Qblox Instruments

Release 0.8.0

Qblox

Dec 09, 2022

GETTING STARTED

The Qblox instruments package contains everything to get started with Qblox instruments (i.e. Python drivers, documention and tutorials).

CHAPTER

ONE

CONTENTS

1.1 Overview

In this section we will have a look at the Qblox instruments and their IO and shortly explain what they are for.

1.1.1 Pulsar

The Pulsar modules are compact qubit control and readout modules (QCM / QRM). They are conveniently controlled over Ethernet and are easily connected to your setup using SMA and SMP connectors. They are also easily combined with other Qblox instruments using Qblox's SYNQ technology.

Front

On the front of a Pulsar module you will find the following components:

- **4 x SMA male connectors**: 4 output for a Pulsar QCM ($O^{[1-4]}$: 5 Vpp @ 50) ; or 2 outputs and 2 input channels for a Pulsar QRM ($O^{[1-2]}$: 1 Vpp @ 50 , $I^{[1-2]}$: 2 Vpp @ 50).
- **4 x SMP female connectors**: Marker output channels (0-3.3 V TTL).
- **6 x status LEDs**: See section *Frontpanel LEDs*.

Back

On the back of a Pulsar module you will find the following components:

- **Power**: 12 V DC power supply input.
- **RJ45**: Host PC connection.
- **3 x USB**:
  - **UART/JTAG**: For debug purposes only.
  - **2 x SYNQ**: For synchronizing multiple Qblox instruments.
- **3 x SMA**:
  - **REF^{in}**: External 10MHz reference clock input (1 Vpp nominal @ 50).
  - **REF^{out}**: 10MHz reference clock output (0-3.3 V @ 50).
  - **TRIG^{in}**: External trigger input (0-3.3 V, high-Z).

1.1.2 SPI Rack

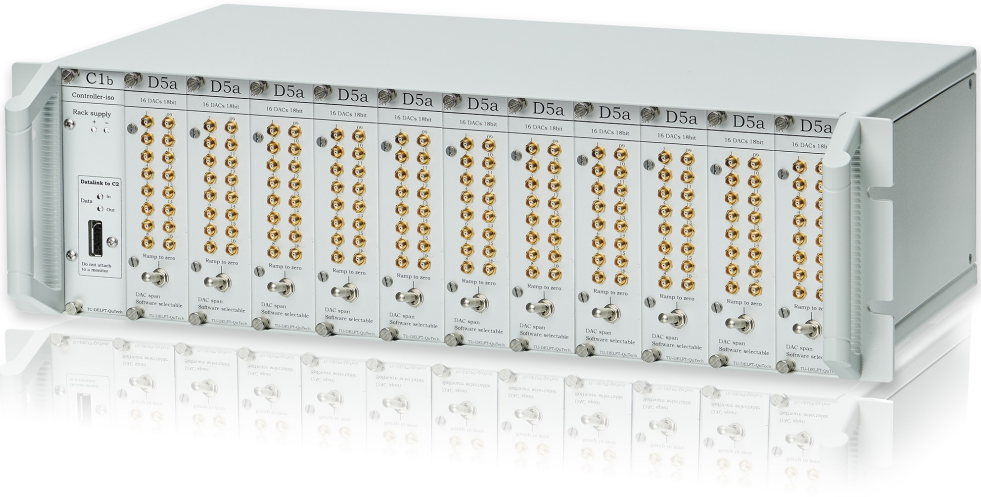
The SPI Rack is a modular system for DC current and DC voltage source modules. Modules are designed to maximize output stability. Together with our Galvanically isolated Control Interface and Isolated power supply ground loops are avoided and interference (like 50 Hz) is minimized.

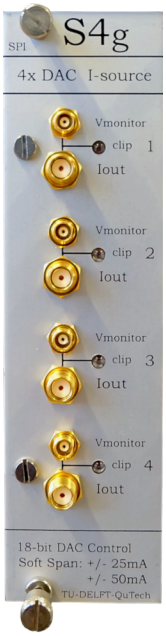
S4g 4 channel current source

On the front of each S4g module you find:

- **4 x SMA**: DC current source with software-selectable range: ±50 mA, ±25 mA, +50 mA, 18 bit resolution.
- **4 x MCX**: Voltage monitor for each channel.

Technical specifications S4g


C1b
Controller-16o
Rack supply
Establish to C2
Data
In
Out
Do not attach to a monitor
To DELFT-QuTech
D5a
16 DACs 18bit
Ramp to zero
DAC span
Software selectable
TU-DELFT-QuTech


SPI
S4g
4x DAC I-source
Vmonitor
clip 1
Iout
Vmonitor
clip 2
Iout
Vmonitor
clip 3
Iout
Vmonitor
clip 4
Iout
18-bit DAC Control
Soft Span: +/- 25mA
+/- 50mA
TU-DELFT-QuTech

D5a 16 channel DAC

On the front of each D5a module you find:

- **16 x MCX**: DC voltage source with software-selectable range (±4, ±2, +4 V), 18 bit resolution, 550 Ohm output impedance.
- **1 x switch**: Switch to ramp all outputs down to zero volt.

Technical specifications D5a

1.2 Installation

In this section we will explain how to install the Qblox instrument package. To do this make sure you have Python 3.8 or newer installed.

Tip: The Python version can be queried by running `$ python --version` in your terminal of choice.

The Qblox instrument package can be installed through PIP, by executing the following command:

```
$ pip install qblox-instruments
```

This will install the most recent version of the package. Please make sure that the version you install is compatible with your current module firmware (See Qblox instruments PyPI and section *Updating*). To install a specific version of the package, execute the following command:

```
$ pip install qblox-instruments==<version>
```

Tip: You can query your installed version by executing `$ pip show qblox-instruments`.

1.3 Connecting a Pulsar

In this section we will explain how to connect a Qblox Pulsar QCM or QRM module to your host PC. Please make sure that you have the Qblox instruments package installed before proceeding (see section *Installation*) and that your host PC has an Ethernet port.

1.3.1 Connecting to a single module

As an example, we will consider a setup composed of:

- A laptop (host PC) with a USB Ethernet adapter.
- A Pulsar QCM, configured to use the default IP configuration `192.168.0.2/24`.

The following steps will allow you to successfully connect the module to your local network:

1. Connect the module to your host PC using an Ethernet cable.
2. Power up the module. The module is ready when all LEDs turn on (See section *Frontpanel LEDs*).
3. Configure the network adapter of the host PC, so that its IP address is within subnet `192.168.0.X` (where `X` is in a range from 3 to 254). Make sure the subnet mask is set to `255.255.255.0`.

   Note: Configuration of a network adapter varies slightly between operating systems. See section *Network adapter configuration* for a Windows, Linux and MacOS description.

   At this point your setup will look similar to the example setup in the figure below:

   After a few seconds, the module should be present on the local network. You can verify this by executing the following command in a terminal of your choice.

   Note: The default IP address of the module is `192.168.0.2`. Replace the IP address of any following instruction accordingly if the IP address of the module was ever changed. See section *Finding the IP address of a module* in case you do not know the IP address.

   ```
   $ ping 192.168.0.2 # Press Ctrl + C to terminate the program
   ```

   If successful, the output should be similar to the following example output:

   ```
   PING 192.168.0.2 (192.168.0.2): 56 data bytes
   64 bytes from 192.168.0.2: icmp_seq=0 ttl=64 time=0.396 ms
   64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=0.232 ms
   64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.261 ms
   ```

4. Finally, connect to the module from your host PC by running the following snippet using a Python 3.8 environment like an interactive shell or a Jupyter Notebook:

   ```
   # Import driver
   from qblox_instruments import Pulsar

   # Connect to module
   pulsar_qcm = Pulsar("qcm", "192.168.0.2")
   ```


QBLOX
PULSAR QCM
QUBIT CONTROL MODULE

Tip: Close the connection to the module using `pulsar_qcm.close()`.

Network adapter configuration

Windows

(a) Go to *Control Panel* > *Network and Internet* > *Network Connections*.

(b) Select the network adapter that is connected to the same network as the module(s) (e.g. USB Ethernet adapter).

(c) Right-click and select *properties*.

(d) Select *Internet Protocol Version 4 (TCP/IPv4)* and click on properties.

(e) Select *Use the following IP address* and specify the desired IP address (e.g. `192.168.0.200`) and subnet mask (i.e. `255.255.255.0`)

(f) Click on *OK* and *Close*.

Ubuntu

(a) Go to *System Settings* > *Connections*.

(b) Select the network adapter that is connected to the same network as the module(s) (e.g. USB Ethernet adapter).

(c) Set *IPv4 method* to *Manual* and specify the desired IP address (e.g. `192.168.0.200`) and subnet mask (i.e. `255.255.255.0`).

(d) Click on *Apply* and close the window.

MacOS

(a) Open *System Preferences* > *Network*.

(b) Select the network adapter that is connected to the same network as the module(s) (e.g. USB Ethernet adapter).

(c) Set *Configure IPv4* to *Manually* and specify the desired IP address (e.g. `192.168.0.200`) and subnet mask (i.e. `255.255.255.0`).

(d) Click on *Apply* and close the window.

Finding the IP address of a module

Here we provide some tips to help find the IP address of an instrument (e.g. a Pulsar QCM or QRM).

Tip: When changing the IP address of the instrument, put a label on it with its new IP address. This will help avoid connectivity issues in the future.

Using plug & play

The easiest way to find the IP address of an instrument or a number of instruments is via plug & play. It should work under the following circumstances.

- The instrument firmware is at least at version 0.7.0.
- You are directly connected to the instrument(s) via Ethernet, or there are only network switches in between. It will NOT work if there is a router or VPN connection in between. Follow steps 1-3 of *Connecting to a single module* if possible. The IP address you configure for your Ethernet connection should not matter, as long as *some* IPv4 address is configured.

Now it should be as simple as running the following command in a terminal of your choice:

```
$ qblox-pnp list
```

That should give you something like this:

```
Devices:
 - 192.168.0.3: Pulsar QRM 0.7.0 with name "pulsar-qrm" and serial␣
↪number 00013_2120_003
```

Using nmap

`qblox-pnp` won't be able to find your instruments across complex networks. For example, it won't work through a VPN. In this case, the nmap tool can help, although you will need to know which subnet your instruments are on (by default `192.168.0.X`) to use it.

Note: nmap will generally find all network devices on a network; it is not restricted to only Qblox instruments! If possible, disconnect all other devices that share the same network, such that only the host PC and the device with the an unknown IP are on the same subnetwork.

To use nmap, follow the instructions for your operating system.

MacOS/Linux

Open a terminal of your choice and run:

```
$ sudo nmap -sn 192.168.0.*
```

The output should look similar to:

```
Starting Nmap 7.91 ( https://nmap.org ) at 2021-01-02 20:16 CET
Nmap scan report for 192.168.0.3
Host is up (0.00029s latency).
MAC Address: 04:91:62:BF:DE:57 (Microchip Technology)
Nmap scan report for 192.168.0.200
Host is up.
Nmap done: 256 IP addresses (2 hosts up) scanned in 13.05 seconds
```

In this case our device has the IP `192.168.0.3` while the network adapter of our host PC has been configured at `192.168.0.200`.

Windows

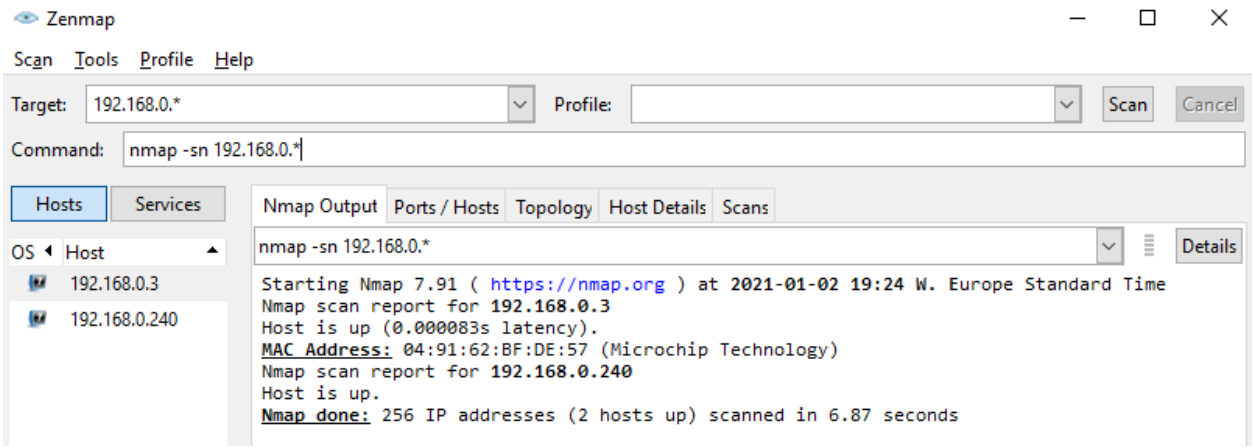
On Windows we recommend installing the latest version of *nmap* that comes with a graphical interface (Zenmap).

(a) Visit download section at nmap.org and download the *Latest stable release self-installer* under the *Microsoft Windows binaries* section.

(b) Run the installer with default options.

(c) Execute **as administrator** the *Nmap - Zenmap GIU* that should appear on your Desktop (or in the start menu)

(d) Type in the *Command* field `nmap -sn 192.168.0.*` and hit *Enter* on your keyboard. After a few seconds the output should look similar to:



In this case our device has the IP `192.168.0.3` while the network adapter of our host PC has been configured at `192.168.0.240`.

1.3.2 Connecting to multiple modules

To be able to control multiple modules (e.g. a Pulsar QCM and QRM) we need to follow the steps described in *Connecting to a single module* except, now:

- Instead of connecting a module directly to the Ethernet adapter of the host PC, we will connect all the modules and the host PC to the same network using, for example, an Ethernet switch.
- The IP address of the modules **must** be changed to avoid IP collisions. See section *Updating* for further instructions on updating the IP address of the modules.

As an example, we will consider a setup composed of:

- A laptop (host PC) with a USB Ethernet adapter.
- A Pulsar QCM.
- A Pulsar QRM.
- A network switch.

The following figure shows the example setup:

The following python code lets us connect to the modules in the example setup:

```
# Import drivers
from qblox_instruments import Pulsar

# Connect to modules
pulsar_qcm = Pulsar("qcm", "192.168.0.2")  # This module uses the default IP␣
↪address.
pulsar_qrm = Pulsar("qrm", "192.168.0.3")  # This module's IP address was␣
↪changed.
```

Note: When using multiple modules in your setup, you might need to synchronize the in- and outputs of the various modules. See section *Synchronization* the learn how to do this.

1.4 Connecting an SPI Rack

In this section we will explain how to setup a Qblox SPI Rack and connect it to your PC. Please make sure that you have the Qblox instruments package installed before proceeding (see section *Installation*) and that your PC has an available USB port.

1.4.1 Connecting to power

The recommended way to connect the SPI Rack to power is by connecting it via a gyrator to mains in parallel to a battery. This will ensure the system remains powered and stable, the gyrator prevents interference signals such as 50 Hz from getting to the SPI Rack by mimicking a large (~40H) inductor.

Note: Please ensure there is a minimum of 1 meter between the gyrator and its power supply in order to effectively prevent interference signals from getting to the SPI Rack. Also ensure that the SPI Rack is not mounted in the same 19 inch rack with line-powered equipment, this includes the gyrator power supply.

1.4.2 Connecting to PC

First, make sure a C1b controller module is fully inserted in the left-most slot of the SPI Rack. Then, connect the C1b to a C2 SPI Isolator box via a display port cable. Now, connect the C2 SPI Isolator box via USB to your PC. Finally, make sure to power switch on the back of the SPI Rack is in the *on* position, and the power supply is switched on as well. The LED on the C1b module should now turn on.

We can now connect to the SPI Rack via the QCoDeS driver provided through the *qblox-instruments* package (see section *Installation*).

Communicating with the SPI Rack

Starting the communication with the SPI Rack is done by simply instantiating an instance of the driver:

```
from qblox_instruments import SpiRack
spi = SpiRack('SPI Rack', 'COM4')
```

SPI Modules

To use an SPI Rack module, simply slide the module into one of the slots of the rack and fasten the screws. You should feel them “click” into place. The address on which to connect to the module is written on a sticker on the side of the module itself. Each of the modules we intend to use, we should add to the driver. This is done simply by:

```
spi.add_spi_module(2, "S4g") # example for an S4g on address 2
```

For more information on how to use the driver please visit the tutorial section.

1.5 Updating

In this section we will explain how to update the firmware and IP address of your module.

First download the latest firmware for the Qblox Pulsar QCM, Pulsar QRM and/or Cluster. The firmware can then be installed using the Qblox Configuration Manager tool, shipped with the Qblox instruments Python package (make sure it's *installed*).

Note: We used to ship the configuration manager with each firmware release, rather than with Qblox instruments. This is, however, an older version of the tool, that will not be able to communicate with instruments running newer firmware versions. The configuration manager shipped with Qblox instruments should however be able to communicate with all firmware versions. If you nonetheless run into problems with it (in particular when downgrading), you may have to update Qblox instruments first:

```
$ pip install --upgrade qblox-instruments
```

Once you've updated or downgraded all your instruments, you may have to install a different version of Qblox instruments again, in order for it to be compatible with the firmware version you installed.

When ready, you can update the firmware of your instruments as follows using a terminal of your choice, replacing `<filename>` with the file you downloaded:

```
$ qblox-cfg 192.168.0.2 update <filename>
```

Note: The default IP address of the module is `192.168.0.2`. Replace the IP address of any following instruction accordingly if the module's IP address was ever changed.

You can also use the configuration manager to update the IP address of your instrument:

```
$ qblox-cfg 192.168.0.2 set-ip <new-ip-address>
```

Tip: The instruments support more than just static IPv4 addresses: you can also configure them to use DHCP, and/or you can add a static IPv6 address if needed. Run `qblox-cfg --help` for more information.

Or to change its *name*:

```
$ qblox-cfg 192.168.0.2 set-name <new-name>
```

You can also perform several operations at once, if you want. For example, this will set the IP address and update the instrument (originally) at 192.168.0.2:

```
$ qblox-cfg 192.168.0.2 set-ip <new-ip-address> update <filename>
```

After executing one of the commands above, follow the instructions given by the Qblox Configuration Manager. The module will reboot, after which the update is complete.

During reboot, the status (S) LED will turn yellow, and the other LEDs will turn off (if you're updating from an old firmware version, the LEDs may also turn purple/red). This indicates that the instrument is rebooting. When the LEDs turn green/blue again, the reboot is finished.

If the module does not finish rebooting within 1-2 minutes, please remove power from the module and wait one minute before powering it on again. This might be required up to two times: once to start the internal update process, and a second time to start using the updated firmware.

When done, please make sure your installed Qblox instruments package is compatible with the installed firmware. See Qblox instruments PyPI and section *Installation*.

Tip: If you're connected to your instrument(s) directly or only via network switches (NOT via VPN or a router), you can also use the instrument name or serial number instead of the IP address to select an instrument. Or you can update all instruments at once:

```
$ qblox-cfg -k all update <filename>
```

Note that this will only update the instruments that the update file you specify is compatible with. The tool will report that it failed if one of the instruments it found wasn't compatible, but (due to the `-k` flag, short for "keep going") it will try to apply the updates to all instruments, instead of stopping after immediately after trying to update an incompatible instrument.

