_setup_epr_sockets** (*bool*) – whether to immediately send "open EPR socket" messages to the quantum node controller upon construction of this connection. If True, the "open EPR socket" messages are for the EPR sockets defined in the *epr_sockets* parameter.

property app_name

Get the application name

Return type str

property node_name

Get the node name

Return type str

property app_id
Get the application ID

Return type int

property network_info

Return type Type[*NetworkInfo*]

property builder

Return type *Builder*

classmethod get_app_ids()

Return type Dict[str, List[int]]

classmethod get_app_names()

Return type Dict[str, Dict[int, str]]

__enter__()
Start a context with this connection.

Quantum operations specified within the connection are automatically compiled into NetQASM subroutines. These subroutines are sent to the quantum node controller, over this connection, when either *flush()* is called or the connection goes out of context which calls __exit__().

```
# Open the connection
with NetQASMConnection(app_name="alice") as alice:
    # Create a qubit
    q = Qubit(alice)
    # Perform a Hadamard
    q.H()
    # Measure the qubit
    m = q.measure()
    # Flush the subroutine to populate the variable `m` with the outcome
    # Alternatively, this can be done by letting the connection
    # go out of context and move the print to after.
    alice.flush()
    print(m)
```

__exit__(*exc_type*, *exc_val*, *exc_tb*)
Called automatically when a connection context ends.

Default behavior is to call the *close* method on the connection.

clear()

Return type None

close(*clear_app=True*, *stop_backend=False*, *exception=False*)
Close a connection.

By default, this method is automatically called when a connection context ends.

Parameters

- **clear_app** (bool) –
- **stop_backend** (bool) –
- **exception** (bool) –

Return type None

property shared_memory
Get this connection's Shared Memory object.

This property should *not* be accessed before any potential setting-up of shared memories has finished. If it cannot be found, an error is raised.

Return type *SharedMemory*

property active_qubits
Get a list of qubits that are currently in use.

"In use" means that the virtual qubit represented by this *Qubit* instance has been allocated and hence its virtual ID cannot be re-used.

Return type List[*Qubit*]

Returns list of active qubits

flush(*block=True*, *callback=None*)
Compile and send all pending operations to the quantum node controller.

All operations that have been added to this connection's Builder (typically by issuing these operations within a connection context) are collected and compiled into a NetQASM subroutine. This subroutine is then sent over the connection to be executed by the quantum node controller.

Parameters

- **block** (bool) – block on receiving the result of executing the compiled subroutine from the quantum node controller.
- **callback** (Optional[Callable]) – if *block* is False, this callback is called when the quantum node controller sends the subroutine results.

Return type None

compile()
Compile the previous SDK commands into a NetQASM subroutine.

This does this the same as calling *flush()*, except it does not send the subroutine to the quantum node controller for execution. This method can hence be used to pre-compile a subroutine and send it later, possibly after filling in concrete values for templates.

Return type Optional[*Subroutine*]

commit_protosubroutine(*protosubroutine*, *block=True*, *callback=None*)
Send a protosubroutine to the quantum node controller.

Takes a *ProtoSubroutine*, i.e. an intermediate representation of the subroutine that comes from the Builder. The ProtoSubroutine is compiled into a *Subroutine* instance.

Parameters

- **protosubroutine** (*ProtoSubroutine*) –
- **block** (bool) –
- **callback** (Optional[Callable]) –

Return type None

commit_subroutine(*subroutine*, *block=True*, *callback=None*)

Parameters

- **subroutine** (*Subroutine*) –

- **block** (bool) –
- **callback** (Optional[Callable]) –

Return type None

block()
Block until a flushed subroutines finishes.

This should be implemented by subclasses.

:raises NotImplementedError

Return type None

new_array(*length=1*, *init_values=None*)
Allocate a new array in the shared memory.

This operation is handled by the connection's Builder. The Builder make sures the relevant NetQASM instructions end up in the subroutine.

Parameters

- **length** (int) – length of the array, defaults to 1
- **init_values** (Optional[List[Optional[int]]]) – list of initial values of the array. If not None, must have the same length as *length*.

Return type *Array*

Returns a handle to the array that can be used in application code

loop(*stop*, *start=0*, *step=1*, *loop_register=None*)
Create a context for code that gets looped.

Each iteration of the loop is associated with an index, which starts at 0 by default. Each iteration the index is increased by *step* (default 1). Looping stops when the index reaches *stop*.

Code inside the context *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

This operation is handled by the connection's Builder. The Builder make sures the NetQASM subroutine contains a loop around the (compiled) code that is inside the context.

Example:

```
with NetQASMConnection(app_name="alice") as alice:
    outcomes = alice.new_array(10)
    with alice.loop(10) as i:
        q = Qubit(alice)
        q.H()
        outcome = outcomes.get_future_index(i)
        q.measure(outcome)
```

Parameters

- **stop** (int) – end of iteration range (exluding)
- **start** (int) – start of iteration range (including), defaults to 0
- **step** (int) – step size of iteration range, defaults to 1
- **loop_register** (Optional[*Register*]) – specific register to be used for holding the loop index. In most cases there is no need to explicitly specify this.

Return type AbstractContextManager[*Register*]

Returns the context object (to be used in a *with ...* expression)

loop_body(*body*, *stop*, *start=0*, *step=1*, *loop_register=None*)
Loop code that is defined in a Python function (body).

The function to loop should have a single argument with that has the *BaseNetQASMConnection* type.

Parameters

- **body** (Callable[[ForwardRef, *RegFuture*], None]) – function to loop
- **stop** (int) – end of iteration range (exluding)
- **start** (int) – start of iteration range (including), defaults to 0
- **step** (int) – step size of iteration range, defaults to 1
- **loop_register** (Union[*Register*, str, None]) – specific register to be used for holding the loop index.

Return type None

loop_until(*max_iterations*)
Create a context with code to be looped until the exit condition is met, or the maximum number of tries has been reached.

The code inside the context is automatically looped (re-run). At the end of each iteration, the exit_condition of the context object is checked. If the condition holds, the loop exits. Otherwise the loop does another iteration. If *max_iterations* iterations have been reached, the loop exits anyway.

Make sure you set the loop_condition on the context object, like e.g.

```
with connection.loop_until(max_iterations=10) as loop:
    q = Qubit(conn)
    m = q.measure()
    constraint = ValueAtMostConstraint(m, 0)
    loop.set_exit_condition(constraint)
```

Parameters **max_iterations** (int) – the maximum number of times to loop

Return type AbstractContextManager[*SdkLoopUntilContext*]

if_eq(*a*, *b*, *body*)
Execute a function if a == b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ne(*a*, *b*, *body*)
Execute a function if a != b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_lt(*a*, *b*, *body*)
Execute a function if a < b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ge(*a*, *b*, *body*)
Execute a function if a > b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ez(*a*, *body*)
Execute a function if a == 0.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_nz(*a*, *body*)
Execute a function if a != 0.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

try_until_success(*max_tries=1*)
TODO docstring

Parameters **max_tries** (int) –

Return type AbstractContextManager[None]

tomography(*preparation*, *iterations*, *progress=True*)
Does a tomography on the output from the preparation specified. The frequencies from X, Y and Z measurements are returned as a tuple (f_X,f_Y,f_Z).

- **Arguments**

  preparation A function that takes a NetQASMConnection as input and prepares a qubit and returns this

  iterations Number of measurements in each basis.

  progress_bar Displays a progress bar

Parameters

- **preparation** (Callable[[*BaseNetQASMConnection*], *Qubit*]) –
- **iterations** (int) –
- **progress** (bool) –

Return type Dict[str, float]

test_preparation(*preparation*, *exp_values*, *conf=2*, *iterations=100*, *progress=True*)
Test the preparation of a qubit. Returns True if the expected values are inside the confidence interval produced from the data received from the tomography function

- **Arguments**

  preparation A function that takes a NetQASMConnection as input and prepares a qubit and returns this

  exp_values The expected values for measurements in the X, Y and Z basis.

  conf Determines the confidence region (+/- conf/sqrt(iterations))

  iterations Number of measurements in each basis.

  progress_bar Displays a progress bar

Parameters

- **preparation** (Callable[[*BaseNetQASMConnection*], *Qubit*]) –
- **exp_values** (Tuple[float, float, float]) –
- **conf** (float) –

- **iterations** (int) –
- **progress** (bool) –

Return type bool

insert_breakpoint(*action*, *role=<BreakpointRole.CREATE: 0>*)

Parameters

- **action** (*BreakpointAction*) –
- **role** (*BreakpointRole*) –

Return type None

2.4.2 Qubit

Qubit representation.

This module contains the *Qubit* class, which are used by application scripts as handles to in-memory qubits.

class netqasm.sdk.qubit.**Qubit**(*conn*, *add_new_command=True*, *ent_info=None*, *virtual_address=None*)

Representation of a qubit that has been allocated in the quantum node.

A *Qubit* instance represents a quantum state that is stored in a physical qubit somewhere in the quantum node. The particular qubit is identified by its virtual qubit ID. To which physical qubit ID this is mapped (at a given time), is handled completely by the quantum node controller and is not known to the *Qubit* itself.

A *Qubit* object can be instantiated in an application script. Such an instantiation is automatically compiled into NetQASM instructions that allocate and initialize a new qubit in the quantum node controller.

A *Qubit* object may also be obtained by SDK functions that return them, like the *create()* method on an *EPRSocket*, which returns the object as a handle to the qubit that is now entangled with one in another node.

Qubit operations like applying gates and measuring them are done by calling methods on a *Qubit* instance.

Parameters

- **conn** (*sdkconn.BaseNetQASMConnection*) –
- **add_new_command** (*bool*) –
- **ent_info** (*Optional[qlink_compat.LinkLayerOKTypeK]*) –
- **virtual_address** (*Optional[int]*) –

__init__(*conn*, *add_new_command=True*, *ent_info=None*, *virtual_address=None*)

Qubit constructor. This is the standard way to allocate a new qubit in an application.

Parameters

- **conn** (*sdkconn.BaseNetQASMConnection*) – connection of the application in which to allocate the qubit
- **add_new_command** (*bool*) – whether to automatically add NetQASM instructions to the current subroutine to allocate and initialize the qubit
- **ent_info** (*Optional[qlink_compat.LinkLayerOKTypeK]*) – entanglement generation information in case this qubit is the result of an entanglement generation request
- **virtual_address** (*Optional[int]*) – explicit virtual ID to use for this qubit. If None, a free ID is automatically chosen.

property connection
Get the NetQASM connection of this qubit

Return type sdkconn.BaseNetQASMConnection

property builder
Get the Builder of this qubit's connection

Return type *Builder*

property qubit_id
Get the qubit ID

Return type `int`

property active

Return type `bool`

property entanglement_info
Get information about the successful link layer request that resulted in this qubit.

Return type Optional[qlink_compat.LinkLayerOKTypeK]

property remote_entangled_node
Get the name of the remote node the qubit is entangled with.

If not entanled, *None* is returned.

Return type `Optional[str]`

assert_active()
Assert that the qubit is active, i.e. allocated.

Return type `None`

measure(*future=None*, *inplace=False*, *store_array=True*, *basis=<QubitMeasureBasis.Z: 2>*, *basis_rotations=None*)
Measure the qubit in the standard basis and get the measurement outcome.

Parameters

- **future** (`Union[Future, RegFuture, None]`) – the *Future* to place the outcome in. If None, a Future is created automatically.
- **inplace** (`bool`) – If False, the measurement is destructive and the qubit is removed from memory. If True, the qubit is left in the post-measurement state.
- **store_array** (`bool`) – whether to store the outcome in an array. If not, it is placed in a register. Only used if *future* is None.
- **basis** (*QubitMeasureBasis*) – in which of the Pauli bases (X, Y or Z) to measure. Default is Z. Ignored if *basis_rotations* is not None.
- **basis_rotations** (`Optional[Tuple[int, int, int]]`) – rotations to apply before measuring in the Z-basis. This can be used to specify arbitrary measurement bases. The 3 values are interpreted as 3 rotation angles, for an X- Y-, and another X-rotation, respectively. Each angle is interpreted as a multiple of pi/16. For example, if *basis_rotations* is (8, 0, 0), an X-rotation is applied with angle 8*pi/16 = pi/2 radians, followed by a Y-rotation of angle 0 and an X-rotation of angle 0. Finally, the measurement is done in the Z-basis.

Return type `Union[Future, RegFuture]`

Returns the Future representing the measurement outcome. It is a *Future* if

the result is in an array (default) or *RegFuture* if the result is in a register.

X()
Apply an X gate on the qubit.

Return type None

Y()
Apply a Y gate on the qubit.

Return type None

Z()
Apply a Z gate on the qubit.

Return type None

T()
Apply a T gate on the qubit.

A T gate is a Z-rotation with angle pi/4.

Return type None

H()
Apply a Hadamard gate on the qubit.

Return type None

K()
Apply a K gate on the qubit.

A K gate moves the |0> state to +|i> (positive Y) and vice versa.

Return type None

S()
Apply an S gate on the qubit.

An S gate is a Z-rotation with angle pi/2.

Return type None

rot_X(*n=0*, *d=0*, *angle=None*)
Do a rotation around the X-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

rot_Y(*n=0*, *d=0*, *angle=None*)
Do a rotation around the Y-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

rot_Z (*n=0*, *d=0*, *angle=None*)
Do a rotation around the Z-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

cnot (*target*)
Apply a CNOT gate between this qubit (control) and a target qubit.

Parameters **target** (*Qubit*) – target qubit. Should have the same connection as this qubit.

Return type None

cphase (*target*)
Apply a CPHASE (CZ) gate between this qubit (control) and a target qubit.

Parameters **target** (*Qubit*) – target qubit. Should have the same connection as this qubit.

Return type None

reset ()
Reset the qubit to the state |0>.

Return type None

free ()
Free the qubit and its virtual ID.

After freeing, the underlying physical qubit can be used to store another state.

Return type None

2.4.3 EPR socket

EPR Socket interface.

class netqasm.sdk.epr_socket.**EPRSocket** (*remote_app_name*, *epr_socket_id=0*, *remote_epr_socket_id=0*, *min_fidelity=100*)
EPR socket class. Used to generate entanglement with a remote node.

An EPR socket represents a connection with a single remote node through which EPR pairs can be generated. Its main interfaces are the *create* and *recv* methods. A typical use case for two nodes is that they both create

an EPR socket to the other node, and during the protocol, one of the nodes does *create* operations on its socket while the other node does *recv* operations.

A *create* operation asks the network stack to initiate generation of EPR pairs with the remote node. Depending on the type of generation, the result of this operation can be qubit objects or measurement outcomes. A *recv* operation asks the network stack to wait for the remote node to initiate generation of EPR pairs. Again, the result can be qubit objects or measurement outcomes.

Each *create* operation on one node must be matched by a *recv* operation on the other node. Since "creating" and "receiving" must happen at the same time, a node that is doing a *create* operation on its socket cannot advance until the other node does the corresponding *recv*. This is different from classical network sockets where a "send" operation (roughly anologous to *create* in an EPR socket) does not block on the remote node receiving it.

An EPR socket is identified by a triple consisting of (1) the remote node ID, (2) the local socket ID and (3) the remote socket ID. Two nodes that want to generate EPR pairs with each other should make sure that the IDs in their local sockets match.

Parameters

- **remote_app_name** (str) –
- **epr_socket_id** (int) –
- **remote_epr_socket_id** (int) –
- **min_fidelity** () –

__init__ (*remote_app_name*, *epr_socket_id=0*, *remote_epr_socket_id=0*, *min_fidelity=100*)

Create an EPR socket. It still needs to be registered with the network stack separately.

Registering and opening the EPR socket is currently done automatically by the connection that uses this EPR socket, specifically when a context is opened with that connection.

Parameters

- **remote_app_name** (str) – name of the remote party (i.e. the role, like "client", not necessarily the node name like "delft")
- **epr_socket_id** (int) – local socket ID, defaults to 0
- **remote_epr_socket_id** (int) – remote socket ID, defaults to 0. Note that this must match with the local socket ID of the remote node's EPR socket.
- **min_fidelity** (int) – minimum desired fidelity for EPR pairs generated over this socket, in percentages (i.e. range 0-100). Defaults to 100.

property conn

Get the underlying NetQASMConnection

Return type *connection.BaseNetQASMConnection*

property remote_app_name

Get the remote application name

Return type str

property remote_node_id

Get the remote node ID

Return type int

property epr_socket_id

Get the EPR socket ID

Return type int

property remote_epr_socket_id
Get the remote EPR socket ID

Return type int

property min_fidelity
Get the desired minimum fidelity

Return type int

create_keep(*number=1*, *post_routine=None*, *sequential=False*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *min_fidelity_all_at_end=None*, *max_tries=None*)
Ask the network stack to generate EPR pairs with the remote node and keep them in memory.

A *create_keep* operation must always be matched by a *recv_keep* operation on the remote node.

If *sequential* is False (default), this operation returns a list of Qubit objects representing the local qubits that are each one half of the generated pairs. These qubits can then be manipulated locally just like locally initialized qubits, by e.g. applying gates or measuring them. Each qubit also contains information about the entanglement generation that lead to its creation, and can be accessed by its *entanglement_info* property.

A typical example for just generating one pair with another node would be:

```
q = epr_socket.create_keep()[0]
# `q` can now be used as a normal qubit
```

If *sequential* is False (default), the all requested EPR pairs are generated at once, before returning the results (qubits or entanglement info objects).

If *sequential* is True, a callback function (*post_routine*) should be specified. After generating one EPR pair, this callback will be called, before generating the next pair. This method can e.g. be used to generate many EPR pairs (more than the number of physical qubits available), by measuring (and freeing up) each qubit before the next pair is generated.

For example:

```
outcomes = alice.new_array(num)

def post_create(conn, q, pair):
    q.H()
    outcome = outcomes.get_future_index(pair)
    q.measure(outcome)
epr_socket.create_keep(number=num, post_routine=post_create, sequential=True)
```

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function for each genated pair. Only used if *sequential* is True. The callback should take three arguments *(conn, q, pair)* where * *conn* is the connection (e.g. *self*) * *q* is the entangled qubit (of type *FutureQubit*) * *pair* is a register holding which pair is handled (0, 1, . . .)
- **sequential** (bool) – whether to use callbacks after each pair, defaults to False
- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter
- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.

- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[*Qubit*]

Returns list of qubits created

create_keep_with_info(*number=1*, *post_routine=None*, *sequential=False*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *min_fidelity_all_at_end=None*)

Same as create_keep but also return the EPR generation information coming from the network stack.

For more information see the documentation of *create_keep*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) –
- **sequential** (bool) –
- **time_unit** (TimeUnit) –
- **max_time** (int) –
- **min_fidelity_all_at_end** (Optional[int]) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

Returns tuple with (1) list of qubits created, (2) list of EprKeepResult objects

create_measure(*number=1*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *basis_local=None*, *basis_remote=None*, *rotations_local=(0, 0, 0)*, *rotations_remote=(0, 0, 0)*, *random_basis_local=None*, *random_basis_remote=None*)

Ask the network stack to generate EPR pairs with the remote node and measure them immediately (on both nodes).

A *create_measure* operation must always be matched by a *recv_measure* operation on the remote node.

This operation returns a list of Linklayer response objects. These objects contain information about the entanglement generation and includes the measurement outcome and basis used. Note that all values are *Future* objects. This means that the current subroutine must be flushed before the values become defined.

An example for generating 10 pairs with another node that are immediately measured:

```
# list of Futures that become defined when subroutine is flushed
outcomes = []
with NetQASMConnection("alice", epr_sockets=[epr_socket]):
    ent_infos = epr_socket.create(number=10, tp=EPRType.M)
    for ent_info in ent_infos:
        outcomes.append(ent_info.measurement_outcome)
```

The basis to measure in can also be specified. There are 3 ways to specify a basis:

- using one of the *EprMeasBasis* variants

- by specifying 3 rotation angles, interpreted as an X-rotation, a Y-rotation and another X-rotation. For example, setting *rotations_local* to (8, 0, 0) means that before measuring, an X-rotation of 8*pi/16 = pi/2 radians is applied to the qubit.
- using one of the *RandomBasis* variants, in which case one of the bases of that variant is chosen at random just before measuring

NOTE: the node that initiates the entanglement generation, i.e. the one that calls *create* on its EPR socket, also controls the measurement bases of the receiving node (by setting e.g. *rotations_remote*). The receiving node cannot change this.

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter
- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.
- **basis_local** (Optional[EprMeasBasis]) – basis to measure in on this node for M-type requests
- **basis_remote** (Optional[EprMeasBasis]) – basis to measure in on the remote node for M-type requests
- **rotations_local** (Tuple[int, int, int]) – rotations to apply before measuring on this node
- **rotations_remote** (Tuple[int, int, int]) – rotations to apply before measuring on remote node
- **random_basis_local** (Optional[RandomBasis]) – random bases to choose from when measuring on this node
- **random_basis_remote** (Optional[RandomBasis]) – random bases to choose from when measuring on the remote node

Return type List[EprMeasureResult]

Returns list of entanglement info objects per created pair.

create_rsp(*number=1*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *basis_local=None*, *rotations_local=(0, 0, 0)*, *random_basis_local=None*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Ask the network stack to do remote preparation with the remote node.

A *create_rsp* operation must always be matched by a *recv_epr* operation on the remote node.

This operation returns a list of Linklayer response objects. These objects contain information about the entanglement generation and includes the measurement outcome and basis used. Note that all values are *Future* objects. This means that the current subroutine must be flushed before the values become defined.

An example for generating 10 pairs with another node that are immediately measured:

```
m: LinkLayerOKTypeM = epr_socket.create_rsp(tp=EPRType.R)[0]
print(m.measurement_outcome)
# remote node now has a prepared qubit
```

The basis to measure in can also be specified. There are 3 ways to specify a basis:

- using one of the *EprMeasBasis* variants

- by specifying 3 rotation angles, interpreted as an X-rotation, a Y-rotation and another X-rotation. For example, setting *rotations_local* to (8, 0, 0) means that before measuring, an X-rotation of 8*pi/16 = pi/2 radians is applied to the qubit.
- using one of the *RandomBasis* variants, in which case one of the bases of that variant is chosen at random just before measuring

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter
- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.
- **basis_local** (Optional[EprMeasBasis]) – basis to measure in on this node for M-type requests
- **rotations_local** (Tuple[int, int, int]) – rotations to apply before measuring on this node
- **random_basis_local** (Optional[RandomBasis]) – random bases to choose from when measuring on this node
- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[EprMeasureResult]

Returns list of entanglement info objects per created pair.

create(*number=1*, *post_routine=None*, *sequential=False*, *tp=<EPRType.K: 0>*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *basis_local=None*, *basis_remote=None*, *rotations_local=(0, 0, 0)*, *rotations_remote=(0, 0, 0)*, *random_basis_local=None*, *random_basis_remote=None*)

Ask the network stack to generate EPR pairs with the remote node.

A *create* operation must always be matched by a *recv* operation on the remote node.

If the type of request is Create and Keep (CK, or just K) and if *sequential* is False (default), this operation returns a list of Qubit objects representing the local qubits that are each one half of the generated pairs. These qubits can then be manipulated locally just like locally initialized qubits, by e.g. applying gates or measuring them. Each qubit also contains information about the entanglement generation that lead to its creation, and can be accessed by its *entanglement_info* property.

A typical example for just generating one pair with another node would be:

```
q = epr_socket.create()[0]
# `q` can now be used as a normal qubit
```

If the type of request is Measure Directly (MD, or just M), this operation returns a list of Linklayer response objects. These objects contain information about the entanglement generation and includes the measurement outcome and basis used. Note that all values are *Future* objects. This means that the current subroutine must be flushed before the values become defined.

An example for generating 10 pairs with another node that are immediately measured:

```
# list of Futures that become defined when subroutine is flushed
outcomes = []
with NetQASMConnection("alice", epr_sockets=[epr_socket]):
    ent_infos = epr_socket.create(number=10, tp=EPRType.M)
    for ent_info in ent_infos:
        outcomes.append(ent_info.measurement_outcome)
```

For "Measure Directly"-type requests, the basis to measure in can also be specified. There are 3 ways to specify a basis:

- using one of the *EprMeasBasis* variants
- by specifying 3 rotation angles, interpreted as an X-rotation, a Y-rotation and another X-rotation. For example, setting *rotations_local* to (8, 0, 0) means that before measuring, an X-rotation of 8*pi/16 = pi/2 radians is applied to the qubit.
- using one of the *RandomBasis* variants, in which case one of the bases of that variant is chosen at random just before measuring

NOTE: the node that initiates the entanglement generation, i.e. the one that calls *create* on its EPR socket, also controls the measurement bases of the receiving node (by setting e.g. *rotations_remote*). The receiving node cannot change this.

If *sequential* is False (default), the all requested EPR pairs are generated at once, before returning the results (qubits or entanglement info objects).

If *sequential* is True, a callback function (*post_routine*) should be specified. After generating one EPR pair, this callback will be called, before generating the next pair. This method can e.g. be used to generate many EPR pairs (more than the number of physical qubits available), by measuring (and freeing up) each qubit before the next pair is generated.

For example:

```
outcomes = alice.new_array(num)

def post_create(conn, q, pair):
    q.H()
    outcome = outcomes.get_future_index(pair)
    q.measure(outcome)
epr_socket.create(number=num, post_routine=post_create, sequential=True)
```

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function for each genated pair. Only used if *sequential* is True. The callback should take three arguments *(conn, q, pair)* where * *conn* is the connection (e.g. *self*) * *q* is the entangled qubit (of type *FutureQubit*) * *pair* is a register holding which pair is handled (0, 1, . . .)
- **sequential** (bool) – whether to use callbacks after each pair, defaults to False
- **tp** (EPRType) – type of entanglement generation, defaults to EPRType.K. Note that corresponding *recv* of the remote node's EPR socket must specify the same type.

- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter
- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.
- **basis_local** (Optional[EprMeasBasis]) – basis to measure in on this node for M-type requests
- **basis_remote** (Optional[EprMeasBasis]) – basis to measure in on the remote node for M-type requests
- **rotations_local** (Tuple[int, int, int]) – rotations to apply before measuring on this node (for M-type requests)
- **rotations_remote** (Tuple[int, int, int]) – rotations to apply before measuring on remote node (for M-type requests)
- **random_basis_local** (Optional[RandomBasis]) – random bases to choose from when measuring on this node (for M-type requests)
- **random_basis_remote** (Optional[RandomBasis]) – random bases to choose from when measuring on the remote node (for M-type requests)

Return type Union[List[*Qubit*], List[EprMeasureResult], List[LinkLayerOKTypeM]]

Returns For K-type requests: list of qubits created. For M-type requests: list of entanglement info objects per created pair.

create_context(*number=1*, *sequential=False*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*)

Create a context that is executed for each generated EPR pair consecutively.

Creates EPR pairs with a remote node and handles each pair by the operations defined in a subsequent context. See the example below.

```
with epr_socket.create_context(number=10) as (q, pair):
    q.H()
    m = q.measure()
```

NOTE: even though all pairs are handled consecutively, they are still generated concurrently by the network stack. By setting *sequential* to True, the network stack only generates the next pair after the context for the previous pair has been executed, similar to using a callback (*post_routine*) in the *create* method.

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **sequential** (bool) – whether to generate pairs sequentially, defaults to False
- **time_unit** (TimeUnit) –
- **max_time** (int) –

Return type AbstractContextManager[Tuple[*FutureQubit*, *RegFuture*]]

recv_keep(*number=1*, *post_routine=None*, *sequential=False*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Ask the network stack to wait for the remote node to generate EPR pairs, which are kept in memory.

A *recv_keep* operation must always be matched by a *create_keep* operation on the remote node. The number of generated pairs must also match.

For more information see the documentation of *create_keep*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function used when *sequential* is True
- **sequential** (bool) – whether to call the callback after each pair generation, defaults to False
- **expect_phi_plus** (bool) – whether to assume that the EPR pairs that are created are in the Phi+ (or Phi_00) state. Defaults to True. If True, the compiler will make sure that if the physical link actually produced another Bell state, the behavior seen by the application is still as if a Phi+ state was actually produced.
- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[*Qubit*]

Returns list of qubits created

recv_keep_with_info(*number=1*, *post_routine=None*, *sequential=False*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Same as recv_keep but also return the EPR generation information coming from the network stack.

For more information see the documentation of *recv_keep*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) –
- **sequential** (bool) –
- **expect_phi_plus** (bool) –
- **min_fidelity_all_at_end** (Optional[int]) –
- **max_tries** (Optional[int]) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

Returns tuple with (1) list of qubits created, (2) list of EprKeepResult objects

recv_measure(*number=1*, *expect_phi_plus=True*)

Ask the network stack to wait for the remote node to generate EPR pairs, which are immediately measured (on both nodes).

A *recv_measure* operation must always be matched by a *create_measure* operation on the remote node. The number and type of generation must also match.

For more information see the documentation of *create_measure*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **expect_phi_plus** (bool) – whether to assume that the EPR pairs that are created are in the Phi+ (or Phi_00) state. Defaults to True. If True, the compiler will make sure that if the physical link actually produced another Bell state, the behavior seen by the application is still as if a Phi+ state was actually produced.

Return type List[EprMeasureResult]

Returns list of entanglement info objects per created pair.

recv_rsp(*number=1*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Ask the network stack to wait for remote state preparation from another node.

A *recv_rsp* operation must always be matched by a *create_rsp* operation on the remote node. The number and type of generation must also match.

For more information see the documentation of *create_rsp*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **expect_phi_plus** (bool) – whether to assume that the EPR pairs that are created are in the Phi+ (or Phi_00) state. Defaults to True. If True, the compiler will make sure that if the physical link actually produced another Bell state, the behavior seen by the application is still as if a Phi+ state was actually produced.
- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[*Qubit*]

Returns list of qubits created

recv_rsp_with_info(*number=1*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Same as recv_rsp but also return the EPR generation information coming from the network stack.

For more information see the documentation of *recv_rsp*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **expect_phi_plus** (bool) –
- **min_fidelity_all_at_end** (Optional[int]) –
- **max_tries** (Optional[int]) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

Returns tuple with (1) list of qubits created, (2) list of EprKeepResult objects

recv(*number=1*, *post_routine=None*, *sequential=False*, *tp=<EPRType.K: 0>*)

Ask the network stack to wait for the remote node to generate EPR pairs.

A *recv* operation must always be matched by a *create* operation on the remote node. See also the documentation of *create*. The number and type of generation must also match.

In case of Measure Directly requests, it is the initiating node (that calls *create*) which specifies the measurement bases. This should not and cannot be done in *recv*.

For more information see the documentation of *create*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function used when *sequential* is True
- **sequential** (bool) – whether to call the callback after each pair generation, defaults to False
- **tp** (EPRType) – type of entanglement generation, defaults to EPRType.K

Return type Union[List[*Qubit*], List[EprMeasureResult], List[LinkLayerOKTypeR]]

Returns For K-type requests: list of qubits created. For M-type requests: list of entanglement info objects per created pair.

recv_context (*number=1*, *sequential=False*)
Receives EPR pair with a remote node (see doc of *create_context()*)

Parameters

- **number** (int) –
- **sequential** (bool) –

2.4.4 Futures

Abstractions for classical runtime values.

This module contains the *BaseFuture* class and its subclasses.

class netqasm.sdk.futures.**Future** (*connection*, *address*, *index*)
Represents a single array entry value that will become available in the future.

See *BaseFuture* for more explanation about Futures.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) –
- **address** (*int*) –
- **index** (*Union[int, Future, operand.Register, RegFuture]*) –

__init__ (*connection*, *address*, *index*)
Future constructor. Typically not used directly.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) – connection through which subroutines are sent that contain the array entry corresponding to this Future
- **address** (*int*) – address of the array
- **index** (*Union[int, Future, operand.Register, RegFuture]*) – index in the array

add(*other*, *mod=None*)
Add another value to this Future's value.

The result is stored in this Future.

Let the quantum node controller add a value to the value represented by this Future. The addition operation is compiled into a subroutine and is fully executed by the quantum node controller. This avoids the need to wait for a subroutine result (which resolves the Future's *value*) and then doing the addition on the Host.

Parameters

- **other** (Union[int, str, *Register*, *BaseFuture*]) – value to add to this Future's value
- **mod** (Optional[int]) – do the addition modulo *mod*

Return type None

get_load_commands(*register*)
Return a list of ProtoSubroutine commands for loading this Future into the specified register.

Parameters **register** (*Register*) –

Return type List[Union[*ICmd*, *BranchLabel*]]

get_address_entry()
Convert this Future to an ArrayEntry object to be used an instruction operand.

Return type *ArrayEntry*

class netqasm.sdk.futures.**Array**(*connection*, *length*, *address*, *init_values=None*, *lineno=None*)
Wrapper around an array in Shared Memory.

An *Array* instance provides methods to inspect and operate on an array that exists in shared memory. They are typically obtained as return values to certain SDK methods.

Elements or slices of the array can be captured as Futures.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) –
- **length** (*int*) –
- **address** (*int*) –
- **init_values** (*Optional[List[Optional[int]]]*) –
- **lineno** (*Optional[HostLine]*) –

__init__(*connection*, *length*, *address*, *init_values=None*, *lineno=None*)
Array constructor. Typically not used directly.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) – connection of the application this array is part of
- **length** (*int*) – length of the array
- **address** (*int*) – address of the array
- **init_values** (*Optional[List[Optional[int]]]*) – initial values of the array. Must have length *length*.
- **lineno** (*Optional[HostLine]*) – line number where the array is created in the Python source code

property lineno
What line in host application file initiated this array

Return type Optional[*HostLine*]

property address

Return type int

property builder

Return type *Builder*

get_future_index(*index*)
Get a Future representing a particular array element

Parameters **index** (Union[int, str, *Register*, *RegFuture*]) –

Return type *Future*

get_future_slice(*s*)
Get a list of Futures each representing one element in a particular array slice

Parameters **s** (slice) –

Return type List[*Future*]

foreach()
Create a context of code that gets called for each element in the array.

Returns a future of the array value at the current index.

Code inside the context *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Example:

```
with NetQASMConnection(app_name="alice") as alice:
    outcomes = alice.new_array(10)
    values = alice.new_array(
        10,
        init_values=[random.randint(0, 1) for _ in range(10)]
    )
    with values.foreach() as v:
        q = Qubit(alice)
        with v.if_eq(1):
            q.H()
        q.measure(future=outcomes.get_future_index(i))
```

Return type *SdkForEachContext*

enumerate()
Create a context of code that gets called for each element in the array and includes a counter.

Returns a tuple (*index*, *future*) where *future* is a Future of the array value at the current index.

Code inside the context *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Example:

```
with NetQASMConnection(app_name="alice") as alice:
    outcomes = alice.new_array(10)
    values = alice.new_array(
        10,
        init_values=[random.randint(0, 1) for _ in range(10)]
    )
    with values.enumerate() as (i, v):
        q = Qubit(alice)
        with v.if_eq(1):
            q.H()
        q.measure(future=outcomes.get_future_index(i))
```

Return type *SdkForEachContext*

undefine()
Undefine (i.e. set to 'None') all elements in the array.

Return type None

class netqasm.sdk.futures.**RegFuture**(*connection*, *reg=None*)
Represents a single register value that will become available in the future.

See *BaseFuture* for more explanation about Futures.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) –
- **reg** (*Optional[operand.Register]*) –

__init__(*connection*, *reg=None*)
RegFuture constructor. Typically not used directly.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) – connection through which subroutines are sent that contain the array entry corresponding to this Future
- **reg** (*Optional[operand.Register]*) – specific NetQASM register that will hold the Future's value. If None, a suitable register is automatically used.

property reg

Return type Optional[*Register*]

add(*other*, *mod=None*)
Add another value to this Future's value.

The result is stored in this Future.

Let the quantum node controller add a value to the value represented by this Future. The addition operation is compiled into a subroutine and is fully executed by the quantum node controller. This avoids the need to wait for a subroutine result (which resolves the Future's *value*) and then doing the addition on the Host.

Parameters

- **other** (Union[int, str, *Register*, *BaseFuture*]) – value to add to this Future's value
- **mod** (Optional[int]) – do the addition modulo *mod*

Return type None

2.4.5 Classical communication

Interface for classical communication between Hosts.

This module contains the *Socket* class which is a base for representing classical communication (sending and receiving classical messages) between Hosts.

class netqasm.sdk.classical_communication.socket.**Socket**(*app_name*, *remote_app_name*, *socket_id=0*, *timeout=None*, *use_callbacks=False*, *log_config=None*)

Base class for classical sockets.

Classical communication is modelled by sockets, which are also widely used in purely classical applications involving communication.

If a node wants to communicate arbitrary classical messages with another node, this communication must be done between the Hosts of these nodes. Both Hosts should instantiate a Socket object with the other Host as 'remote'. Upon creation, the Socket objects try to connect to each other. Only after this has succeeded, the sockets can be used.

The main methods of a Socket are *send* and *recv*, which are used to send a message to the remote Host, and wait to receive a message from the other Host, respectively. Messages are str (string) objects. To send a number, convert it to a string before sending, and convert it back after receiving.

There are some variations on the *send* and *recv* methods which may be useful in specific scenarios. See their own documentation for their use.

NOTE: At the moment, Sockets are not part of the compilation process yet. Therefore, they don't need to be part of a Connection, and operations on Sockets do not need to be flushed before they are executed (they are executed immediately). This also means that e.g. a *recv* operation, which is blocking by default, acutally blocks the whole application script. **So, if any quantum operations should be executed before such a `recv` statement, make sure that these operations are flushed before blocking on `recv`.**

Implementations (subclasses) of Sockets may be quite different, depending on the runtime environment. A physical setup (with real hardware) may implement this as TCP sockets. A simulator might use inter-thread communication (see e.g. *ThreadSocket*), or another custom object type.

Parameters

- **app_name** (*str*) –
- **remote_app_name** (*str*) –
- **socket_id** (*int*) –
- **timeout** (*Optional[float]*) –
- **use_callbacks** (*bool*) –
- **log_config** (*Optional[config.LogConfig]*) –

__init__(*app_name*, *remote_app_name*, *socket_id=0*, *timeout=None*, *use_callbacks=False*, *log_config=None*)

Socket constructor.

Parameters

- **app_name** (*str*) – application/Host name of this socket's owner
- **remote_app_name** (*str*) – application/Host name of this socket's remote

- **socket_id** (*int*) – local ID to use for this socket
- **timeout** (*Optional[float]*) – maximum amount of real time to try to connect to the remote socket
- **use_callbacks** (*bool*) – whether to call the *recv_callback* method upon receiving a message
- **log_config** (*Optional[config.LogConfig]*) – logging configuration for this socket

abstract send(*msg*)
Send a message to the remote node.

Parameters **msg** (str) –

Return type None

abstract recv(*block=True*, *timeout=None*, *maxsize=None*)
Receive a message from the remote node.

Parameters

- **block** (bool) –
- **timeout** (Optional[float]) –
- **maxsize** (Optional[int]) –

Return type str

send_structured(*msg*)
Sends a structured message (with header and payload) to the remote node.

Parameters **msg** (*StructuredMessage*) –

Return type None

recv_structured(*block=True*, *timeout=None*, *maxsize=None*)
Receive a message (with header and payload) from the remote node.

Parameters

- **block** (bool) –
- **timeout** (Optional[float]) –
- **maxsize** (Optional[int]) –

Return type *StructuredMessage*

send_silent(*msg*)
Sends a message without logging

Parameters **msg** (str) –

Return type None

recv_silent(*block=True*, *timeout=None*, *maxsize=None*)
Receive a message without logging

Parameters

- **block** (bool) –
- **timeout** (Optional[float]) –
- **maxsize** (Optional[int]) –

Return type str

recv_callback(*msg*)

This method gets called when a message is received.

Subclass to define behaviour.

NOTE: This only happens if *self.use_callbacks* is set to *True*.

Parameters **msg** (str) –

Return type None

conn_lost_callback()

This method gets called when the connection is lost.

Subclass to define behaviour.

NOTE: This only happens if *self.use_callbacks* is set to *True*.

Return type None

CHAPTER

THREE

NETQASM PACKAGE

3.1 netqasm.backend

3.1.1 netqasm.backend.executor

NetQASM execution interface for simulators.

This module provides the *Executor* class which can be used by simulators as a base class for executing NetQASM instructions.

class netqasm.backend.executor.**EprCmdData**(*subroutine_id*, *ent_results_array_address*, *q_array_address*, *request*, *tot_pairs*, *pairs_left*)

Bases: object

Container for info about EPR pending requests.

Parameters

- **subroutine_id** (int) –
- **ent_results_array_address** (int) –
- **q_array_address** (Optional[int]) –
- **request** (Optional[LinkLayerCreate]) –
- **tot_pairs** (int) –
- **pairs_left** (int) –

subroutine_id: int

ent_results_array_address: int

q_array_address: Optional[int]

request: Optional[netqasm.qlink_compat.LinkLayerCreate]

tot_pairs: int

pairs_left: int

netqasm.backend.executor.**inc_program_counter**(*method*)

Decorator that automatically increases the current program counter.

Should be used on functions that interpret a single NetQASM instruction.

class netqasm.backend.executor.**Executor**(*name=None*, *instr_log_dir=None*, ***kwargs*)

Bases: object

Base class for entities that execute NetQASM applications.

An Executor represents the component in a quantum node controller that handles the registration and execution of NetQASM applications. It can execute NetQASM subroutines by interpreting their instructions.

This base class provides handlers for classical NetQASM instructions. These are methods with names *_instr_XXX*. Methods that handle quantum instructions are a no-op and should be overridden by subclasses. Entanglement instructions are handled and forwarded to the network stack.

Parameters

- **name** (Optional[str]) –
- **instr_log_dir** (Optional[str]) –

instr_logger_class
alias of *netqasm.logging.output.InstrLogger*

__init__(*name=None*, *instr_log_dir=None*, ***kwargs*)
Executor constructor.

Parameters

- **name** (Optional[str]) – name of the executor for logging purposes, defaults to None
- **instr_log_dir** (Optional[str]) – directory to log instructions to, defaults to None

property name
Get the name of this executor.

Return type str

Returns name

property node_id
Get the ID of the node this Executor runs on

Raises **NotImplementedError** – This should be overridden by a subclass

Return type int

Returns ID of the node

set_instr_logger(*instr_log_dir*)
Let the executor use an instruction logger that logs to *instr_log_dir*

Parameters **instr_log_dir** (str) – path to the log directory

Return type None

classmethod get_instr_logger(*node_name*, *instr_log_dir*, *executor*, *force_override=False*)

Parameters

- **node_name** (str) –
- **instr_log_dir** (str) –
- **executor** (*Executor*) –
- **force_override** (bool) –

Return type *InstrLogger*

property network_stack
Get the network stack (if any) connected to this Executor.

Return type Optional[*BaseNetworkStack*]

Returns the network stack

init_new_application(*app_id*, *max_qubits*)
Register a new application.

Parameters

- **app_id** (int) – App ID of the application.
- **max_qubits** (int) – Maximum number of qubits the application is allowed to allocate at the same time.

Return type None

setup_epr_socket(*epr_socket_id*, *remote_node_id*, *remote_epr_socket_id*)
Instruct the Executor to open an EPR Socket.

The Executor forwards this instruction to the Network Stack.

Parameters

- **epr_socket_id** (int) – ID of local EPR socket
- **remote_node_id** (int) – ID of remote node
- **remote_epr_socket_id** (int) – ID of remote EPR socket

Yield [description]

Return type Generator[Any, None, None]

stop_application(*app_id*)
Stop an application and clear all qubits and classical memories.

Parameters **app_id** (int) – ID of the application to stop

Yield [description]

Return type Generator[Any, None, None]

consume_execute_subroutine(*subroutine*)
Consume the generator returned by *execute_subroutine*.

Parameters **subroutine** (*subrt_module.Subroutine*) – subroutine to execute

Return type None

execute_subroutine(*subroutine*)
Execute a NetQASM subroutine.

This is a generator to allow simulators to yield at certain points during execution, e.g. to yield control to a asynchronous runtime.

Parameters **subroutine** (*subrt_module.Subroutine*) – subroutine to execute

Yield [description]

Return type Generator[Any, None, None]

allocate_new_qubit_unit_module(*app_id*, *num_qubits*)

Parameters

- **app_id** (int) –
- **num_qubits** (int) –

Return type None

3.1.2 netqasm.backend.messages

Definitions of messages between the Host and the quantum node controller.

class netqasm.backend.messages.**MessageHeader**
Bases: _ctypes.Structure

classmethod len()

id
Structure/Union member

length
Structure/Union member

class netqasm.backend.messages.**MessageType**(*value*)
Bases: enum.Enum

An enumeration.

INIT_NEW_APP = 0

OPEN_EPR_SOCKET = 1

SUBROUTINE = 2

STOP_APP = 3

SIGNAL = 4

class netqasm.backend.messages.**Message**
Bases: _ctypes.Structure

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

class netqasm.backend.messages.**InitNewAppMessage**(*app_id=0*, *max_qubits=0*)
Bases: *netqasm.backend.messages.Message*

Message sent to the quantum node controller to register a new application.

TYPE = 0

app_id
Structure/Union member

max_qubits
Structure/Union member

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

class netqasm.backend.messages.**OpenEPRSocketMessage**(*app_id=0*, *epr_socket_id=0*, *remote_node_id=0*, *remote_epr_socket_id=0*, *min_fidelity=100*)
Bases: *netqasm.backend.messages.Message*

Message sent to the quantum node controller to open an EPR socket.

TYPE = 1

app_id
Structure/Union member

epr_socket_id
Structure/Union member

remote_node_id
Structure/Union member

remote_epr_socket_id
Structure/Union member

min_fidelity
Structure/Union member

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

class netqasm.backend.messages.**SubroutineMessage**(*subroutine*)
Bases: object

Message sent to the quantum node controller to execute a subroutine.

Parameters **subroutine** (Union[bytes, *Subroutine*]) –

TYPE = 2

__init__(*subroutine*)
NOTE this message does not subclass from *Message* since it contains a subroutine which is defined separately and not as a *ctype* for now. Still this class defines the methods *__bytes__* and *deserialize_from* so that it can be packed and unpacked. The packed form of the message is:

```
| TYP | SUBROUTINE ... |
```

Parameters **subroutine** (Union[bytes, *Subroutine*]) –

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

class netqasm.backend.messages.**StopAppMessage**(*app_id=0*)
Bases: *netqasm.backend.messages.Message*

Message sent to the quantum node controller to stop/finish an application.

TYPE = 3

app_id
Structure/Union member

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

class netqasm.backend.messages.**Signal**(*value*)
Bases: enum.Enum

An enumeration.

STOP = 0

class netqasm.backend.messages.**SignalMessage**(*signal=<Signal.STOP: 0>*)
Bases: *netqasm.backend.messages.Message*

Message sent to the quantum node controller with a specific signal.

Currently only used with the SquidASM simulator backend.

Parameters **signal** (*Signal*) –

TYPE = 4

signal
Structure/Union member

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

netqasm.backend.messages.**deserialize_host_msg**(*raw*)
Convert a serialized message into a *Message* object

Parameters **raw** (bytes) – serialized message (string of bytes)

Return type *Message*

Returns deserialized message object

class netqasm.backend.messages.**ReturnMessage**
Bases: *netqasm.backend.messages.Message*

Base class for messages from the quantum node controller to the Host.

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

class netqasm.backend.messages.**ReturnMessageType**(*value*)
Bases: enum.Enum

An enumeration.

DONE = 0

ERR = 1

RET_ARR = 2

RET_REG = 3

class netqasm.backend.messages.**MsgDoneMessage**(*msg_id=0*)
Bases: *netqasm.backend.messages.ReturnMessage*

Message to the Host that a subroutine has finished.

TYPE = 0

msg_id
Structure/Union member

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

class netqasm.backend.messages.**ErrorCode**(*value*)
Bases: enum.Enum

An enumeration.

GENERAL = 0

NO_QUBIT = 1

UNSUPP = 2

class netqasm.backend.messages.**ErrorMessage**(*err_code*)
Bases: *netqasm.backend.messages.ReturnMessage*

Message to the Host that an error occurred at the quantum node controller.

TYPE = 1

err_code
Structure/Union member

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

class netqasm.backend.messages.**ReturnArrayMessageHeader**
Bases: _ctypes.Structure

Header for a message with a returned array coming from the quantum node controller.

classmethod len()

address
Structure/Union member

length
Structure/Union member

class netqasm.backend.messages.**ReturnArrayMessage**(*address*, *values*)
Bases: object

Message with a returned array coming from the quantum node controller.

TYPE = 2

__init__(*address*, *values*)
NOTE this message does not subclass from *ReturnMessage* since the values is of variable length. Still this class defines the methods *__bytes__* and *deserialize_from* so that it can be packed and unpacked.

The packed form of the message is:

```
| ADDRESS | LENGTH | VALUES ... |
```

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

class netqasm.backend.messages.**ReturnRegMessage**(*register*, *value*)
Bases: *netqasm.backend.messages.ReturnMessage*

Message with a returned register coming from the quantum node controller.

TYPE = 3

register
Structure/Union member

value
Structure/Union member

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

type
Structure/Union member

netqasm.backend.messages.**deserialize_return_msg**(*raw*)
Convert a serialized 'return' message into a *Message* object

Parameters **raw** (bytes) – serialized message (string of bytes)

Return type *Message*

Returns deserialized message object

3.1.3 netqasm.backend.network_stack

Network stack interface for simulators.

This module provides the *BaseNetworkStack* class which can be used by simulators as a base class for modeling the network stack.

class netqasm.backend.network_stack.**Address**(*node_id*, *epr_socket_id*)
Bases: object

Parameters

- **node_id** (int) –
- **epr_socket_id** (int) –

node_id: int

epr_socket_id: int

class netqasm.backend.network_stack.**BaseNetworkStack**
Bases: abc.ABC

Base class for a (extremly simplified) network stack in simulators.

This class can be used as a simplified network stack that can handle requests that adhere to the QLink-Interface.

abstract put(*request*)
Put a link layer request to the network stack

Parameters **request** (Union[LinkLayerCreate, LinkLayerRecv]) –

Return type None

abstract setup_epr_socket(*epr_socket_id*, *remote_node_id*, *remote_epr_socket_id*, *timeout=1.0*)

Ask the network stack to open an EPR socket.

Parameters

- **epr_socket_id** (int) –
- **remote_node_id** (int) –
- **remote_epr_socket_id** (int) –
- **timeout** (float) –

Return type Generator[Any, None, None]

abstract get_purpose_id(*remote_node_id*, *epr_socket_id*)

Parameters

- **remote_node_id** (int) –
- **epr_socket_id** (int) –

Return type int

3.1.4 netqasm.backend.qnodeos

Quantum node controller interface for simulators.

This module provides the *QNodeController* class which can be used by simulators as a base class for modeling the quantum node controller.

class netqasm.backend.qnodeos.**QNodeController**(*name*, *instr_log_dir=None*, *flavour=None*, ***kwargs*)

Bases: object

Class for representing a Quantum Node Controller in a simulation.