Tip: By default, the tool will ask you to confirm that you really want to update after printing a summary of the operations it will be performing. You can get more information by adding `-v` or `-vv` to the command line, or by reading the log file. Alternatively, if you want to automate `qblox-cfg`, you may want to disable the prompt (a.k.a. non-interactive mode), which you can do with `-y`.

Tip: In general, the Qblox Configuration Manager tool can do much more than what's listed here. Run `qblox-cfg --help` for more information.

1.6 Addressing

The most robust way to address any instrument connected via Ethernet, is to configure it using a static IP address and then use that address directly. As we've seen in earlier sections, Pulsars ship with IP address 192.168.0.2 by default, expecting to be in the 192.168.0.0/24 subnet. However, having to remember IP addresses is not very user-friendly, and if the instruments are to be connected through an existing network, static IP addresses may not even be an option: most networks use DHCP to automatically configure the IP addresses of the endpoints connected to it. In the latter case, ask your system administrator to be sure.

If you have to use DHCP, or just don't want to have to hardcode IP addresses into your notebooks, you have two options, but both put some requirements on your network infrastructure:

- *Addressing instruments by their name or serial number using Qblox plug & play.* This will only work when you are on the same local area network (LAN) as the instruments; it will *not* work if there is a router or VPN in between.
- *Addressing instruments by their hostname via (local) DNS.* This requires DHCP and a router with local DNS support. Note that most consumer-grade routers *don't* support this, but enterprise hardware usually does.

These four types of addresses (IP address, name, serial number, and DNS hostname) are collectively referred to as device identifiers. They are internally resolved via the `resolve()` function; refer to its API documentation for the complete syntax supported by it. Both `qblox-cfg` and the Pulsar & Cluster instrument drivers in Python support all types of identifiers.

1.6.1 Listing all instruments on a network

You can easily determine which instruments are present on a network and what their IP addresses, names, and serial numbers are:

```
$ qblox-pnp list
Devices:
 - 192.168.0.3: Pulsar QRM 0.7.0 with name "test" and serial number 00013_2120_
↪003
```

Note however that, since this uses Qblox Plug & Play, it will only work if you're connected to the instrument(s) directly or only via network switches.

1.6.2 Verifying which instrument you're connected to

Once you have a connection, you might want to ensure that you're actually connected to the right instrument. You can do this with the `identify()` method on an instrument connection object: calling this will make the module blink its front-panel LEDs for a few seconds. You can do the same from the command line using `qblox-pnp identify <name-or-serial-number>`. Alternatively, if you're not physically near the instrument but you do know what its name is supposed to be, you can use the `get_name()` method to retrieve it from an open connection.

1.6.3 Renaming instruments

By default, Pulsar QRMs come programmed with the name and hostname pulsar-qrm, and Pulsar QCMs come programmed with pulsar-qcm. You can change this name using `set_name()` in Python, or using the `set-name` subcommand of `qblox-cfg` (see *Updating*). Instrument names may be any single line of characters not including backslashes or underscores.

Warning: Instrument names are case sensitive, so be aware that an instrument named `A` is different from a instrument named `a`.

Hostnames have more stringent requirements. They must be lowercase, at most 63 characters long, and besides letters and numbers only the `-` (dash) is allowed. If you want to use local DNS, make sure that you configure a name that complies with these rules to avoid confusion. If you configure a name that isn't a valid hostname, the instrument will change its hostname to a best-effort representation of the invalid name.

Note: Name changes go into effect immediately for resolution via Qblox plug & play, but the hostname will only change after you reboot the instrument.

1.7 What's next

1.7.1 Quantify

To get the most out of your Qblox instruments, we advise that you install Quantify; our measurement control package with built-in support for Qblox instruments. You can do this by following the installation guide.

Please consult Quantify's documentation for more information and tutorials.

1.7.2 Learn more

If you want to learn more about our instruments, we advise that you read on. The following pages consist of documentation and tutorials for both beginning and advanced users. The pages will go into more detail about the internal operations of the instruments and will teach you how to operate our instruments effectively and efficiently.

1.8 General

In this section we explain general concepts of the Qblox instruments, like general instrument control and status.

1.8.1 Control

All Qblox instruments, excluding the SPI Rack, are controlled over Ethernet using a Python driver based on QCoDeS. We recommend using this driver as it provides easy and clear access to all functionality of the instrument; even if you use a different lab-framework as the overhead of QCoDeS is minimal.

Underneath the QCoDeS driver layer, the control software is built upon the SCPI standard, as also reflected by the *API reference*. This means that all communication with the instrument happens using the master/slave paradigm, where the host PC is the master and **always** responsible for initiating communication by issuing SCPI commands to the instrument. Of course, all of this is abstracted away at the driver level, so you don't have to have in-depth knowledge of the standard. However, if you are familiar with it, you will have access to all the default SCPI functionality that you are used to, like `get_idn()` (**IDN?*), `reset()` (**RST*) and `clear()` (**CLS*), albeit with a slightly more readable name.

Reset

We advise resetting the instrument before executing any experiment to get the instrument into a well-defined state, thereby improving reproducibility of the experiment. Resetting the instrument is easily achieved by calling `reset()`. This will reset the instrument status and configuration to the default values. It will reset all SCPI registers, including any reported error. It will also clear all stored Q1ASM programs, waveforms and acquisitions.

There are many use cases where you want to store the instrument's settings before resetting, for instance to be able to easily reproduce an experiment. For this, we advise to use the snapshot feature of QCoDeS.

Errors

Instrument errors are reported using SCPI's system error registers, which can be read using `get_num_system_error()` (*SYSTem:ERRor:COUNt?*) and `get_system_error()` (*SYSTem:ERRor:NEXT?*). However, like mentioned before, this is all abstracted away at the driver level. This means that the errors are automatically read and reported to you using exceptions. Any driver function can throw these exceptions and you need to make sure these are handled appropriately, for instance by using try statements.

1.8.2 Clocking

The instruments need a 10 MHz reference clock to operate. It is used to derive clocks for the instrument's internal logic and data converters. Each instrument can either generate this 10 MHz reference clock internally or it can be generated externally and provided through the REF^{in} SMA connector (10 MHz, 1 Vpp nominal @ 50) [see section *Overview*]. Using the external reference source can be useful for synchronizing the instrument with other instruments in you setup, Qblox's or others. The `reference_source()` parameter can be used to select which reference clock is used to clock the instrument. We recommend to set the reference right after resetting the instrument with `reset()`. Whichever reference clock is selected, the reference clock is also output using the REF^{out} SMA connector (10 MHz, 0-3.3 V @ 50). This output can be used as reference clock for other instruments. Be aware that the input and output reference clocks are purposely not phase aligned to aid synchronization of Qblox instruments (see section *Synchronization*).

1.8.3 Status

The status of the instrument conveys the general operational condition of the instrument. This is derived from multiple internal components, like PLLs and temperature sensors. The instrument's status is updated every millisecond and stored in the standard SCPI registers. It can be queried through these registers [e.g. through `get_status_byte()` (**STB?*)], but a more convenient way of reading out the general instrument status is calling `get_system_state()`. This returns the following status and accompanying flags that elaborate on the status:

- **Status**:
  - `BOOTING`: Instrument is booting.
  - `OKAY`: Instrument is operational.
  - `CRITICAL`: Instrument has encountered an error (see flags below), but it has been corrected.
  - `ERROR`: Instrument has encountered an error (see flags below), which needs to be fixed urgently.
- **Flags**:
  - `CARRIER PLL UNLOCKED`: No reference clock found.
  - `FPGA PLL UNLOCKED`: No reference clock found.
  - `LO PLL UNLOCKED`: No reference clock found (only for RF modules).
  - `FPGA TEMPERATURE OUT-OF-RANGE`: FPGA temperature has surpassed 80°C.
  - `CARRIER TEMPERATURE OUT-OF-RANGE`: Carrier board temperature has surpassed 100°C.
  - `AFE TEMPERATURE OUT-OF-RANGE`: Analog frontend board temperature has surpassed 100°C.
  - `LO TEMPERATURE OUT-OF-RANGE`: Local oscillator board temperature has surpassed 100°C.
  - `BACKPLANE TEMPERATURE OUT-OF-RANGE`: Backplane board temperature has surpassed 100°C (only for Cluster).

The instrument status is persistent through the state *critical*, so a way to reset it is required. This can be simply done by calling the `clear()` to clear the state or by completely resetting the instrument by calling `reset()`.

Frontpanel LEDs

The LEDs on the frontpanel of the Qblox instruments are used as a visual indication of the status of the instrument. The LED colors indicate the following status:

S	**White**	Okay and idle (no connections).
	Green	Okay.
	Yellow	Booting (other LEDs are off).
	Orange	Critical.
	Red	Error.
R	**Green**	External reference clock selected.
	Blue	Internal reference clock selected
	Red	No reference clock found.
I/O	**Green**	Channel idle.
	Purple	Sequencer connected to channel is armed.
	Blue	Sequencer connected to channel is running.
	Red	Sequencer connected to channel failed.
	Orange	Output values are clipping.

It's also possible for all LEDs to show a combination of red and purple. On older firmware versions, this happens naturally while the device is rebooting, but otherwise, you've stumbled onto a firmware bug! If you're not already using the latest firmware, please *update* your device. If the problem persists, please contact us.

1.9 Pulsar

In this section we will discuss the architecture of the Pulsar instruments and their rich feature set.

1.9.1 QCM Overview

The Qubit Control Module is an instrument completely dedicated to qubit control using parametrized pulses. The pulses are stored as waveform envelopes in memory and can be parametrized by changing gain and offset and additionally phase if also modulated. This parametrization is controlled by the AWG paths of the sequencer, which each have two waveform paths (from here on referred to as path 0 and 1). Using parametrization, the output of these paths can either be independent signals or modulated IQ signals. The two paths of each sequencer can, in turn, be connected to any output pair of the instrument (i.e. $O^{1/2}$ and $O^{3/4}$) to control one or more qubits per output. Additionally, the sequencers also control four marker output channels.

The figure below shows the architecture of the Pulsar QCM. Please see the *Features* section for more information on the numbered features in the figure.

Note: This figure is outdated and will be updated soon.

1.9.2 QRM Overview

The Qubit Readout Module is an instrument that targets qubit readout. To accomplish this, it employs a similar architecture to the QCM: besides two outputs ($O^{1/2}$), the module has an acquisition path that operates on two inputs (i.e. $I^{1/2}$) using two processing paths (from here on also referred to as path 0 and 1). Using parametrization, each sequencer can target one qubit for readout, allowing multiplexed readout of qubits on the same channel. The AWG paths can generate the readout pulses and the acquisition paths can process the returned readout data. The acquisition path supports three acquisition modes:

- *Scope*: Returns the raw input data.
- *Integration*: Returns the result after integrating the input data; optionally based on an integration function stored in memory.
- *Thresholded*: Returns the binary qubit value after thresholding the integrated value.

The results of the acquisitions are returned to the user.

The figure below shows the architecture of the Pulsar QRM. Please see the *Features* section for more information on the numbered features in the figure.

Note: This figure is outdated and will be updated soon.

1.9.3 Features

1. SYNQ & trigger

The Qblox SYNQ technology and trigger input enable simple and quick synchronization over multiple instruments. See section *Synchronization* for more information.

2. Sequencer

The sequencers are the heart(s) of the Pulsar instruments. They orchestrate the experiment using a custom low-latency sequence processor specifically designed for quantum experiments. Furthermore, they each achieve that by controlling a dedicated AWG path and, in case of a Pulsar QRM, acquisition path, which enables parametrized pulse generation and readout. Each instrument has a 6 of these sequencers to target multiple qubits with one instrument. See section *Sequencer* for more information on how to program and control them.

3. Gain

Each sequencer has a dedicated gain step for both path 0 and 1, which can be statically configured using the `sequencer#.gain_awg_path#()` parameters. However, the gain can also be dynamically controlled using the *set_awg_gain* instruction of the sequence processor which enables pulse parametrization (see section *Instructions*). The static and dynamic gain controls are complementary.

Note: If modulated IQ signals are used for an output pair, the gain `sequencer#.gain_awg_path#()` has to be the same for both paths.

4. Offset

Each sequencer has a dedicated offset step for both path 0 and 1, which can be statically configured using the `sequencer#.offs_awg_path#()` parameters. However, the offset can also be dynamically controlled using the *set_awg_offs* instruction of the sequence processor which enables pulse parametrization. (see section *Instructions*). The static and dynamic offset controls are complementary.

Note: This offset is applied to the signals before the mixer and cannot be used for DC offset correction if the mixer is enabled.

---

5. NCO & IQ mixer

Each sequencer has a dedicated numerically controlled oscillator and IQ mixer. The NCO can be used to track the qubit phase (at a fixed frequency) and the IQ mixer can be used to modulate the output.

The frequency of the NCO and phase can be statically controlled using the `sequencer#.nco_freq()` and `sequencer#.nco_phase_offs()` parameters. However, the phase of the NCO can also be dynamically controlled using the *set_freq*, *reset_ph*, *set_ph* and *set_ph_delta* instructions of the sequence processor, which enables pulse parametrization and execution of virtual Z-gates (see section *Instructions*). The static and dynamic phase control is complementary. The modulation is enabled using the `sequencer#.mod_en_awg parameter()`. The demodulation is enabled using the `sequencer#.demod_en_acq parameter()`.

6. Sequencer multiplexer

A multiplexer that allows any sequencer to be connected to any output pair. Multiple sequencers can also be connected to a single output pair. This, in combination with the dedicated NCO and IQ mixer per sequencer, enables easy and flexible targeting of multiple qubits on a single channel. The multiplexer can be statically configured through the `sequencer#.channel_map_path#_out#_en()` parameters.

Note: The output of each sequencer is complementary. Be aware of potential output clipping when connecting multiple sequencers to a single output.

---

7. Mixer correction

The mixer correction is used to correct imperfections in an external mixer used for up or down conversion. Every sequencer can apply mixer corrections to the phase and gain between path 0 and 1 to correct for frequency dependant phase and/or gain imbalance in the external mixer. On top of that, a DC offset can be applied to each output to correct for frequency independant offset imperfections in the external mixer. The mixer correction is statically configured using the `sequencer#.mixer_corr_phase_offset_degree()`, `sequencer#.mixer_corr_gain_ratio()` and `out#_offset()` parameters.

8. High-speed data converters

The Pulsar instruments use state-of-art 1Gbps 16-bit DACs and 1Gbps 12-bit ADCs. The dynamic output range of the Pulsar QRM and QCM's DACs are 5 Vpp and 1 Vpp respectively and 50 Ω terminated. The maximum input range of the Pulsar QRM's ADCs is 2 Vpp and 50 Ω terminated.

9. Marker output channels

Each sequencer has control over the four marker output channels, with the control of each sequencer being OR'ed to create the final marker outputs. The markers can be dynamically controlled with the *set_mrk* instruction of the sequence processor (see section *Instructions*), but can also be overwritten with the static marker overwrite parameters `sequencer#.marker_ovr_en()` and `sequencer#.marker_ovr_value()`. The marker output range is 0-3.3 V TTL. In RF modules *set_mrk* is also used to toggle the switches before the outputs/inputs to enable the respective output/input.

10. Input gain

Dedicated amplifiers to provide additional gain to the input signals. The gain can vary between -6dB and 26dB and can be set using the `in#_amp_gain()` parameters.

1.10 Sequencer

This section will explain how the sequencers of the Pulsar QCM and QRM are controlled. Every sequencer is controlled using the same functions and parameters, which either take the sequencer index as a parameter or indicate which sequencer they operate on based on the index in their name.

Note: As of version 0.5.0 of the Pulsar QRM, new functionality has been added to the acquisition path (e.g. real-time demodulation, (weighed) integration, discretization, averaging, binning). More details about this functionality will be added to the documentation as soon as possible. For now, please have a look at the binned acquisition tutorial to get started.

1.10.1 Overview

The sequencers are split into the sequence processor, AWG and acquisition paths as shown in the figures below. Each sequence processor controls one AWG path and, in case of the Pulsar QRM, one acquisition path. The AWG path and acquisition path are discussed in more detail in section *Pulsar*. Each sequencer processor is, in turn, split into a classical and real-time pipeline. The classical pipeline is responsible for any classical instructions related to program flow or arithmetic and the real-time pipeline is responsible for real-time instructions that are used to create the experiment timeline.

Fig. 1: Pulsar QCM sequencer with AWG path.

Fig. 2: Pulsar QRM sequencer with AWG and acquisition paths.

The sequencers are started and stopped by calling the `arm_sequencer()`, `start_sequencer()` and `stop_sequencer()` functions. Once started they will execute the sequence described in the next section.

1.10.2 Sequence

The sequencers are programmed with a sequence using the `sequencer#.sequence()` function parameter. This parameter expects a sequence in the form of a JSON compatible file that contains the waveform, weight, acquistion and program information. The JSON file is expected to adhere to the following format:

- **waveforms**: Indicates that the following waveforms are intended for the AWG path.
  - **waveform name**: Replace by string containing the waveform name.
    - **data**: List of floating point values to express the waveform.
    - **index**: Integer index used by the Q1ASM program to refer to the waveform.
- **weights**: Indicates that the following weight functions are intended for the integration units of the acquisition path (only used by the Pulsar QRM).
  - **weight name**: Replace by string containing the weight name.
    - **data**: List of floating point values to express the weight.
    - **index**: Integer index used by the Q1ASM program to refer to the weight.
- **acquisitions**: Indicates that the following acquisitions are available for the acquisition path to refer to (only used by the Pulsar QRM).
  - **acquisition name**: Replace by string containing the acquisition name.
    - **num_bins**: Number of bins in acquisition.
    - **index**: Integer index used by the Q1ASM program to refer to the acquisition.
- **program**: Single string containing the entire sequence processor Q1ASM program.

Example of a sequence JSON file.

```
{
    "waveforms": {
        "gaussian": {
            "data": [
                0.007575676774442599355, 0.5812730178734145, 0.
↪5812730178734145, 0.007575676774442599355
            ],
            "index": 0
        },
        "sine": {
            "data": [0.0, 1.0, 1.2246467991473532e-16, -1.0],
            "index": 1
        }
    },
    "weights": {
        "gaussian": {
            "data": [0.007575676774442599355, 0.5812730178734145, 0.
↪5812730178734145, 0.007575676774442599355],
            "index": 0
        },
        "sine": {
            "data": [0.0, 1.0, 1.2246467991473532e-16, -1.0],
            "index": 1
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "acquisitions": {
        "binned": {
            "num_bins": 100000,
            "index": 0
        },
        "averaged": {
            "num_bins": 1,
            "index": 1
        }
    },
    "program": "\nplay 0,1,4 #Play waveforms and wait 4ns.\nacquire 1,
↪0,16380 #Acquire wait for scope mode acquisition to finish.\nstop
↪#Stop.\n"
}
```

Program

The sequence programs are written in the custom Q1ASM assembly language described in the following sections. All sequence processor instructions are executed by the classical pipeline and the real-time instructions are also executed by the real-time pipeline. These latter instructions are intended to control the AWG and acquisition paths in a real-time fashion. Once processed by the classical pipeline they are queued in the real-time pipeline awaiting further execution. A total of 32 instructions can be queued and once the queue is full, the classical part will stall on any further real-time instructions.

Once execution of the real-time instructions by the real-time pipeline is started, care must be taken to not cause an underrun of the queue. An underrun will potentially cause undetermined real-time behaviour and desynchronize any synchronized sequencers. Therefore, when this is detected, the sequencer is completely stopped. A likely cause of underruns is a loop with a very short (i.e. < 24ns) real-time run-time, since the jump of a loop takes some cycles to be execute by the classical pipeline.

Finally, be aware that moving data into a register using an instruction takes a cycle to complete. This means that when an instruction reads from a register that the previous instruction has written to, a *nop* instruction must be placed in between these consecutive instructions for the value to be correctly read.

The state of the sequencers, including any errors, can be queried through `get_sequencer_state()`.

Instructions

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
Control						

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
illegal	–	–	–	–	–	Instruction that should not be executed. If it is executed, the sequencer will stop with the illegal instruction flag set.
stop	–	–	–	–	–	Instruction that stops the sequencer.
nop	–	–	–	–	–	No operation instruction, that does nothing. It is used to pass a single cycle in the classic part of the sequencer without any operations.
Jumps						

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
jmp	Immediate, Register, Label	–	–	–	–	Jump to the next instruction indicated by *argument 0*.
jge	Register	Immediate	Immediate, Register, Label	–	–	If *argument 0* is greater or equal to *argument 1*, jump to the instruction indicated by *argument 2*.
jlt	Register	Immediate	Immediate, Register, Label	–	–	If *argument 0* is less than *argument 1*, jump to the instruction indicated by *argument 2*.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
loop	Register	Immediate, Register, Label	–	–	–	Subtract *argument 0* by one and jump to the instruction indicated by *argument 1* until *argument 0* reaches zero.
Arithmetic						
move	Immediate, Register	Register	–	–	–	*Argument 0* is moved / copied to *argument 1*.
not	Immediate, Register	Register	–	–	–	Bit-wise invert *argument 0* and move the result to *argument 1*.
add	Register	Immediate, Register	Register	–	–	Add *argument 1* to *argument 0* and move the result to *argument 2*.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
sub	Register	Immediate, Register	Register	–	–	Subtract *argument 1* from *argument 0* and move the result to *argument 2*.
and	Register	Immediate, Register	Register	–	–	Bit-wise AND *argument 0* and *argument 1* and move the result to *argument 2*.
or	Register	Immediate, Register	Register	–	–	Bit-wise OR *argument 0* and *argument 1* and move the result to *argument 2*.
xor	Register	Immediate, Register	Register	–	–	Bit-wise XOR *argument 0* and *argument 1* and move the result to *argument 2*.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
asl	Register	Immediate, Register	Register	–	–	Bit-wise left-shift *argument 0* by *argument 1* number of bits and move the result to *argument 2*.
asr	Register	Immediate, Register	Register	–	–	Bit-wise right-shift *argument 0* by *argument 1* number of bits and move the result to *argument 2*.
Software request						
sw_req	Immediate, Register	–	–	–	–	Generate software request interrupt with *argument 0* value being passed as interrupt argument (currently not implemented).
Real-time pipeline instructions						

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
set_mrk	Immediate, Register	–	–	–	–	Set marker output channels to *argument 0* (bits 0-3), where the bit index corresponds to the channel index for baseband modules. For QCM-RF module, bit indices 0 & 1 correspond to output enable 1 and 2 respectively; indices 2 & 3 correspond to marker outputs 2 and 1 respectively. For QRM-RF module, bit indices 0 & 1 correspond to input 1 and output 1 switches respectively; indices 2 & 3 correspond to marker outputs 1 and 2 respectively. The

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
set_freq	Immediate, Register	–	–	–	–	Set the frequency of the NCO used by the AWG and acquisition using *argument 0*. The frequency is divided into 4e9 steps between -500 and 500 MHz and expressed as an integer between -2e9 and 2e9. (e.g. 1 MHz=4e6). The frequency parameter is cached and only applied when the *upd_param*, *play*, *acquire* or *acquired_weighed* instructions are executed.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
reset_ph	–	–	–	–	–	Reset the absolute phase of the NCO used by the AWG and acquisition to 0°. This also resets any relative phase offsets that were already statically or dynamically set. The reset is cached and only applied when the *upd_param*, *play*, *acquire* or *acquired_weighed* instructions are executed.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
set_ph	Immediate, Register	–	–	–	–	Set the relative phase of the NCO used by the AWG and acquisition using *argument 0*. The phase is divided into 1e9 steps between 0° and 360°, expressed as an integer between 0 and 1e9 (e.g 45°=125e6). The phase parameter is cached and only updated when the *upd_param*, play`, *acquire* or *acquired_weighed* instructions are executed.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
set_ph_delta	Immediate, Register	–	–	–	–	Set an offset on top of the relative phase of the NCO used by the AWG and acquisition. The offset is applied on top of the phase set using *set_ph*. See *set_ph* for more details regarding the argument. The phase parameter is cached and only updated when the *upd_param*, *play*, *acquire* or *acquired_weighed* instructions are executed.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
set_awg_gain	Immediate, Register	Immediate, Register	–	–	–	Set AWG gain for path 0 using *argument 0* and path 1 using *argument 1*. Both gain values are divided in 2**sample path width steps. The parameters are cached and only updated when the *upd_param*, *play*, *acquire* or *acquired_weighed* instructions are executed. The arguments are either all set through immediates or registers.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
set_awg_offs	Immediate, Register	Immediate, Register	–	–	–	Set AWG gain for path 0 using *argument 0* and path 1 using *argument 1*. Both offset values are divided in 2**sample path width steps. The parameters are cached and only updated when the *upd_param*, *play*, *acquire* or *acquired_weighed* instructions are executed. The arguments are either all set through immediates or registers.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
upd_param	Immediate	–	–	–	–	Update the marker, phase, phase offset, gain and offset parameters set using their respective instructions and then wait for *argument 0* number of nanoseconds.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
play	Immediate, Register	Immediate, Register	Immediate	–	–	Update the marker, phase, phase offset, gain and offset parameters set using their respective instructions, start playing AWG waveforms stored at indexes *argument 0* on path 0 and *argument 1* on path 1 and finally wait for *argument 2* number of nanoseconds. The arguments are either all set through immediates or registers.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
acquire	Immediate	Immediate, Register	Immediate	–	–	Update the marker, phase, phase offset, gain and offset parameters set using their respective instruction, start the acquisition referred to using index *argument 0* and store the bin data in bin index *argument 1*, finally wait for *argument 2* number of nanoseconds. Integration is executed using a square weight with a preset length through the associated QCoDeS parameter. The arguments are either all set through immediates or registers.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
acquire_weighed	Immediate	Immediate, Register	Immediate, Register	Immediate, Register	Immediate	Update the marker, phase, phase offset, gain and offset parameters set using their respective instruction, start the acquisition referred to using index *argument 0* and store the bin data in bin index *argument 1*, finally wait for *argument 4* number of nanoseconds. Integration is executed using weights stored at indexes *argument 2* for path 0 and *argument 3* for path 1. The arguments are either all set through immediates or registers.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
acquire_ttl	Immediate	Immediate, Register	Immediate	Immediate	–	Update the marker, phase, phase offset, gain and offset parameters set using their respective instruction, start the TTL trigger acquisition referred to using index *argument 0* and store the bin data in bin index *argument 1*, enable the acquisition by writing 1 to *argument 2*, finally wait for *argument 3* number of nanoseconds. The TTL trigger acquisition has to be actively disabled afterwards by writing 0 to *argument 2*.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Argument 3	Argument 4	Description
wait	Immediate, Register	–	–	–	–	Wait for *argument 0* number of nanoseconds.
wait_trigger	Immediate, Register	–	–	–	–	Wait for external trigger and then wait for *argument 0* number of nanoseconds.
wait_sync	Immediate, Register	–	–	–	–	Wait for SYNQ to complete on all connected sequencers over all connected instruments and then wait for *argument 0* number of nanoseconds.

Note: The duration argument for *upd_param*, *play*, *acquire*, *acquire_weighed*, *wait*, *wait_trigger* and *wait_sync* needs to a be multiple of 4ns. This will be reduced to 1ns in the future.

Arguments

Arguments	Format	Description
Immediate	#	32-bit decimal value (e.g. `1000`)
Register	R#	Register address in range 0 to 63 (e.g. `R0`) pointing to a 32-bit unsigned integer
Label	@label	Label name string (e.g. `@main`)

Labels

Any instruction can be preceded by a label. This label can be used as a reference to that specific instruction. In other words, it can be used as a goto-point by any instruction that can alter program flow (i.e. *jmp*, *jge*, *jlt* and *loop*). The label must be followed by a ':' character and a whitespace before the actual referenced instruction.

Example

This is a simple example of a Q1ASM program. It enables each marker channel output for 1s and then stops.

```
      move      1,R0        # Start at marker output channel 0 (move 1 into R0)
      nop                   # Wait a cycle for R0 to be available.

loop: set_mrk   R0          # Set marker output channels to R0
      upd_param 1000        # Update marker output channels and wait 1s.
      asl       R0,1,R0     # Move to next marker output channel (left-shift␣
↪R0).
      nop                   # Wait a cycle for R0 to be available.
      jlt       R0,16,@loop # Loop until all 4 marker output channels have␣
↪been set once.

      set_mrk   0           # Reset marker output channels.
      upd_param 4           # Update marker output channels.
      stop                  # Stop sequencer.
```

Waveforms

The waveforms are expressed as a list of floating point values in the range of 1.0 to -1.0 with a resolution of one nanosecond per sample. The AWG path uses these waveforms to parametrically generate pulses on its outputs.

Waveform playback is started by the *play* instructions. Each waveform is paired with an index, which is used by this instruction to refer to the associated waveform. The waveform is then completely played irrespective of further sequence processor instructions, except when the sequence processor issues the playback of another waveform, in which case the waveform will be stopped and the new waveform will start. When waveforms are not played back-to-back, the intermediate time will be filled by samples with a value of zero.

The programmed waveforms can be retrieved using `get_waveforms()`.

Weights

The weights are expressed as a list of floating point values in the range of 1.0 to -1.0 with a resolution of one nanosecond per sample. The integration units in the acquisition path apply (i.e. multiply) these weights during the integration process when the acquisition path is triggered for weighed integration.

Weighed integration is triggered by the *acquire_weighed* instruction. Each weight is paired with an index, which is used by this instruction to refer to the associated weight. The weight is then played, like the waveforms discussed in the previous section and determines the length of the integration. The weighed integration process continues irrespective of further sequence processor instructions, except when the sequence processor issues another acquisition using the *acquire* or *acquire_weighed* instructions, in which case the integration will be stopped, the result will be stored and a new integration will start.

The programmed weights can be retrieved using `get_weights()`.

Acquisitions

Acquisitions are started by the *acquire*, *acquire_weighed* or *acquire_ttl* instructions and will trigger the capture of 16k input samples on both inputs. This mode of operation is called *scope mode* and will store the raw input samples in a temporary buffer. Every time an acquisition is started, this temporary memory is overwritten, so it is vital to move the samples from the temporary buffer to a more lasting location before the start of the next acquisition. This is done by calling `store_scope_acquisition()`, which moves the samples into the specified acquisition in the acquisition list of the sequencer, located in the RAM of the instrument. Multiple acquisitions can be stored in this list before being retrieved from the instrument by simply calling `get_acquisitions()`. Acquisitions are returned as a dictionary of acquisitions. Scope mode data is located under the *scope* key as lists of floating point values in a range of 1.0 to -1.0 with a resolution of one nanosecond per sample, as well as an indication if the ADC was out-of-range during the measurement.

Note: Before calling `store_scope_acquisition()`, be sure to call `get_sequencer_state()` and `get_acquisition_state()` in that order. This ensures that both the sequencer has finished and that there is an acquisition ready.

The acquisition path also has an averaging function set through the `scope_acq_avg_mode_en_path#()` parameters. This enables the automatic accumulation of acquisitions, where sample *N* of acquisition *M* is automatically accumulated to sample *N* of acquisition *M+1*. This happens while the acquisition is still in the temporary buffer, so after the desired number of averaging acquisitions is completed, call `store_scope_acquisition()` to store the accumulated result in the acquisition list. Once retrieved from the instrument, the accumulated samples will automatically be divided by the number of averages to get the actual averaged acquisition result.

Tip: For debug purposes, the acquisition path can also be triggered using a trigger level, where if the input exceeds this level, an acquisition is started. See the `sequencer#.trigger_mode_acq_path#()` and `sequencer#.trigger_level_acq_path#()` parameters for more information.

TTL trigger acquisitions

The *acquire_ttl* instruction is used to detect TTL triggers on one of the instrument's inputs when it exceeds a configurable threshold. The input is selected using `sequencer#.ttl_acq_input_select()`, while the threshold is set using `sequencer#.ttl_acq_threshold()`. The instruction enables the TTL trigger acquisition path by setting the enable argument to 1 and needs to actively disable the path again by using the same instruction with its enable argument set to 0. While the TTL trigger acquisition path is enabled, real-time integration and discretization is disabled.

```
acquire_ttl 0,0,1,1000 #Acquire triggers and wait for 1000ns
acquire_ttl 0,0,0,4    #Stop acquiring triggers and wait for 4ns
stop                   #Stop sequencer
```

When a rising edge is detected on the input that exceeds the threshold while the path is enabled, a TTL trigger is detected and its ADC value at which it was detected is stored in the bin specified by the instruction's bin index argument. No new TTL triggers will be detected until a new rising edge exceeds the configured threshold.

When multiple triggers are detected while the path is enabled, the storage strategy is determined by the `sequencer#.ttl_acq_auto_bin_incr_en()` parameter. When `sequencer#.ttl_acq_auto_bin_incr_en()` is enabled, the bin index is automatically incremented everytime a TTL trigger is detected. This gives the ability to store every ADC value individually and allows you to count the number of TTL triggers by counting the number of valid bins stored at the end of the sequence. Alternatively, when `sequencer#.ttl_acq_auto_bin_incr_en()` is disabled the same bin is reused for every detected TTL trigger. This allows averaging the ADC values in hardware and allows you to count the number of TTL triggers by looking at the average count of that bin at the end of the sequence.

The resulting acquisition data can be retrieved using `get_acquisitions()`. The ADC values at which TTL triggers were detected are stored in the *integration* key as lists of floating point values in a range of 1.0 to -1.0. They replace the integration results of the regular acquisitions.

1.10.3 Continuous waveform mode

The sequencer also supports a continuous waveform mode of operation, where the waveform playback control of sequence processor is completely bypassed and a single waveform is just played back on a loop. This mode can be enabled using the `sequencer#.cont_mode_en_awg_path#()` parameter and the waveform can be selected using the `sequencer#.cont_mode_waveform_idx_awg_path#()` parameter. The waveforms used in this mode must be a multiple of four samples long (i.e. 4ns).

When in continuous mode, simply program, arm, start and stop the sequencer using the regular control functions and parameters (i.e. `sequencer#.sequence()`, `arm_sequencer()`, `start_sequencer()` and `stop_sequencer()`). However, be aware that the sequencer processor can still control parts of the AWG path, like phase, gain and offset, while the sequencer operates in this mode. Therefore, we advise to program the sequence processor with a single *stop* instruction.

Note: We realise that the current way of controlling this mode is not optimal, so in the near future we will be implementing additional driver support to streamline this mode.

1.11 Synchronization

In this section we explain how to synchronize multiple instruments in your setup including Qblox Pulsar series instruments. Synchronization is based on two aspects:

1. A shared reference clock, preferably phase aligned, so that all instruments use the same reference to base their operations on.
2. A synchronized start event, so that all instruments start their operations simultaneously.

The following subsections will go into more detail on how to achieve both aspects.

1.11.1 Reference clock

Like most instruments the Qblox instruments use a 10 MHz clock as a time reference. To synchronize multiple instruments in your setup you will need to connect such a reference clock to the REF^{in} SMA connector of the instruments (10 MHz, 1 Vpp nominal @ 50) (see section *Overview*) and set the `reference_source()` parameter to external. Connecting the reference can be done in two ways:

1. Through a clock distribution module that distributes a reference clock provided by a reference clock source to all instruments in the setup as shown in the figure below. Care has to be taken that all distributed reference clocks are length matched to keep the clocks phase aligned.

2. Through daisy-chaining the reference clock from one Qblox instrument to the next as shown in the figure below. The Qblox instruments have been configured so that when a 50 cm coaxial cable is used to connect the REF^{out} SMA connector of one instrument to the REF^{in} SMA connector of the next, the instrument's reference clocks are phase align to one another. This removes the need of an additional clock distribution module. The first instrument in the daisy-chain can either use an internal reference source or an external reference if you wish to connect additional non-Qblox instruments. All other Qblox instruments need to be configured to use external reference sources.

1.11.2 SYNQ

To synchronize the start event of the instruments, Qblox SYNQ technology can be used to greatly simplify the process. To use this SYNQ technology, the Qblox instruments need to be daisy-chained using the two SYNC ports (see section *Overview*) as shown in the figure below. Additionally, the `sequencer#.sync_en()` parameter needs to be set for every sequencer in the instrument participating in the experiment and these same sequencers need to execute the *wait_sync* instruction (see section *Instructions*). Note, these last two steps also need to done when only using a single Qblox instrument. The Qblox SYNQ technology will then automatically align the timing of every participating sequencer in all Qblox instruments to within 300 ps of one another.

Additionally, the marker output channels can be controlled by the sequencers to trigger other non-Qblox instruments, thereby synchronizing them with the Qblox instruments. However, care needs to be taken to compensate for any trigger delay caused by the connection or the triggered instrument itself.

1.11.3 Trigger

If desired, the Qblox instruments can also be triggered by other non-Qblox instruments. To achieve this, simply connect the trigger signal to the $TRIG^{in}$ SMA connector (0-3.3 V, high-Z) (see section *Overview*) as shown in the figure below and have any sequencer in the Qblox instrument participating in the experiment execute the *wait_trigger* instruction (see section *Instructions*).

1.12 Troubleshooting

"Have you tried turning it off and on again?" - The IT Crowd

Below you will find a table with common problems and potential solutions for Qblox Pulsar series instruments. If your problem is not listed or you are not able to fix your problem, please contact support@qblox.com for help.

Problem	Solutions
The status (S) LED is not green or white.	If the LED is red, it indicates a serious error that needs to be resolved. Query `get_system_state()` to see what the problem is.
	If the LED is orange, the LED indicates a critical warning, that an error has occurred but has been resolved. Query `get_system_state()` see what the problem is.
	If the LED is yellow, the device is (trying to) boot. If this state persists for more than 1-2 try power-cycling the device manually.
The reference clock (R) LED is not green.	If the LED is red, no reference clock is found. Most likely you have selected the external input as reference source, but have not connected the reference.
	If the LED is blue, the internal reference clock is selected. Use the parameter `reference_source()` to change this if necessary.
The channel (I/O) LEDs are not green.	If the LED is red, an error has occurred in one or more of the connected sequencers. Query `get_sequencer_state()` to see what the problem is.
	If the LED is purple or blue, one or more of the connected sequencers is armed or running. Query `get_sequencer_state()` to see what the sequencers are doing.
	If the LED is orange, one or more of the connected sequencers is causing the output to clip. Query `get_sequencer_state()` to see what the sequencers are doing.
I cannot connect to the instrument.	Make sure that the Ethernet cables are firmly

See also:

An IPython notebook version of this tutorial can be downloaded here:

`cont_wave_mode.ipynb`

1.13 Continuous waveform mode

In this tutorial we will demonstrate continuous waveform mode (see Continuous waveform mode). In addition, we will observe the output on an oscilloscope to demonstrate the results.