A QNodeController represents a physical quantum node controller that handles messages coming from the Host, lets the Executor execute subroutines, and sends results back to the Host.

Parameters

- **name** (str) –
- **instr_log_dir** (Optional[str]) –
- **flavour** (Optional[*Flavour*]) –

__init__(*name*, *instr_log_dir=None*, *flavour=None*, ***kwargs*)

QNodeController constructor.

Parameters

- **name** (str) – name used for logging purposes
- **instr_log_dir** (Optional[str]) – directory used to write log files to
- **flavour** (Optional[*Flavour*]) – which NetQASM flavour this quantum node controller should expect and be able to interpret

abstract stop()

Return type None

property finished

Return type bool

handle_netqasm_message(*msg_id*, *msg*)

Parameters

- **msg_id** (int) –
- **msg** (*Message*) –

Return type Generator[Any, None, None]

property has_active_apps

Return type bool

property network_stack

Return type Optional[*BaseNetworkStack*]

add_network_stack(*network_stack*)

Parameters **network_stack** (*BaseNetworkStack*) –

Return type None

3.2 netqasm.lang

3.2.1 netqasm.lang.encoding

netqasm.lang.encoding.**NETQASM_VERSION**
alias of netqasm.lang.encoding.c_ubyte_Array_2

class netqasm.lang.encoding.**Metadata**
Bases: _ctypes.Structure

app_id
Structure/Union member

netqasm_version
Structure/Union member

class netqasm.lang.encoding.**OptionalInt**(*value*)
Bases: _ctypes.Structure

type
Structure/Union member

value
Structure/Union member

class netqasm.lang.encoding.**RegisterName**(*value*)
Bases: enum.Enum

An enumeration.

R = 0

C = 1

Q = 2

M = 3

class netqasm.lang.encoding.**Register**
Bases: _ctypes.Structure

padding
Structure/Union member

register_index
Structure/Union member

register_name
Structure/Union member

class netqasm.lang.encoding.**Address**
Bases: _ctypes.Structure

address
Structure/Union member

class netqasm.lang.encoding.**ArrayEntry**
Bases: _ctypes.Structure

address
Structure/Union member

index
Structure/Union member

class netqasm.lang.encoding.**ArraySlice**
Bases: _ctypes.Structure

address
Structure/Union member

start
Structure/Union member

stop
Structure/Union member

class netqasm.lang.encoding.**Command**(**args*, ***kwargs*)
Bases: _ctypes.Structure

id
Structure/Union member

netqasm.lang.encoding.**add_padding**(*fields*)
Used to add correct amount of padding for commands to make them fixed-length

class netqasm.lang.encoding.**NoOperandCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

class netqasm.lang.encoding.**RegCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

reg
Structure/Union member

class netqasm.lang.encoding.**RegRegCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

reg0
Structure/Union member

reg1
Structure/Union member

class netqasm.lang.encoding.**MeasCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

outcome
Structure/Union member

padding
Structure/Union member

qubit
Structure/Union member

class netqasm.lang.encoding.**RegImmImmCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

imm0
Structure/Union member

imm1
Structure/Union member

padding
Structure/Union member

reg
Structure/Union member

class netqasm.lang.encoding.**RegRegImmImmCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

imm0
Structure/Union member

imm1
Structure/Union member

padding
Structure/Union member

reg0
Structure/Union member

reg1
Structure/Union member

class netqasm.lang.encoding.**RegRegImm4Command**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

imm0
Structure/Union member

imm1
Structure/Union member

imm2
Structure/Union member

imm3
Structure/Union member

padding
Structure/Union member

reg0
Structure/Union member

reg1
Structure/Union member

class netqasm.lang.encoding.**RegRegRegCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

reg0
Structure/Union member

reg1
Structure/Union member

reg2
Structure/Union member

class netqasm.lang.encoding.**RegRegRegRegCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

reg0
Structure/Union member

reg1
Structure/Union member

reg2
Structure/Union member

reg3
Structure/Union member

class netqasm.lang.encoding.**ImmCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

imm
Structure/Union member

padding
Structure/Union member

class netqasm.lang.encoding.**ImmImmCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

imm0
Structure/Union member

imm1
Structure/Union member

padding
Structure/Union member

class netqasm.lang.encoding.**RegRegImmCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

imm
Structure/Union member

padding
Structure/Union member

reg0
Structure/Union member

reg1
Structure/Union member

class netqasm.lang.encoding.**RegImmCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

imm
Structure/Union member

padding
Structure/Union member

reg
Structure/Union member

class netqasm.lang.encoding.**RegEntryCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

entry
Structure/Union member

id
Structure/Union member

padding
Structure/Union member

reg
Structure/Union member

class netqasm.lang.encoding.**RegAddrCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

addr
Structure/Union member

id
Structure/Union member

padding
Structure/Union member

reg
Structure/Union member

class netqasm.lang.encoding.**ArrayEntryCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

entry
Structure/Union member

id
Structure/Union member

padding
Structure/Union member

class netqasm.lang.encoding.**ArraySliceCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

slice
Structure/Union member

class netqasm.lang.encoding.**SingleRegisterCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

register
Structure/Union member

class netqasm.lang.encoding.**ArrayCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

address
Structure/Union member

id
Structure/Union member

padding
Structure/Union member

size
Structure/Union member

class netqasm.lang.encoding.**AddrCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

addr
Structure/Union member

id
Structure/Union member

padding
Structure/Union member

class netqasm.lang.encoding.**Reg5Command**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

id
Structure/Union member

padding
Structure/Union member

reg0
Structure/Union member

reg1
Structure/Union member

reg2
Structure/Union member

reg3
Structure/Union member

reg4
Structure/Union member

class netqasm.lang.encoding.**RecvEPRCommand**(**args*, ***kwargs*)
Bases: *netqasm.lang.encoding.Command*

ent_results_array
Structure/Union member

epr_socket_id
Structure/Union member

id
Structure/Union member

padding
Structure/Union member

qubit_address_array
Structure/Union member

remote_node_id
Structure/Union member

3.2.2 netqasm.lang.instr

3.2.3 netqasm.lang.instr.base

class netqasm.lang.instr.base.**NetQASMInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*)
Bases: abc.ABC

Base NetQASM instruction class.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –

id: int = -1

mnemonic: str = ''

lineno: Optional[*netqasm.util.log.HostLine*] = None

abstract property operands

Return type List[*Operand*]

abstract classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

Return type *NetQASMInstruction*

abstract serialize()

Return type bytes

abstract classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

Return type *NetQASMInstruction*

writes_to()
Returns a list of Registers that this instruction writes to

Return type List[*Register*]

property debug_str

class netqasm.lang.instr.base.**NoOperandInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*)
Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with no operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg=None*)
Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 Register operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

reg: ***netqasm.lang.operand.Register*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegRegInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 2 Register operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –

reg0: *netqasm.lang.operand.Register* **= None**

reg1: *netqasm.lang.operand.Register* **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegImmImmInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 Register operand followed by 2 Immediate operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

reg: *netqasm.lang.operand.Register* **= None**

imm0: *netqasm.lang.operand.Immediate* **= None**

imm1: *netqasm.lang.operand.Immediate* **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegRegImmImmInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 2 Register operands followed by 2 Immediate operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

reg0: ***netqasm.lang.operand.Register*** **= None**

reg1: ***netqasm.lang.operand.Register*** **= None**

imm0: ***netqasm.lang.operand.Immediate*** **= None**

imm1: ***netqasm.lang.operand.Immediate*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegRegImm4Instruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *imm0=None*, *imm1=None*, *imm2=None*, *imm3=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 2 Register operands followed by 4 Immediate operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –
- **imm2** (Optional[*Immediate*]) –
- **imm3** (Optional[*Immediate*]) –

reg0: *netqasm.lang.operand.Register* = **None**

reg1: *netqasm.lang.operand.Register* = **None**

imm0: *netqasm.lang.operand.Immediate* = **None**

imm1: *netqasm.lang.operand.Immediate* = **None**

imm2: *netqasm.lang.operand.Immediate* = **None**

imm3: *netqasm.lang.operand.Immediate* = **None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegRegRegInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 3 Register operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –

reg0: *netqasm.lang.operand.Register* = **None**

reg1: *netqasm.lang.operand.Register* = **None**

reg2: *netqasm.lang.operand.Register* = **None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegRegRegRegInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*, *reg3=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 4 Register operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –
- **reg3** (Optional[*Register*]) –

reg0: *netqasm.lang.operand.Register* = **None**

reg1: *netqasm.lang.operand.Register* = **None**

reg2: *netqasm.lang.operand.Register* = **None**

reg3: *netqasm.lang.operand.Register* = **None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**ImmInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *imm=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 Immediate operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **imm** (Optional[*Immediate*]) –

imm: ***netqasm.lang.operand.Immediate*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**ImmImmInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 2 Immediate operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

imm0: ***netqasm.lang.operand.Immediate*** **= None**

imm1: ***netqasm.lang.operand.Immediate*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegRegImmInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *imm=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 2 Register operands and one Immediate operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

reg0: ***netqasm.lang.operand.Register*** **= None**

reg1: ***netqasm.lang.operand.Register*** **= None**

imm: ***netqasm.lang.operand.Immediate*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegImmInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg=None*, *imm=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 Register operand and one Immediate operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

reg: ***netqasm.lang.operand.Register*** **= None**

imm: ***netqasm.lang.operand.Immediate*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegEntryInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg=None*, *entry=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 Register operand and one ArrayEntry operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **entry** (Optional[*ArrayEntry*]) –

reg: ***netqasm.lang.operand.Register*** **= None**

entry: ***netqasm.lang.operand.ArrayEntry*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**RegAddrInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg=None*, *address=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 Register operand and one Address operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **address** (Optional[*Address*]) –

reg: ***netqasm.lang.operand.Register*** **= None**

address: ***netqasm.lang.operand.Address*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**ArrayEntryInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *entry=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 ArrayEntry operand and one Address operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **entry** (Optional[*ArrayEntry*]) –

entry: ***netqasm.lang.operand.ArrayEntry*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**ArraySliceInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *slice=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 ArraySlice operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **slice** (Optional[*ArraySlice*]) –

slice: ***netqasm.lang.operand.ArraySlice*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**AddrInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *address=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 1 Address operand.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **address** (Optional[*Address*]) –

address: ***netqasm.lang.operand.Address*** **= None**

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**Reg5Instruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*, *reg3=None*, *reg4=None*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

An instruction with 5 Register operands.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –
- **reg3** (Optional[*Register*]) –
- **reg4** (Optional[*Register*]) –

reg0: ***netqasm.lang.operand.Register*** **= None**

reg1: *netqasm.lang.operand.Register* = None

reg2: *netqasm.lang.operand.Register* = None

reg3: *netqasm.lang.operand.Register* = None

reg4: *netqasm.lang.operand.Register* = None

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.base.**DebugInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *text=''*)

Bases: *netqasm.lang.instr.base.NetQASMInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **text** (str) –

text: str = ''

property operands

Return type List[*Operand*]

classmethod deserialize_from(*raw*)

Parameters **raw** (bytes) –

serialize()

Return type bytes

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

3.2.4 netqasm.lang.instr.core

class netqasm.lang.instr.core.**SingleQubitInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.base.RegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –

- **reg** (Optional[*Register*]) –

property qreg

abstract to_matrix()

Return type ndarray

class netqasm.lang.instr.core.**TwoQubitInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*)

Bases: *netqasm.lang.instr.base.RegRegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –

property qreg0

property qreg1

abstract to_matrix()

abstract to_matrix_target_only()

class netqasm.lang.instr.core.**RotationInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.base.RegImmImmInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

property qreg

property angle_num

property angle_denom

abstract to_matrix()

classmethod from_operands(*operands*)

Parameters **operands** (List[Union[*Operand*, int]]) –

class netqasm.lang.instr.core.**ControlledRotationInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.base.RegRegImmImmInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

property qreg0

property qreg1

property angle_num

property angle_denom

abstract to_matrix()

class netqasm.lang.instr.core.**ClassicalOpInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*)

Bases: *netqasm.lang.instr.base.RegRegRegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –

writes_to()

Returns a list of Registers that this instruction writes to

Return type List[*Register*]

property regout

property regin0

property regin1

class netqasm.lang.instr.core.**ClassicalOpModInstruction**(*id=- 1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*, *reg3=None*)

Bases: *netqasm.lang.instr.base.RegRegRegRegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –
- **reg3** (Optional[*Register*]) –

writes_to()

Returns a list of Registers that this instruction writes to

Return type List[*Register*]

property regout

property regin0

property regin1

property regmod

class netqasm.lang.instr.core.**QAllocInstruction**(*id=1*, *mnemonic='qalloc'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.base.RegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: int = 1

mnemonic: str = 'qalloc'

property qreg

class netqasm.lang.instr.core.**InitInstruction**(*id=2*, *mnemonic='init'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.base.RegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 2**

mnemonic: **str = 'init'**

property qreg

class netqasm.lang.instr.core.**ArrayInstruction**(*id=3*, *mnemonic='array'*, *lineno=None*, *reg=None*, *address=None*)

Bases: *netqasm.lang.instr.base.RegAddrInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **address** (Optional[*Address*]) –

id: **int = 3**

mnemonic: **str = 'array'**

property size

class netqasm.lang.instr.core.**SetInstruction**(*id=4*, *mnemonic='set'*, *lineno=None*, *reg=None*, *imm=None*)

Bases: *netqasm.lang.instr.base.RegImmInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

id: **int = 4**

mnemonic: **str = 'set'**

writes_to()

Returns a list of Registers that this instruction writes to

Return type List[*Register*]

class netqasm.lang.instr.core.**StoreInstruction**(*id=5*, *mnemonic='store'*, *lineno=None*, *reg=None*, *entry=None*)

Bases: *netqasm.lang.instr.base.RegEntryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **entry** (Optional[*ArrayEntry*]) –

id: **int = 5**

mnemonic: str = 'store'

class netqasm.lang.instr.core.**LoadInstruction**(*id=6*, *mnemonic='load'*, *lineno=None*, *reg=None*, *entry=None*)

Bases: *netqasm.lang.instr.base.RegEntryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **entry** (Optional[*ArrayEntry*]) –

id: int = 6

mnemonic: str = 'load'

writes_to()

Returns a list of Registers that this instruction writes to

Return type List[*Register*]

class netqasm.lang.instr.core.**UndefInstruction**(*id=7*, *mnemonic='undef'*, *lineno=None*, *entry=None*)

Bases: *netqasm.lang.instr.base.ArrayEntryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **entry** (Optional[*ArrayEntry*]) –

id: int = 7

mnemonic: str = 'undef'

class netqasm.lang.instr.core.**LeaInstruction**(*id=8*, *mnemonic='lea'*, *lineno=None*, *reg=None*, *address=None*)

Bases: *netqasm.lang.instr.base.RegAddrInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **address** (Optional[*Address*]) –

id: int = 8

mnemonic: str = 'lea'

writes_to()

Returns a list of Registers that this instruction writes to

Return type List[*Register*]

class netqasm.lang.instr.core.**JmpInstruction**(*id=9*, *mnemonic='jmp'*, *lineno=None*, *imm=None*)

Bases: *netqasm.lang.instr.base.ImmInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **imm** (Optional[*Immediate*]) –

id: int = 9

mnemonic: str = 'jmp'

property line

class netqasm.lang.instr.core.**BranchUnaryInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg=None*, *imm=None*)

Bases: *netqasm.lang.instr.base.RegImmInstruction*

Represents an instruction to branch to a certain line, depending on a unary expression.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

property line

abstract check_condition(*a*)

Return type bool

class netqasm.lang.instr.core.**BezInstruction**(*id=10*, *mnemonic='bez'*, *lineno=None*, *reg=None*, *imm=None*)

Bases: *netqasm.lang.instr.core.BranchUnaryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

id: int = 10

mnemonic: str = 'bez'

check_condition(*a*)

Parameters **a** (int) –

Return type bool

class netqasm.lang.instr.core.**BnzInstruction**(*id=11*, *mnemonic='bnz'*, *lineno=None*, *reg=None*, *imm=None*)

Bases: *netqasm.lang.instr.core.BranchUnaryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

id: int = 11

mnemonic: str = 'bnz'

check_condition(*a*)

Parameters **a** (int) –

Return type bool

class netqasm.lang.instr.core.**BranchBinaryInstruction**(*id=-1*, *mnemonic=''*, *lineno=None*, *reg0=None*, *reg1=None*, *imm=None*)

Bases: *netqasm.lang.instr.base.RegRegImmInstruction*

Represents an instruction to branch to a certain line, depending on a binary expression.

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

property line

abstract check_condition(*a*, *b*)

Return type bool

class netqasm.lang.instr.core.**BeqInstruction**(*id=12*, *mnemonic='beq'*, *lineno=None*, *reg0=None*, *reg1=None*, *imm=None*)

Bases: *netqasm.lang.instr.core.BranchBinaryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

id: int = 12

mnemonic: str = 'beq'

check_condition(*a*, *b*)

Parameters

- **a** (int) –
- **b** (int) –

Return type bool

class netqasm.lang.instr.core.**BneInstruction**(*id=13*, *mnemonic='bne'*, *lineno=None*, *reg0=None*, *reg1=None*, *imm=None*)

Bases: *netqasm.lang.instr.core.BranchBinaryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

id: int = 13

mnemonic: str = 'bne'

check_condition(*a*, *b*)

Parameters

- **a** (int) –
- **b** (int) –

Return type bool

class netqasm.lang.instr.core.**BltInstruction**(*id=14*, *mnemonic='blt'*, *lineno=None*, *reg0=None*, *reg1=None*, *imm=None*)

Bases: *netqasm.lang.instr.core.BranchBinaryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

id: int = 14

mnemonic: str = 'blt'

check_condition(*a*, *b*)

Parameters

- **a** (int) –
- **b** (int) –

Return type bool

class netqasm.lang.instr.core.**BgeInstruction**(*id=15*, *mnemonic='bge'*, *lineno=None*, *reg0=None*, *reg1=None*, *imm=None*)

Bases: *netqasm.lang.instr.core.BranchBinaryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm** (Optional[*Immediate*]) –

id: int = 15

mnemonic: str = 'bge'

check_condition(*a*, *b*)

Parameters

- **a** (int) –
- **b** (int) –

Return type bool

class netqasm.lang.instr.core.**AddInstruction**(*id=16*, *mnemonic='add'*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*)

Bases: *netqasm.lang.instr.core.ClassicalOpInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –

id: int = 16

mnemonic: str = 'add'

class netqasm.lang.instr.core.**SubInstruction**(*id=17*, *mnemonic='sub'*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*)

Bases: *netqasm.lang.instr.core.ClassicalOpInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –

- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –

id: int = 17

mnemonic: str = 'sub'

class netqasm.lang.instr.core.**AddmInstruction**(*id=18*, *mnemonic='addm'*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*, *reg3=None*)

Bases: *netqasm.lang.instr.core.ClassicalOpModInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –
- **reg3** (Optional[*Register*]) –

id: int = 18

mnemonic: str = 'addm'

class netqasm.lang.instr.core.**SubmInstruction**(*id=19*, *mnemonic='subm'*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*, *reg3=None*)

Bases: *netqasm.lang.instr.core.ClassicalOpModInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –
- **reg3** (Optional[*Register*]) –

id: int = 19

mnemonic: str = 'subm'

class netqasm.lang.instr.core.**MeasInstruction**(*id=32*, *mnemonic='meas'*, *lineno=None*, *reg0=None*, *reg1=None*)

Bases: *netqasm.lang.instr.base.RegRegInstruction*

Parameters

- **id** (int) –

- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –

id: int = 32

mnemonic: str = 'meas'

writes_to()
Returns a list of Registers that this instruction writes to

Return type List[*Register*]

property qreg

property creg

class netqasm.lang.instr.core.**MeasBasisInstruction**(*id=41*, *mnemonic='meas_basis'*, *lineno=None*, *reg0=None*, *reg1=None*, *imm0=None*, *imm1=None*, *imm2=None*, *imm3=None*)
Bases: *netqasm.lang.instr.base.RegRegImm4Instruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –
- **imm2** (Optional[*Immediate*]) –
- **imm3** (Optional[*Immediate*]) –

id: int = 41

mnemonic: str = 'meas_basis'

writes_to()
Returns a list of Registers that this instruction writes to

Return type List[*Register*]

property qreg

property creg

property angle_num_x1

property angle_num_y

property angle_num_x2

property angle_denom

class netqasm.lang.instr.core.**CreateEPRInstruction**(*id=33*, *mnemonic='create_epr'*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*, *reg3=None*, *reg4=None*)

Bases: *netqasm.lang.instr.base.Reg5Instruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –
- **reg3** (Optional[*Register*]) –
- **reg4** (Optional[*Register*]) –

id: int = 33

mnemonic: str = 'create_epr'

property remote_node_id

property epr_socket_id

property qubit_addr_array

property arg_array

property ent_results_array

class netqasm.lang.instr.core.**RecvEPRInstruction**(*id=34*, *mnemonic='recv_epr'*, *lineno=None*, *reg0=None*, *reg1=None*, *reg2=None*, *reg3=None*)

Bases: *netqasm.lang.instr.base.RegRegRegRegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **reg2** (Optional[*Register*]) –
- **reg3** (Optional[*Register*]) –

id: int = 34

mnemonic: str = 'recv_epr'

property remote_node_id

property epr_socket_id

property qubit_addr_array

property ent_results_array

class netqasm.lang.instr.core.**WaitAllInstruction**(*id=35*, *mnemonic='wait_all'*, *lineno=None*, *slice=None*)

Bases: *netqasm.lang.instr.base.ArraySliceInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **slice** (Optional[*ArraySlice*]) –

id: int = 35

mnemonic: str = 'wait_all'

class netqasm.lang.instr.core.**WaitAnyInstruction**(*id=36*, *mnemonic='wait_any'*, *lineno=None*, *slice=None*)

Bases: *netqasm.lang.instr.base.ArraySliceInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **slice** (Optional[*ArraySlice*]) –

id: int = 36

mnemonic: str = 'wait_any'

class netqasm.lang.instr.core.**WaitSingleInstruction**(*id=37*, *mnemonic='wait_single'*, *lineno=None*, *entry=None*)

Bases: *netqasm.lang.instr.base.ArrayEntryInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **entry** (Optional[*ArrayEntry*]) –

id: int = 37

mnemonic: str = 'wait_single'

class netqasm.lang.instr.core.**QFreeInstruction**(*id=38*, *mnemonic='qfree'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.base.RegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: int = 38

mnemonic: str = 'qfree'

property qreg

class netqasm.lang.instr.core.**RetRegInstruction**(*id=39*, *mnemonic='ret_reg'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.base.RegInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: int = 39

mnemonic: str = 'ret_reg'

class netqasm.lang.instr.core.**RetArrInstruction**(*id=40*, *mnemonic='ret_arr'*, *lineno=None*, *address=None*)

Bases: *netqasm.lang.instr.base.AddrInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **address** (Optional[*Address*]) –

id: int = 40

mnemonic: str = 'ret_arr'

class netqasm.lang.instr.core.**BreakpointInstruction**(*id=100*, *mnemonic='breakpoint'*, *lineno=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.base.ImmImmInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

id: int = 100

mnemonic: str = 'breakpoint'

property action

property role

3.2.5 netqasm.lang.instr.flavour

class netqasm.lang.instr.flavour.**InstrMap**(*id_map=None*, *name_map=None*)
Bases: object

Parameters

- **id_map** (Optional[Dict[int, Type[*NetQASMInstruction*]]]) –
- **name_map** (Optional[Dict[str, Type[*NetQASMInstruction*]]]) –

id_map: Optional[Dict[int, Type[*netqasm.lang.instr.base.NetQASMInstruction*]]] = None

name_map: Optional[Dict[str, Type[*netqasm.lang.instr.base.NetQASMInstruction*]]] = Non

class netqasm.lang.instr.flavour.**Flavour**(*flavour_specific*)
Bases: abc.ABC

A Flavour represents an explicit instruction set that adheres to the Core NetQASM specification. Typically, a flavour is used for each specific target hardware.

A flavour 'inherits' all classical instructions from the core, but can specify explicitly the quantum instructions that the hardware supports, by listing the corresponding instruction classes.

Examples of flavours are the Vanilla flavour (with instructions defined in vanilla.py) and the Nitrogen-Vacancy (NV) flavour (instructions in nv.py).