We can perform this tutorial with either a Pulsar or a Cluster QCM/QRM . We use the term 'QxM' encompassing both QCM and QRM modules.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.13.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
```

(continues on next page)

(continued from previous page)

```
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('cluster-mm',), value='cluster-
↪mm')
```

Pulsar QxM

Run these cells after selecting the your Pulsar module. Skip to the *Cluster QxM section* below if you have selected a Cluster module.

[ ]:
```
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qxm = Pulsar(f"{device_name}", ip_address)
qxm.reset()
print(f"{device_name} connected at {ip_address}")
print(qxm.get_system_state())
```

Skip to the next section *(Generate Waveform)* if you are not using a cluster.

Cluster QxM

First we connect to the Cluster using its IP address. Go to the *Pulsar QxM section* if you are using a Pulsar.

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]
```

(continues on next page)

(continued from previous page)

```
# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.0.2
```

We then find all available cluster modules to connect to them individually.

[4]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QxM module from the available modules in your Cluster:")
display(connect_qxm)
```

```
{'module3': 'QRM', 'module5': 'QCM-RF', 'module8': 'QCM', 'module16': 'QRM-RF'}

Select the QxM module from the available modules in your Cluster:
```

```
Dropdown(options=('module3', 'module5', 'module8', 'module16'), value='module3
↪')
```

Finally, we connect to the selected Cluster module.

[6]:
```
# Connect to the cluster QxM module
module = connect_qxm.value
qxm = getattr(cluster, module)
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QCM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

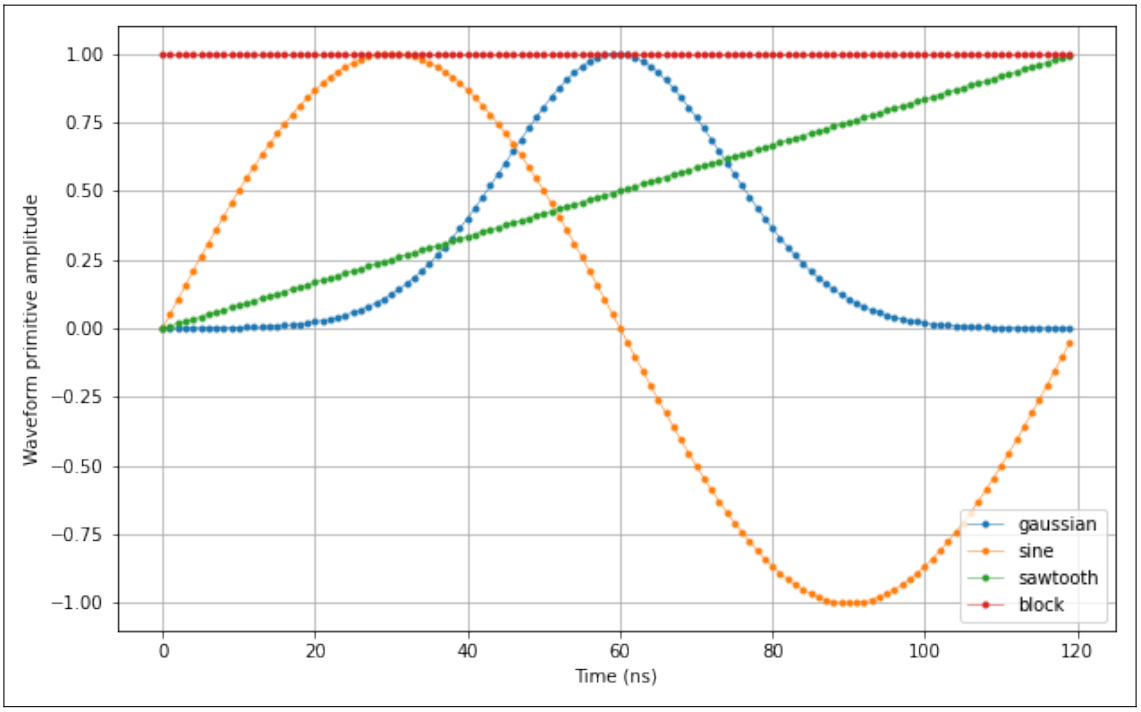
1.13.2 Generate waveforms

Next, we are going to generate a couple of waveforms that we are going to upload to the instrument in the next step.

[7]:
```
# Waveform parameters
waveform_length = 120  # nanoseconds (needs to be a multiple of 4 ns)

# Waveform dictionary (data will hold the samples and index will be used to␣
↪select the waveforms in the instrument).
waveforms = {
    "gaussian": {
        "data": scipy.signal.gaussian(
            waveform_length, std=0.12 * waveform_length
        ).tolist(),
        "index": 0,
    },
    "sine": {
        "data": [
            math.sin((2 * math.pi / waveform_length) * i)
            for i in range(0, waveform_length)
        ],
        "index": 1,
    },
    "sawtooth": {
        "data": [(1.0 / (waveform_length)) * i for i in range(0, waveform_
↪length)],
        "index": 2,
    },
    "block": {"data": [1.0 for i in range(0, waveform_length)], "index": 3},
}
```

Let's plot the waveforms to see what we have created.

[8]:
```
time = numpy.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())),␣
↪1)
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(10, 10 / 1.61))

for wf, d in waveforms.items():
    ax.plot(time[: len(d["data"])], d["data"], ".-", linewidth=0.5, label=wf)

ax.legend(loc=4)
ax.yaxis.grid()
ax.xaxis.grid()
ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

matplotlib.pyplot.draw()
matplotlib.pyplot.show()
```



1.13.3 Upload waveforms

Now that we know that the waveforms are what we expect them to be, let's upload them to the instrument. To do this we need to store the waveforms in a JSON file together with a Q1ASM program for the sequence processor. Since we are going to use continuous waveform mode, the sequence processor will be bypassed and the Q1ASM program can be trivial (i.e. `stop`).

[9]:
```
# Sequence program.
seq_prog = "stop"

# Check waveform length.
for name in waveforms:
    assert (
        len(waveforms[name]["data"]) % 4
    ) == 0, (
        "In continuous waveform mode the lenght of a waveform must be a␣
↪multiple of 4!"
    )

# Add sequence program and waveforms to single dictionary and write to JSON␣
↪file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": {},
    "program": seq_prog,
}
with open("cont_wave_mode.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
```

(continues on next page)

(continued from previous page)

```
    file.close()
```

Let's write the JSON file to the instruments. We will use sequencer 0 and 1, which will drive outputs $O^{[1-2]}$ and $O^{[3-4]}$ respectively.

[10]:
```
# Upload sequence.
qxm.sequencer0.sequence("cont_wave_mode.json")
qxm.sequencer1.sequence("cont_wave_mode.json")
```

1.13.4 Play waveforms

The waveforms have been uploaded to the instrument. Now we need to configure the instrument to run in continuous waveform mode. We do this by setting the following parameters of the sequencers.

[11]:
```
# Configure the sequencers to run in continuous waveform mode.
for sequencer in [qxm.sequencer0, qxm.sequencer1]:
    sequencer.cont_mode_en_awg_path0(True)
    sequencer.cont_mode_en_awg_path1(True)

# Map sequencers to specific outputs (but first disable all sequencer␣
↪connections).
for sequencer in qxm.sequencers:
    for out in range(0, 4):
        if hasattr(sequencer, "channel_map_path{}_out{}_en".format(out % 2,␣
↪out)):
            sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)

# If it is a QRM, we only map sequencer 0 to the outputs.
qxm.sequencer0.channel_map_path0_out0_en(True)
qxm.sequencer0.channel_map_path1_out1_en(True)
if qxm.is_qcm_type:
    qxm.sequencer1.channel_map_path0_out2_en(True)
    qxm.sequencer1.channel_map_path1_out3_en(True)

# Set specific waveform to specific output.
qxm.sequencer0.cont_mode_waveform_idx_awg_path0(0)  # Gaussian on O1
qxm.sequencer0.cont_mode_waveform_idx_awg_path1(1)  # Sine on O2
qxm.sequencer1.cont_mode_waveform_idx_awg_path0(2)  # Sawtooth on O3
qxm.sequencer1.cont_mode_waveform_idx_awg_path1(3)  # DC on O4
```

Now let's start playback.

[12]:
```
# Arm and start both sequencers.
qxm.arm_sequencer(0)
qxm.arm_sequencer(1)
qxm.start_sequencer()

# Print status of both sequencers (should now say Q1 stopped, because of the␣
↪stop instruction).
print(qxm.get_sequencer_state(0))
print(qxm.get_sequencer_state(1))
```

```
Status: STOPPED, Flags: NONE
Status: STOPPED, Flags: NONE
```

1.13.5 Check waveforms

The instrument is now running in continuous waveform mode. Now let's connect an oscilloscope and check the outputs. We connect all output channels of the QXM to two/four channels of an oscilloscope. On the scope we are able to see that all waveforms are being generated correctly:

Outputs: Yellow=O^1, Blue=O^2, Purple=O^3 and Green=O^4

1.13.6 Stop

Finally, let's stop the playback and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[13]:
```
# Stop both sequencers.
qxm.stop_sequencer()

# Print status of both sequencers (should now say it is stopped).
print(qxm.get_sequencer_state(0))
print(qxm.get_sequencer_state(1))
print()

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qxm.print_readable_snapshot(update=True)
```

(continues on next page)

(continued from previous page)

```
# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

```
Status: STOPPED, Flags: FORCED_STOP
Status: STOPPED, Flags: FORCED_STOP
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`basic_sequencing.ipynb`

1.14 Basic sequencing

In this tutorial we will demonstrate basic sequencer based operations (see section Sequencer). This includes creating a sequence, consisting of waveforms and a simple Q1ASM program, and executing this sequence synchronously on multiple sequencers.

The sequence is going to consecutively play two waveforms, a gaussian and block with a duration of 20ns each, with an increasing wait period in between them. We will increase the wait period 20ns a 100 times after which the sequence is stopped. The sequence will also trigger marker output 1 at every interval, so that the sequence can be easily monitored on an oscilloscope.

We can perform this tutorial with either a Pulsar or a Cluster QCM/QRM . We use the term 'QxM' encompassing both QCM and QRM modules.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.14.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('cluster-mm',), value='cluster-
↪mm')
```

Pulsar QxM

Run these cells after selecting the your Pulsar module. Skip to the *Cluster QxM section* below if you have selected a Cluster module.

[6]:
```
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qxm = Pulsar(f"{device_name}", ip_address)
qxm.reset()
print(f"{device_name} connected at {ip_address}")
print(qxm.get_system_state())
```

```
pulsar-qcm connected at 192.168.0.3
Status: OKAY, Flags: NONE, Slot flags: NONE
```

Skip to the next section *(Generate Waveform)* if you are not using a cluster.

Cluster QxM

First we connect to the Cluster using its IP address. Go to the *Pulsar QxM section* if you are using a Pulsar.

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]


# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()

print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.1.0
```

We then find all available cluster modules to connect to them individually.

[4]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QxM module from the available modules in your Cluster:")
display(connect_qxm)
```

```
{'module2': 'QCM', 'module4': 'QRM'}

Select the QxM module from the available modules in your Cluster:
```

```
Dropdown(options=('module2', 'module4'), value='module2')
```

Finally, we connect to the selected Cluster module.

[5]:
```
# Connect to the cluster QxM module
module = connect_qxm.value
qxm = getattr(cluster, module)
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QCM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

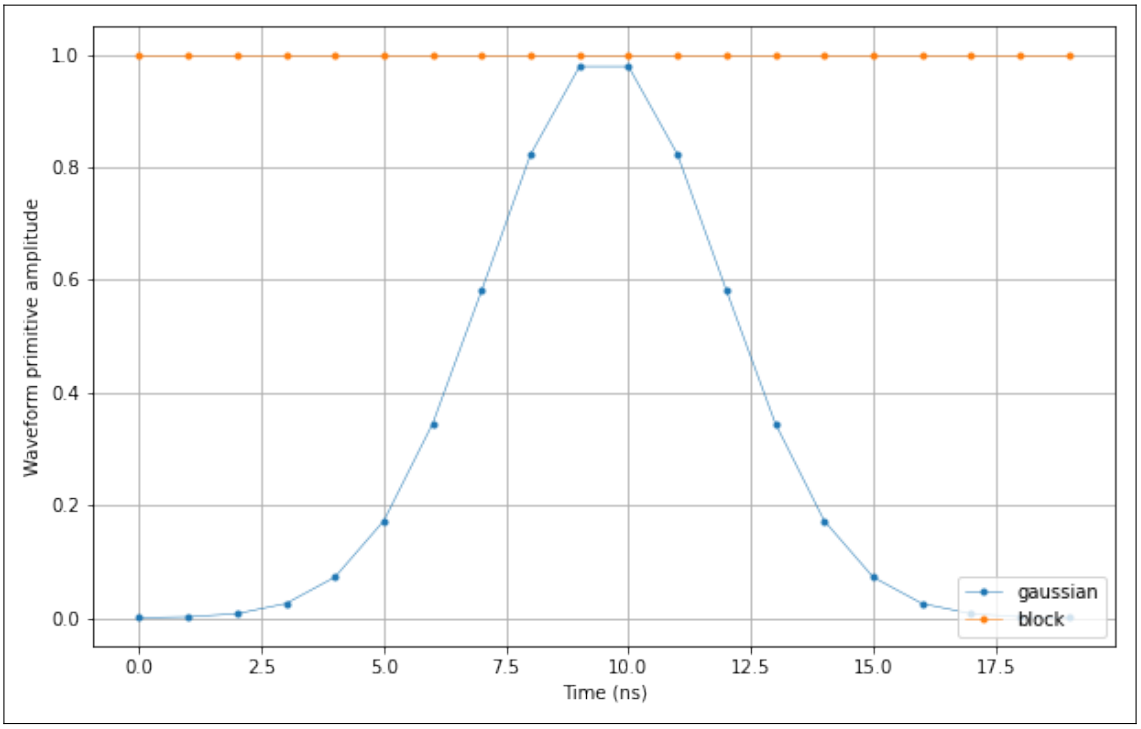
1.14.2 Generate waveforms

Next, we need to create the gaussian and block waveforms for the sequence.

[6]:
```
# Waveform parameters
waveform_length = 20  # nanoseconds

# Waveform dictionary (data will hold the samples and index will be used to␣
↪select the waveforms in the instrument).
waveforms = {
    "gaussian": {
        "data": scipy.signal.gaussian(
            waveform_length, std=0.12 * waveform_length
        ).tolist(),
        "index": 0,
    },
    "block": {"data": [1.0 for i in range(0, waveform_length)], "index": 1},
}
```

Let's plot the waveforms to see what we have created.

[7]:
```
time = numpy.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())),␣
↪1)
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(10, 10 / 1.61))

for wf, d in waveforms.items():
    ax.plot(time[: len(d["data"])], d["data"], ".-", linewidth=0.5, label=wf)

ax.legend(loc=4)
ax.yaxis.grid()
ax.xaxis.grid()
ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

matplotlib.pyplot.draw()
matplotlib.pyplot.show()
```



1.14.3 Create Q1ASM program

Now that we have the waveforms for the sequence, we need a Q1ASM program that sequences the waveforms as previously described.

[8]:
```
# Sequence program.
seq_prog = """
       move      100,R0   #Loop iterator.
       move      20,R1    #Initial wait period in ns.
       wait_sync 4        #Wait for sequencers to synchronize and then wait␣
↪another 4ns.

loop:  set_mrk   1        #Set marker output 1.
       play      0,1,4    #Play a gaussian and a block on output path 0 and 1␣
↪respectively and wait 4ns.
       set_mrk   0        #Reset marker output 1.
       upd_param 16       #Update parameters and wait the remaining 16ns of␣
↪the waveforms.

       wait      R1       #Wait period.

       play      1,0,20   #Play a block and a gaussian on output path 0 and 1␣
↪respectively and wait 20ns.
       wait      1000     #Wait a 1us in between iterations.
       add       R1,20,R1 #Increase wait period by 20ns.
       loop      R0,@loop #Subtract one from loop iterator.

       stop               #Stop the sequence after the last iteration.
"""
```

1.14.4 Upload sequence

Now that we have the waveforms and Q1ASM program, we can combine them in a sequence stored in a JSON file.

[9]:
```
# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": {},
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

Let's write the JSON file to the instruments. We will use sequencer 0 and 1, which will drive outputs $O^{[1-2]}$ and $O^{[3-4]}$ respectively.

[ ]:
```
# Upload sequence.
qxm.sequencer0.sequence("sequence.json")
qxm.sequencer1.sequence("sequence.json")
```

1.14.5 Play sequence

The sequence has been uploaded to the instrument. Now we need to configure the sequencers in the instrument to use the `wait_sync` instruction at the start of the Q1ASM program to synchronize.

[11]:
```
# Configure the sequencers to synchronize.
qxm.sequencer0.sync_en(True)
qxm.sequencer1.sync_en(True)

# Map sequencers to specific outputs (but first disable all sequencer␣
↪connections).
for sequencer in qxm.sequencers:
    for out in range(0, 4):
        if hasattr(sequencer, "channel_map_path{}_out{}_en".format(out % 2,␣
↪out)):
            sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)

# If it is a QRM, we only map sequencer 0 to the outputs.
qxm.sequencer0.channel_map_path0_out0_en(True)
qxm.sequencer0.channel_map_path1_out1_en(True)
if qxm.is_qcm_type:
    qxm.sequencer1.channel_map_path0_out2_en(True)
    qxm.sequencer1.channel_map_path1_out3_en(True)
```

Now let's start the sequence. If you want to observe the sequence, this is the time to connect an oscilloscope to marker output 1 and one or more of the four outputs. Configure the oscilloscope to trigger on the marker output 1.

[12]:
```
# Arm and start both sequencers.
qxm.arm_sequencer(0)
qxm.arm_sequencer(1)
qxm.start_sequencer()

# Print status of both sequencers.
print(qxm.get_sequencer_state(0))
print(qxm.get_sequencer_state(1))
```

```
Status:
Status: STOPPED, Flags: NONE
Status: STOPPED, Flags: NONE
```

1.14.6 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Stop both sequencers.
qxm.stop_sequencer()

# Print status of both sequencers (should now say it is stopped).
print(qxm.get_sequencer_state(0))
print(qxm.get_sequencer_state(1))
print()

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qxm.print_readable_snapshot(update=True)

# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`advanced_sequencing.ipynb`

1.15 Advanced sequencing

In this tutorial we will demonstrate advanced sequencer based operations, where we focus on waveform parametrization (see section Sequencer). We will demonstrate this by creating a sequence that will show various sequencer features, including complex looping constructs, dynamic gain control, hardware-based modulation and marker output control.

The sequence itself will use four waveform envelopes with a duration of 1s each; a gaussian, a sine, a sawtooth and a block. We will have several nested loops in the sequence. The first loop will increase the wait period between the start of the iteration and playback of the waveform envelope and also increase the gain of the waveform envelope on every iteration. At the end of this loop a second loop will do the

inverse operations. A third loop will loop over the first and second loops to iterate over the four waveform envelopes. And finally a fourth loop will function as an infinite loop over the third loop. At the same time, the sequence will also control marker output 1 and create a trigger point at the start of each iteration of the first and second loops as well an "enable" during playback. Finally, each waveform envelope will be modulated at 10MHz.

The result of this sequence, when observed on an oscilloscope, will be iterating waveform envelopes that will be sliding over the modulation frequency with varying gain, encapsulated by an "enable" on the marker output. We highly recommend that you take a look at it, to get an impression of what is possible with the sequencers.

We can perform this tutorial with either a Pulsar QCM/QRM or a Cluster QCM/QRM . We use the term 'QxM' encompassing both QCM and QRM modules.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.15.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
```

(continues on next page)

(continued from previous page)

```
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('cluster-mm',), value='cluster-
↪mm')
```

Pulsar QxM

Run this cells after selecting the your Pulsar module. Skip to the *Cluster QxM section* below if you have selected a Cluster module.

[ ]:
```
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qxm = Pulsar(f"{device_name}", ip_address)
qxm.reset()
print(f"{device_name} connected at {ip_address}")
print(qxm.get_system_state())
```

Skip to the next section *(Generate Waveform)* if you are not using a cluster.

Cluster QxM

First we connect to the Cluster using its IP address. Go to the *Pulsar QxM section* if you are using a Pulsar.

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster
cluster = Cluster(device_name, ip_address)
```

(continues on next page)

(continued from previous page)

```
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.1.0
```

We then find all available cluster modules to connect to them individually.

[4]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QxM module from the available modules in your Cluster:")
display(connect_qxm)
```

```
{'module2': 'QCM', 'module4': 'QRM'}

Select the QxM module from the available modules in your Cluster:
```

```
Dropdown(options=('module2', 'module4'), value='module2')
```

Finally, we connect to the selected Cluster module.

[5]:
```
# Connect to the cluster QxM module
module = connect_qxm.value
qxm = getattr(cluster, module)
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QCM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

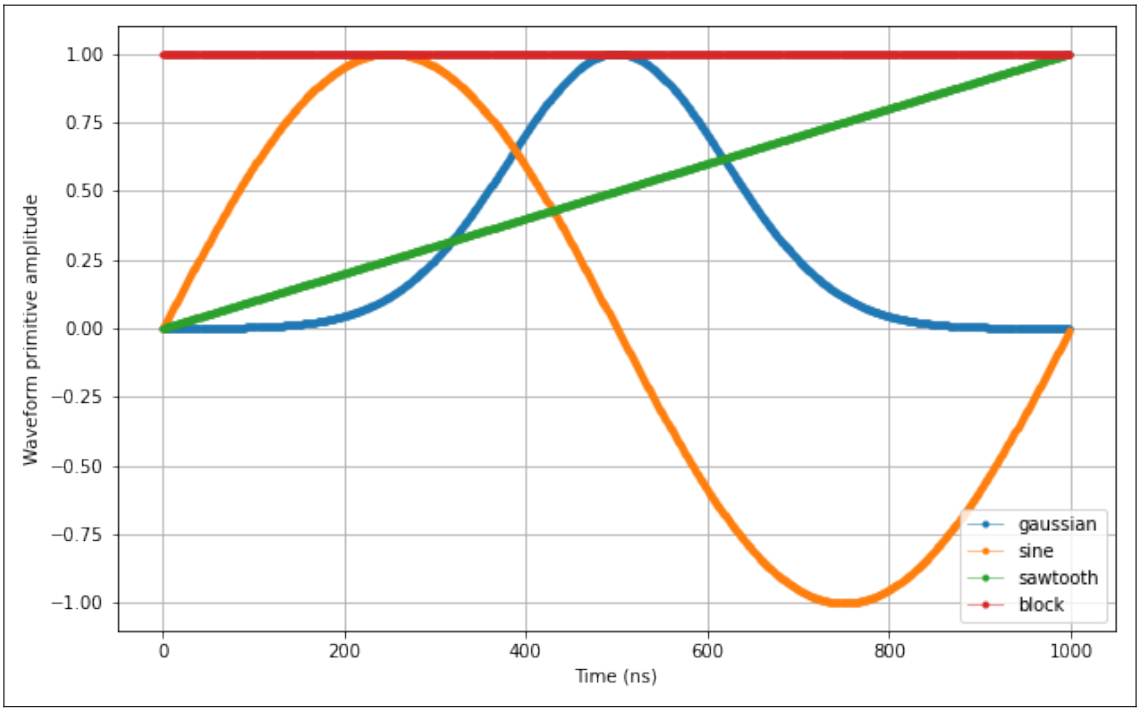
1.15.2 Generate waveforms

Next, we need to create the gaussian, sine, sawtooth and block waveform envelopes for the sequence.

[6]:
```
# Waveform parameters
waveform_length = 1000  # nanoseconds

# Waveform dictionary (data will hold the samples and index will be used to␣
↪select the waveforms in the instrument).
waveforms = {
    "gaussian": {
        "data": scipy.signal.gaussian(
            waveform_length, std=0.12 * waveform_length
        ).tolist(),
        "index": 0,
    },
    "sine": {
        "data": [
            math.sin((2 * math.pi / waveform_length) * i)
            for i in range(0, waveform_length)
        ],
        "index": 1,
    },
    "sawtooth": {
        "data": [(1.0 / (waveform_length)) * i for i in range(0, waveform_
↪length)],
        "index": 2,
    },
    "block": {"data": [1.0 for i in range(0, waveform_length)], "index": 3},
}
```

Let's plot the waveforms to see what we have created.

[7]:
```
time = numpy.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())),␣
↪1)
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(10, 10 / 1.61))

for wf, d in waveforms.items():
    ax.plot(time[: len(d["data"])], d["data"], ".-", linewidth=0.5, label=wf)

ax.legend(loc=4)
ax.yaxis.grid()
ax.xaxis.grid()
ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

matplotlib.pyplot.draw()
matplotlib.pyplot.show()
```



1.15.3 Create Q1ASM program

Now that we have the waveforms for the sequence, we need a Q1ASM program that sequences the waveforms as previously described.

[8]:
```
# Sequence program.
seq_prog = """
                  wait_sync     4                   #Wait for synchronization
                  reset_ph                          #Reset absolute phase
                  upd_param     4                   #Update all parameters and␣
↪wait 4ns
start:            move          4,R0                #Init number of waveforms
                  move          0,R1                #Init waveform index

mult_wave_loop:   move          166,R2              #Init number of single␣
↪wave loops (increasing wait)
                  move          166,R3              #Init number of single␣
↪wave loops (decreasing wait)
                  move          24,R4               #Init number of dynamic␣
↪wait time (total of 4us)
                  move          3976,R5             #Init number of dynamic␣
↪wait time remainder
                  move          32768,R6            #Init gain (Maximum gain)

sngl_wave_loop_0: move          800,R7              #Init number of long wait␣
↪loops (total of 40ms)
                  set_mrk       15                  #Set marker to 0xF
                  upd_param     4                   #Update all parameters and␣
↪wait 4ns
```

(continues on next page)

(continued from previous page)

```
                  set_mrk      0                    #Set marker to 0
                  upd_param    96                   #Update all parameters and␣
↪wait 96ns

                  wait         R4                   #Dynamic wait
                  add          R4,24,R4             #Increase wait

                  set_mrk      1                    #Set marker to 1
                  play         R1,R1,996            #Play waveform and wait␣
↪996ns
                  set_mrk      0                    #Set marker to 0
                  upd_param    4                    #Update all parameters and␣
↪wait for 4ns

                  wait         R5                   #Compensate previous␣
↪dynamic wait
                  sub          R5,24,R5             #Decrease wait

                  sub          R6,98,R6             #Decrease gain
                  nop
                  set_awg_gain R6,R6                #Set gain

long_wait_loop_0: wait         50000                #Wait 50 us
                  loop         R7,@long_wait_loop_0 #Wait total of 40ms
                  loop         R2,@sngl_wave_loop_0 #Repeat single wave loop

sngl_wave_loop_1: move         800,R7               #Init number of long wait␣
↪loops (total of 40ms)
                  set_mrk      15                   #Set marker to 0xF
                  upd_param    8                    #Update all parameters and␣
↪wait 8ns
                  set_mrk      0                    #Set marker to 0
                  upd_param    92                   #Update all parameters and␣
↪wait 92ns

                  wait         R4                   #Dynamic wait
                  sub          R4,24,R4             #Decrease wait

                  set_mrk      1                    #Set marker to 1
                  play         R1,R1,996            #Play waveform and wait␣
↪996ns
                  set_mrk      0                    #Set marker to 0
                  upd_param    4                    #Update all parameters and␣
↪wait 4ns

                  wait         R5                   #Compensate previous␣
↪dynamic wait
                  add          R5,24,R5             #Increase wait

                  sub          R6,98,R6             #Decrease gain
                  nop
                  set_awg_gain R6,R6                #Set gain
```

(continues on next page)

(continued from previous page)

```
long_wait_loop_1: wait          50000                #Wait for 50 us
                  loop          R7,@long_wait_loop_1 #Wait total of 40ms
                  loop          R3,@sngl_wave_loop_1 #Repeat single wave loop

                  add           R1,1,R1              #Adjust waveform index
                  loop          R0,@mult_wave_loop   #Repeat with next waveform␣
↪envelope
                  jmp           @start               #Repeat entire sequence
"""
```

1.15.4 Upload sequence

Now that we have the waveforms and Q1ASM program, we can combine them in a sequence stored in a JSON file.

[9]:
```
# Add sequence to single dictionary and write to JSON file.
seqeunce = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": {},
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(seqeunce, file, indent=4)
    file.close()
```

Let's write the JSON file to the instruments. We will use sequencer 0 and 1, which will drive outputs $O^{[1-2]}$ and $O^{[3-4]}$ respectively.

[10]:
```
# Upload sequence.
qxm.sequencer0.sequence("sequence.json")
qxm.sequencer1.sequence("sequence.json")
```

1.15.5 Play sequence

The sequence has been uploaded to the instrument. Now we need to configure the sequencers in the instrument to use the `wait_sync` instruction at the start of the Q1ASM program to synchronize and to enable the hardware-based modulation at 10MHz.

[11]:
```
# Configure the sequencers to synchronize and enable modulation at 10MHz.
qxm.sequencer0.sync_en(True)
qxm.sequencer0.mod_en_awg(True)
qxm.sequencer0.nco_freq(10e6)
qxm.sequencer1.sync_en(True)
qxm.sequencer1.mod_en_awg(True)
qxm.sequencer1.nco_freq(10e6)

# Map sequencers to specific outputs (but first disable all sequencer␣
↪connections).
```

(continues on next page)

(continued from previous page)

```
for sequencer in qxm.sequencers:
    for out in range(0, 4):
        if hasattr(sequencer, "channel_map_path{}_out{}_en".format(out % 2,␣
↪out)):
            sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)

# If it is a QRM, we only map sequencer 0 to the outputs.
qxm.sequencer0.channel_map_path0_out0_en(True)
qxm.sequencer0.channel_map_path1_out1_en(True)
if qxm.is_qcm_type:
    qxm.sequencer1.channel_map_path0_out2_en(True)
    qxm.sequencer1.channel_map_path1_out3_en(True)
```

Now let's start the sequence. If you want to observe the sequence, this is the time to connect an oscilloscope to marker output 1 and one or more of the four outputs. Configure the oscilloscope to trigger on marker output 1.

[12]:
```
# Arm and start sequencers.
qxm.arm_sequencer(0)
qxm.arm_sequencer(1)
qxm.start_sequencer()

# Print status of sequencers.
print(qxm.get_sequencer_state(0))
print(qxm.get_sequencer_state(1))
```

```
Status:
Status: RUNNING, Flags: NONE
Status: RUNNING, Flags: NONE
```

Before we continue, have you looked at the oscilloscope? Pretty nifty right? This is just an example. Imagine what else you can do with the power of the sequencers to control and/or speed up your experiments.

1.15.6 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Stop both sequencers.
qxm.stop_sequencer()

# Print status of both sequencers (should now say it is stopped).
print(qxm.get_sequencer_state(0))
print(qxm.get_sequencer_state(1))
print()

# Uncomment the following to print an overview of the instrument parameters.
# print("Snapshot:")
# qxm.print_readable_snapshot(update=True)
```

(continues on next page)

(continued from previous page)

```
# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`mixer_correction.ipynb`

1.16 Mixer correction

In this tutorial we will demonstrate the ability to compensate for output mixer non-idealities and observe the changes using an oscilloscope.

Mixer non-idealities can lead to unwanted spurs on the output (LO/RF/IF feedthrough and other spurious products) and they can be compensated by applying adjustments to the I/Q outputs: phase offset, gain ratio and DC offset. This solution applies to both baseband QCM/QRM products using external mixers as well as QCM-RF and QRM-RF products.

The tutorial is designed for all Qblox baseband products: Pulsar QRM, Pulsar QCM, Cluster QRM, Cluster QCM. We use the term 'QxM' encompassing both QCM and QRM modules. We will adjust all the parameters listed above and observe the changes to the I/Q outputs directly on an oscilloscope.

For QCM-RF and QRM-RF products, one can refer to the 'mixer calibration' section of the tutorial on RF-control.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.16.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('pulsar-qrm', 'pulsar-qcm'),␣
↪value='pulsar-qrm')
```

Pulsar QxM

Run these cells after selecting the your Pulsar module. Skip to the *Cluster QxM section* below if you have selected a Cluster module.

[3]:
```
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device
qxm = Pulsar(f"{device_name}", ip_address)
qxm.reset()  # reset QxM
print(f"{device_name} connected at {ip_address}")
print(qxm.get_system_state())
```

```
pulsar-qrm connected at 192.168.0.4
Status: OKAY, Flags: NONE, Slot flags: NONE
```

Skip to the next section *(Setup Sequencer)* if you are not using a cluster.

Cluster QxM

First we connect to the Cluster using its IP address. Go to the *Pulsar QxM section* if you are using a Pulsar.

[ ]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

We then find all available cluster modules to connect to them individually.

[ ]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QxM module from the available modules in your Cluster:")
display(connect_qxm)
```

Finally, we connect to the selected Cluster module.

[13]:
```
# Connect to the cluster QxM module
module = connect_qxm.value
qxm = getattr(cluster, module)
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QCM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

1.16.2 Setup Sequencer

The easiest way to view the influence of the mixer correction is to mix the NCO sin and cos with I and Q values of 1 (fullscale). The instrument output would be simple sinusoids with a 90deg phase offset and identical amplitude.

We use sequencer 0 to set I and Q values of 1 (fullscle) using DC offset and we mix those with the NCO signals.

[4]:
```
# Program sequence we will not use.
sequence = {"waveforms": {}, "weights": {}, "acquisitions": {}, "program":
↪"stop"}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
qxm.sequencer0.sequence(sequence)

# Program fullscale DC offset on I & Q, turn on NCO and enable modulation.
qxm.sequencer0.offset_awg_path0(1.0)
qxm.sequencer0.offset_awg_path1(1.0)
qxm.sequencer0.nco_freq(10e6)
qxm.sequencer0.mod_en_awg(True)
```

1.16.3 Control sliders

Create control sliders for the parameters described in the introduction. Each time the value of a parameter is updated, the sequencer is automatically stopped from the embedded firmware for safety reasons and has to be manually restarted.

The sliders cover the valid parameter range. If the code below is modified to input invalid values, the firmware will not program the values.

Please connect the I/Q outputs to an oscilloscope and set to trigger continuously on the I channel at 0V. Execute the code below, move the sliders and observe the result on the oscilloscope.

[5]:
```
def set_offset_I(offset_I):
    qxm.out0_offset(offset_I)
    qxm.arm_sequencer(0)
    qxm.start_sequencer(0)


def set_offset_Q(offset_Q):
    qxm.out1_offset(offset_Q)
    qxm.arm_sequencer(0)
    qxm.start_sequencer(0)


def set_gain_ratio(gain_ratio):
    qxm.sequencer0.mixer_corr_gain_ratio(gain_ratio)
    qxm.arm_sequencer(0)
    qxm.start_sequencer(0)


def set_phase_offset(phase_offset):
```

(continues on next page)

(continued from previous page)

```
    qxm.sequencer0.mixer_corr_phase_offset_degree(phase_offset)
    qxm.arm_sequencer(0)
    qxm.start_sequencer(0)


interact(
    set_offset_I, offset_I=widgets.FloatSlider(min=-1.0, max=1.0, step=0.01,␣
↪start=0.0)
)
interact(
    set_offset_Q, offset_Q=widgets.FloatSlider(min=-1.0, max=1.0, step=0.01,␣
↪start=0.0)
)
interact(
    set_gain_ratio,
    gain_ratio=widgets.FloatSlider(min=0.5, max=2.0, step=0.1, start=1.0),
)
interact(
    set_phase_offset,
    phase_offset=widgets.FloatSlider(min=-45.0, max=45.0, step=1.0, start=0.0),
)
```

```
interactive(children=(FloatSlider(value=0.0, description='offset_I', max=1.0,␣
↪min=-1.0, step=0.01), Output()),...
```

```
interactive(children=(FloatSlider(value=0.0, description='offset_Q', max=1.0,␣
↪min=-1.0, step=0.01), Output()),...
```

```
interactive(children=(FloatSlider(value=0.5, description='gain_ratio', max=2.0,
↪ min=0.5), Output()), _dom_clas...
```

```
interactive(children=(FloatSlider(value=0.0, description='phase_offset',␣
↪max=45.0, min=-45.0, step=1.0), Outpu...
```

[5]:
```
<function __main__.set_phase_offset(phase_offset)>
```

1.16.4 Stop

Finally, let's stop the sequencers and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Stop sequencers.
qxm.stop_sequencer()

# Print sequencer status (should now say it is stopped).
print(qxm.get_sequencer_state(0))

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qxm.print_readable_snapshot(update=True)
```

(continues on next page)

(continued from previous page)

```
# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`scope_acquisition.ipynb`

1.17 Scope acquisition

In this tutorial we will demonstrate the sequencer based scope acquisition procedure allowing you to inspect and process the raw input of the QRM. We will also have a look at how to average multiple scope acquisitions in hardware (see section Acquisition). We will show this by using a QRM and directly connecting outputs $O^{[1-2]}$ to inputs $I^{[1-2]}$ respectively. We will then use the QRM's sequencers to sequence waveforms on the outputs and simultaneously acquire the resulting waveforms on the inputs.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.17.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[ ]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

Pulsar QRM

Choose the Pulsar QRM and run the following cell. Skip to the *Cluster QRM section* if you selected a Cluster module.

[ ]:
```
# Close existing connections to the Pulsar modules
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qrm = Pulsar(f"{device_name}", ip_address)
qrm.reset()
print(f"{device_name} connected at {ip_address}")
print(qrm.get_system_state())
```

Skip to the next section *(Generate Waveform)* if you are not using a cluster.

Cluster QRM

First we connect to the Cluster using its IP address. Go to the *Pulsar QRM section* if you are using a Pulsar.

[ ]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]


# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

We then find all available cluster modules to connect to them individually.