Parameters **flavour_specific** (List[Type[*NetQASMInstruction*]]) –

get_instr_by_id(*id*)

Parameters **id** (int) –

get_instr_by_name(*name*)

Parameters **name** (str) –

abstract property instrs

class netqasm.lang.instr.flavour.**VanillaFlavour**
Bases: *netqasm.lang.instr.flavour.Flavour*

property instrs

class netqasm.lang.instr.flavour.**NVFlavour**
Bases: *netqasm.lang.instr.flavour.Flavour*

property instrs

3.2.6 netqasm.lang.instr.nv

class netqasm.lang.instr.nv.**GateXInstruction**(*id=20*, *mnemonic='x'*, *lineno=None*, *reg=None*)
Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: int = 20

mnemonic: **str = 'x'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.nv.**GateYInstruction**(*id=21*, *mnemonic='y'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 21**

mnemonic: **str = 'y'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.nv.**GateZInstruction**(*id=22*, *mnemonic='z'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 22**

mnemonic: **str = 'z'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.nv.**GateHInstruction**(*id=23*, *mnemonic='h'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 23**

mnemonic: **str = 'h'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.nv.**RotXInstruction**(*id=27*, *mnemonic='rot_x'*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.RotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

id: int = 27

mnemonic: str = 'rot_x'

to_matrix()

Return type ndarray

class netqasm.lang.instr.nv.**RotYInstruction**(*id=28*, *mnemonic='rot_y'*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.RotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

id: int = 28

mnemonic: str = 'rot_y'

to_matrix()

Return type ndarray

class netqasm.lang.instr.nv.**RotZInstruction**(*id=29*, *mnemonic='rot_z'*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.RotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –

- **imm1** (Optional[*Immediate*]) –

id: **int = 29**

mnemonic: **str = 'rot_z'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.nv.**ControlledRotXInstruction**(*id=30*, *mnemonic='crot_x'*, *lineno=None*, *reg0=None*, *reg1=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.ControlledRotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

id: **int = 30**

mnemonic: **str = 'crot_x'**

to_matrix()

Return type ndarray

to_matrix_target_only()

Return type ndarray

class netqasm.lang.instr.nv.**ControlledRotYInstruction**(*id=31*, *mnemonic='crot_y'*, *lineno=None*, *reg0=None*, *reg1=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.ControlledRotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

id: **int = 31**

mnemonic: **str = 'crot_y'**

to_matrix()

Return type ndarray

to_matrix_target_only()

Return type ndarray

3.2.7 netqasm.lang.instr.vanilla

class netqasm.lang.instr.vanilla.**GateXInstruction**(*id=20*, *mnemonic='x'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: int = 20

mnemonic: str = 'x'

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**GateYInstruction**(*id=21*, *mnemonic='y'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: int = 21

mnemonic: str = 'y'

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**GateZInstruction**(*id=22*, *mnemonic='z'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 22**

mnemonic: **str = 'z'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**GateHInstruction**(*id=23*, *mnemonic='h'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 23**

mnemonic: **str = 'h'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**GateSInstruction**(*id=24*, *mnemonic='s'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 24**

mnemonic: **str = 's'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**GateKInstruction**(*id=25*, *mnemonic='k'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: **int = 25**

mnemonic: **str = 'k'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**GateTInstruction**(*id=26*, *mnemonic='t'*, *lineno=None*, *reg=None*)

Bases: *netqasm.lang.instr.core.SingleQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –

id: int = 26

mnemonic: str = 't'

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**RotXInstruction**(*id=27*, *mnemonic='rot_x'*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.RotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

id: int = 27

mnemonic: str = 'rot_x'

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**RotYInstruction**(*id=28*, *mnemonic='rot_y'*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.RotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –

- **imm1** (Optional[*Immediate*]) –

id: **int = 28**

mnemonic: **str = 'rot_y'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**RotZInstruction**(*id=29*, *mnemonic='rot_z'*, *lineno=None*, *reg=None*, *imm0=None*, *imm1=None*)

Bases: *netqasm.lang.instr.core.RotationInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg** (Optional[*Register*]) –
- **imm0** (Optional[*Immediate*]) –
- **imm1** (Optional[*Immediate*]) –

id: **int = 29**

mnemonic: **str = 'rot_z'**

to_matrix()

Return type ndarray

class netqasm.lang.instr.vanilla.**CnotInstruction**(*id=30*, *mnemonic='cnot'*, *lineno=None*, *reg0=None*, *reg1=None*)

Bases: *netqasm.lang.instr.core.TwoQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –

id: **int = 30**

mnemonic: **str = 'cnot'**

to_matrix()

Return type ndarray

to_matrix_target_only()

Return type ndarray

class netqasm.lang.instr.vanilla.**CphaseInstruction**(*id=31*, *mnemonic='cphase'*, *lineno=None*, *reg0=None*, *reg1=None*)

Bases: *netqasm.lang.instr.core.TwoQubitInstruction*

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –

id: int = 31

mnemonic: str = 'cphase'

to_matrix()

Return type ndarray

to_matrix_target_only()

Return type ndarray

class netqasm.lang.instr.vanilla.**MovInstruction**(*id=41*, *mnemonic='mov'*, *lineno=None*, *reg0=None*, *reg1=None*)

Bases: *netqasm.lang.instr.core.TwoQubitInstruction*

Move source qubit to target qubit (target is overwritten)

Parameters

- **id** (int) –
- **mnemonic** (str) –
- **lineno** (Optional[*HostLine*]) –
- **reg0** (Optional[*Register*]) –
- **reg1** (Optional[*Register*]) –

id: int = 41

mnemonic: str = 'mov'

to_matrix()

Return type ndarray

to_matrix_target_only()

Return type ndarray

3.2.8 netqasm.lang.ir

class netqasm.lang.ir.**GenericInstr**(*value*)

Bases: enum.Enum

An enumeration.

QALLOC = 1

INIT = 2

ARRAY = 3

```
SET = 4
STORE = 5
LOAD = 6
UNDEF = 7
LEA = 8
JMP = 9
BEZ = 10
BNZ = 11
BEQ = 12
BNE = 13
BLT = 14
BGE = 15
ADD = 16
SUB = 17
ADDM = 18
SUBM = 19
X = 20
Y = 21
Z = 22
H = 23
S = 24
K = 25
T = 26
ROT_X = 27
ROT_Y = 28
ROT_Z = 29
CNOT = 30
CPHASE = 31
MEAS = 32
MEAS_BASIS = 33
CREATE_EPR = 34
RECV_EPR = 35
WAIT_ALL = 36
WAIT_ANY = 37
WAIT_SINGLE = 38
QFREE = 39
```

RET_REG = 40

RET_ARR = 41

CROT_X = 42

CROT_Y = 43

CROT_Z = 44

MOV = 45

BREAKPOINT = 46

class netqasm.lang.ir.**BreakpointAction**(*value*)
Bases: enum.Enum

An enumeration.

NOP = 0

DUMP_LOCAL_STATE = 1

DUMP_GLOBAL_STATE = 2

class netqasm.lang.ir.**BreakpointRole**(*value*)
Bases: enum.Enum

An enumeration.

CREATE = 0

RECEIVE = 1

netqasm.lang.ir.**instruction_to_string**(*instr*)

netqasm.lang.ir.**flip_branch_instr**(*instr*)

Parameters **instr** (*GenericInstr*) –

Return type *GenericInstr*

netqasm.lang.ir.**string_to_instruction**(*instr_str*)

class netqasm.lang.ir.**ICmd**(*instruction*, *args=None*, *operands=None*, *lineno=None*)
Bases: object

Parameters

- **instruction** (GenericInstr) –
- **args** (*List[int]*) –
- **operands** (*List[T_ProtoOperand]*) –
- **lineno** (*Optional[log.HostLine]*) –

instruction: *GenericInstr*

args: List[int] = None

operands: List[T_ProtoOperand] = None

lineno: Optional[log.HostLine] = None

property debug_str

class netqasm.lang.ir.**BranchLabel**(*name*, *lineno=None*)
Bases: object

Parameters

- **name** (*str*) –
- **lineno** (*Optional[log.HostLine]*) –

name: str

lineno: Optional[log.HostLine] = None

property debug_str

class netqasm.lang.ir.**ProtoSubroutine**(*commands=None*, *arguments=None*, *netqasm_version=(0, 10)*, *app_id=None*)

Bases: object

A *ProtoSubroutine* object represents a preliminary subroutine that consists of general 'commands' that might not yet be valid NetQASM instructions. These commands can include labels, or instructions with immediates that still need to be converted to registers.

ProtoSubroutines can optionally have *arguments*, which are yet-to-be-defined variables that are used in one or more of the commands in the ProtoSubroutine. So, a ProtoSubroutine can be seen as a function which takes certain parameters (arguments). Concrete values for arguments can be given by instantiating (using the *instantiate* method).

ProtoSubroutine`s are currently only used by the sdk and the text parser (netqasm.parser.text). In both cases they are converted into `Subroutine objects before given to other package components.

Parameters

- **commands** (Optional[List[Union[*ICmd*, *BranchLabel*]]]) –
- **arguments** (Optional[List[str]]) –
- **netqasm_version** (Tuple[int, int]) –
- **app_id** (Optional[int]) –

property netqasm_version

Return type Tuple[int, int]

property app_id

Return type Optional[int]

property commands

Return type List[Union[*ICmd*, *BranchLabel*]]

property arguments

Return type List[str]

instantiate(*app_id*, *arguments*)

Parameters

- **app_id** (int) –
- **arguments** (Dict[str, int]) –

Return type None

3.2.9 netqasm.lang.operand

class netqasm.lang.operand.**Operand**
Bases: object

class netqasm.lang.operand.**Immediate**(*value*)
Bases: *netqasm.lang.operand.Operand*

Parameters **value** (int) –

value: int

class netqasm.lang.operand.**Register**(*name*, *index*)
Bases: *netqasm.lang.operand.Operand*

Parameters

- **name** (*RegisterName*) –
- **index** (int) –

name: *netqasm.lang.encoding.RegisterName*

index: int

property cstruct

classmethod from_raw(*raw*)

Parameters **raw** (*Register*) –

class netqasm.lang.operand.**Address**(*address*)
Bases: *netqasm.lang.operand.Operand*

Parameters **address** (int) –

address: int

property cstruct

classmethod from_raw(*raw*)

Parameters **raw** (*Address*) –

class netqasm.lang.operand.**ArrayEntry**(*address*, *index*)
Bases: *netqasm.lang.operand.Operand*

Parameters

- **address** (*Address*) –
- **index** (Union[*Register*, int]) –

address: *netqasm.lang.operand.Address*

index: Union[*netqasm.lang.operand.Register*, int]

property cstruct

classmethod from_raw(*raw*)

Parameters **raw** (*ArrayEntry*) –

class netqasm.lang.operand.**ArraySlice**(*address*, *start*, *stop*)
Bases: *netqasm.lang.operand.Operand*

Parameters

- **address** (*Address*) –

- **start** (Union[*Register*, int]) –
- **stop** (Union[*Register*, int]) –

address: *netqasm.lang.operand.Address*

start: Union[*netqasm.lang.operand.Register*, int]

stop: Union[*netqasm.lang.operand.Register*, int]

property cstruct

classmethod from_raw(*raw*)

Parameters **raw** (*ArraySlice*) –

class netqasm.lang.operand.**Label**(*name*)

Bases: object

Parameters **name** (str) –

name: str

class netqasm.lang.operand.**Template**(*name*)

Bases: *netqasm.lang.operand.Operand*

An operand that does not have a concrete value (it can be filled in later).

Parameters **name** (str) –

name: str

3.2.10 netqasm.lang.parsing

3.2.11 netqasm.lang.parsing.binary

class netqasm.lang.parsing.binary.**Deserializer**(*flavour*)

Bases: object

Deserializes raw bytes into a Subroutine, given a Flavour. NetQASMInstructions are immediately created from the binary encoding.

(This is in contrast with the parsing.text module, which first converts the input to a *ProtoSubroutine*, consisting of ICmd s, before transforming it into a *Subroutine* containing *NetQASMInstruction* s.)

Parameters **flavour** (*Flavour*) –

deserialize_subroutine(*raw*)

Parameters **raw** (bytes) –

Return type *Subroutine*

deserialize_command(*raw*)

Parameters **raw** (bytes) –

Return type *NetQASMInstruction*

netqasm.lang.parsing.binary.**deserialize**(*data*, *flavour=None*)

Convert a binary encoding into a Subroutine object. The Vanilla flavour is used by default.

Parameters

- **data** (bytes) –

- **flavour** (Optional[*Flavour*]) –

Return type *Subroutine*

3.2.12 netqasm.lang.parsing.text

netqasm.lang.parsing.text.**parse_text_protosubroutine**(*text*)

Convert a text representation of a subroutine into a ProtoSubroutine object.

Parameters **text** (str) –

Return type *ProtoSubroutine*

netqasm.lang.parsing.text.**parse_text_subroutine**(*subroutine*, *assign_branch_labels=True*, *make_args_operands=True*, *replace_constants=True*, *flavour=None*)

Convert a text representation of a subroutine into a Subroutine object.

Internally, first a *ProtoSubroutine* object is created, consisting of *ICmd`s. This is then converted into a `Subroutine* using *assemble_subroutine*.

Parameters

- **subroutine** (str) –
- **flavour** (Optional[*Flavour*]) –

Return type *Subroutine*

netqasm.lang.parsing.text.**assemble_subroutine**(*pre_subroutine*, *assign_branch_labels=True*, *make_args_operands=True*, *replace_constants=True*, *flavour=None*)

Convert a *ProtoSubroutine* into a *Subroutine*, given a Flavour (default: vanilla).

Parameters

- **pre_subroutine** (*ProtoSubroutine*) –
- **flavour** (Optional[*Flavour*]) –

Return type *Subroutine*

netqasm.lang.parsing.text.**parse_register**(*register*)

Parameters **register** (str) –

Return type *Register*

netqasm.lang.parsing.text.**parse_address**(*address*)

Parameters **address** (str) –

Return type Union[*Address*, *ArraySlice*, *ArrayEntry*]

netqasm.lang.parsing.text.**get_current_registers**(*commands*)

Parameters **commands** (List[Union[*ICmd*, *BranchLabel*]]) –

Return type Set[str]

3.2.13 netqasm.lang.subroutine

NetQASM subroutine definitions.

This module contains the *Subroutine* class which represents a static (not being executed) NetQASM subroutine.

class netqasm.lang.subroutine.**Subroutine**(*instructions=None*, *arguments=None*, *netqasm_version=(0, 10)*, *app_id=None*)

Bases: object

A *Subroutine* object represents a subroutine consisting of valid instructions, i.e. objects deriving from *NetQASMInstruction*.

Subroutines can optionally have *arguments*, which are yet-to-be-defined variables that are used in one or more of the instructions in the Subroutine. So, a Subroutine can be seen as a function which takes certain parameters (arguments). Concrete values for arguments can be given by instantiating (using the *instantiate* method).

`Subroutine`s are executed by `Executor`s.

Parameters

- **instructions** (Optional[List[*NetQASMInstruction*]]) –
- **arguments** (Optional[List[str]]) –
- **netqasm_version** (Tuple[int, int]) –
- **app_id** (Optional[int]) –

property netqasm_version

Return type Tuple[int, int]

property app_id

Return type Optional[int]

property instructions

Return type List[*NetQASMInstruction*]

property arguments

Return type List[str]

instantiate(*app_id*, *arguments=None*)

Parameters

- **app_id** (int) –
- **arguments** (Optional[Dict[str, int]]) –

Return type None

property cstructs

3.2.14 netqasm.lang.symbols

class netqasm.lang.symbols.**Symbols**
Bases: object

COMMENT_START = '//'

BRANCH_END = ':'

MACRO_START = '$'

ADDRESS_START = '@'

ARGS_BRACKETS = '()'

ARGS_DELIM = ','

INDEX_BRACKETS = '[]'

SLICE_DELIM = ':'

TEMPLATE_BRACKETS = '{}'

PREAMBLE_START = '#'

PREAMBLE_NETQASM = 'NETQASM'

PREAMBLE_APPID = 'APPID'

PREAMBLE_DEFINE = 'DEFINE'

PREAMBLE_DEFINE_BRACKETS = '{}'

3.3 netqasm.logging

3.3.1 netqasm.logging.output

netqasm.logging.output.**should_ignore_instr**(*instr*)

netqasm.logging.output.**reset_struct_loggers**()

netqasm.logging.output.**save_all_struct_loggers**()

class netqasm.logging.output.**StructuredLogger**(*filepath*)
Bases: abc.ABC

log(**args*, ***kwargs*)

save()

class netqasm.logging.output.**InstrLogger**(*filepath*, *executor*)
Bases: *netqasm.logging.output.StructuredLogger*

Parameters **filepath** (str) –

log(**args*, ***kwargs*)

save()

class netqasm.logging.output.**NetworkLogger**(*filepath*)
Bases: *netqasm.logging.output.StructuredLogger*

log(**args*, ***kwargs*)

save()

class netqasm.logging.output.**SocketOperation**(*value*)
Bases: enum.Enum

An enumeration.

SEND = 'SEND'

RECV = 'RECV'

WAIT_RECV = 'WAIT_RECV'

class netqasm.logging.output.**ClassCommLogger**(*filepath*)
Bases: *netqasm.logging.output.StructuredLogger*

log(**args*, ***kwargs*)

save()

class netqasm.logging.output.**AppLogger**(*filepath*, *log_config*)
Bases: *netqasm.logging.output.StructuredLogger*

log(**args*, ***kwargs*)

save()

netqasm.logging.output.**get_new_app_logger**(*app_name*, *log_config*)

3.3.2 netqasm.logging.glob

netqasm.logging.glob.**get_netqasm_logger**(*sub_logger=None*)

Parameters **sub_logger** (Optional[str]) –

Return type Logger

netqasm.logging.glob.**set_log_level**(*level*)

Parameters **level** (Union[int, str]) –

Return type None

netqasm.logging.glob.**get_log_level**(*effective=True*)

Parameters **effective** (bool) –

Return type int

3.4 netqasm.runtime

3.4.1 netqasm.runtime.app_config

class netqasm.runtime.app_config.**AppConfig**(*app_name*, *node_name*, *main_func*, *log_config*, *inputs*)
Bases: object

Parameters

- **app_name** (str) –
- **node_name** (str) –
- **main_func** (Callable) –

- **log_config** (Optional[*LogConfig*]) –
- **inputs** (Dict[str, Any]) –

app_name: str

node_name: str

main_func: Callable

log_config: Optional[*netqasm.sdk.config.LogConfig*]

inputs: Dict[str, Any]

netqasm.runtime.app_config.**default_app_config**(*app_name*, *main_func*)

Parameters

- **app_name** (str) –
- **main_func** (Callable) –

Return type *AppConfig*

3.4.2 netqasm.runtime.application

NetQASM application definitions.

NetQASM applications are modeled as pieces of Python code together with metadata. Generally, applications are multi-node, i.e. they consist of separate chunks of code that are run by separate nodes.

To distinguish the notion of a multiple-node-spanning collection of code and single-node piece of code, the following terminology is used:

- A *Program* is code that runs on a single node. It is a Python script whose code is executed on (1) the Host component of that node and (2) the quantum node controller of that node.
- An *Application* is a collection of Programs (specifically, one Program per node).

class netqasm.runtime.application.**Program**(*party*, *entry*, *args*, *results*)

Bases: object

Program running on one specific node. Part of a multi-node application.

Parameters

- **party** (str) – name of the party or role in the multi-node application (protocol). E.g. a blind computation application may have two parties: "client" and "server". Note that the party name is not necessarily the name of the node this Program runs on (which may be, e.g. "Delft").
- **entry** (Callable) – entry point of the Program. This must be Python function.
- **args** (List[str]) – list of argument names that the entry point expects
- **results** (List[str]) – list of result names that are keys in the dictionary that this Program returns on completion

party: str

entry: Callable

args: List[str]

results: List[str]

class netqasm.runtime.application.**AppMetadata**(*name*, *description*, *authors*, *version*)
Bases: object

Metadata about a NetQASM application.

Parameters

- **name** (str) – name of the application
- **description** (str) – description of the application
- **authors** (List[str]) – list of authors of the application
- **version** (str) – application version

name: str

description: str

authors: List[str]

version: str

class netqasm.runtime.application.**Application**(*programs*, *metadata*)
Bases: object

Static NetQASM application (or protocol) information.

Parameters

- **programs** (List[*Program*]) – list of programs for each of the parties that are involved in this application (or protocol).
- **metadata** (Optional[*AppMetadata*]) – application metadata

programs: List[*netqasm.runtime.application.Program*]

metadata: Optional[*netqasm.runtime.application.AppMetadata*]

class netqasm.runtime.application.**ApplicationInstance**(*app*, *program_inputs*, *network*, *party_alloc*, *logging_cfg*)
Bases: object

Instantiation of a NetQASM application with concrete input values and configuration of the underlying network.

Parameters

- **app** (*Application*) – static application info
- **program_inputs** (Dict[str, Dict[str, Any]]) – program input values for each of the application's programs
- **network** (Optional[*NetworkConfig*]) – configuration for a simulated network
- **party_alloc** (Dict[str, str]) – mapping of application parties to nodes in the network
- **logging_cfg** (Optional[*LogConfig*]) – logging configuration

app: *netqasm.runtime.application.Application*

program_inputs: Dict[str, Dict[str, Any]]

network: Optional[*netqasm.runtime.interface.config.NetworkConfig*]

party_alloc: Dict[str, str]

logging_cfg: **Optional[*netqasm.sdk.config.LogConfig*]**

class netqasm.runtime.application.**ApplicationOutput**
Bases: object

Results of a finished run of an ApplicationInstance. Should be subclassed.

netqasm.runtime.application.**load_yaml_file**(*path*)

Parameters **path** (str) –

Return type Any

netqasm.runtime.application.**app_instance_from_path**(*app_dir=None*)
Create an Application Instance based on files in a directory. Uses the current working directory if *app_dir* is None.

Parameters **app_dir** (Optional[str]) –

Return type *ApplicationInstance*

netqasm.runtime.application.**default_app_instance**(*programs*)
Create an Application Instance with programs that take no arguments.

Parameters **programs** (List[Tuple[str, Callable]]) –

Return type *ApplicationInstance*

netqasm.runtime.application.**network_cfg_from_path**(*app_dir=None*, *network_config_file=None*)

Parameters

- **app_dir** (Optional[str]) –
- **network_config_file** (Optional[str]) –

Return type Optional[*NetworkConfig*]

netqasm.runtime.application.**post_function_from_path**(*app_dir=None*, *post_function_file=None*)

Parameters

- **app_dir** (Optional[str]) –
- **post_function_file** (Optional[str]) –

Return type Optional[Callable]

3.4.3 netqasm.runtime.cli

Command-line interface of the *netqasm* executable.

This module defines the commands that may be used when using *netqasm* as a program on the command line, as well as handlers for these commands.

3.4.4 netqasm.runtime.debug

netqasm.runtime.debug.**get_qubit_state**(*qubit*, *reduced_dm=True*)

netqasm.runtime.debug.**run_application**(*app_instance*, *post_function=None*, *in-str_log_dir=None*, *results_file=None*, *use_app_config=True*)

Parameters **app_instance** (*ApplicationInstance*) –

3.4.5 netqasm.runtime.env

Tools for dealing with files related to NetQASM application execution.

netqasm.runtime.env.**load_app_config_file**(*app_dir*, *app_name*)

Return type Any

netqasm.runtime.env.**get_roles_config_path**(*app_dir*)

netqasm.runtime.env.**load_roles_config**(*roles_config_file*)

netqasm.runtime.env.**load_app_files**(*app_dir*)

Return type Dict[str, str]

netqasm.runtime.env.**get_log_dir**(*app_dir*)

netqasm.runtime.env.**get_timed_log_dir**(*log_dir*)

netqasm.runtime.env.**get_post_function_path**(*app_dir*)

netqasm.runtime.env.**load_post_function**(*post_function_file*)

netqasm.runtime.env.**get_results_path**(*timed_log_dir*)

netqasm.runtime.env.**new_folder**(*path*, *template='teleport'*, *quiet=False*)
Used by the CLI to create an app folder template

Parameters

- **path** (*str*) – Path to the directory
- **template** (*str*) – Which pre-defined app to use as template
- **quiet** (*bool*) – Whether to print info to stdout or not (default *False*)

netqasm.runtime.env.**init_folder**(*path*, *quiet=False*)
Used by the CLI to initialize a directory by adding missing config files.

Parameters

- **path** (*str*) – Path to the directory
- **quiet** (*bool*) – Whether to print info to stdout or not (default *False*)

netqasm.runtime.env.**file_creation_notify**(*func*)
Decorator for notification about file creation

netqasm.runtime.env.**get_example_apps**()

3.4.6 netqasm.runtime.hardware

Execution of application scripts without setting up a backend.

The *run_applications* function simply spawns a thread for each of the applications given to it, and runs the Python script of each application. The relevant quantum node controllers are expected to be setup elsewhere, e.g. as real hardware that is connected to the machine that runs *run_applications*.

netqasm.runtime.hardware.**run_application**(*app_instance*, *post_function=None*, *results_file=None*, *use_app_config=True*)

Parameters **app_instance** (*ApplicationInstance*) –

netqasm.runtime.hardware.**save_results**(*results*, *results_file*)

3.4.7 netqasm.runtime.interface

3.4.8 netqasm.runtime.interface.config

class netqasm.runtime.interface.config.**QuantumHardware**(*value*)

Bases: enum.Enum

An enumeration.

Generic = 'Generic'

NV = 'NV'

TrappedIon = 'TrappedIon'

class netqasm.runtime.interface.config.**NoiseType**(*value*)

Bases: enum.Enum

An enumeration.