[ ]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QRM module from the available modules in your Cluster:")
display(connect_qxm)
```

Finally, we connect to the selected Cluster module.

[50]:
```
# Connect to the cluster QRM
qrm = getattr(
    cluster, connect_qxm.value
)  # Connect to the module that you have chosen above
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QRM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

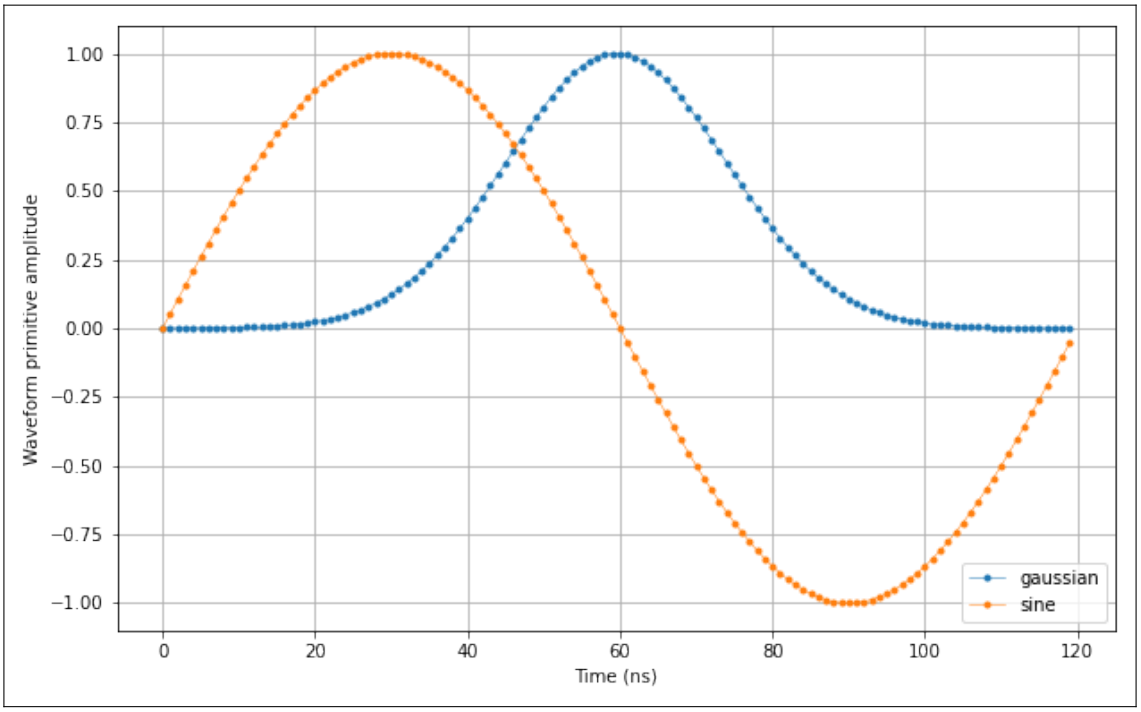
1.17.2 Generate waveforms

Next, we need to create the gaussian and block waveforms for the sequence.

[5]:
```
# Waveform parameters
waveform_length = 120  # nanoseconds

# Waveform dictionary (data will hold the samples and index will be used to␣
↪select the waveforms in the instrument).
waveforms = {
    "gaussian": {
        "data": scipy.signal.gaussian(
            waveform_length, std=0.12 * waveform_length
        ).tolist(),
        "index": 0,
    },
    "sine": {
        "data": [
            math.sin((2 * math.pi / waveform_length) * i)
            for i in range(0, waveform_length)
        ],
        "index": 1,
    },
}
```

Let's plot the waveforms to see what we have created.

[6]:
```
time = numpy.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())),␣
↪1)
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(10, 10 / 1.61))

for wf, d in waveforms.items():
    ax.plot(time[: len(d["data"])], d["data"], ".-", linewidth=0.5, label=wf)

ax.legend(loc=4)
ax.yaxis.grid()
ax.xaxis.grid()
ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

matplotlib.pyplot.draw()
matplotlib.pyplot.show()
```



1.17.3 Specify acquisitions

We also need to specify the acquisitions so that the instrument can allocate the required memory for it's acquisition list. In this case we will create 5 acquisition specifications that each create a single bin. However, we will not be using the bins in this turorial.

[7]:
```
# Acquisitions
acquisitions = {
    "single": {"num_bins": 1, "index": 0},
    "multiple_0": {"num_bins": 1, "index": 1},
    "multiple_1": {"num_bins": 1, "index": 2},
    "multiple_2": {"num_bins": 1, "index": 3},
    "avg": {"num_bins": 1, "index": 4},
}
```

1.17.4 Create Q1ASM program

Now that we have the waveforms and acquisitions specification for the sequence, we need a simple Q1ASM program that sequences and acquires the waveforms.

[8]:
```
# Sequence program.
seq_prog = """
play    0,1,4     #Play waveforms and wait 4ns.
acquire 0,0,16380 #Acquire waveforms and wait remaining duration of scope␣
↪acquisition.
stop              #Stop.
"""
```

1.17.5 Upload sequence

Now that we have the waveform and acquisition specifications and Q1ASM program, we can combine them in a sequence stored in a JSON file.

[9]:
```
# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

Let's write the JSON file to the instruments. We will use sequencer 0, which will drive outputs $O^{[1-2]}$ and acquire on inputs $I^{[1-2]}$.

[10]:
```
# Upload sequence.
qrm.sequencer0.sequence("sequence.json")
```

1.17.6 Play sequence

The sequence has been uploaded to the instrument. Now we need to configure the sequencers to trigger the acquisition with the `acquire` instruction.

[11]:
```
# Configure the sequencer to trigger the scope acquisition.
qrm.scope_acq_sequencer_select(0)
qrm.scope_acq_trigger_mode_path0("sequencer")
qrm.scope_acq_trigger_mode_path1("sequencer")

# Map sequencer to specific outputs (but first disable all sequencer␣
↪connections)
for sequencer in qrm.sequencers:
    for out in range(0, 2):
        sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)
```

Now let's start the sequence.

[12]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Print status of sequencer.
print("Status:")
print(qrm.get_sequencer_state(0))
```

```
Status:
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

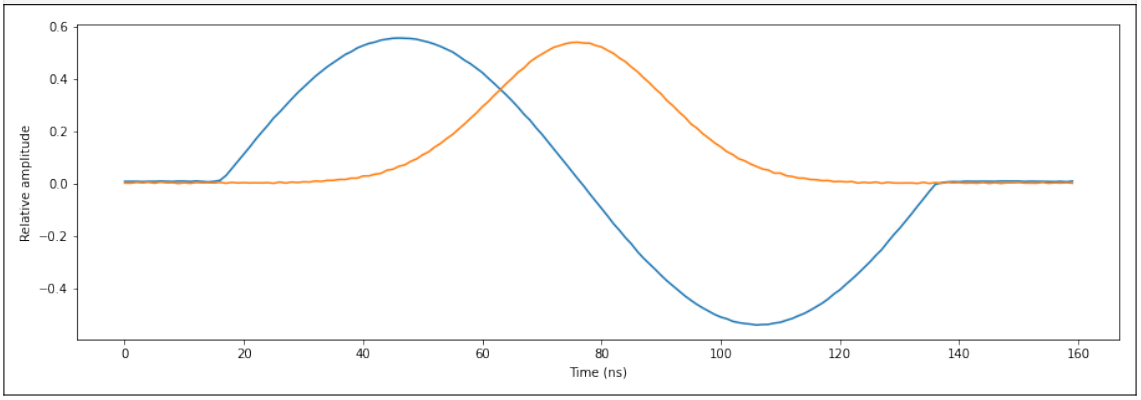
1.17.7 Retrieve acquisition

The waveforms have now been sequenced on the outputs and acquired on the inputs. Lets make sure that the sequencer has finished it's acquisition and then retrieve the resulting data. The acquisition data is stored in a temporary memory in the instrument's FPGA. We need to first move the data from this memory into the into the instrument's acquisition list. From there we can retrieve it from the instrument.

[13]:
```
# Wait for the acquisition to finish with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "single")

# Get acquisition list from instrument.
single_acq = qrm.get_acquisitions(0)
```

Let's plot the result.

[14]:
```
# Plot acquired signal on both inputs.
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(single_acq["single"]["acquisition"]["scope"]["path0"]["data"][130:290])
ax.plot(single_acq["single"]["acquisition"]["scope"]["path1"]["data"][130:290])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



1.17.8 Retrieve multiple acquisitions

We can also run the sequence multiple times consecutively and store the acquisition data in the instrument's acquisition list before retrieving them all in one go. To demonstrate this we will run the same sequence three times and vary the output gain for each run to create a clear distinction between the acquisitions.

[15]:
```
# First run
qrm.sequencer0.gain_awg_path0(0.33)
qrm.sequencer0.gain_awg_path1(0.33)

qrm.arm_sequencer(0)
qrm.start_sequencer()

qrm.get_acquisition_state(0, 1)
```

(continues on next page)

(continued from previous page)

```
qrm.store_scope_acquisition(0, "multiple_0")

# Second run
qrm.sequencer0.gain_awg_path0(0.66)
qrm.sequencer0.gain_awg_path1(0.66)

qrm.arm_sequencer(0)
qrm.start_sequencer()

qrm.get_acquisition_state(0, 1)

qrm.store_scope_acquisition(0, "multiple_1")

# Second run
qrm.sequencer0.gain_awg_path0(1)
qrm.sequencer0.gain_awg_path1(1)

qrm.arm_sequencer(0)
qrm.start_sequencer()

qrm.get_acquisition_state(0, 1)

qrm.store_scope_acquisition(0, "multiple_2")

# Get acquisition list from instrument.
multiple_acq = qrm.get_acquisitions(0)
```

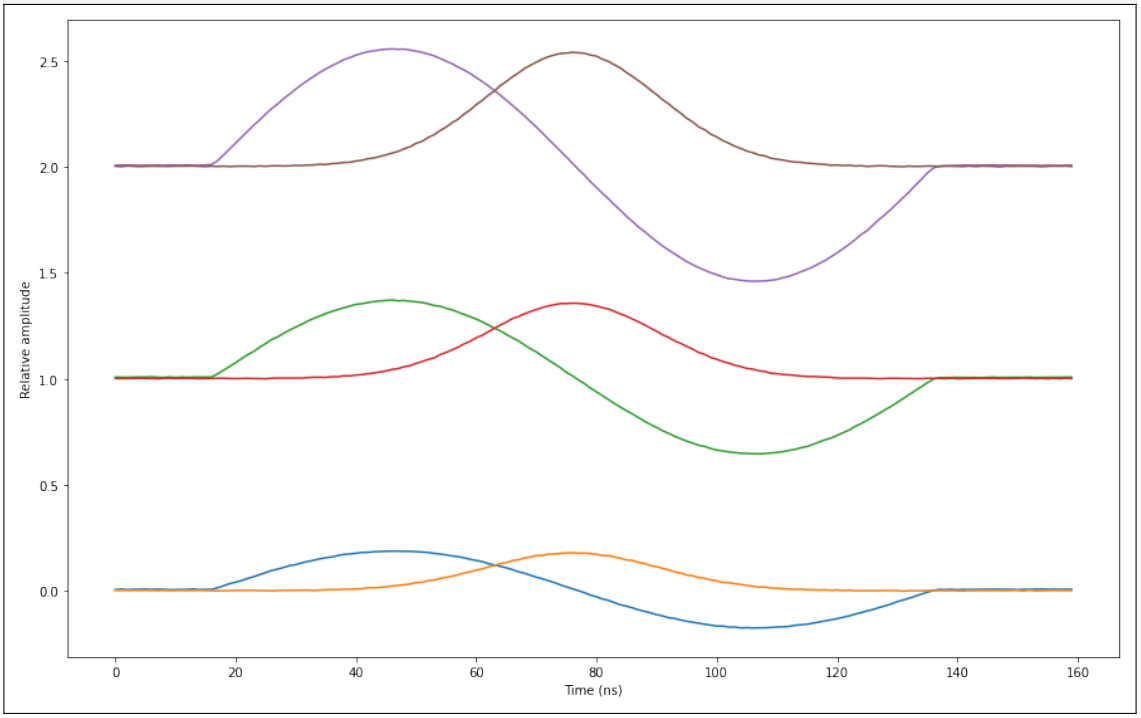
Let's plot the result again.

[16]:
```
# Plot acquired signals (add acquisition index to separate acquisitions in␣
↪plot).
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 1.61))
for acq_idx in range(0, 3):
    ax.plot(
        numpy.array(
            multiple_acq["multiple_{}".format(acq_idx)]["acquisition"]["scope
↪"][
                "path0"
            ]["data"][130:290]
        )
        + acq_idx
    )
    ax.plot(
        numpy.array(
            multiple_acq["multiple_{}".format(acq_idx)]["acquisition"]["scope
↪"][
                "path1"
            ]["data"][130:290]
        )
        + acq_idx
    )
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlabel("Time (ns)")
    ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



1.17.9 Hardware-based averaging

We can also use hardware in the instrument itself to automatically accumulate acquisition data on-the-fly. This can be used to do averaging, by dividing the final accumulated result by the number of accumulations. To use this feature, we first need to modify the Q1ASM to run the sequence multiple consecutive times.

[17]:
```
# Sequence program.
seq_prog = """
      move    1000,R0   #Loop iterator.

loop: play    0,1,4     #Play waveforms and wait 4ns.
      acquire 4,0,16380 #Acquire waveforms and wait remaining duration of␣
↪scope acquisition.
      loop    R0,@loop  #Run until number of iterations is done.

      stop              #Stop.
"""
```

Next, we need to program, configure and start the sequencer. This time we will also configure the sequencer to run in averaging mode.

[18]:
```
# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
```

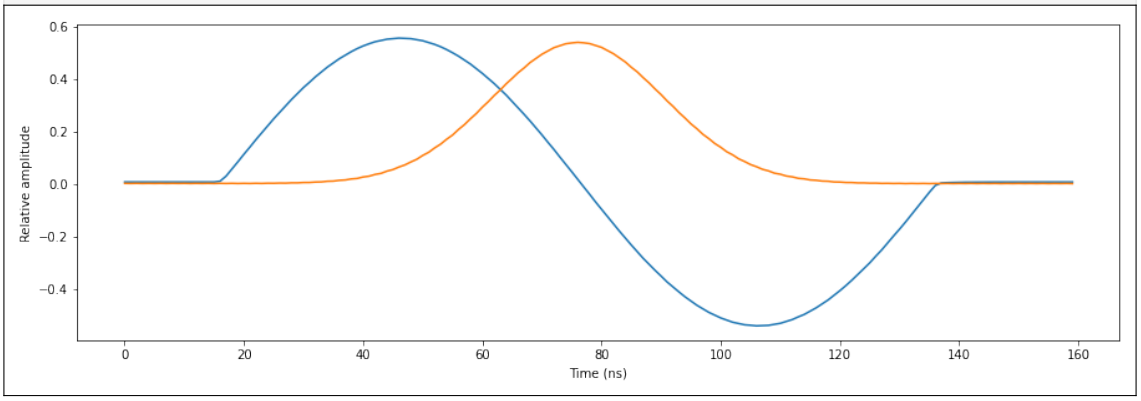
(continues on next page)

(continued from previous page)

```
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence_avg.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()

# Upload sequence.
qrm.sequencer0.sequence("sequence_avg.json")

# Enable hardware averaging
qrm.scope_acq_avg_mode_en_path0(True)
qrm.scope_acq_avg_mode_en_path1(True)

# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for sequence and acquisitions to finish.
qrm.get_acquisition_state(0, 1)

# Move accumulated result from temporary memory to the instrument's acquisition␣
↪list.
qrm.store_scope_acquisition(0, "avg")

# Get acquisition list from instrument.
avg_acq = qrm.get_acquisitions(0)
```

The sequence has now run and accumulated a 1000 times. Time to finish the averaging process and print the result.

[19]:
```
# Plot results.
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(numpy.array(avg_acq["avg"]["acquisition"]["scope"]["path0"]["data
↪"][130:290]))
ax.plot(numpy.array(avg_acq["avg"]["acquisition"]["scope"]["path1"]["data
↪"][130:290]))
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



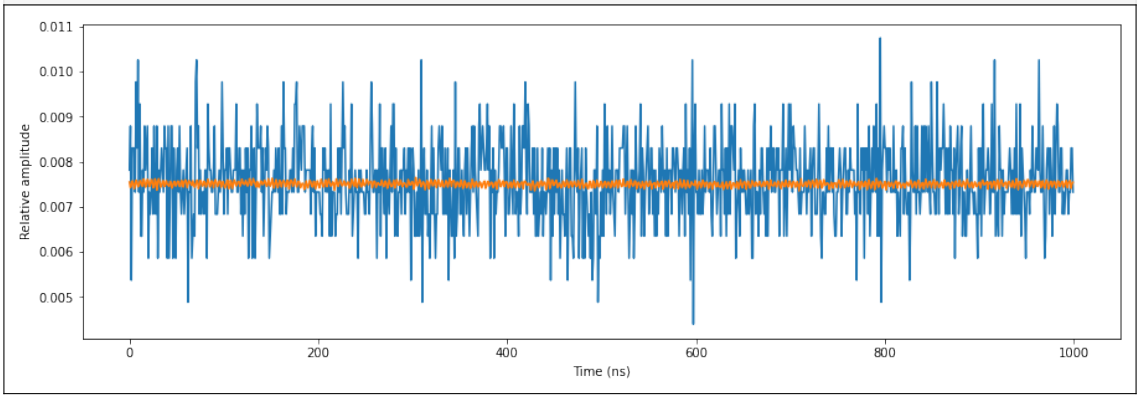
Note that the accumulated acquisitions have already been averaged when the data is returned. The instrument keeps track of the number of accumulations and divides the result upon returning the data. The number of accumulations is also available for review.

[20]:
```
# Print number of averages
print(avg_acq["avg"]["acquisition"]["scope"]["path0"]["avg_cnt"])
print(avg_acq["avg"]["acquisition"]["scope"]["path1"]["avg_cnt"])
```

```
1000
1000
```

To show that the hardware averaging worked, let's compare and zoom in on the data from the very first acquisition in this tutorial and the very last.

[21]:
```
# Plot results
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(
    numpy.array(
        single_acq["single"]["acquisition"]["scope"]["path0"]["data"][1000:
↪2000]
    )
)
ax.plot(numpy.array(avg_acq["avg"]["acquisition"]["scope"]["path0"]["data
↪"][1000:2000]))
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



1.17.10 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Stop sequencer.
qrm.stop_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0))
print()
```

(continues on next page)

(continued from previous page)

```
# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qrm.print_readable_snapshot(update=True)

# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`binned_acquisition.ipynb`

1.18 Binned acquisition

In this tutorial we will demonstrate the sequencer based acquisition binning procedure. The binning process is applied on the input path after real-time demodulation, (weighed) integration, IQ rotation and discretization. It allows storing both the integration and discretization results on the fly without intervention of the host PC in up to 131072 bins. It also allows averaging those bins on the fly as well (see section Acquisition). We will show this by using a QRM and directly connecting outputs $O^{[1-2]}$ to inputs $I^{[1-2]}$ respectively. We will then use the QRM's sequencers to sequence waveforms on the outputs and simultaneously acquire the resulting waveforms on the inputs.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.18.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('cluster-mm',), value='cluster-
↪mm')
```

Pulsar QRM

Choose the Pulsar QRM, run the following cell. Skip to the *Cluster QRM section* if you selected a Cluster module.

[3]:
```
# Close existing connections to the Pulsar modules
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qrm = Pulsar(f"{device_name}", ip_address)
qrm.reset()
print(f"{device_name} connected at {ip_address}")
print(qrm.get_system_state())
```

```
pulsar-qrm connected at 192.168.0.4
Status: OKAY, Flags: NONE, Slot flags: NONE
```

Skip to *Generate Waveform and Weights* if you have not selected a Cluster module.

Cluster QRM

First we connect to the Cluster using its IP address. Go to the *Pulsar QRM section* if you are using a Pulsar.