NoNoise = 'NoNoise'

Depolarise = 'Depolarise'

DiscreteDepolarise = 'DiscreteDepolarise'

Bitflip = 'Bitflip'

class netqasm.runtime.interface.config.**Qubit**(*id*, *t1*, *t2*)

Bases: object

Parameters

- **id** (int) –
- **t1** (float) –
- **t2** (float) –

id: int

t1: float

t2: float

class netqasm.runtime.interface.config.**Node**(*name*, *hardware*, *qubits*, *gate_fidelity=1.0*)

Bases: object

Parameters

- **name** (str) –
- **hardware** (*QuantumHardware*) –
- **qubits** (List[*Qubit*]) –
- **gate_fidelity** (float) –

name: str

hardware: *netqasm.runtime.interface.config.QuantumHardware*

qubits: List[*netqasm.runtime.interface.config.Qubit*]

gate_fidelity: float = 1.0

class netqasm.runtime.interface.config.**Link**(*name*, *node_name1*, *node_name2*, *noise_type*, *fidelity*)

Bases: object

Parameters

- **name** (str) –
- **node_name1** (str) –
- **node_name2** (str) –
- **noise_type** (*NoiseType*) –
- **fidelity** (float) –

name: str

node_name1: str

node_name2: str

noise_type: *netqasm.runtime.interface.config.NoiseType*

fidelity: float

class netqasm.runtime.interface.config.**NetworkConfig**(*nodes*, *links*)

Bases: object

Parameters

- **nodes** (List[*Node*]) –
- **links** (List[*Link*]) –

nodes: List[*netqasm.runtime.interface.config.Node*]

links: List[*netqasm.runtime.interface.config.Link*]

netqasm.runtime.interface.config.**default_network_config**(*node_names*, *hardware=<QuantumHardware.Generic: 'Generic'>*)

Create a config for a fully connected network of nodes with the given names

Parameters

- **node_names** (List[str]) –
- **hardware** (*QuantumHardware*) –

Return type *NetworkConfig*

netqasm.runtime.interface.config.**parse_network_config**(*cfg*)

Return type *NetworkConfig*

netqasm.runtime.interface.config.**network_cfg_from_file**(*network_config_file=None*)

Parameters **network_config_file** (Optional[str]) –

Return type *NetworkConfig*

3.4.9 netqasm.runtime.interface.logging

class netqasm.runtime.interface.logging.**QubitGroup**(*is_entangled*, *qubit_ids*, *state*)

Bases: object

Parameters

- **is_entangled** (Optional[bool]) –
- **qubit_ids** (List[List[Union[str, int]]]) –
- **state** (Optional[Tuple[Tuple[complex, complex], Tuple[complex, complex]]]) –

is_entangled: Optional[bool]

qubit_ids: List[List[Union[str, int]]]

state: Optional[Tuple[Tuple[complex, complex], Tuple[complex, complex]]]

class netqasm.runtime.interface.logging.**EntanglementType**(*value*)

Bases: enum.Enum

An enumeration.

CK = 'CK'

MD = 'MD'

class netqasm.runtime.interface.logging.**EntanglementStage**(*value*)

Bases: enum.Enum

An enumeration.

START = 'start'

FINISH = 'finish'

class netqasm.runtime.interface.logging.**InstrLogEntry**(*WCT*, *SIT*, *AID*, *SID*, *PRC*, *HLN*, *HFL*, *INS*, *OPR*, *ANG*, *QID*, *VID*, *OUT*, *QGR*, *LOG*)

Bases: object

Parameters

- **WCT** (str) –
- **SIT** (int) –
- **AID** (int) –
- **SID** (int) –
- **PRC** (int) –
- **HLN** (int) –
- **HFL** (str) –

- **INS** (str) –
- **OPR** (List[str]) –
- **ANG** (Optional[Dict[str, int]]) –
- **QID** (List[int]) –
- **VID** (List[int]) –
- **OUT** (Optional[int]) –
- **QGR** (Optional[Dict[int, *QubitGroup*]]) –
- **LOG** (str) –

WCT: str
Wall clock time. Format is Python's *datetime.now()*.

SIT: int
Time in NetSquid simulation, in nanoseconds.

AID: int
App ID, used internally by the backend.

SID: int
Subroutine ID. Used internally by the Executor.

PRC: int
Program counter. Used internally by the Executor.

HLN: int
Host line number. Line number in source file (.py) related to what is currently executed. The line is in the file given by HFL (see below).

HFL: str
Host file. Source file (.py) of current executed instruction.

INS: str
Mnemonic of the NetQASM instruction being executed.

OPR: List[str]
Operands (register, array-entries..). List of "op=val" strings

ANG: Optional[Dict[str, int]]
Angle represented as the fraction num/den. For non-rotation instructions, ANG is None. For rotation instructions ANG is a dictionary with 2 entries: 'num' (an int) and 'den' (an int).

QID: List[int]
Physical qubit IDs of qubits part of the current operation.

VID: List[int]
Virtual qubit IDs of qubits part of the current operation.

OUT: Optional[int]
Measurement outcome. Only set in case of a measurement instruction.

QGR: Optional[Dict[int, *netqasm.runtime.interface.logging.QubitGroup*]]
Dictionary specifying groups of qubits across the network.

LOG: str
Human-readable message.

class netqasm.runtime.interface.logging.**NetworkLogEntry**(*WCT*, *SIT*, *TYP*, *INS*, *BAS*, *MSR*, *NOD*, *PTH*, *QID*, *QGR*, *LOG*)

Bases: object

Parameters

- **WCT** (str) –
- **SIT** (int) –
- **TYP** (Optional[*EntanglementType*]) –
- **INS** (*EntanglementStage*) –
- **BAS** (Optional[List[int]]) –
- **MSR** (List[int]) –
- **NOD** (List[str]) –
- **PTH** (List[str]) –
- **QID** (List[int]) –
- **QGR** (Optional[Dict[int, *QubitGroup*]]) –
- **LOG** (str) –

WCT: str
Wall clock time. Format is Python's *datetime.now()*.

SIT: int
Time in NetSquid simulation, in nanoseconds.

TYP: Optional[*netqasm.runtime.interface.logging.EntanglementType*]
Entanglement generation type(Measure Directly or Create and Keep). For the 'start' entanglement stage (see INS below), this value is None since at this stage the value cannot be determined yet. For the 'finish' stage, the correct value is filled in, however.

INS: *netqasm.runtime.interface.logging.EntanglementStage*
Entanglement generation stage(start or finish).

BAS: Optional[List[int]]
Bases in which the two qubits(one on each end) were measured in . Only applies to the Measure Directly case. It is *None* otherwise.

MSR: List[int]
Measurement outcomes of the two qubits(one on each end). Only applies to the Measure Directly case. It is *None* otherwise.

NOD: List[str]
Node names involved in this entanglement operation.

PTH: List[str]
Path of links used for entanglement generation. Links are identified using their names.

QID: List[int]
Physical qubit IDs of qubits part of the current operation.

QGR: Optional[Dict[int, *netqasm.runtime.interface.logging.QubitGroup*]]
Dictionary specifying groups of qubits across the network, as they are after the current operation.

LOG: str
Human-readable message.

class netqasm.runtime.interface.logging.**ClassCommLogEntry**(*WCT*, *HLN*, *HFL*, *INS*, *MSG*, *SEN*, *REC*, *SOD*, *LOG*)

Bases: object

Parameters

- **WCT** (str) –
- **HLN** (int) –
- **HFL** (str) –
- **INS** (str) –
- **MSG** (str) –
- **SEN** (str) –
- **REC** (str) –
- **SOD** (str) –
- **LOG** (str) –

WCT: str
Wall clock time. Format is Python's *datetime.now()*.

HLN: int
Host line number. Line number in source file (.py) related to what is currently executed. The line is in the file given by HFL(see below).

HFL: str
Host file. Source file (.py) of current executed instruction.

INS: str
Classical operation being performed.

MSG: str
Message that is sent or received.

SEN: str
Name(role) of the sender.

REC: str
Name(role) of the receiver.

SOD: str
Socket ID(used internally).

LOG: str
Human-readable message.

class netqasm.runtime.interface.logging.**AppLogEntry**(*WCT*, *HLN*, *HFL*, *LOG*)

Bases: object

Parameters

- **WCT** (str) –
- **HLN** (int) –
- **HFL** (str) –
- **LOG** (str) –

WCT: str
Wall clock time. Format is Python's *datetime.now()*.

HLN: int
Host line number. Line number in source file (.py) related to what is currently executed. The line is in the file given by HFL(see below).

HFL: str

LOG: str
Human-readable message.

3.4.10 netqasm.runtime.interface.results

3.4.11 netqasm.runtime.process_logs

netqasm.runtime.process_logs.**process_log**(*log_dir*)

netqasm.runtime.process_logs.**make_last_log**(*log_dir*)

netqasm.runtime.process_logs.**create_app_instr_logs**(*log_dir*)

3.4.12 netqasm.runtime.settings

class netqasm.runtime.settings.**Simulator**(*value*)
Bases: enum.Enum

An enumeration.

NETSQUID = 'netsquid'

NETSQUID_SINGLE_THREAD = 'netsquid_single_thread'

SIMULAQRON = 'simulaqron'

DEBUG = 'debug'

class netqasm.runtime.settings.**Formalism**(*value*)
Bases: enum.Enum

An enumeration.

STAB = 'stab'

KET = 'ket'

DM = 'dm'

class netqasm.runtime.settings.**Flavour**(*value*)
Bases: enum.Enum

An enumeration.

VANILLA = 'vanilla'

NV = 'nv'

netqasm.runtime.settings.**set_simulator**(*simulator*)

netqasm.runtime.settings.**get_simulator**()

netqasm.runtime.settings.**set_is_using_hardware**(*b*)

netqasm.runtime.settings.**get_is_using_hardware**()

3.5 netqasm.sdk

3.5.1 netqasm.sdk.builder

Conversion from Python code into an NetQASM subroutines.

This module contains the *Builder* class, which is used by a Connection to transform Python application script code into NetQASM subroutines.

class netqasm.sdk.builder.**LabelManager**
Bases: object

Simple manager class for providing unique branch labels.

new_label(*start_with=''*)

Parameters **start_with** (str) –

Return type str

class netqasm.sdk.builder.**SdkIfContext**(*id*, *builder*, *condition*, *op0*, *op1*)
Bases: object

Context object for if statements in SDK code such as *with conn.if_eq()*.

Parameters

- **id** (int) –
- **builder** (*Builder*) –
- **condition** (*GenericInstr*) –
- **op0** (Union[int, ForwardRef, ForwardRef, None]) –
- **op1** (Union[int, ForwardRef, ForwardRef, None]) –

class netqasm.sdk.builder.**SdkForEachContext**(*id*, *builder*, *array*, *return_index*)
Bases: object

Context object for foreach statements in SDK code such as *with conn.foreach()*.

Parameters

- **id** (int) –
- **builder** (*Builder*) –
- **array** (*Array*) –
- **return_index** (bool) –

class netqasm.sdk.builder.**SdkLoopUntilContext**(*id*, *builder*, *max_iterations*)
Bases: object

Context object for loop_until() statements in SDK code.

Parameters

- **id** (int) –
- **builder** (*Builder*) –

- **max_iterations** (int) –

set_exit_condition(*constraint*)
Set the exit condition for this while loop.

Parameters **constraint** (SdkConstraint) –

Return type None

property exit_condition
Get the exit condition for this while loop.

Return type Optional[SdkConstraint]

set_cleanup_code(*cleanup_code*)
Set the cleanup code for this while loop.

Parameters **cleanup_code** (Callable[[ForwardRef], None]) –

Return type None

property cleanup_code
Get the cleanup code for this while loop.

Return type Optional[Callable[[ForwardRef], None]]

set_loop_register(*register*)
Set the register used that holds the iteration index for this while loop.

Parameters **register** (*RegFuture*) –

Return type None

property loop_register
Get the register used that holds the iteration index for this while loop.

Return type Optional[*RegFuture*]

property max_iterations
Get the maximum number of iterations for this while loop.

Return type int

class netqasm.sdk.builder.**Builder**(*connection*, *app_id*, *hardware_config=None*, *log_config=None*, *compiler=None*, *return_arrays=True*)
Bases: object

Object that transforms Python script code into `ProtoSubroutine`s.

A Connection uses a Builder to handle statements in application script code. The Builder converts the statements into pseudo-NetQASM instructions that are assembled into a ProtoSubroutine. When the connectin flushes, the ProtoSubroutine is is compiled into a NetQASM subroutine.

Parameters

- **connection** (BaseNetQASMConnection) –
- **app_id** (*int*) –
- **hardware_config** (*Optional[HardwareConfig]*) –
- **log_config** (*Optional[LogConfig]*) –
- **compiler** (*Optional[Type[SubroutineTranspiler]]*) –
- **return_arrays** (*bool*) –

__init__(*connection*, *app_id*, *hardware_config=None*, *log_config=None*, *compiler=None*, *return_arrays=True*)

Builder constructor. Typically not used directly by the Host script.

Parameters

- **connection** (BaseNetQASMConnection) – Connection that this builder builds for
- **app_id** (*int*) – ID of the application as given by the quantum node controller
- **max_qubits** – maximum number of qubits allowed (as registered with the quantum node controller)
- **log_config** (*Optional[LogConfig]*) – logging configuration, typically just passed as-is by the connection object
- **compiler** (*Optional[Type[SubroutineTranspiler]]*) – which compiler class to use for the translation from ProtoSubroutine to Subroutine
- **return_arrays** (*bool*) – whether to add ret_arr NetQASM instructions at the end of each subroutine (for all arrays that are used in the subroutine). May be set to False if the quantum node controller does not support returning arrays.
- **hardware_config** (*Optional[HardwareConfig]*) –

property app_id

Return type int

inactivate_qubits()

Return type None

new_qubit_id()

Return type int

alloc_array(*length=1*, *init_values=None*)

Parameters

- **length** (int) –
- **init_values** (Optional[List[Optional[int]]]) –

Return type *Array*

new_register(*init_value=0*)

Parameters **init_value** (int) –

Return type *RegFuture*

subrt_add_pending_commands(*commands*)

Parameters **commands** (List[Union[*ICmd*, *BranchLabel*]]) –

Return type None

subrt_add_pending_command(*command*)

Parameters **command** (Union[*ICmd*, *BranchLabel*]) –

Return type None

subrt_pop_all_pending_commands()

Return type List[Union[*ICmd*, *BranchLabel*]]

subrt_pop_pending_subroutine()

Return type Optional[*ProtoSubroutine*]

subrt_compile_subroutine(*pre_subroutine*)
Convert a ProtoSubroutine into a Subroutine.

Parameters **pre_subroutine** (*ProtoSubroutine*) –

Return type *Subroutine*

property committed_subroutines

Return type List[*Subroutine*]

if_context_enter(*context_id*)

Parameters **context_id** (int) –

Return type None

if_context_exit(*context_id*, *condition*, *op0*, *op1*)

Parameters

- **context_id** (int) –
- **condition** (*GenericInstr*) –
- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **op1** (Union[int, ForwardRef, ForwardRef, None]) –

Return type None

sdk_epr_keep(*role*, *params*, *reset_results_array=False*)
Build commands for an EPR keep operation and return the result futures.

Parameters

- **role** (EPRRole) –
- **params** (EntRequestParams) –
- **reset_results_array** (bool) –

Return type Tuple[List[*Qubit*], *Array*]

sdk_epr_measure(*role*, *params*)
Build commands for an EPR measure operation and return the result futures.

Parameters

- **role** (EPRRole) –
- **params** (EntRequestParams) –

Return type List[EprMeasureResult]

sdk_epr_rsp_create(*params*)
Build commands for a 'create remote state preparation' EPR operation and return the result futures.

Parameters **params** (EntRequestParams) –

Return type List[EprMeasureResult]

sdk_epr_rsp_recv(*params*)
Build commands for a 'receive remote state preparation' EPR operation and return the created qubits and result futures.

Parameters **params** (EntRequestParams) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

sdk_create_epr_keep(*params*)
Build commands for a 'create and keep' EPR operation and return the created qubits and result futures.

Parameters **params** (EntRequestParams) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

sdk_recv_epr_keep(*params*)
Build commands for a 'receive and keep' EPR operation and return the created qubits and result futures.

Parameters **params** (EntRequestParams) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

sdk_create_epr_measure(*params*)
Build commands for a 'create and measure' EPR operation and return the result futures.

Parameters **params** (EntRequestParams) –

Return type List[EprMeasureResult]

sdk_recv_epr_measure(*params*)
Build commands for a 'receive and measure' EPR operation and return the result futures.

Parameters **params** (EntRequestParams) –

Return type List[EprMeasureResult]

sdk_create_epr_rsp(*params*)
Build commands for a 'create remote state preperation' EPR operation and return the result futures.

Parameters **params** (EntRequestParams) –

Return type List[EprMeasureResult]

sdk_recv_epr_rsp(*params*)
Build commands for a 'receive remote state preparation' EPR operation and return the created qubits and result futures.

Parameters **params** (EntRequestParams) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

sdk_loop_context(*stop*, *start=0*, *step=1*, *loop_register=None*)
Build commands for a 'loop' context and return the context object.

Parameters

- **stop** (int) –
- **start** (int) –
- **step** (int) –
- **loop_register** (Union[*Register*, str, None]) –

Return type Iterator[*Register*]

sdk_loop_body(*body*, *stop*, *start=0*, *step=1*, *loop_register=None*)
Build commands for looping the code in the specified body.

Parameters

- **body** (Callable[[ForwardRef, *RegFuture*], None]) –
- **stop** (int) –

- **start** (int) –
- **step** (int) –
- **loop_register** (Union[*Register*, str, None]) –

Return type None

sdk_if_eq(*op0*, *op1*, *body*)
An effective if-statement where body is a function executing the clause for a == b

Parameters

- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **op1** (Union[int, ForwardRef, ForwardRef]) –
- **body** (Callable[[ForwardRef], None]) –

Return type None

sdk_if_ne(*op0*, *op1*, *body*)
An effective if-statement where body is a function executing the clause for a != b

Parameters

- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **op1** (Union[int, ForwardRef, ForwardRef]) –
- **body** (Callable[[ForwardRef], None]) –

Return type None

sdk_if_lt(*op0*, *op1*, *body*)
An effective if-statement where body is a function executing the clause for a < b

Parameters

- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **op1** (Union[int, ForwardRef, ForwardRef]) –
- **body** (Callable[[ForwardRef], None]) –

Return type None

sdk_if_ge(*op0*, *op1*, *body*)
An effective if-statement where body is a function executing the clause for a >= b

Parameters

- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **op1** (Union[int, ForwardRef, ForwardRef]) –
- **body** (Callable[[ForwardRef], None]) –

Return type None

sdk_if_ez(*op0*, *body*)
An effective if-statement where body is a function executing the clause for a == 0

Parameters

- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **body** (Callable[[ForwardRef], None]) –

Return type None

sdk_if_nz (*op0*, *body*)
An effective if-statement where body is a function executing the clause for a != 0

Parameters

- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **body** (Callable[[ForwardRef], None]) –

Return type None

sdk_new_if_context (*condition*, *op0*, *op1*)
Build commands for an 'if' context and return the context object.

Parameters

- **condition** (*GenericInstr*) –
- **op0** (Union[int, ForwardRef, ForwardRef]) –
- **op1** (Union[int, ForwardRef, ForwardRef, None]) –

Return type *SdkIfContext*

sdk_new_foreach_context (*array*, *return_index*)
Build commands for an 'foreach' context and return the context object.

Parameters

- **array** (*Array*) –
- **return_index** (bool) –

Return type *SdkForEachContext*

sdk_new_loop_until_context (*max_iterations*)
Build commands for a 'loop_until' context and return the context object.

Parameters **max_iterations** (int) –

Return type Iterator[*SdkLoopUntilContext*]

sdk_try_context (*max_tries=1*)
Build commands for a 'try' context.

Parameters **max_tries** (int) –

Return type Iterator[None]

sdk_create_epr_context (*params*)
Build commands for an EPR context and return an iterator over the EPR qubits and indices created in this context.

Parameters **params** (EntRequestParams) –

Return type Iterator[Tuple[*FutureQubit*, *RegFuture*]]

3.5.2 netqasm.sdk.classical_communication

3.5.3 netqasm.sdk.classical_communication.broadcast_channel

Interface for classical broadcasting between Hosts.

This module contains the *BroadcastChannel* class which is a base for representing classical broadcasting (sending classical messages to all other nodes in the network) between Hosts.

class netqasm.sdk.classical_communication.broadcast_channel.**BroadcastChannel**(*app_name*, *timeout=None*, *use_callbacks=False*)

Bases: abc.ABC

Socket for sending messages to all nodes in the network (broadcasting).

A BroadcastChannel can be used by Hosts to broadcast a message to all other nodes in the network. It is very similar to a Socket object, just without an explicit single remote node.

A BroadcastChannel is a local object that each node (that wants to send and receive broadcast messages) must instantiate themselves.

Parameters

- **app_name** (str) –
- **timeout** (Optional[float]) –
- **use_callbacks** (bool) –

__init__(*app_name*, *timeout=None*, *use_callbacks=False*)

BroadcastChannel constructor.

Parameters

- **app_name** (str) – application/Host name of this channel's constructor
- **timeout** (Optional[float]) – maximum time to try and create this channel before aborting
- **use_callbacks** (bool) – whether to use the *recv_callback* and *conn_lost_callback* callback methods

abstract send(*msg*)

Broadcast a message to all remote nodes.

Parameters **msg** (str) –

Return type None

abstract recv(*block=True*)

Receive a message that was broadcast and from whom.

Parameters **block** (bool) –

Return type Tuple[str, str]

recv_callback(*remote_app_name*, *msg*)

This method gets called when a message is received.

Subclass to define behaviour.

NOTE: This only happens if *self.use_callbacks* is set to *True*.

Parameters

- **remote_app_name** (str) –
- **msg** (str) –

Return type None

conn_lost_callback()
This method gets called when the connection is lost.

Subclass to define behaviour.

NOTE: This only happens if *self.use_callbacks* is set to *True*.

Return type None

class netqasm.sdk.classical_communication.broadcast_channel.**BroadcastChannelBySockets**(*app_n*
re-
*mote_*
***kwa*

Bases: *netqasm.sdk.classical_communication.broadcast_channel.BroadcastChannel*

Implementation of a BroadcastChannel using a Socket for each remote node.

Technically this is a multicast channel since the receiving nodes must be explicitly listed. It simply uses one-to-one sockets for every remote node.

Parameters

- **app_name** (str) –
- **remote_app_names** (List[str]) –

__init__(*app_name*, *remote_app_names*, ***kwargs*)
BroadcastChannel constructor.

Parameters

- **app_name** (str) – application/Host name of self
- **remote_app_names** (List[str]) – list of receiving remote Hosts

send(*msg*)
Broadcast a message to all remote nodes.

Parameters **msg** (str) –

Return type None

recv(*block=True*, *timeout=None*)
Receive a message that was broadcast.

Parameters

- **block** (*bool*) – Whether to block for an available message
- **timeout** (*float, optional*) – Optionally use a timeout for trying to recv a message. Only used if *block=True*.

Returns (remote_node_name, msg)

Return type tuple

Raises **RuntimeError** – If *block=False* and there is no available message

3.5.4 netqasm.sdk.classical_communication.thread_socket.broadcast_channel

BroadcastChannel implementation using ThreadSockets.

class netqasm.sdk.classical_communication.thread_socket.broadcast_channel.**ThreadBroadcastCl**

Bases: *netqasm.sdk.classical_communication.broadcast_channel.BroadcastChannelBySockets*

Parameters

- **app_name** (str) –
- **remote_app_names** (List[str]) –

3.5.5 netqasm.sdk.classical_communication.message

class netqasm.sdk.classical_communication.message.**StructuredMessage**(*header*, *payload*)

Bases: object

Parameters

- **header** (str) –
- **payload** (str) –

header: str

payload: str

3.5.6 netqasm.sdk.classical_communication.socket

Interface for classical communication between Hosts.

This module contains the *Socket* class which is a base for representing classical communication (sending and receiving classical messages) between Hosts.

class netqasm.sdk.classical_communication.socket.**Socket**(*app_name*, *remote_app_name*, *socket_id=0*, *timeout=None*, *use_callbacks=False*, *log_config=None*)

Bases: abc.ABC

Base class for classical sockets.

Classical communication is modelled by sockets, which are also widely used in purely classical applications involving communication.

If a node wants to communicate arbitrary classical messages with another node, this communication must be done between the Hosts of these nodes. Both Hosts should instantiate a Socket object with the other Host as 'remote'. Upon creation, the Socket objects try to connect to each other. Only after this has succeeded, the sockets can be used.

The main methods of a Socket are *send* and *recv*, which are used to send a message to the remote Host, and wait to receive a message from the other Host, respectively. Messages are str (string) objects. To send a number, convert it to a string before sending, and convert it back after receiving.

There are some variations on the *send* and *recv* methods which may be useful in specific scenarios. See their own documentation for their use.

NOTE: At the moment, Sockets are not part of the compilation process yet. Therefore, they don't need to be part of a Connection, and operations on Sockets do not need to be flushed before they are executed (they are executed immediately). This also means that e.g. a *recv* operation, which is blocking by default, acutally blocks the whole application script. **So, if any quantum operations should be executed before such a `recv` statement, make sure that these operations are flushed before blocking on `recv`.**

Implementations (subclasses) of Sockets may be quite different, depending on the runtime environment. A physical setup (with real hardware) may implement this as TCP sockets. A simulator might use inter-thread communication (see e.g. *ThreadSocket*), or another custom object type.

Parameters

- **app_name** (*str*) –
- **remote_app_name** (*str*) –
- **socket_id** (*int*) –
- **timeout** (*Optional[float]*) –
- **use_callbacks** (*bool*) –
- **log_config** (*Optional[config.LogConfig]*) –

__init__(*app_name*, *remote_app_name*, *socket_id=0*, *timeout=None*, *use_callbacks=False*, *log_config=None*)

Socket constructor.

Parameters

- **app_name** (*str*) – application/Host name of this socket's owner
- **remote_app_name** (*str*) – application/Host name of this socket's remote
- **socket_id** (*int*) – local ID to use for this socket
- **timeout** (*Optional[float]*) – maximum amount of real time to try to connect to the remote socket
- **use_callbacks** (*bool*) – whether to call the *recv_callback* method upon receiving a message
- **log_config** (*Optional[config.LogConfig]*) – logging configuration for this socket

abstract send(*msg*)

Send a message to the remote node.