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.0.2
```

We then find all available cluster modules to connect to them individually.

[6]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QRM module from the available modules in your Cluster:")
display(connect_qxm)
```

```
{'module2': 'QCM', 'module4': 'QRM', 'module6': 'QCM-RF', 'module8': 'QRM-RF'}

Select the QRM module from the available modules in your Cluster:
```

```
Dropdown(options=('module2', 'module4', 'module6', 'module8'), value='module2')
```

Finally, we connect to the selected Cluster module.

[7]:
```
# Connect to the cluster QRM
qrm = getattr(
```

(continues on next page)

(continued from previous page)

```
    cluster, connect_qxm.value
)  # Connect to the module that you have chosen above
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QRM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

1.18.2 Generate waveforms and weights

Next, we need to create the waveforms used by the sequence for playback on the outputs as well as weights used by the sequence for weighed integrations.To keep it straightforward, we use the DC offset from the sequencers as our waveforms and define waveform weights in the cell below.

[4]:
```
# Waveform and weight parameters
waveform_weight_length = 600  # nanoseconds

# These will be used as weights in the "Weighed acquisition" section
waveforms_weights = {
    "gaussian": {
        "data": scipy.signal.gaussian(
            waveform_weight_length, std=0.12 * waveform_weight_length
        ).tolist(),
        "index": 0,
    },
    "sine": {
        "data": [
            math.sin((2 * math.pi / waveform_weight_length) * i)
            for i in range(0, waveform_weight_length)
        ],
        "index": 1,
    },
    "block": {"data": [1.0 for _ in range(0, waveform_weight_length)], "index":
↪ 2},
}
```

1.18.3 Specify acquisitions

We also need to specify the acquisitions so that the instrument can allocate the required memory for it's acquisition list. In this case we will create 4 acquisition specifications that each create multiple bins.

[5]:
```
# Acquisitions
acquisitions = {
    "non_weighed": {"num_bins": 10, "index": 0},
    "weighed": {"num_bins": 10, "index": 1},
    "large": {"num_bins": 131072, "index": 2},
    "avg": {"num_bins": 10, "index": 3},
}
```

1.18.4 Create Q1ASM program

Now that we have the waveform and acquisition specifications for the sequence, we need a simple Q1ASM program that sequences the waveforms and triggers the acquisitions. In this case we will simply trigger 10 non-weighed acquisitions and store each acquisition in a separate bin.

[6]:
```
# Sequence program.
seq_prog = """
      move    0,R0        #Loop iterator.
      nop

loop: acquire 0,R0,1200   #Acquire bins and store them in "non_weighed"␣
↪acquisition.
      add     R0,1,R0     #Increment iterator
      nop                 #Wait a cycle for R0 to be available.
      jlt     R0,10,@loop #Run until number of iterations is done.

      stop                #Stop.
"""
```

1.18.5 Upload sequence

Now that we have the waveform, weights and acquisition specifications and Q1ASM program, we can combine them in a sequence stored in a JSON file.

[7]:
```
# Add sequence program, waveforms, weights and acquistitions to single␣
↪dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms_weights,
    "weights": waveforms_weights,
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

Let's write the JSON file to the instruments. We will use sequencer 0, which will drive outputs $O^{[1-2]}$ and acquire on inputs $I^{[1-2]}$.

[8]:
```
# Upload sequence.
qrm.sequencer0.sequence("sequence.json")
```

1.18.6 Play sequence

The sequence has been uploaded to the instrument. Now we need to configure the sequencers. To keep it simple we will set a DC signal on the outputs of the instrument by enabling the sequencer offsets and disabling the modulation. These DC signals will then be acquired through the inputs, so we will also disable the demodulation on the input path. Furthermore, since we are running non-weighed integrations we need to specify the integration length. This integration length will be used for every non-weighed integration moving forward. We will also put the integration result phase rotation to 0 degrees and discretation threshold to 0.

[9]:
```
# Configure scope mode
qrm.scope_acq_sequencer_select(0)
qrm.scope_acq_trigger_mode_path0("sequencer")
qrm.scope_acq_trigger_mode_path1("sequencer")

# Configure the sequencer
qrm.sequencer0.offset_awg_path0(0.5)
qrm.sequencer0.offset_awg_path1(0.5)
qrm.sequencer0.mod_en_awg(False)
qrm.sequencer0.demod_en_acq(False)
qrm.sequencer0.integration_length_acq(1000)
qrm.sequencer0.phase_rotation_acq(0)
qrm.sequencer0.discretization_threshold_acq(0)

# Map sequencer to specific outputs (but first disable all sequencer␣
↪connections)
for sequencer in qrm.sequencers:
    for out in range(0, 2):
        sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)
```

Now let's start the sequence.

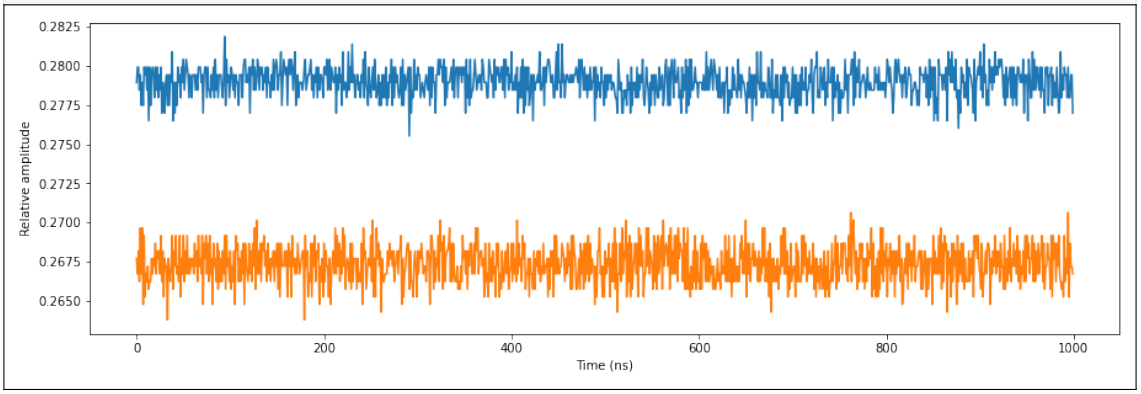
[10]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0, 1))
```

```
Status:
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

1.18.7 Retrieve acquisition

Next we will have a quick look at the input signal, so that we can compare it to the integration results. Since we are integrating over a DC signal we are expecting the integration results to be roughly equal to the average DC value.

[11]:
```
# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "non_weighed")

# Get acquisition list from instrument.
non_weighed_acq = qrm.get_acquisitions(0)["non_weighed"]

# Plot acquired signal on both inputs.
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(non_weighed_acq["acquisition"]["scope"]["path0"]["data"][0:1000])
ax.plot(non_weighed_acq["acquisition"]["scope"]["path1"]["data"][0:1000])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



To check if the integration results match with what we expect, we need to divide the integration results by the integration length which was set through the corresponding QCoDeS parameter. Note that the 'valid' key of the dictionary indicates if the bin was actually set during the sequence.

[12]:
```
int_len = qrm.sequencer0.integration_length_acq()
bins = non_weighed_acq["acquisition"]["bins"]
bins["integration"]["path0"] = [(val / int_len) for val in bins["integration"][
↪"path0"]]
bins["integration"]["path1"] = [(val / int_len) for val in bins["integration"][
↪"path1"]]
bins
```

[12]:
```
{'integration': {'path0': [0.27915486077186125,
   0.2791612115290669,
   0.27915192965315094,
   0.27901367855398146,
   0.2791001465559355,
   0.2791299462628236,
   0.2789799706888129,
```

(continues on next page)

(continued from previous page)

```
   0.2790600879335613,
   0.279057156814851,
   0.27897557401074746],
  'path1': [0.2676213971665853,
   0.26770151441133366,
   0.2676560820713239,
   0.26742061553492913,
   0.2675017098192477,
   0.26750122129946263,
   0.2674181729360039,
   0.26754714215925746,
   0.26734391792867607,
   0.26737909135319976]},
 'threshold': [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
 'avg_cnt': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

1.18.8 Weighed acquisition

Next we will show weighed integrations. To do this, we will need to modify the sequence program slightly and reupload the program. We will be using a the gaussian weight to integrate over input path 0 and the sine weight to integrate over input path 1. The integration length of a weighed integration is determined by the weight length.

[13]:
```
# Sequence program.
seq_prog = """
      move            0,R0            #Loop iterator.
      move            0,R1            #Weight for path 0.
      move            1,R2            #Weight for path 1.
      nop

loop: acquire_weighed 1,R0,R1,R2,1200 #Acquire bins and store them in "weighed
↪" acquisition.
      add             R0,1,R0         #Increment iterator
      nop                             #Wait a cycle for R0 to be available.
      jlt             R0,10,@loop     #Run until number of iterations is done.

      stop                            #Stop.
"""
```

[14]:
```
# Add sequence program, waveforms, weights and acquistitions to single␣
↪dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms_weights,
    "weights": waveforms_weights,
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

[15]:
```
# Upload sequence.
qrm.sequencer0.sequence("sequence.json")
```

Let's start the sequence and retrieve the results.

[16]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0, 1))
```

```
Status:
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

[17]:
```
# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Get acquisition list from instrument.
weighed_acq = qrm.get_acquisitions(0)["weighed"]
```

To check if the integration results match with what we expect, we need to divide the integration results by the integration length again. In this case the integration length is determined by the length of the weights.

[18]:
```
int_len = waveform_weight_length
bins = weighed_acq["acquisition"]["bins"]
bins["integration"]["path0"] = [(val / int_len) for val in bins["integration"][
↪"path0"]]
bins["integration"]["path1"] = [(val / int_len) for val in bins["integration"][
↪"path1"]]
bins
```

[18]:
```
{'integration': {'path0': [0.0838375615631583,
   0.08379245402794945,
   0.0838248975170451,
   0.08380893490833831,
   0.08373342157077765,
   0.0837548154917014,
   0.08381608371642879,
   0.08383202245997753,
   0.08376547844431308,
   0.08379022621544653],
  'path1': [-5.6059653985437636e-05,
   -3.018027660831874e-05,
   3.383960084091271e-05,
   6.327051148681764e-06,
   -1.1236511909867344e-06,
   1.715626436135604e-05,
   -0.00011699814177924457,
   -5.624102918653143e-05,
   -3.3345592069512524e-05,
   -0.00010118747457863478]},
 'threshold': [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
```

(continues on next page)

(continued from previous page)

```
 'avg_cnt': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

1.18.9 Large number of bins

The QRM supports up to 131072 bins. To show that, we need to change the program slightly again. We will use the non-weighed acquisition program, but now we will loop over the maximum number of acquisitions while storing each result in a separate bin.

[19]:
```
# Sequence program.
seq_prog = """
      move    0,R0            #Loop iterator.
      nop

loop: acquire 2,R0,1200       #Acquire bins and store them in "large"␣
↪acquisition.
      add     R0,1,R0         #Increment iterator
      nop                     #Wait a cycle for R0 to be available.
      jlt     R0,131072,@loop #Run until number of iterations is done.

      stop                    #Stop.
"""
```

[20]:
```
# Add sequence program, waveforms, weights and acquistitions to single␣
↪dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms_weights,
    "weights": waveforms_weights,
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

[21]:
```
# Upload sequence.
qrm.sequencer0.sequence("sequence.json")
```

Let's start the sequence and retrieve the results.

[22]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0, 1))
```

```
Status:
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

[23]:
```
# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Get acquisition list from instrument.
large_acq = qrm.get_acquisitions(0)["large"]
```

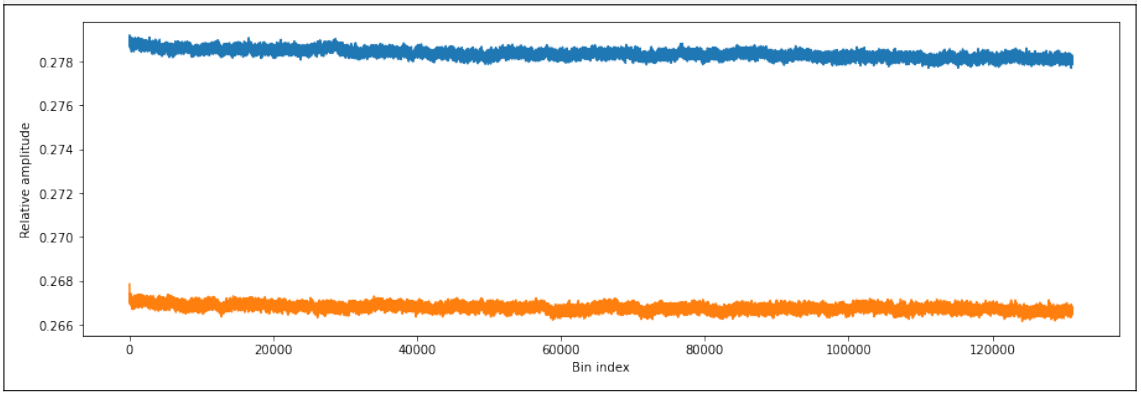
Since the number of bins is now to large to simply print, we will check the number of bins and we will check the bins for NaN values which indicate that a bin is not written.

[24]:
```
int_len = qrm.sequencer0.integration_length_acq()
bins = large_acq["acquisition"]["bins"]
bins["integration"]["path0"] = [(val / int_len) for val in bins["integration"][
↪"path0"]]
bins["integration"]["path1"] = [(val / int_len) for val in bins["integration"][
↪"path1"]]

print("Number of bins: {}".format(len(bins["avg_cnt"])))
for it, val in enumerate(bins["integration"]["path0"]):
    if math.isnan(val):
        Exception("NaN found at index {}.".format(it))
for it, val in enumerate(bins["integration"]["path0"]):
    if math.isnan(val):
        Exception("NaN found at index {}.".format(it))
for it, val in enumerate(bins["integration"]["path0"]):
    if math.isnan(val):
        Exception("NaN found at index {}.".format(it))
print("All values are valid.")
```

```
Number of bins: 131072
All values are valid.
```

We will also plot the integration results in every bin to visualize the contents.

[25]:
```
# Plot bins
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(bins["integration"]["path0"])
ax.plot(bins["integration"]["path1"])
ax.set_xlabel("Bin index")
ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



1.18.10 Averaging

As you might have noticed, the acquisition results also contain an average counter. This average counter reflects the number of times a bin as been averaged during the sequence. Each time a the sequence writes to the same bin the results are automatically accumulated and the average counter is increased. Upon retrieval of the acquisition results, each result is divided by the average counter and therefore automatically averaged. To show this, we will change the sequence one last time. This time we will average 10 bins a 1000 times each.

[26]:
```
# Sequence program.
seq_prog = """
      move    0,R1         #Average iterator.

avg:  move    0,R0         #Bin iterator.
      nop

loop: acquire 3,R0,1200    #Acquire bins and store them in "avg" acquisition.
      add     R0,1,R0      #Increment bin iterator
      nop                  #Wait a cycle for R0 to be available.
      jlt     R0,10,@loop  #Run until number of avg iterations is done.
      add     R1,1,R1      #Increment avg iterator
      nop                  #Wait a cycle for R1 to be available.
      jlt     R1,1000,@avg #Run until number of average iterations is done.

      stop                 #Stop.
"""
```

[27]:
```
# Add sequence program, waveforms, weights and acquistitions to single␣
↪dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms_weights,
    "weights": waveforms_weights,
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

[28]:
```
# Upload sequence.
qrm.sequencer0.sequence("sequence.json")
```

Let's start the sequence and retrieve the results.

[29]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0, 1))
```

```
Status:
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

Note that the average count of each bin is now set to a 1000.

[30]:
```
# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Get acquisition list from instrument.
avg_acq = qrm.get_acquisitions(0)["avg"]
```

[31]:
```
int_len = qrm.sequencer0.integration_length_acq()
bins = avg_acq["acquisition"]["bins"]
bins["integration"]["path0"] = [(val / int_len) for val in bins["integration"][
↪"path0"]]
bins["integration"]["path1"] = [(val / int_len) for val in bins["integration"][
↪"path1"]]
bins
```

[31]:
```
{'integration': {'path0': [0.2783555486077186,
   0.2783564631167562,
   0.2783504118221788,
   0.2783517176355642,
   0.27834840449438203,
   0.27835067464582314,
   0.2783542623351246,
   0.2783474118221788,
   0.2783547987298486,
   0.2783543351245726],
  'path1': [0.26682932535417686,
   0.2668245642403517,
   0.26682411724474836,
   0.2668218031265267,
   0.2668268925256473,
   0.26682839179286766,
   0.2668272222765022,
   0.2668283277967758,
   0.2668281695163654,
   0.2668296311675623]},
 'threshold': [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
 'avg_cnt': [1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000]}
```

1.18.11 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Stop sequencer.
qrm.stop_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0))
print()
```

(continues on next page)

(continued from previous page)

```
# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qrm.print_readable_snapshot(update=True)

# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`nco_control.ipynb`

1.19 Numerically Controlled Oscillator

In this tutorial we will demonstrate how to use the numerically controlled oscillator (NCO) during an experiment by: 1. *Changing the modulation frequency*, e.g. for rapid spectroscopy measurements 2. *Setting the phase*, e.g. for virtual Z gates

This tutorial has multiple sections showcasing how the hardware behaves in certain edge cases. These are optional and can be skipped.

We will show this by using a QRM and directly connecting outputs $O^{[1-2]}$ to inputs $I^{[1-2]}$ respectively. We will then use the QRM's sequencers to sequence waveforms on the outputs and simultaneously acquire the resulting waveforms on the inputs.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.19.1 Getting Started

How the NCO works

Modulation

The NCO modulates any AWG output at a certain frequency, such that we only need to store the envelope of any waveform to create a wave with the correct frequency and phase. To enable IQ up- and downconversion to RF frequencies, the device will generate a phase shifted signal on both signal paths. If modulation is enabled, the value of the NCO will be multiplied with the awg output and forwarded to path 0/1 as follows:

$$\text{path}_{0,\text{out}} = \frac{1}{\sqrt{2}}(\cos(\omega t)\text{awg}_0 + \sin(\omega t)\text{awg}_1) \tag{1}$$

$$\text{path}_{1,\text{out}} = \frac{1}{\sqrt{2}}(-\sin(\omega t)\text{awg}_0 + \cos(\omega t)\text{awg}_1) \tag{2}$$

Note the factor $\sqrt{2}$, which is required to preserve the total signal amplitude and prevents clipping. These two outputs can then be used within a QRM-RF module or by an external mixer to generate RF pulses.

Demodulation

Usually, we are interested in the envelope of the acquired signal instead of the oscillating values, in particular when integrating. Therefore, the NCO can also be used to demodulate the signal before integration. If demodulation is enabled, the signal is again multiplied with the NCO, this time with a prefactor $\sqrt{2}$. Thus, if we modulate and demodulate the same signal, we obtain the original awg output:

$$\text{path}_{0,\text{in}} = \sqrt{2}(\cos(\omega t)\text{in}_0 - \sin(\omega t)\text{in}_1) = \cos(\omega t)\cos(\omega t)\text{awg}_0 + \sin(\omega t)\sin(\omega t)\text{awg}_0 = \text{awg}_0 \quad (3)$$

$$\text{path}_{1,\text{in}} = \sqrt{2}(\cos(\omega t)\text{in}_1 + \sin(\omega t)\text{in}_0) = \cos(\omega t)\cos(\omega t)\text{awg}_1 + \sin(\omega t)\sin(\omega t)\text{awg}_1 = \text{awg}_1 \quad (4)$$

Setup

First, we are going to import the required packages and connect to the instrument.

[1]:
```
# Import ipython widgets
import json

import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np

# Set up the environment.
from IPython.display import display
from scipy.signal import spectrogram, welch, gaussian

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package.