Parameters **msg** (str) –

Return type None

abstract recv(*block=True*, *timeout=None*, *maxsize=None*)

Receive a message from the remote node.

Parameters

- **block** (bool) –
- **timeout** (Optional[float]) –

- **maxsize** (Optional[int]) –

Return type str

send_structured(*msg*)
Sends a structured message (with header and payload) to the remote node.

Parameters **msg** (*StructuredMessage*) –

Return type None

recv_structured(*block=True*, *timeout=None*, *maxsize=None*)
Receive a message (with header and payload) from the remote node.

Parameters

- **block** (bool) –
- **timeout** (Optional[float]) –
- **maxsize** (Optional[int]) –

Return type *StructuredMessage*

send_silent(*msg*)
Sends a message without logging

Parameters **msg** (str) –

Return type None

recv_silent(*block=True*, *timeout=None*, *maxsize=None*)
Receive a message without logging

Parameters

- **block** (bool) –
- **timeout** (Optional[float]) –
- **maxsize** (Optional[int]) –

Return type str

recv_callback(*msg*)
This method gets called when a message is received.

Subclass to define behaviour.

NOTE: This only happens if *self.use_callbacks* is set to *True*.

Parameters **msg** (str) –

Return type None

conn_lost_callback()
This method gets called when the connection is lost.

Subclass to define behaviour.

NOTE: This only happens if *self.use_callbacks* is set to *True*.

Return type None

3.5.7 netqasm.sdk.classical_communication.thread_socket.socket

Classical socket implementation using a hub that is shared across threads.

This module contains the ThreadSocket class which is an implementation of the Socket interface that can be used by Hosts that run in a multi-thread simulation.

netqasm.sdk.classical_communication.thread_socket.socket.**trim_msg**(*msg*)

Parameters **msg** (str) –

Return type str

netqasm.sdk.classical_communication.thread_socket.socket.**log_send**(*method*)

netqasm.sdk.classical_communication.thread_socket.socket.**log_send_structured**(*method*)

netqasm.sdk.classical_communication.thread_socket.socket.**log_recv**(*method*)

netqasm.sdk.classical_communication.thread_socket.socket.**log_recv_structured**(*method*)

class netqasm.sdk.classical_communication.thread_socket.socket.**ThreadSocket**(*app_name*, *remote_app_name*, *socket_id=0*, *timeout=None*, *use_callbacks=False*, *log_config=None*)

Bases: *netqasm.sdk.classical_communication.socket.Socket*

Classical socket implementation for multi-threaded simulations.

This implementation should be used when the simulation consists of a single process, with a thread for each Host. A global SocketHub instance must be available and shared between all threads. The ThreadSocket instance for a Host can then 'send' and 'receive' messages to/from other Hosts by writing/reading from the shared SocketHub memory.

Currently only used with the SquidASM simulator backend.

Parameters

- **app_name** (str) –
- **remote_app_name** (str) –
- **socket_id** (int) –
- **timeout** (Optional[float]) –
- **use_callbacks** (bool) –
- **log_config** (Optional[*LogConfig*]) –

__init__(*app_name*, *remote_app_name*, *socket_id=0*, *timeout=None*, *use_callbacks=False*, *log_config=None*)

ThreadSocket constructor.

Parameters

- **app_name** (str) – application/Host name of this socket's owner
- **remote_app_name** (str) – remote application/Host name
- **socket_id** (int) – local ID to use for this socket

- **timeout** (Optional[float]) – maximum amount of real time to try to connect to the remote socket
- **use_callbacks** (bool) – whether to call the *recv_callback* method upon receiving a message
- **log_config** (Optional[*LogConfig*]) – logging configuration for this socket

classmethod get_comm_logger(*app_name*, *comm_log_dir*)

Parameters

- **app_name** (str) –
- **comm_log_dir** (str) –

Return type *ClassCommLogger*

property app_name

Return type str

property remote_app_name

Return type str

property id

Return type int

property key

Return type Tuple[str, str, int]

property remote_key

Return type Tuple[str, str, int]

property connected

Return type bool

property use_callbacks

Return type bool

send(*msg: str*) → None
Send a message to the remote node.

Parameters **msg** (str) –

Return type None

recv(**args*, ***kwargs*)
Receive a message from the remote node.

send_structured(*msg:* netqasm.sdk.classical_communication.message.StructuredMessage)
Sends a structured message (with header and payload) to the remote node.

Parameters **msg** (*StructuredMessage*) –

recv_structured(**args*, ***kwargs*)
Receive a message (with header and payload) from the remote node.

wait()
Waits until the connection gets lost

Return type None

send_silent(*msg*)
Sends a message without logging

Parameters **msg** (str) –

Return type None

recv_silent(*block=True*, *timeout=None*, *maxsize=None*)
Receive a message without logging

Parameters

- **block** (bool) –
- **timeout** (Optional[float]) –
- **maxsize** (Optional[int]) –

Return type str

class netqasm.sdk.classical_communication.thread_socket.socket.**StorageThreadSocket**(*app_name*, *remote_app_name*, ***kwargs*)

Bases: *netqasm.sdk.classical_communication.thread_socket.socket.ThreadSocket*

Parameters

- **app_name** (str) –
- **remote_app_name** (str) –

__init__(*app_name*, *remote_app_name*, ***kwargs*)
ThreadSocket that simply stores any message comming in

Parameters

- **app_name** (str) –
- **remote_app_name** (str) –

recv_callback(*msg*)
This method gets called when a message is received.

Subclass to define behaviour.

NOTE: This only happens if *self.use_callbacks* is set to *True*.

Parameters **msg** (str) –

Return type None

3.5.8 netqasm.sdk.classical_communication.thread_socket.socket_hub

Global management for classical sockets that live in separate threads.

This module contains the _SocketHub that manages sockets that are in separate threads and need to communicate with each other.

netqasm.sdk.classical_communication.thread_socket.socket_hub.**reset_socket_hub**()

Return type None

3.5.9 netqasm.sdk.config

class netqasm.sdk.config.**LogConfig**(*track_lines=False*, *log_subroutines_dir=None*, *comm_log_dir=None*, *app_dir=None*, *lib_dirs=None*, *log_dir=None*, *split_runs=True*)

Bases: object

Parameters

- **track_lines** (bool) –
- **log_subroutines_dir** (Optional[str]) –
- **comm_log_dir** (Optional[str]) –
- **app_dir** (Optional[str]) –
- **lib_dirs** (Optional[List[str]]) –
- **log_dir** (Optional[str]) –
- **split_runs** (bool) –

track_lines: bool = False

log_subroutines_dir: Optional[str] = None

comm_log_dir: Optional[str] = None

app_dir: Optional[str] = None

lib_dirs: Optional[List[str]] = None

log_dir: Optional[str] = None

split_runs: bool = True

3.5.10 netqasm.sdk.connection

Interface to quantum node controllers.

This module provides the *BaseNetQASMConnection* class which represents the connection with a quantum node controller.

class netqasm.sdk.connection.**BaseNetQASMConnection**(*app_name*, *node_name=None*, *app_id=None*, *max_qubits=5*, *hardware_config=None*, *log_config=None*, *epr_sockets=None*, *compiler=None*, *return_arrays=True*, *_init_app=True*, *_setup_epr_sockets=True*)

Bases: abc.ABC

Base class for representing connections to a quantum node controller.

A BaseNetQASMConnection instance provides an interface for Host programs to interact with a quantum node controller like QNodeOS, which controls the quantum hardware.

The interaction with the quantum node controller includes registering applications, opening EPR sockets, sending NetQASM subroutines, and getting execution results,

A BaseNetQASMConnection instance also provides a 'context' for the Host to run its code in. Code within this context is compiled into NetQASM subroutines and then sent to the quantum node controller.

Parameters

- **app_name** (*str*) –
- **node_name** (*Optional[str]*) –
- **app_id** (*Optional[int]*) –
- **max_qubits** (*int*) –
- **hardware_config** (*Optional[HardwareConfig]*) –
- **log_config** (*Optional[LogConfig]*) –
- **epr_sockets** (*Optional[List[esck.EPRSocket]]*) –
- **compiler** (*Optional[Type[SubroutineTranspiler]]*) –
- **return_arrays** (*bool*) –
- **_init_app** (*bool*) –
- **_setup_epr_sockets** (*bool*) –

__init__(*app_name*, *node_name=None*, *app_id=None*, *max_qubits=5*, *hardware_config=None*, *log_config=None*, *epr_sockets=None*, *compiler=None*, *return_arrays=True*, *_init_app=True*, *_setup_epr_sockets=True*)

BaseNetQASMConnection constructor.

In most cases, you will want to instantiate a subclass of this.

Parameters

- **app_name** (*str*) – Name of the application. Specifically, this is the name of the program that runs on this particular node. So, *app_name* can often be seen as the name of the "role" within the multi-party application or protocol. For example, in a Blind Computation protocol, the two roles may be "client" and "server"; the *app_name* of a particular *BaseNetQASMConnection* instance may then e.g. be "client".
- **node_name** (*Optional[str]*) – name of the Node that is controlled by the quantum node controller that we connect to. The Node name may be different from the *app_name*, and e.g. reflect its geographical location, like a city name. If None, the Node name is obtained by querying the global *NetworkInfo* by using the *app_name*.
- **app_id** (*Optional[int]*) – ID of this application. An application registered in the quantum node controller using this connection will use this app ID. If None, a unique ID will be chosen (unique among possible other applications that were registered through other BaseNetQASMConnection instances).
- **max_qubits** (*int*) – maximum number of qubits that can be in use at the same time by applications registered through this connection. Defaults to 5.
- **hardware_config** (*Optional[HardwareConfig]*) – configuration object with info about the underlying hardware. Used by the Builder of this Connection. When None, a generic configuration object is created with the default qubit count of 5.
- **log_config** (*Optional[LogConfig]*) – configuration object specifying what to log.
- **epr_sockets** (*Optional[List[esck.EPRSocket]]*) – list of EPR sockets. If *_setup_epr_sockets* is True, these EPR sockets are automatically opened upon entering this connection's context.

- **compiler** (*Optional[Type[SubroutineTranspiler]]*) – the class that is used to instantiate the compiler. If None, a compiler is used that can compile for the hardware of the node this connection is to.
- **return_arrays** (*bool*) – whether to add "return array"-instructions at the end of subroutines. A reason to set this to False could be that a quantum node controller does not support returning arrays back to the Host.
- **_init_app** (*bool*) – whether to immediately send a "register application" message to the quantum node controller upon construction of this connection.
- **_setup_epr_sockets** (*bool*) – whether to immediately send "open EPR socket" messages to the quantum node controller upon construction of this connection. If True, the "open EPR socket" messages are for the EPR sockets defined in the *epr_sockets* parameter.

property app_name
Get the application name

Return type str

property node_name
Get the node name

Return type str

property app_id
Get the application ID

Return type int

property network_info

Return type Type[*NetworkInfo*]

property builder

Return type *Builder*

classmethod get_app_ids()

Return type Dict[str, List[int]]

classmethod get_app_names()

Return type Dict[str, Dict[int, str]]

clear()

Return type None

close(*clear_app=True*, *stop_backend=False*, *exception=False*)
Close a connection.

By default, this method is automatically called when a connection context ends.

Parameters

- **clear_app** (bool) –
- **stop_backend** (bool) –
- **exception** (bool) –

Return type None

property shared_memory
Get this connection's Shared Memory object.

This property should *not* be accessed before any potential setting-up of shared memories has finished. If it cannot be found, an error is raised.

Return type *SharedMemory*

property active_qubits
Get a list of qubits that are currently in use.

"In use" means that the virtual qubit represented by this *Qubit* instance has been allocated and hence its virtual ID cannot be re-used.

Return type List[*Qubit*]

Returns list of active qubits

flush(*block=True*, *callback=None*)
Compile and send all pending operations to the quantum node controller.

All operations that have been added to this connection's Builder (typically by issuing these operations within a connection context) are collected and compiled into a NetQASM subroutine. This subroutine is then sent over the connection to be executed by the quantum node controller.

Parameters

- **block** (bool) – block on receiving the result of executing the compiled subroutine from the quantum node controller.
- **callback** (Optional[Callable]) – if *block* is False, this callback is called when the quantum node controller sends the subroutine results.

Return type None

compile()
Compile the previous SDK commands into a NetQASM subroutine.

This does this the same as calling *flush()*, except it does not send the subroutine to the quantum node controller for execution. This method can hence be used to pre-compile a subroutine and send it later, possibly after filling in concrete values for templates.

Return type Optional[*Subroutine*]

commit_protosubroutine(*protosubroutine*, *block=True*, *callback=None*)
Send a protosubroutine to the quantum node controller.

Takes a *ProtoSubroutine*, i.e. an intermediate representation of the subroutine that comes from the Builder. The ProtoSubroutine is compiled into a *Subroutine* instance.

Parameters

- **protosubroutine** (*ProtoSubroutine*) –
- **block** (bool) –
- **callback** (Optional[Callable]) –

Return type None

commit_subroutine(*subroutine*, *block=True*, *callback=None*)

Parameters

- **subroutine** (*Subroutine*) –
- **block** (bool) –

- **callback** (Optional[Callable]) –

Return type None

block()
Block until a flushed subroutines finishes.

This should be implemented by subclasses.

:raises NotImplementedError

Return type None

new_array(*length=1*, *init_values=None*)
Allocate a new array in the shared memory.

This operation is handled by the connection's Builder. The Builder make sures the relevant NetQASM instructions end up in the subroutine.

Parameters

- **length** (int) – length of the array, defaults to 1
- **init_values** (Optional[List[Optional[int]]]) – list of initial values of the array. If not None, must have the same length as *length*.

Return type *Array*

Returns a handle to the array that can be used in application code

loop(*stop*, *start=0*, *step=1*, *loop_register=None*)
Create a context for code that gets looped.

Each iteration of the loop is associated with an index, which starts at 0 by default. Each iteration the index is increased by *step* (default 1). Looping stops when the index reaches *stop*.

Code inside the context *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

This operation is handled by the connection's Builder. The Builder make sures the NetQASM subroutine contains a loop around the (compiled) code that is inside the context.

Example:

```
with NetQASMConnection(app_name="alice") as alice:
    outcomes = alice.new_array(10)
    with alice.loop(10) as i:
        q = Qubit(alice)
        q.H()
        outcome = outcomes.get_future_index(i)
        q.measure(outcome)
```

Parameters

- **stop** (int) – end of iteration range (exluding)
- **start** (int) – start of iteration range (including), defaults to 0
- **step** (int) – step size of iteration range, defaults to 1
- **loop_register** (Optional[*Register*]) – specific register to be used for holding the loop index. In most cases there is no need to explicitly specify this.

Return type AbstractContextManager[*Register*]

Returns the context object (to be used in a *with ...* expression)

loop_body(*body*, *stop*, *start=0*, *step=1*, *loop_register=None*)
Loop code that is defined in a Python function (body).

The function to loop should have a single argument with that has the *BaseNetQASMConnection* type.

Parameters

- **body** (Callable[[ForwardRef, *RegFuture*], None]) – function to loop
- **stop** (int) – end of iteration range (exluding)
- **start** (int) – start of iteration range (including), defaults to 0
- **step** (int) – step size of iteration range, defaults to 1
- **loop_register** (Union[*Register*, str, None]) – specific register to be used for holding the loop index.

Return type None

loop_until(*max_iterations*)
Create a context with code to be looped until the exit condition is met, or the maximum number of tries has been reached.

The code inside the context is automatically looped (re-run). At the end of each iteration, the exit_condition of the context object is checked. If the condition holds, the loop exits. Otherwise the loop does another iteration. If *max_iterations* iterations have been reached, the loop exits anyway.

Make sure you set the loop_condition on the context object, like e.g.

```
with connection.loop_until(max_iterations=10) as loop:
    q = Qubit(conn)
    m = q.measure()
    constraint = ValueAtMostConstraint(m, 0)
    loop.set_exit_condition(constraint)
```

Parameters **max_iterations** (int) – the maximum number of times to loop

Return type AbstractContextManager[*SdkLoopUntilContext*]

if_eq(*a*, *b*, *body*)
Execute a function if a == b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ne(*a*, *b*, *body*)
Execute a function if a != b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_lt(*a*, *b*, *body*)

Execute a function if a < b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ge(*a*, *b*, *body*)

Execute a function if a > b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ez(*a*, *body*)

Execute a function if a == 0.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_nz(*a*, *body*)

Execute a function if a != 0.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

try_until_success(*max_tries=1*)
TODO docstring

Parameters **max_tries** (int) –

Return type AbstractContextManager[None]

tomography(*preparation*, *iterations*, *progress=True*)
Does a tomography on the output from the preparation specified. The frequencies from X, Y and Z measurements are returned as a tuple (f_X,f_Y,f_Z).

- **Arguments**

  preparation A function that takes a NetQASMConnection as input and prepares a qubit and returns this

  iterations Number of measurements in each basis.

  progress_bar Displays a progress bar

Parameters

- **preparation** (Callable[[*BaseNetQASMConnection*], *Qubit*]) –
- **iterations** (int) –
- **progress** (bool) –

Return type Dict[str, float]

test_preparation(*preparation*, *exp_values*, *conf=2*, *iterations=100*, *progress=True*)
Test the preparation of a qubit. Returns True if the expected values are inside the confidence interval produced from the data received from the tomography function

- **Arguments**

  preparation A function that takes a NetQASMConnection as input and prepares a qubit and returns this

  exp_values The expected values for measurements in the X, Y and Z basis.

  conf Determines the confidence region (+/- conf/sqrt(iterations))

  iterations Number of measurements in each basis.

  progress_bar Displays a progress bar

Parameters

- **preparation** (Callable[[*BaseNetQASMConnection*], *Qubit*]) –
- **exp_values** (Tuple[float, float, float]) –
- **conf** (float) –
- **iterations** (int) –
- **progress** (bool) –

Return type bool

insert_breakpoint(*action*, *role=<BreakpointRole.CREATE: 0>*)

Parameters

- **action** (*BreakpointAction*) –
- **role** (*BreakpointRole*) –

Return type None

class netqasm.sdk.connection.**DebugConnection**(**args*, ***kwargs*)

Bases: *netqasm.sdk.connection.BaseNetQASMConnection*

Connection that mocks most of the *BaseNetQASMConnection* logic.

Subroutines that are flushed are simply stored in this object. No actual connection is made.

node_ids: Dict[str, int] = {}

__init__(**args*, ***kwargs*)

A connection that simply stores the subroutine it commits

property shared_memory

Get this connection's Shared Memory object.

This property should *not* be accessed before any potential setting-up of shared memories has finished. If it cannot be found, an error is raised.

Return type *SharedMemory*

property active_qubits

Get a list of qubits that are currently in use.

"In use" means that the virtual qubit represented by this *Qubit* instance has been allocated and hence its virtual ID cannot be re-used.

Return type List[*Qubit*]

Returns list of active qubits

property app_id

Get the application ID

Return type int

property app_name

Get the application name

Return type str

block()

Block until a flushed subroutines finishes.

This should be implemented by subclasses.

:raises NotImplementedError

Return type None

property builder

Return type *Builder*

clear()

Return type None

close(*clear_app=True*, *stop_backend=False*, *exception=False*)
Close a connection.

By default, this method is automatically called when a connection context ends.

Parameters

- **clear_app** (bool) –
- **stop_backend** (bool) –
- **exception** (bool) –

Return type None

commit_protosubroutine(*protosubroutine*, *block=True*, *callback=None*)
Send a protosubroutine to the quantum node controller.

Takes a *ProtoSubroutine*, i.e. an intermediate representation of the subroutine that comes from the Builder. The ProtoSubroutine is compiled into a *Subroutine* instance.

Parameters

- **protosubroutine** (*ProtoSubroutine*) –
- **block** (bool) –
- **callback** (Optional[Callable]) –

Return type None

commit_subroutine(*subroutine*, *block=True*, *callback=None*)

Parameters

- **subroutine** (*Subroutine*) –
- **block** (bool) –
- **callback** (Optional[Callable]) –

Return type None

compile()
Compile the previous SDK commands into a NetQASM subroutine.

This does this the same as calling *flush()*, except it does not send the subroutine to the quantum node controller for execution. This method can hence be used to pre-compile a subroutine and send it later, possibly after filling in concrete values for templates.

Return type Optional[*Subroutine*]

flush(*block=True*, *callback=None*)
Compile and send all pending operations to the quantum node controller.

All operations that have been added to this connection's Builder (typically by issuing these operations within a connection context) are collected and compiled into a NetQASM subroutine. This subroutine is then sent over the connection to be executed by the quantum node controller.

Parameters

- **block** (bool) – block on receiving the result of executing the compiled subroutine from the quantum node controller.
- **callback** (Optional[Callable]) – if *block* is False, this callback is called when the quantum node controller sends the subroutine results.

Return type None

classmethod get_app_ids()

Return type Dict[str, List[int]]

classmethod get_app_names()

Return type Dict[str, Dict[int, str]]

if_eq(*a*, *b*, *body*)
Execute a function if a == b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ez(*a*, *body*)
Execute a function if a == 0.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ge(*a*, *b*, *body*)
Execute a function if a > b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_lt(*a*, *b*, *body*)
Execute a function if a < b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value

- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_ne(*a*, *b*, *body*)
Execute a function if a != b.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **b** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

if_nz(*a*, *body*)
Execute a function if a != 0.

Code inside the callback function *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Parameters

- **a** (Union[int, ForwardRef, ForwardRef]) – a classical value
- **body** (Callable[[ForwardRef], None]) – function to execute if condition is true

Return type None

insert_breakpoint(*action*, *role=<BreakpointRole.CREATE: 0>*)

Parameters

- **action** (*BreakpointAction*) –
- **role** (*BreakpointRole*) –

Return type None

loop(*stop*, *start=0*, *step=1*, *loop_register=None*)
Create a context for code that gets looped.

Each iteration of the loop is associated with an index, which starts at 0 by default. Each iteration the index is increased by *step* (default 1). Looping stops when the index reaches *stop*.

Code inside the context *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

This operation is handled by the connection's Builder. The Builder make sures the NetQASM subroutine contains a loop around the (compiled) code that is inside the context.

Example:

```
with NetQASMConnection(app_name="alice") as alice:
    outcomes = alice.new_array(10)
    with alice.loop(10) as i:
```

(continues on next page)

(continued from previous page)

```
        q = Qubit(alice)
        q.H()
        outcome = outcomes.get_future_index(i)
        q.measure(outcome)
```

Parameters

- **stop** (int) – end of iteration range (exluding)
- **start** (int) – start of iteration range (including), defaults to 0
- **step** (int) – step size of iteration range, defaults to 1
- **loop_register** (Optional[*Register*]) – specific register to be used for holding the loop index. In most cases there is no need to explicitly specify this.

Return type AbstractContextManager[*Register*]

Returns the context object (to be used in a *with ...* expression)

loop_body(*body*, *stop*, *start=0*, *step=1*, *loop_register=None*)

Loop code that is defined in a Python function (body).

The function to loop should have a single argument with that has the *BaseNetQASMConnection* type.

Parameters

- **body** (Callable[[ForwardRef, *RegFuture*], None]) – function to loop
- **stop** (int) – end of iteration range (exluding)
- **start** (int) – start of iteration range (including), defaults to 0
- **step** (int) – step size of iteration range, defaults to 1
- **loop_register** (Union[*Register*, str, None]) – specific register to be used for holding the loop index.

Return type None

loop_until(*max_iterations*)

Create a context with code to be looped until the exit condition is met, or the maximum number of tries has been reached.

The code inside the context is automatically looped (re-run). At the end of each iteration, the exit_condition of the context object is checked. If the condition holds, the loop exits. Otherwise the loop does another iteration. If *max_iterations* iterations have been reached, the loop exits anyway.

Make sure you set the loop_condition on the context object, like e.g.

```
```

with connection.loop_until(max_iterations=10) as loop: q = Qubit(conn) m = q.measure() constraint = ValueAtMostConstraint(m, 0) loop.set_exit_condition(constraint)

Parameters **max_iterations** (int) – the maximum number of times to loop

Return type AbstractContextManager[*SdkLoopUntilContext*]

property network_info

Return type Type[*NetworkInfo*]

new_array(*length=1*, *init_values=None*)
Allocate a new array in the shared memory.

This operation is handled by the connection's Builder. The Builder make sures the relevant NetQASM instructions end up in the subroutine.

Parameters

- **length** (int) – length of the array, defaults to 1
- **init_values** (Optional[List[Optional[int]]]) – list of initial values of the array. If not None, must have the same length as *length*.

Return type *Array*

Returns a handle to the array that can be used in application code

property node_name
Get the node name

Return type str

test_preparation(*preparation*, *exp_values*, *conf=2*, *iterations=100*, *progress=True*)
Test the preparation of a qubit. Returns True if the expected values are inside the confidence interval produced from the data received from the tomography function

- **Arguments**

  preparation A function that takes a NetQASMConnection as input and prepares a qubit and returns this

  exp_values The expected values for measurements in the X, Y and Z basis.

  conf Determines the confidence region (+/- conf/sqrt(iterations))

  iterations Number of measurements in each basis.

  progress_bar Displays a progress bar

Parameters

- **preparation** (Callable[[*BaseNetQASMConnection*], *Qubit*]) –
- **exp_values** (Tuple[float, float, float]) –
- **conf** (float) –
- **iterations** (int) –
- **progress** (bool) –

Return type bool

tomography(*preparation*, *iterations*, *progress=True*)
Does a tomography on the output from the preparation specified. The frequencies from X, Y and Z measurements are returned as a tuple (f_X,f_Y,f_Z).

- **Arguments**

  preparation A function that takes a NetQASMConnection as input and prepares a qubit and returns this

  iterations Number of measurements in each basis.

  progress_bar Displays a progress bar

Parameters

- **preparation** (Callable[[*BaseNetQASMConnection*], *Qubit*]) –
- **iterations** (int) –
- **progress** (bool) –

Return type Dict[str, float]

try_until_success(*max_tries=1*)
TODO docstring

Parameters **max_tries** (int) –

Return type AbstractContextManager[None]

class netqasm.sdk.connection.**DebugNetworkInfo**
Bases: *netqasm.sdk.network.NetworkInfo*

classmethod get_node_id_for_app(*app_name*)
Returns the node id for the app with the given name

Parameters **app_name** (str) –

Return type int

classmethod get_node_name_for_app(*app_name*)
Returns the node name for the app with the given name

Parameters **app_name** (str) –

Return type str

3.5.11 netqasm.sdk.epr_socket

EPR Socket interface.

class netqasm.sdk.epr_socket.**EPRSocket**(*remote_app_name*, *epr_socket_id=0*, *remote_epr_socket_id=0*, *min_fidelity=100*)
Bases: abc.ABC

EPR socket class. Used to generate entanglement with a remote node.

An EPR socket represents a connection with a single remote node through which EPR pairs can be generated. Its main interfaces are the *create* and *recv* methods. A typical use case for two nodes is that they both create an EPR socket to the other node, and during the protocol, one of the nodes does *create* operations on its socket while the other node does *recv* operations.

A *create* operation asks the network stack to initiate generation of EPR pairs with the remote node. Depending on the type of generation, the result of this operation can be qubit objects or measurement outcomes. A *recv* operation asks the network stack to wait for the remote node to initiate generation of EPR pairs. Again, the result can be qubit objects or measurement outcomes.

Each *create* operation on one node must be matched by a *recv* operation on the other node. Since "creating" and "receiving" must happen at the same time, a node that is doing a *create* operation on its socket cannot advance until the other node does the corresponding *recv*. This is different from classical network sockets where a "send" operation (roughly anologous to *create* in an EPR socket) does not block on the remote node receiving it.

An EPR socket is identified by a triple consisting of (1) the remote node ID, (2) the local socket ID and (3) the remote socket ID. Two nodes that want to generate EPR pairs with each other should make sure that the IDs in their local sockets match.

Parameters

- **remote_app_name** (str) –
- **epr_socket_id** (int) –
- **remote_epr_socket_id** (int) –
- **min_fidelity** (int) –

__init__ (*remote_app_name*, *epr_socket_id=0*, *remote_epr_socket_id=0*, *min_fidelity=100*)
Create an EPR socket. It still needs to be registered with the network stack separately.

Registering and opening the EPR socket is currently done automatically by the connection that uses this EPR socket, specifically when a context is opened with that connection.

Parameters

- **remote_app_name** (str) – name of the remote party (i.e. the role, like "client", not necessarily the node name like "delft")
- **epr_socket_id** (int) – local socket ID, defaults to 0
- **remote_epr_socket_id** (int) – remote socket ID, defaults to 0. Note that this must match with the local socket ID of the remote node's EPR socket.
- **min_fidelity** (int) – minimum desired fidelity for EPR pairs generated over this socket, in percentages (i.e. range 0-100). Defaults to 100.

property conn
Get the underlying NetQASMConnection

Return type *connection.BaseNetQASMConnection*

property remote_app_name
Get the remote application name

Return type str

property remote_node_id
Get the remote node ID

Return type int

property epr_socket_id
Get the EPR socket ID

Return type int

property remote_epr_socket_id
Get the remote EPR socket ID

Return type int

property min_fidelity
Get the desired minimum fidelity

Return type int

create_keep (*number=1*, *post_routine=None*, *sequential=False*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *min_fidelity_all_at_end=None*, *max_tries=None*)
Ask the network stack to generate EPR pairs with the remote node and keep them in memory.

A *create_keep* operation must always be matched by a *recv_keep* operation on the remote node.

If *sequential* is False (default), this operation returns a list of Qubit objects representing the local qubits that are each one half of the generated pairs. These qubits can then be manipulated locally just like locally

initialized qubits, by e.g. applying gates or measuring them. Each qubit also contains information about the entanglement generation that lead to its creation, and can be accessed by its *entanglement_info* property.

A typical example for just generating one pair with another node would be:

```
q = epr_socket.create_keep()[0]
# `q` can now be used as a normal qubit
```

If *sequential* is False (default), the all requested EPR pairs are generated at once, before returning the results (qubits or entanglement info objects).

If *sequential* is True, a callback function (*post_routine*) should be specified. After generating one EPR pair, this callback will be called, before generating the next pair. This method can e.g. be used to generate many EPR pairs (more than the number of physical qubits available), by measuring (and freeing up) each qubit before the next pair is generated.

For example:

```
outcomes = alice.new_array(num)

def post_create(conn, q, pair):
    q.H()
    outcome = outcomes.get_future_index(pair)
    q.measure(outcome)
epr_socket.create_keep(number=num, post_routine=post_create, sequential=True)
```

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function for each genated pair. Only used if *sequential* is True. The callback should take three arguments *(conn, q, pair)* where * *conn* is the connection (e.g. *self*) * *q* is the entangled qubit (of type *FutureQubit*) * *pair* is a register holding which pair is handled (0, 1, . . .)
- **sequential** (bool) – whether to use callbacks after each pair, defaults to False
- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter
- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.
- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[*Qubit*]

Returns list of qubits created

create_keep_with_info(*number=1*, *post_routine=None*, *sequential=False*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *min_fidelity_all_at_end=None*)

Same as create_keep but also return the EPR generation information coming from the network stack.

For more information see the documentation of *create_keep*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) –
- **sequential** (bool) –
- **time_unit** (TimeUnit) –
- **max_time** (int) –
- **min_fidelity_all_at_end** (Optional[int]) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

Returns tuple with (1) list of qubits created, (2) list of EprKeepResult objects

create_measure(*number=1*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *basis_local=None*, *basis_remote=None*, *rotations_local=(0, 0, 0)*, *rotations_remote=(0, 0, 0)*, *random_basis_local=None*, *random_basis_remote=None*)

Ask the network stack to generate EPR pairs with the remote node and measure them immediately (on both nodes).

A *create_measure* operation must always be matched by a *recv_measure* operation on the remote node.

This operation returns a list of Linklayer response objects. These objects contain information about the entanglement generation and includes the measurement outcome and basis used. Note that all values are *Future* objects. This means that the current subroutine must be flushed before the values become defined.

An example for generating 10 pairs with another node that are immediately measured:

```
# list of Futures that become defined when subroutine is flushed
outcomes = []
with NetQASMConnection("alice", epr_sockets=[epr_socket]):
    ent_infos = epr_socket.create(number=10, tp=EPRType.M)
    for ent_info in ent_infos:
        outcomes.append(ent_info.measurement_outcome)
```

The basis to measure in can also be specified. There are 3 ways to specify a basis:

- using one of the *EprMeasBasis* variants
- by specifying 3 rotation angles, interpreted as an X-rotation, a Y-rotation and another X-rotation. For example, setting *rotations_local* to (8, 0, 0) means that before measuring, an X-rotation of 8*pi/16 = pi/2 radians is applied to the qubit.
- using one of the *RandomBasis* variants, in which case one of the bases of that variant is chosen at random just before measuring

NOTE: the node that initiates the entanglement generation, i.e. the one that calls *create* on its EPR socket, also controls the measurement bases of the receiving node (by setting e.g. *rotations_remote*). The receiving node cannot change this.

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1

- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter
- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.
- **basis_local** (Optional[EprMeasBasis]) – basis to measure in on this node for M-type requests
- **basis_remote** (Optional[EprMeasBasis]) – basis to measure in on the remote node for M-type requests
- **rotations_local** (Tuple[int, int, int]) – rotations to apply before measuring on this node
- **rotations_remote** (Tuple[int, int, int]) – rotations to apply before measuring on remote node
- **random_basis_local** (Optional[RandomBasis]) – random bases to choose from when measuring on this node
- **random_basis_remote** (Optional[RandomBasis]) – random bases to choose from when measuring on the remote node

Return type List[EprMeasureResult]

Returns list of entanglement info objects per created pair.

create_rsp(*number=1*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *basis_local=None*, *rotations_local=(0, 0, 0)*, *random_basis_local=None*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Ask the network stack to do remote preparation with the remote node.

A *create_rsp* operation must always be matched by a *recv_epr* operation on the remote node.

This operation returns a list of Linklayer response objects. These objects contain information about the entanglement generation and includes the measurement outcome and basis used. Note that all values are *Future* objects. This means that the current subroutine must be flushed before the values become defined.

An example for generating 10 pairs with another node that are immediately measured:

```
m: LinkLayerOKTypeM = epr_socket.create_rsp(tp=EPRType.R)[0]
print(m.measurement_outcome)
# remote node now has a prepared qubit
```

The basis to measure in can also be specified. There are 3 ways to specify a basis:

- using one of the *EprMeasBasis* variants
- by specifying 3 rotation angles, interpreted as an X-rotation, a Y-rotation and another X-rotation. For example, setting *rotations_local* to (8, 0, 0) means that before measuring, an X-rotation of 8*pi/16 = pi/2 radians is applied to the qubit.
- using one of the *RandomBasis* variants, in which case one of the bases of that variant is chosen at random just before measuring

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter

- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.
- **basis_local** (Optional[EprMeasBasis]) – basis to measure in on this node for M-type requests
- **rotations_local** (Tuple[int, int, int]) – rotations to apply before measuring on this node
- **random_basis_local** (Optional[RandomBasis]) – random bases to choose from when measuring on this node
- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[EprMeasureResult]

Returns list of entanglement info objects per created pair.

create(*number=1*, *post_routine=None*, *sequential=False*, *tp=<EPRType.K: 0>*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*, *basis_local=None*, *basis_remote=None*, *rotations_local=(0, 0, 0)*, *rotations_remote=(0, 0, 0)*, *random_basis_local=None*, *random_basis_remote=None*)

Ask the network stack to generate EPR pairs with the remote node.

A *create* operation must always be matched by a *recv* operation on the remote node.

If the type of request is Create and Keep (CK, or just K) and if *sequential* is False (default), this operation returns a list of Qubit objects representing the local qubits that are each one half of the generated pairs. These qubits can then be manipulated locally just like locally initialized qubits, by e.g. applying gates or measuring them. Each qubit also contains information about the entanglement generation that lead to its creation, and can be accessed by its *entanglement_info* property.

A typical example for just generating one pair with another node would be:

```
q = epr_socket.create()[0]
# `q` can now be used as a normal qubit
```

If the type of request is Measure Directly (MD, or just M), this operation returns a list of Linklayer response objects. These objects contain information about the entanglement generation and includes the measurement outcome and basis used. Note that all values are *Future* objects. This means that the current subroutine must be flushed before the values become defined.

An example for generating 10 pairs with another node that are immediately measured:

```
# list of Futures that become defined when subroutine is flushed
outcomes = []
with NetQASMConnection("alice", epr_sockets=[epr_socket]):
    ent_infos = epr_socket.create(number=10, tp=EPRType.M)
```

(continues on next page)

(continued from previous page)

```
    for ent_info in ent_infos:
        outcomes.append(ent_info.measurement_outcome)
```

For "Measure Directly"-type requests, the basis to measure in can also be specified. There are 3 ways to specify a basis:

- using one of the *EprMeasBasis* variants
- by specifying 3 rotation angles, interpreted as an X-rotation, a Y-rotation and another X-rotation. For example, setting *rotations_local* to (8, 0, 0) means that before measuring, an X-rotation of 8*pi/16 = pi/2 radians is applied to the qubit.
- using one of the *RandomBasis* variants, in which case one of the bases of that variant is chosen at random just before measuring

NOTE: the node that initiates the entanglement generation, i.e. the one that calls *create* on its EPR socket, also controls the measurement bases of the receiving node (by setting e.g. *rotations_remote*). The receiving node cannot change this.

If *sequential* is False (default), the all requested EPR pairs are generated at once, before returning the results (qubits or entanglement info objects).

If *sequential* is True, a callback function (*post_routine*) should be specified. After generating one EPR pair, this callback will be called, before generating the next pair. This method can e.g. be used to generate many EPR pairs (more than the number of physical qubits available), by measuring (and freeing up) each qubit before the next pair is generated.

For example:

```
outcomes = alice.new_array(num)

def post_create(conn, q, pair):
    q.H()
    outcome = outcomes.get_future_index(pair)
    q.measure(outcome)
epr_socket.create(number=num, post_routine=post_create, sequential=True)
```

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function for each genated pair. Only used if *sequential* is True. The callback should take three arguments *(conn, q, pair)* where * *conn* is the connection (e.g. *self*) * *q* is the entangled qubit (of type *FutureQubit*) * *pair* is a register holding which pair is handled (0, 1, . . .)
- **sequential** (bool) – whether to use callbacks after each pair, defaults to False
- **tp** (EPRType) – type of entanglement generation, defaults to EPRType.K. Note that corresponding *recv* of the remote node's EPR socket must specify the same type.
- **time_unit** (TimeUnit) – which time unit to use for the *max_time* parameter
- **max_time** (int) – maximum number of time units (see *time_unit*) the Host is willing to wait for entanglement generation of a single pair. If generation does not succeed within this time, the whole subroutine that this request is part of is reset and run again by the quantum node controller.
- **basis_local** (Optional[EprMeasBasis]) – basis to measure in on this node for M-type requests

- **basis_remote** (Optional[EprMeasBasis]) – basis to measure in on the remote node for M-type requests
- **rotations_local** (Tuple[int, int, int]) – rotations to apply before measuring on this node (for M-type requests)
- **rotations_remote** (Tuple[int, int, int]) – rotations to apply before measuring on remote node (for M-type requests)
- **random_basis_local** (Optional[RandomBasis]) – random bases to choose from when measuring on this node (for M-type requests)
- **random_basis_remote** (Optional[RandomBasis]) – random bases to choose from when measuring on the remote node (for M-type requests)

Return type Union[List[*Qubit*], List[EprMeasureResult], List[LinkLayerOKTypeM]]

Returns For K-type requests: list of qubits created. For M-type requests: list of entanglement info objects per created pair.

create_context(*number=1*, *sequential=False*, *time_unit=<TimeUnit.MICRO_SECONDS: 0>*, *max_time=0*)

Create a context that is executed for each generated EPR pair consecutively.

Creates EPR pairs with a remote node and handles each pair by the operations defined in a subsequent context. See the example below.

```
with epr_socket.create_context(number=10) as (q, pair):
    q.H()
    m = q.measure()
```

NOTE: even though all pairs are handled consecutively, they are still generated concurrently by the network stack. By setting *sequential* to True, the network stack only generates the next pair after the context for the previous pair has been executed, similar to using a callback (*post_routine*) in the *create* method.

Parameters

- **number** (int) – number of EPR pairs to generate, defaults to 1
- **sequential** (bool) – whether to generate pairs sequentially, defaults to False
- **time_unit** (TimeUnit) –
- **max_time** (int) –

Return type AbstractContextManager[Tuple[*FutureQubit*, *RegFuture*]]

recv_keep(*number=1*, *post_routine=None*, *sequential=False*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Ask the network stack to wait for the remote node to generate EPR pairs, which are kept in memory.

A *recv_keep* operation must always be matched by a *create_keep* operation on the remote node. The number of generated pairs must also match.

For more information see the documentation of *create_keep*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function used when *sequential* is True

- **sequential** (bool) – whether to call the callback after each pair generation, defaults to False
- **expect_phi_plus** (bool) – whether to assume that the EPR pairs that are created are in the Phi+ (or Phi_00) state. Defaults to True. If True, the compiler will make sure that if the physical link actually produced another Bell state, the behavior seen by the application is still as if a Phi+ state was actually produced.
- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[*Qubit*]

Returns list of qubits created

recv_keep_with_info(*number=1*, *post_routine=None*, *sequential=False*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)

Same as recv_keep but also return the EPR generation information coming from the network stack.

For more information see the documentation of *recv_keep*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) –
- **sequential** (bool) –
- **expect_phi_plus** (bool) –
- **min_fidelity_all_at_end** (Optional[int]) –
- **max_tries** (Optional[int]) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

Returns tuple with (1) list of qubits created, (2) list of EprKeepResult objects

recv_measure(*number=1*, *expect_phi_plus=True*)

Ask the network stack to wait for the remote node to generate EPR pairs, which are immediately measured (on both nodes).

A *recv_measure* operation must always be matched by a *create_measure* operation on the remote node. The number and type of generation must also match.

For more information see the documentation of *create_measure*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **expect_phi_plus** (bool) – whether to assume that the EPR pairs that are created are in the Phi+ (or Phi_00) state. Defaults to True. If True, the compiler will make sure that if the physical link actually produced another Bell state, the behavior seen by the application is still as if a Phi+ state was actually produced.

Return type List[EprMeasureResult]

Returns list of entanglement info objects per created pair.

recv_rsp(*number=1*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)
Ask the network stack to wait for remote state preparation from another node.

A *recv_rsp* operation must always be matched by a *create_rsp* operation on the remote node. The number and type of generation must also match.

For more information see the documentation of *create_rsp*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **expect_phi_plus** (bool) – whether to assume that the EPR pairs that are created are in the Phi+ (or Phi_00) state. Defaults to True. If True, the compiler will make sure that if the physical link actually produced another Bell state, the behavior seen by the application is still as if a Phi+ state was actually produced.
- **min_fidelity_all_at_end** (Optional[int]) – the minimum fidelity that *all* entangled qubits should ideally still have at the moment the last qubit has been generated. For example, when specifying *number=2* and *min_fidelity_all_at_end=80*, the the program will automatically try to make sure that both qubits have a fidelity of at least 80% when the second qubit has been generated. It will attempt to do this by automatically retrying the entanglement generation if the fidelity constraint is not satisfied. This is however an *attempt*, and not a guarantee!.
- **max_tries** (Optional[int]) – maximum number of re-tries should be made to try and achieve the *min_fidelity_all_at_end* constraint.

Return type List[*Qubit*]

Returns list of qubits created

recv_rsp_with_info(*number=1*, *expect_phi_plus=True*, *min_fidelity_all_at_end=None*, *max_tries=None*)
Same as recv_rsp but also return the EPR generation information coming from the network stack.

For more information see the documentation of *recv_rsp*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **expect_phi_plus** (bool) –
- **min_fidelity_all_at_end** (Optional[int]) –
- **max_tries** (Optional[int]) –

Return type Tuple[List[*Qubit*], List[EprKeepResult]]

Returns tuple with (1) list of qubits created, (2) list of EprKeepResult objects

recv(*number=1*, *post_routine=None*, *sequential=False*, *tp=<EPRType.K: 0>*)
Ask the network stack to wait for the remote node to generate EPR pairs.

A *recv* operation must always be matched by a *create* operation on the remote node. See also the documentation of *create*. The number and type of generation must also match.

In case of Measure Directly requests, it is the initiating node (that calls *create*) which specifies the measurement bases. This should not and cannot be done in *recv*.

For more information see the documentation of *create*.

Parameters

- **number** (int) – number of pairs to generate, defaults to 1
- **post_routine** (Optional[Callable]) – callback function used when *sequential* is True
- **sequential** (bool) – whether to call the callback after each pair generation, defaults to False
- **tp** (EPRType) – type of entanglement generation, defaults to EPRType.K

Return type Union[List[*Qubit*], List[EprMeasureResult], List[LinkLayerOKTypeR]]

Returns For K-type requests: list of qubits created. For M-type requests: list of entanglement info objects per created pair.

recv_context(*number=1*, *sequential=False*)
Receives EPR pair with a remote node (see doc of *create_context()*)

Parameters

- **number** (int) –
- **sequential** (bool) –

3.5.12 netqasm.sdk.external

3.5.13 netqasm.sdk.futures

Abstractions for classical runtime values.

This module contains the *BaseFuture* class and its subclasses.

exception netqasm.sdk.futures.**NoValueError**
Bases: RuntimeError

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception netqasm.sdk.futures.**NonConstantIndexError**
Bases: RuntimeError

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

netqasm.sdk.futures.**as_int_when_value**(*cls*)
A decorator for the *BaseFuture* class which makes is behave like an *int* when the property *value* is not *None*.

class netqasm.sdk.futures.**BaseFuture**(*connection*)
Bases: int

Base class for Future-like objects.

A Future represents a classical value that becomes available at some point in the future. At the moment, a Future always represents an integer value.

Futures have a *value* property that is either *None* (when the value is not yet available), or has a concrete integer value. Executing a subroutine on the quantum node controller makes the value property go from *None* to a concrete value, granted that the subroutine sets the value of whatever the Future represents. When the *value*

property has a concrete value, the Future behaves like an *int* and can then also be used in Python expressions just like integers.

Typically, a Future instance is obtained as the result of an SDK function, and not directly instantiated in application code. For example, calling *measure()* on a Qubit returns a Future representing the measurement outcome. Only after the subroutine containing the operations has been executed by the quantum node controller (by flushing the subroutine), the measurement outcome will have an actual value, and the Future can be resolved into an *int*.

Parameters **connection** (*sdkconn.BaseNetQASMConnection*) –

property builder

Return type *Builder*

property value

Get the value of the future. If it's not set yet, *None* is returned.

Return type Optional[int]

add(*other*, *mod=None*)

Add another value to this Future's value.

The result is stored in this Future.

Let the quantum node controller add a value to the value represented by this Future. The addition operation is compiled into a subroutine and is fully executed by the quantum node controller. This avoids the need to wait for a subroutine result (which resolves the Future's *value*) and then doing the addition on the Host.

Parameters

- **other** (Union[int, str, *Register*, *BaseFuture*]) – value to add to this Future's value
- **mod** (Optional[int]) – do the addition modulo *mod*

Return type None

if_eq(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_ne(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_lt(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_ge(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_ez()

Return type *SdkIfContext*

if_nz()

Return type *SdkIfContext*

as_integer_ratio()
Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_length(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

conjugate(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

denominator(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

from_bytes(*byteorder*, ***, *signed=False*)
Return the integer represented by the given array of bytes.

bytes Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value.

signed Indicates whether two's complement is used to represent the integer.

imag(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

numerator(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

real(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

to_bytes(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

class netqasm.sdk.futures.**Future**(*connection*, *address*, *index*)
Bases: *netqasm.sdk.futures.BaseFuture*

Represents a single array entry value that will become available in the future.

See *BaseFuture* for more explanation about Futures.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) –
- **address** (*int*) –
- **index** (*Union[int, Future, operand.Register, RegFuture]*) –

__init__(*connection*, *address*, *index*)
Future constructor. Typically not used directly.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) – connection through which subroutines are sent that contain the array entry corresponding to this Future
- **address** (*int*) – address of the array
- **index** (*Union[int, Future, operand.Register, RegFuture]*) – index in the array

add(*other*, *mod=None*)
Add another value to this Future's value.

The result is stored in this Future.

Let the quantum node controller add a value to the value represented by this Future. The addition operation is compiled into a subroutine and is fully executed by the quantum node controller. This avoids the need to wait for a subroutine result (which resolves the Future's *value*) and then doing the addition on the Host.

Parameters

- **other** (Union[int, str, *Register*, *BaseFuture*]) – value to add to this Future's value
- **mod** (Optional[int]) – do the addition modulo *mod*

Return type None

get_load_commands(*register*)
Return a list of ProtoSubroutine commands for loading this Future into the specified register.

Parameters **register** (*Register*) –

Return type List[Union[*ICmd*, *BranchLabel*]]

get_address_entry()
Convert this Future to an ArrayEntry object to be used an instruction operand.

Return type *ArrayEntry*

as_integer_ratio()
Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_length(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

property builder

Return type *Builder*

conjugate(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

denominator(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

from_bytes(*byteorder*, ***, *signed=False*)
Return the integer represented by the given array of bytes.

bytes Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value.

signed Indicates whether two's complement is used to represent the integer.

if_eq(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_ez()

Return type *SdkIfContext*

if_ge(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_lt(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_ne(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_nz()

Return type *SdkIfContext*

imag(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

numerator(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

real(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

to_bytes(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

property value
Get the value of the future. If it's not set yet, *None* is returned.

Return type Optional[int]

class netqasm.sdk.futures.**RegFuture**(*connection*, *reg=None*)
Bases: *netqasm.sdk.futures.BaseFuture*

Represents a single register value that will become available in the future.

See *BaseFuture* for more explanation about Futures.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) –
- **reg** (*Optional[operand.Register]*) –

__init__(*connection*, *reg=None*)
RegFuture constructor. Typically not used directly.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) – connection through which subroutines are sent that contain the array entry corresponding to this Future
- **reg** (*Optional[operand.Register]*) – specific NetQASM register that will hold the Future's value. If None, a suitable register is automatically used.

property reg

Return type Optional[*Register*]

add(*other*, *mod=None*)
Add another value to this Future's value.

The result is stored in this Future.

Let the quantum node controller add a value to the value represented by this Future. The addition operation is compiled into a subroutine and is fully executed by the quantum node controller. This avoids the need to wait for a subroutine result (which resolves the Future's *value*) and then doing the addition on the Host.

Parameters

- **other** (Union[int, str, *Register*, *BaseFuture*]) – value to add to this Future's value
- **mod** (Optional[int]) – do the addition modulo *mod*

Return type None

as_integer_ratio()
Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_length(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

property builder

Return type *Builder*

conjugate(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

denominator(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

from_bytes(*byteorder*, ***, *signed=False*)
Return the integer represented by the given array of bytes.

bytes Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value.

signed Indicates whether two's complement is used to represent the integer.

if_eq(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_ez()

Return type *SdkIfContext*

if_ge(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_lt(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_ne(*other*)

Parameters **other** (*Optional[T_CValue]*) –

Return type *SdkIfContext*

if_nz()

Return type *SdkIfContext*

imag(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

numerator(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

real(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

to_bytes(**args*, ***kwargs*)
Check if the value is set, otherwise raise an error

property value
Get the value of the future. If it's not set yet, *None* is returned.

Return type Optional[int]

class netqasm.sdk.futures.**Array**(*connection*, *length*, *address*, *init_values=None*, *lineno=None*)
Bases: object

Wrapper around an array in Shared Memory.

An *Array* instance provides methods to inspect and operate on an array that exists in shared memory. They are typically obtained as return values to certain SDK methods.

Elements or slices of the array can be captured as Futures.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) –
- **length** (*int*) –
- **address** (*int*) –
- **init_values** (*Optional[List[Optional[int]]]*) –
- **lineno** (*Optional[HostLine]*) –

__init__(*connection*, *length*, *address*, *init_values=None*, *lineno=None*)
Array constructor. Typically not used directly.

Parameters

- **connection** (*sdkconn.BaseNetQASMConnection*) – connection of the application this array is part of
- **length** (*int*) – length of the array
- **address** (*int*) – address of the array
- **init_values** (*Optional[List[Optional[int]]]*) – initial values of the array. Must have length *length*.
- **lineno** (*Optional[HostLine]*) – line number where the array is created in the Python source code

property lineno
What line in host application file initiated this array

Return type Optional[*HostLine*]

property address

Return type int

property builder

Return type *Builder*

get_future_index(*index*)
Get a Future representing a particular array element

Parameters **index** (Union[int, str, *Register*, *RegFuture*]) –

Return type *Future*

get_future_slice(*s*)
Get a list of Futures each representing one element in a particular array slice

Parameters **s** (slice) –

Return type List[*Future*]

foreach()

Create a context of code that gets called for each element in the array.

Returns a future of the array value at the current index.

Code inside the context *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Example:

```
with NetQASMConnection(app_name="alice") as alice:
    outcomes = alice.new_array(10)
    values = alice.new_array(
        10,
        init_values=[random.randint(0, 1) for _ in range(10)]
    )
    with values.foreach() as v:
        q = Qubit(alice)
        with v.if_eq(1):
            q.H()
        q.measure(future=outcomes.get_future_index(i))
```

Return type *SdkForEachContext*

enumerate()

Create a context of code that gets called for each element in the array and includes a counter.

Returns a tuple (*index*, *future*) where *future* is a Future of the array value at the current index.

Code inside the context *must* be compilable to NetQASM, that is, it should only contain quantum operations and/or classical values that are are stored in shared memory (arrays and registers). No classical communication is allowed.

Example:

```
with NetQASMConnection(app_name="alice") as alice:
    outcomes = alice.new_array(10)
    values = alice.new_array(
        10,
        init_values=[random.randint(0, 1) for _ in range(10)]
    )
    with values.enumerate() as (i, v):
        q = Qubit(alice)
        with v.if_eq(1):
            q.H()
        q.measure(future=outcomes.get_future_index(i))
```

Return type *SdkForEachContext*

undefine()

Undefine (i.e. set to 'None') all elements in the array.

Return type None

3.5.14 netqasm.sdk.network

class netqasm.sdk.network.**NetworkInfo**
Bases: object

Global information about the current quantum network environment.

This class is a container for static functions that provide information about the current network setting. Applications may use this information to e.g. obtain node IDs or map party names to nodes.

Concrete runtime contexts (like a simulator, or a real hardware setup) should override these methods to provide the information specific to that context.

abstract classmethod get_node_id_for_app(*app_name*)
Returns the node id for the app with the given name

Parameters **app_name** (str) –

Return type int

abstract classmethod get_node_name_for_app(*app_name*)
Returns the node name for the app with the given name

Parameters **app_name** (str) –

Return type str

3.5.15 netqasm.sdk.progress_bar

class netqasm.sdk.progress_bar.**ProgressBar**(*maxitr*)
Bases: object

Parameters **maxitr** (int) –

increase()

Return type None

update()

Return type None

close()

Return type None

3.5.16 netqasm.sdk.qubit

Qubit representation.

This module contains the *Qubit* class, which are used by application scripts as handles to in-memory qubits.

exception netqasm.sdk.qubit.**QubitNotActiveError**
Bases: MemoryError

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class netqasm.sdk.qubit.**QubitMeasureBasis**(*value*)
Bases: enum.Enum

An enumeration.

X = 0

Y = 1

Z = 2

class netqasm.sdk.qubit.**Qubit**(*conn*, *add_new_command=True*, *ent_info=None*, *virtual_address=None*)
Bases: object

Representation of a qubit that has been allocated in the quantum node.

A *Qubit* instance represents a quantum state that is stored in a physical qubit somewhere in the quantum node. The particular qubit is identified by its virtual qubit ID. To which physical qubit ID this is mapped (at a given time), is handled completely by the quantum node controller and is not known to the *Qubit* itself.

A *Qubit* object can be instantiated in an application script. Such an instantiation is automatically compiled into NetQASM instructions that allocate and initialize a new qubit in the quantum node controller.

A *Qubit* object may also be obtained by SDK functions that return them, like the *create()* method on an *EPRSocket*, which returns the object as a handle to the qubit that is now entangled with one in another node.

Qubit operations like applying gates and measuring them are done by calling methods on a *Qubit* instance.

Parameters

- **conn** (*sdkconn.BaseNetQASMConnection*) –
- **add_new_command** (*bool*) –
- **ent_info** (*Optional[qlink_compat.LinkLayerOKTypeK]*) –
- **virtual_address** (*Optional[int]*) –

__init__(*conn*, *add_new_command=True*, *ent_info=None*, *virtual_address=None*)
Qubit constructor. This is the standard way to allocate a new qubit in an application.

Parameters

- **conn** (*sdkconn.BaseNetQASMConnection*) – connection of the application in which to allocate the qubit
- **add_new_command** (*bool*) – whether to automatically add NetQASM instructions to the current subroutine to allocate and initialize the qubit
- **ent_info** (*Optional[qlink_compat.LinkLayerOKTypeK]*) – entanglement generation information in case this qubit is the result of an entanglement generation request
- **virtual_address** (*Optional[int]*) – explicit virtual ID to use for this qubit. If None, a free ID is automatically chosen.

property connection
Get the NetQASM connection of this qubit

Return type sdkconn.BaseNetQASMConnection

property builder
Get the Builder of this qubit's connection

Return type *Builder*

property qubit_id
Get the qubit ID

Return type int

property active

Return type bool

property entanglement_info
Get information about the successful link layer request that resulted in this qubit.

Return type Optional[qlink_compat.LinkLayerOKTypeK]

property remote_entangled_node
Get the name of the remote node the qubit is entangled with.

If not entanled, *None* is returned.

Return type Optional[str]

assert_active()
Assert that the qubit is active, i.e. allocated.

Return type None

measure(*future=None*, *inplace=False*, *store_array=True*, *basis=<QubitMeasureBasis.Z: 2>*, *basis_rotations=None*)
Measure the qubit in the standard basis and get the measurement outcome.

Parameters

- **future** (Union[*Future*, *RegFuture*, None]) – the *Future* to place the outcome in. If None, a Future is created automatically.
- **inplace** (bool) – If False, the measurement is destructive and the qubit is removed from memory. If True, the qubit is left in the post-measurement state.
- **store_array** (bool) – whether to store the outcome in an array. If not, it is placed in a register. Only used if *future* is None.
- **basis** (*QubitMeasureBasis*) – in which of the Pauli bases (X, Y or Z) to measure. Default is Z. Ignored if *basis_rotations* is not None.
- **basis_rotations** (Optional[Tuple[int, int, int]]) – rotations to apply before measuring in the Z-basis. This can be used to specify arbitrary measurement bases. The 3 values are interpreted as 3 rotation angles, for an X- Y-, and another X-rotation, respectively. Each angle is interpreted as a multiple of pi/16. For example, if *basis_rotations* is (8, 0, 0), an X-rotation is applied with angle 8*pi/16 = pi/2 radians, followed by a Y-rotation of angle 0 and an X-rotation of angle 0. Finally, the measurement is done in the Z-basis.

Return type Union[*Future*, *RegFuture*]

Returns the Future representing the measurement outcome. It is a *Future* if

the result is in an array (default) or *RegFuture* if the result is in a register.

X()
Apply an X gate on the qubit.

Return type None

Y()
Apply a Y gate on the qubit.

Return type None

Z()
Apply a Z gate on the qubit.

Return type None

T()
Apply a T gate on the qubit.

A T gate is a Z-rotation with angle pi/4.

Return type None

H()
Apply a Hadamard gate on the qubit.

Return type None

K()
Apply a K gate on the qubit.

A K gate moves the |0> state to +|i> (positive Y) and vice versa.

Return type None

S()
Apply an S gate on the qubit.

An S gate is a Z-rotation with angle pi/2.

Return type None

rot_X(*n=0*, *d=0*, *angle=None*)
Do a rotation around the X-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

rot_Y(*n=0*, *d=0*, *angle=None*)
Do a rotation around the Y-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

rot_Z(*n=0*, *d=0*, *angle=None*)
Do a rotation around the Z-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

cnot(*target*)
Apply a CNOT gate between this qubit (control) and a target qubit.

Parameters **target** (*Qubit*) – target qubit. Should have the same connection as this qubit.

Return type None

cphase(*target*)
Apply a CPHASE (CZ) gate between this qubit (control) and a target qubit.

Parameters **target** (*Qubit*) – target qubit. Should have the same connection as this qubit.

Return type None

reset()
Reset the qubit to the state |0>.

Return type None

free()
Free the qubit and its virtual ID.

After freeing, the underlying physical qubit can be used to store another state.

Return type None

class netqasm.sdk.qubit.**FutureQubit**(*conn*, *future_id*)
Bases: *netqasm.sdk.qubit.Qubit*

A Qubit that will be available in the future.

This class is very similar to the *Future* class which is used for classical values. A *FutureQubit* acts like a *Qubit* so that all qubit operations can be applied on it. FutureQubits are typically the result of EPR creating requests, where they represent the qubits that will be available when EPR generation has finished.

Parameters

- **conn** (*sdkconn.BaseNetQASMConnection*) –
- **future_id** (*Future*) –

__init__(*conn*, *future_id*)
FutureQubit constructor. Typically not used directly.

Parameters

- **conn** (*sdkconn.BaseNetQASMConnection*) – connection through which subroutines are sent that contain this qubit
- **future_id** (*Future*) – the virtual ID this qubit will have

H()
Apply a Hadamard gate on the qubit.

Return type None

K()
Apply a K gate on the qubit.

A K gate moves the |0> state to +|i> (positive Y) and vice versa.

Return type None

S()
Apply an S gate on the qubit.

An S gate is a Z-rotation with angle pi/2.

Return type None

T()
Apply a T gate on the qubit.

A T gate is a Z-rotation with angle pi/4.

Return type None

X()
Apply an X gate on the qubit.

Return type None

Y()
Apply a Y gate on the qubit.

Return type None

Z()
Apply a Z gate on the qubit.

Return type None

property active

Return type bool

assert_active()
Assert that the qubit is active, i.e. allocated.

Return type None

property builder
Get the Builder of this qubit's connection

Return type *Builder*

cnot(*target*)
Apply a CNOT gate between this qubit (control) and a target qubit.

Parameters **target** (*Qubit*) – target qubit. Should have the same connection as this qubit.

Return type None

property connection
Get the NetQASM connection of this qubit

Return type sdkconn.BaseNetQASMConnection

cphase(*target*)
Apply a CPHASE (CZ) gate between this qubit (control) and a target qubit.

Parameters **target** (*Qubit*) – target qubit. Should have the same connection as this qubit.

Return type None

free()
Free the qubit and its virtual ID.

After freeing, the underlying physical qubit can be used to store another state.

Return type None

measure(*future=None*, *inplace=False*, *store_array=True*, *basis=<QubitMeasureBasis.Z: 2>*, *basis_rotations=None*)
Measure the qubit in the standard basis and get the measurement outcome.

Parameters

- **future** (Union[*Future*, *RegFuture*, None]) – the *Future* to place the outcome in. If None, a Future is created automatically.
- **inplace** (bool) – If False, the measurement is destructive and the qubit is removed from memory. If True, the qubit is left in the post-measurement state.
- **store_array** (bool) – whether to store the outcome in an array. If not, it is placed in a register. Only used if *future* is None.
- **basis** (*QubitMeasureBasis*) – in which of the Pauli bases (X, Y or Z) to measure. Default is Z. Ignored if *basis_rotations* is not None.
- **basis_rotations** (Optional[Tuple[int, int, int]]) – rotations to apply before measuring in the Z-basis. This can be used to specify arbitrary measurement bases. The 3 values are interpreted as 3 rotation angles, for an X- Y-, and another X-rotation, respectively. Each angle is interpreted as a multiple of pi/16. For example, if *basis_rotations* is (8, 0, 0), an X-rotation is applied with angle 8*pi/16 = pi/2 radians, followed by a Y-rotation of angle 0 and an X-rotation of angle 0. Finally, the measurement is done in the Z-basis.

Return type Union[*Future*, *RegFuture*]

Returns the Future representing the measurement outcome. It is a *Future* if

the result is in an array (default) or *RegFuture* if the result is in a register.

property qubit_id
Get the qubit ID

Return type int

reset()
Reset the qubit to the state |0>.

Return type None

rot_X(*n=0*, *d=0*, *angle=None*)
Do a rotation around the X-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

rot_Y (*n=0*, *d=0*, *angle=None*)
Do a rotation around the Y-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

rot_Z (*n=0*, *d=0*, *angle=None*)
Do a rotation around the Z-axis of the specified angle.

The angle is interpreted as * pi / 2 ^d` radians. For example, (n, d) = (1, 2) represents an angle of pi/4 radians. If *angle* is specified, *n* and *d* are ignored and this instruction is automatically converted into a sequence of (n, d) rotations such that the discrete (n, d) values approximate the original angle.

Parameters

- **n** (*Union[int, Template]*) – numerator of discrete angle specification. Can be a Template, in which case the subroutine containing this command should first be instantiated before flushing.
- **d** (*int*) – denomerator of discrete angle specification
- **angle** (*Optional[float]*) – exact floating-point angle, defaults to None

property entanglement_info
Get information about the successful link layer request that resulted in this qubit.

Return type Optional[qlink_compat.LinkLayerOKTypeK]

property remote_entangled_node
Get the name of the remote node the qubit is entangled with.

If not entanled, *None* is returned.

Return type Optional[str]

3.5.17 netqasm.sdk.shared_memory

Abstractions for the classical memory shared between the Host and the quantum node controller.

class netqasm.sdk.shared_memory.**RegisterGroup**
Bases: object

A register group (like "R", or "Q") in shared memory.

netqasm.sdk.shared_memory.**setup_registers**()

Return type Dict[*RegisterName*, *RegisterGroup*]

class netqasm.sdk.shared_memory.**Arrays**
Bases: object

has_array(*address*)

Parameters **address** (int) –

Return type bool

init_new_array(*address*, *length*)

Parameters

- **address** (int) –
- **length** (int) –

Return type None

class netqasm.sdk.shared_memory.**SharedMemory**
Bases: object

Representation of the classical memory that is shared between the Host and the quantum node controller.

Each application is associated with a single *SharedMemory*. Although the name contains "shared", it is typically only used in one "direction": the quantum node controller writes to it, and the Host reads from it.

Shared memory consists of two types of memory: registers and arrays. The quantum node controller writes to the shared memory if it executes a "return" NetQASM instruction (*ret_reg* or *ret_arr*). This is typically done at the end of a subroutine. The Host can then read the values from the shared memory, which means that the Host effectively receives the returned values.

When trying to use a value represented by a *Future*, it will try to get the value from the shared memory. So, *only* after the quantum node controller has executed a subroutine containing the relevant *ret_reg* or *ret_arr* NetQASM instruction, the shared memory is updated and the value from the Future can be used.

The specific runtime context determines how a shared memory is implemented and by which component it is controlled. In the case of a physical setup, where the Host and the quantum node controller may be separate devices, "writing to the shared memory" may simply be "sending values from the quantum node controller to the Host". On the other hand, a simulator that runs the Host and the quantum node controller in the same process might e.g. use a global shared object.

get_register(*register*)

Parameters **register** (Union[str, *Register*]) –

Return type Optional[int]

set_register(*register*, *value*)

Parameters

- **register** (Union[str, *Register*]) –

- **value** (int) –

Return type None

get_array_part(*address*, *index*)

Parameters

- **address** (int) –
- **index** (Union[int, slice]) –

Return type Union[None, int, List[Optional[int]]]

set_array_part(*address*, *index*, *value*)

Parameters

- **address** (int) –
- **index** (Union[int, slice]) –
- **value** (Union[None, int, List[Optional[int]]]) –

init_new_array(*address*, *length=1*, *new_array=None*)

Parameters

- **address** (int) –
- **length** (int) –
- **new_array** (Optional[List[Optional[int]]]) –

Return type None

class netqasm.sdk.shared_memory.**SharedMemoryManager**

Bases: object

Global object that manages shared memories. Typically used by simulators.

This class can be used as a global object that stores *SharedMemory* objects. Simulators that simulate Hosts and the quantum node controllers in the same process may use this to have a single location for creating and accessing shared memories.

classmethod create_shared_memory(*node_name*, *key=None*)

Parameters

- **node_name** (str) –
- **key** (Optional[int]) –

Return type *SharedMemory*

classmethod get_shared_memory(*node_name*, *key=None*)

Parameters

- **node_name** (str) –
- **key** (Optional[int]) –

Return type Optional[*SharedMemory*]

classmethod reset_memories()

Return type None

3.5.18 netqasm.sdk.toolbox

3.5.19 netqasm.sdk.toolbox.gates

netqasm.sdk.toolbox.gates.**t_inverse**(*q*)

Performs an inverse of the T gate by applying the T gate 7 times.

Parameters **q** (*Qubit*) –

Return type None

netqasm.sdk.toolbox.gates.**toffoli_gate**(*control1*, *control2*, *target*)

Performs a Toffoli gate with *control1* and *control2* as control qubits and *target* as target, using CNOTS, Ts and Hadamard gates.

See https://en.wikipedia.org/wiki/Toffoli_gate

Parameters

- **control1** (*Qubit*) –
- **control2** (*Qubit*) –
- **target** (*Qubit*) –

Return type None

3.5.20 netqasm.sdk.toolbox.measurements

netqasm.sdk.toolbox.measurements.**parity_meas**(*qubits*, *bases*)

Performs a parity measurement on the provided qubits in the Pauli bases specified by 'bases'. *bases* should be a string with letters in 'IXYZ' and optionally start with '-'. If *bases* starts with '-', then the measurement outcome is flipped. The *basis* should have the same length as the number of qubits provided or +1 if starts with '-'. If more than one letter of 'bases' is not identity, then an ancilla qubit will be used, which is created using the connection of the first qubit.

Parameters

- **qubits** (List[*Qubit*]) – The qubits to measure
- **bases** (*str*) – What parity meas to perform.

Returns The measurement outcome

Return type class:~.sdk.futures.Future

3.5.21 netqasm.sdk.toolbox.multi_node

netqasm.sdk.toolbox.multi_node.**create_ghz**(*down_epr_socket=None*, *up_epr_socket=None*, *down_socket=None*, *up_socket=None*, *do_corrections=False*)

Local protocol to create a GHZ state between mutliples nodes.

EPR pairs are generated in a line and turned into a GHZ state by performing half of a Bell measurement. That is, CNOT and H are applied but only the control qubit is measured. If *do_corrections=False* (default) this measurement outcome is returned along with the qubit to be able to know what corrections might need to be applied. If the node is at the start or end of the line, the measurement outcome 0 is always returned since there is no measurement performed. The measurement outcome indicates if the next node in the line should flip its qubit to get the standard GHZ state: $|0\rangle^{\otimes n} + |1\rangle^{\otimes n}$.

On the other hand if *do_corrections=True*, then the classical sockets *down_socket* and/or *up_socket* will be used to communicate the outcomes and automatically perform the corrections.

Depending on if down_epr_socket and/or up_epr_socket is specified the node, either takes the role of the:

- "start", which intialises the process and creates an EPR with the next node using the *up_epr_socket*.
- "middle", which receives an EPR pair on the *down_epr_socket* and then creates one on the *up_epr_socket*.
- "end", which receives an EPR pair on the *down_epr_socket*.

NOTE There has to be exactly one "start" and exactly one "end" but zero or more "middle". NOTE Both *down_epr_socket* and *up_epr_socket* cannot be *None*.

Parameters

- **down_epr_socket** (sdk.epr_socket.esck.EPRSocket) – The esck.EPRSocket to be used for receiving EPR pairs from downstream.
- **up_epr_socket** (sdk.epr_socket.esck.EPRSocket) – The esck.EPRSocket to be used for create EPR pairs upstream.
- **down_socket** (*sdk.classical_communication.socket.Socket*) – The classical socket to be used for sending corrections, if *do_corrections* = *True*.
- **up_socket** (*sdk.classical_communication.socket.Socket*) – The classical socket to be used for sending corrections, if *do_corrections* = *True*.
- **do_corrections** (*bool*) – If corrections should be applied to make the GHZ in the standard form $|0\rangle^{\otimes n} + |1\rangle^{\otimes n}$ or not.

Returns Of the form *(q, m)* where *q* is the qubit part of the state and *m* is the measurement outcome.

Return type tuple

3.5.22 netqasm.sdk.toolbox.sim_states

3.5.23 netqasm.sdk.toolbox.state_prep

netqasm.sdk.toolbox.state_prep.**set_qubit_state**(*qubit*, *phi=0.0*, *theta=0.0*)

Assuming that the qubit is in the state $|0\rangle$, this function rotates the state to $\cos(\theta/2)|0\rangle + e^{i\phi}\sin(\theta/2)|1\rangle$.

Parameters

- **qubit** (*sdk.qubit.Qubit*) – The qubit to prepare the state.
- **phi** (*float*) – Angle around Z-axis from X-axis
- **theta** (*float*) – Angle from Z-axis

Return type None

netqasm.sdk.toolbox.state_prep.**get_angle_spec_from_float**(*angle*, *tol=0.0001*)

Tries to find the shortest sequence of (n, d) such that $abs(\sum_i n_i \pi/2^{d_i} - angle) < tol$ This is to find a sequence of rotations for a given angle.

Parameters

- **angle** (*float*) – The angle to approximate
- **tol** (*float*) – Tolerance to use

Return type List[Tuple[int, int]]

3.6 netqasm.util

3.6.1 netqasm.util.error

exception netqasm.util.error.**NetQASMSyntaxError**
Bases: SyntaxError

args

filename
exception filename

lineno
exception lineno

msg
exception msg

offset
exception offset

print_file_and_line
exception print_file_and_line

text
exception text

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception netqasm.util.error.**NetQASMInstrError**
Bases: ValueError

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception netqasm.util.error.**NoCircuitRuleError**
Bases: RuntimeError

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception netqasm.util.error.**NotAllocatedError**
Bases: RuntimeError

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception netqasm.util.error.**SubroutineAbortedError**
Bases: RuntimeError

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

3.6.2 netqasm.util.log

class netqasm.util.log.**HostLine**(*filename*, *lineno*)
Bases: object

class netqasm.util.log.**LineTracker**(*log_config*)
Bases: object

__init__(*log_config*)

Parameters **log_config** (*LogConfig*) –

get_line()

Return type Optional[*HostLine*]

3.6.3 netqasm.util.quantum_gates

netqasm.util.quantum_gates.**get_rotation_matrix**(*axis*, *angle*)
Returns a single-qubit rotation matrix given an axis and an angle

Return type ndarray

netqasm.util.quantum_gates.**get_controlled_rotation_matrix**(*axis*, *angle*)

Return type ndarray

netqasm.util.quantum_gates.**gate_to_matrix**(*instr*, *angle=None*)
Returns the matrix representation of a quantum gate

netqasm.util.quantum_gates.**are_matrices_equal**(**matrices*)
Checks if two matrices are equal, disregarding any global phase

3.6.4 netqasm.util.states

netqasm.util.states.**bloch_sphere_rep**(*mat*)
Computes polar coordinates in Bloch sphere given a single-qubit density matrix

Parameters **mat** (*numpy.ndarray*) – The single-qubit density matrix

Returns tuple

Return type (theta, phi, r)

3.6.5 netqasm.util.string

netqasm.util.string.**group_by_word**(*line*, *seperator=' '*, *brackets=None*)
Groups a string by words and contents within brackets

Parameters **line** (str) –

Return type List[str]

netqasm.util.string.**is_variable_name**(*variable*)

netqasm.util.string.**is_number**(*number*)

netqasm.util.string.**is_float**(*value*)

netqasm.util.string.**rspaces**(*val*, *min_chars=4*)

3.6.6 netqasm.util.thread

netqasm.util.thread.**as_completed**(*futures*, *names=None*, *sleep_time=0*)

3.6.7 netqasm.util.yaml

netqasm.util.yaml.**load_yaml**(*file_path*)

netqasm.util.yaml.**dump_yaml**(*data*, *file_path*)

CHAPTER

FOUR

KNOWN ISSUES

When using the SquidASM simulator, when the simulation of an application finishes, you may see a warning coming from NetSquid saying `WARNING: a [sic] expression handler missing`. This seems to be a bug relating to NetSquid and is outside the control of `netqasm` but should not impact the simulation of the application.

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

T

U

V

W

X

Y

Z