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
display(connect)
device_name = connect.value
```

```
Dropdown(description='Select Device', options=('cluster-mm',), value='cluster-
↪mm')
```

Pulsar QRM

If you chose the Pulsar QRM, run the following cell. Skip to the *Cluster QRM section* if you selected a Cluster module.

[3]:
```
# Close existing connections to the Pulsar modules
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qrm = Pulsar(f"{device_name}", ip_address)
qrm.reset()
cluster = None  # In absence of a cluster
print(f"{device_name} connected at {ip_address}")
print(qrm.get_system_state())
```

```
---------------------------------------------------------------------------
ConnectionError                           Traceback (most recent call last)
Cell In [3], line 10
      7 ip_address = device_list[device_keys[device_number]]["identity"]["ip"]
      9 # Connect to device and reset
---> 10 qrm = Pulsar(f"{device_name}", ip_address, debug=True)
     11 qrm.reset()
     12 cluster = None  # In absence of a cluster

File c:\Users\David Quinn\anaconda3\envs\quantify-env\lib\site-packages\qcodes\
↪instrument\base.py:517, in AbstractInstrumentMeta.__call__(cls, *args,␣
↪**kwargs)
    512 def __call__(cls, *args: Any, **kwargs: Any) -> Any:
    513     """
    514     Overloads `type.__call__` to add code that runs only if __init__␣
↪completes
    515     successfully.
    516     """
--> 517     new_inst = super().__call__(*args, **kwargs)
    518     is_abstract = new_inst._is_abstract()
    519     if is_abstract:

File c:\users\david quinn\documents\qblox tutorials\qblox_instruments\qblox_
↪instruments\qcodes_drivers\pulsar.py:64, in Pulsar.__init__(self, name,␣
↪identifier, port, debug, dummy_type)
     62 if identifier is None:
     63     identifier = name
---> 64 super().__init__(identifier, port, debug, dummy_type)
     65 Instrument.__init__(self, name)
     67 # Add QCoDeS parameters

File c:\users\david quinn\documents\qblox tutorials\qblox_instruments\qblox_
↪instruments\native\pulsar.py:118, in Pulsar.__init__(self, identifier, port,␣
↪debug, dummy_type)
```

(continues on next page)

(continued from previous page)

```
    113 instrument_types = {
    114     InstrumentType.QCM: PulsarQcm,
    115     InstrumentType.QRM: PulsarQrm,
    116 }
    117 if self.instrument_type not in instrument_types:
--> 118     raise ConnectionError(
    119         "Unsupported instrument type detected ({})".format(self.
↪instrument_type)
    120     )
    121 intrument = instrument_types[self.instrument_type]
    122 self._scpi = intrument(self._transport, debug)

ConnectionError: Unsupported instrument type detected (MM)
```

Skip to the next section *Frequency sweeps* if you are not using a cluster.

Cluster QRM

First we connect to the Cluster using its IP address. Go to the *Pulsar QRM section* if you are using a Pulsar.

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.0.2
```

We then find all available cluster modules to connect to them individually.

[4]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]
```

(continues on next page)

(continued from previous page)

```
# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QRM module from the available modules in your Cluster:")
display(connect_qxm)
```

```
{'module8': 'QRM', 'module11': 'QRM-RF'}

Select the QRM module from the available modules in your Cluster:
```

```
Dropdown(options=('module8', 'module11'), value='module8')
```

Finally, we connect to the selected Cluster module.

[5]:
```
# Connect to the cluster QRM
qrm = getattr(
    cluster, connect_qxm.value
)  # Connect to the module that you have chosen above
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QRM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

1.19.2 Frequency sweeps

One of the most common experiments is to test the frequency response of the system, e.g. to find the resonance frequencies of a qubit or a resonator. For the purpose of this tutorial, we will sweep the full frequency range supported by the QRM. To improve accuracy we can use the maximum integration time and multiple averages. This does not change the overall measurement time much, as most of it is used for the setup.

[6]:
```
start_freq = -500e6
stop_freq = 500e6

n_averages = 10
MAXIMUM_SCOPE_ACQUISITION_LENGTH  = 16384
```

In this tutorial, we will analyze the raw data measured by the scope acquisition of the QRM. For this we will define a simple helper function using scipy.signal.spectrogram and scipy.signal.welch. The spectrogram shows the frequency spectrum of the QRM output as a function of time, to visualize the frequency sweeps we are doing. Welch's method is used to compute the input power as a function of frequency (power spectral density). This way we obtain the response of the system to find features of interest, e.g. a resonance.

[7]:
```
# Power as function of frequency and time by chunking the data
def plot_spectrogram(time_series: np.ndarray) -> None:
    f_sample = 1e9  # All devices have 1 GSPS sample rate
    fig, ax = plt.subplots(1, 2)

    f, t, Sxx = spectrogram(time_series, f_sample, return_onesided=False,␣
```

(continues on next page)

(continued from previous page)

```
↪detrend=False)

    idx = np.argsort(f)
    f = f[idx] / 1e6
    Sxx = Sxx[idx]

    spec = ax[0].pcolormesh(t, f, Sxx, shading="auto", cmap="YlOrRd")
    cb = fig.colorbar(spec)
    cb.set_label("Power Spectral Density [V$^2$/Hz]")
    ax[0].set_ylabel("Frequency [MHz]")
    ax[0].set_xlabel("Time [s]")

    f, Pxx = welch(time_series, f_sample, return_onesided=False, detrend=False)

    idx = np.argsort(f)
    f = f[idx] / 1e6
    Pxx = Pxx[idx]

    ax[1].semilogy(f, Pxx)
    ax[1].set_xlabel("Frequency [MHz]")
    ax[1].set_ylabel("Power Spectral Density [V$^2$/Hz]")
    fig.tight_layout()
    plt.show()
```

And two more helper functions for plotting the amplitude of an array of I, Q values and a scope acquisition:

[8]:
```
def plot_amplitude(x, I_data, Q_data):
    amplitude = np.abs(I_data + 1j * Q_data)

    plt.plot(x/1e6, amplitude)
    plt.xlabel("Frequency [MHz]")
    plt.ylabel("Integration [V]")
    plt.show()

def plot_scope(trace, t_min:int, t_max:int):
    x = np.arange(t_min, t_max)
    plt.plot(x, np.real(trace[t_min:t_max]))
    plt.plot(x, np.imag(trace[t_min:t_max]))
    plt.ylabel("Scope [V]")
    plt.xlabel("Time [ns]")
    plt.show()
```

Setting up the QRM

We set up a modulated DC offset:

[9]:
```
# Configure the channel map
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)

# Set DC Offset
qrm.sequencer0.offset_awg_path0(1)
qrm.sequencer0.offset_awg_path1(1)

# Enable modulation and demodulation. Note that the scope is not demodulated
qrm.sequencer0.mod_en_awg(True)
qrm.sequencer0.demod_en_acq(True)

# Enable hardware averaging for the scope
qrm.scope_acq_avg_mode_en_path0(True)
qrm.scope_acq_avg_mode_en_path1(True)

qrm.sequencer0.integration_length_acq(MAXIMUM_SCOPE_ACQUISITION_LENGTH)
```

NCO sweep controlled by the host computer

As a baseline, we will run a simple frequency sweep controlled by the host computer using QCoDeS. We do this by setting a DC offset and modulating it with the NCO. This is quite slow, so we will only measure 200 steps to keep the measurement time reasonable.

[10]:
```
n_steps = 200

step_freq = (stop_freq - start_freq) / n_steps
print(f"Step size {step_freq/1e6} MHz")

nco_sweep_range = np.arange(start_freq, stop_freq, step_freq)
```

```
Step size 5.0 MHz
```

Now we set up a simple program that acquires data and averages. The frequency will be set later with the QCoDeS interface.

[11]:
```
acquisitions = {"acq": {"num_bins": 1, "index": 0}}

# Sequence program.
slow_sweep = f"""
    move    {n_averages}, R0            # Average iterator.

avg_loop:
    reset_ph
    upd_param 200
    acquire 0, 0, {MAXIMUM_SCOPE_ACQUISITION_LENGTH}

    loop    R0, @avg_loop
```

(continues on next page)

(continued from previous page)

```
    stop
"""

# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": {},
    "weights": {},
    "acquisitions": acquisitions,
    "program": slow_sweep,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

Next, we prepare the QRM for the measurement:

[12]:
```
qrm.sequencer0.sequence("sequence.json")
```

Now we can run the frequency sweep. This is simply a loop where we set the frequency with QCoDeS and then run the program defined above. We measure the run time using the `%%time` Jupyter magic command.

[13]:
```
%%time
data = []
for nco_val in nco_sweep_range:
    # Set the frequency
    qrm.sequencer0.nco_freq(nco_val)

    # Run the program
    qrm.arm_sequencer(0)
    qrm.start_sequencer()

    # Wait for the sequencer to stop with a timeout period of one minute.
    qrm.get_acquisition_state(0, 1)

    # Move acquisition data from temporary memory to acquisition list.
    qrm.store_scope_acquisition(0, "acq")

    # Get acquisition list from instrument.
    data.append(qrm.get_acquisitions(0)["acq"])

    # Clear acquisition data so we do not average the results from different␣
↪frequencies
    qrm.sequencer0.delete_acquisition_data("acq")
```

```
CPU times: total: 2.58 s
Wall time: 10.1 s
```

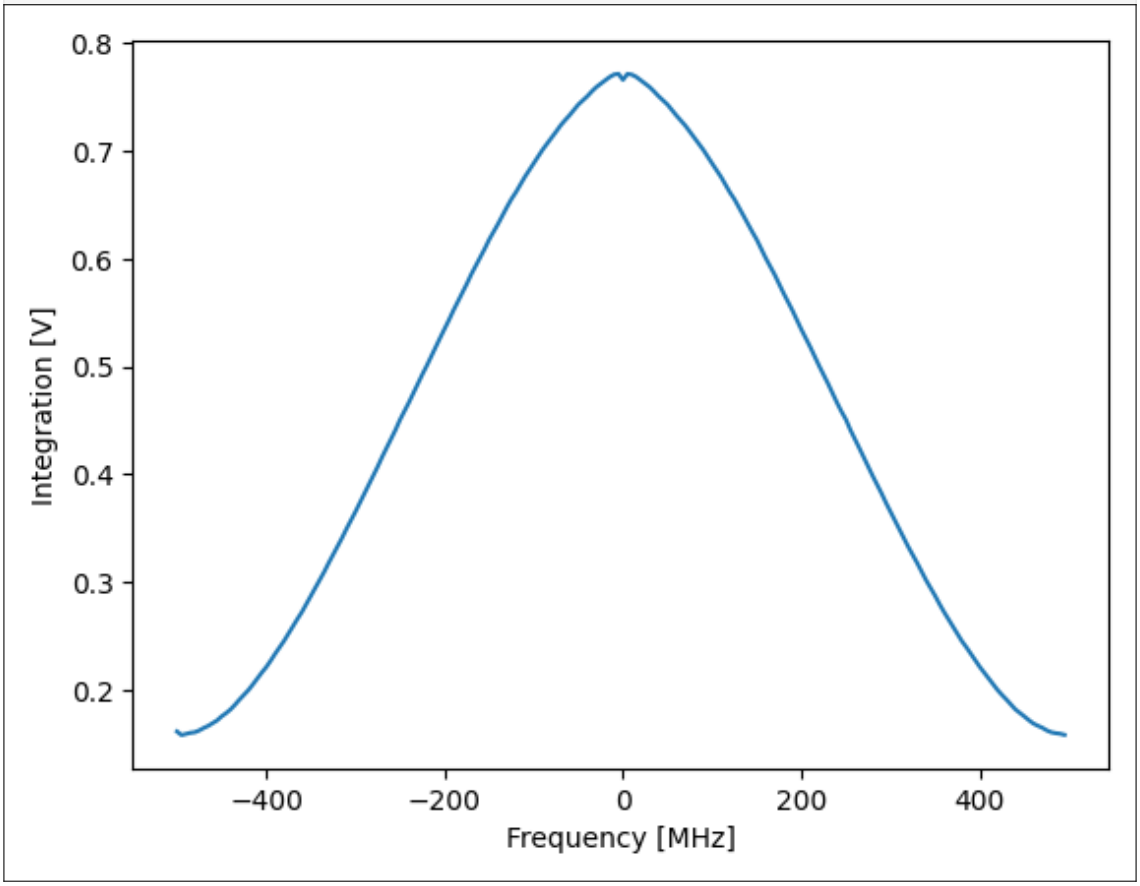
Plotting the acquired integration data, we can see the frequency behavior of the QRM.

[14]:
```
I_data = (
    np.asarray([d["acquisition"]["bins"]["integration"]["path0"][0] for d in␣
```

(continues on next page)

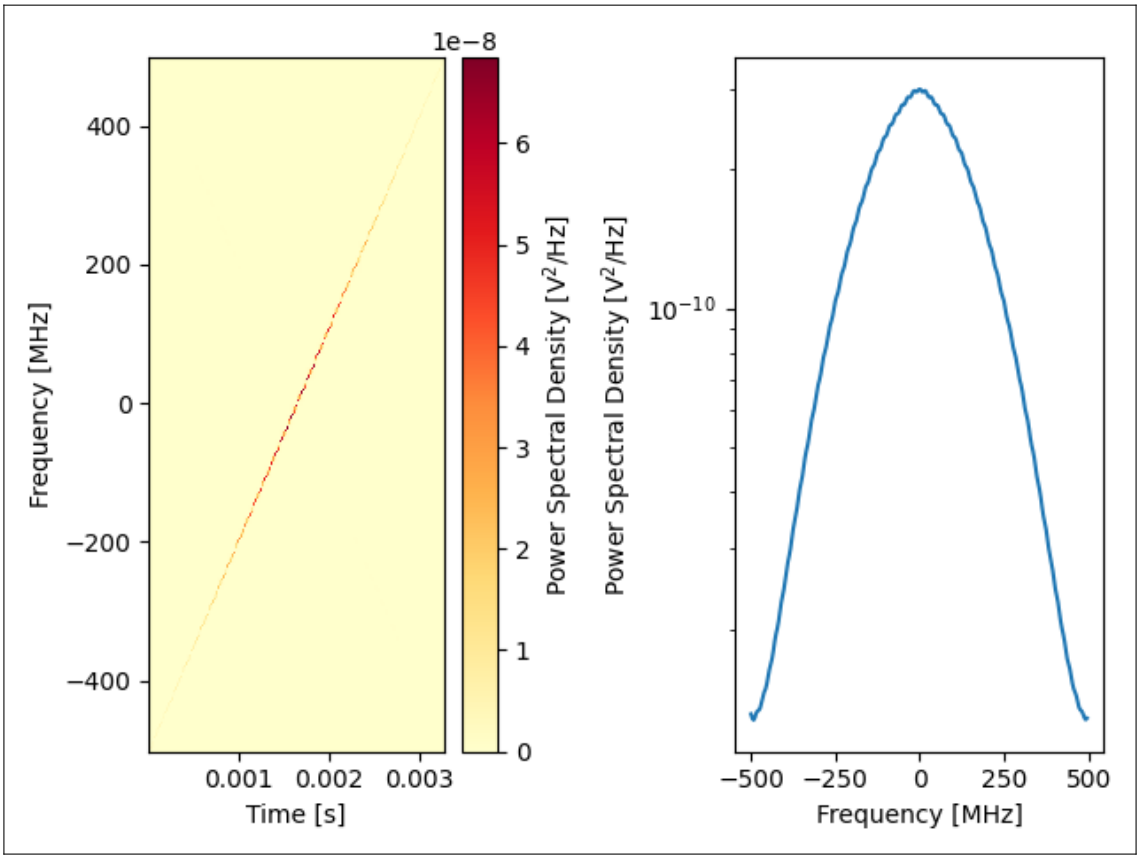
(continued from previous page)

```
↪data])
    / MAXIMUM_SCOPE_ACQUISITION_LENGTH
)
Q_data = (
    np.asarray([d["acquisition"]["bins"]["integration"]["path1"][0] for d in␣
↪data])
    / MAXIMUM_SCOPE_ACQUISITION_LENGTH
)
plot_amplitude(nco_sweep_range, I_data, Q_data)
```



We can see that the output amplitude decreases with frequency, this is expected due to the analog filters. We can also analyze the accumulated scope data with a spectrogram. This takes a few seconds, as there are 16384 data points per frequency step. Note that the time axis of the spectrogram refers to measurement time (16.4us * 200 steps ≈ 3.3ms) and not the wall clock time, which is significantly longer.

[15]:
```
data_scope = (
    np.asarray([d["acquisition"]["scope"]["path0"]["data"] for d in data]).
↪flatten()
    + 1j
    * np.asarray([d["acquisition"]["scope"]["path1"]["data"] for d in data]).
↪flatten()
)

plot_spectrogram(data_scope)
```



Spectroscopy using Q1ASM

Now we will run the same spectroscopy experiment using Q1ASM to change the NCO frequency in real time. First, we set up the QRM for continuous wave output and binned acquisition with many bins. This is significantly faster than using QCoDeS. The maximum number of points that can be measured this way is 131072 per sequencer, which is the maximum number of bins.

The sequencer program can fundamentally only support integer values. However, the NCO has a frequency resolution of 0.25 Hz and supports 10^9 phase values. Therefore, frequencies in the sequencer program must be given as integer multiple of $1/4$ Hz, and phases as integer multiple of $360/10^9$ degree.

[16]:
```
n_steps = 200

step_freq = (stop_freq - start_freq) / n_steps
print(f"{n_steps} steps with step size {step_freq/1e6} MHz")

# Convert frequencies to multiples of 0.25 Hz
nco_int_start_freq = int(4 * start_freq)
nco_int_step_freq = int(4 * step_freq)
```

```
200 steps with step size 5.0 MHz
```

[17]:
```
acquisitions = {"acq": {"num_bins": n_steps, "index": 0}}

setup = f"""
```

(continues on next page)

(continued from previous page)

```
    move {n_averages}, R2

avg_loop:
    move    0, R0           # frequency
    move    0, R1           # step counter
"""

# To get a negative starting frequency, we substract a positive number from 0
if start_freq <= 0:
    setup += f"""
    sub R0, {-nco_int_start_freq}, R0
    """
else:
    setup += f"""
    add R0, {nco_int_start_freq}, R0
    """

spectroscopy = (
    setup
    + f"""
    reset_ph
    set_freq        R0
    upd_param       200

nco_set:
    set_freq        R0                          # Set the frequency
    add             R0, {nco_int_step_freq}, R0  # Update the frequency␣
↪register
    upd_param       200                         # Wait for time of flight
    acquire         0, R1, {MAXIMUM_SCOPE_ACQUISITION_LENGTH}
    add             R1, 1, R1
    nop
    jlt             R1, {n_steps}, @nco_set      # Loop over all frequencies

    loop            R2, @avg_loop

    stop                                    # Stop
"""
)

# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": {},
    "weights": {},
    "acquisitions": acquisitions,
    "program": spectroscopy,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
```

Now we prepare the QRM for measurement.

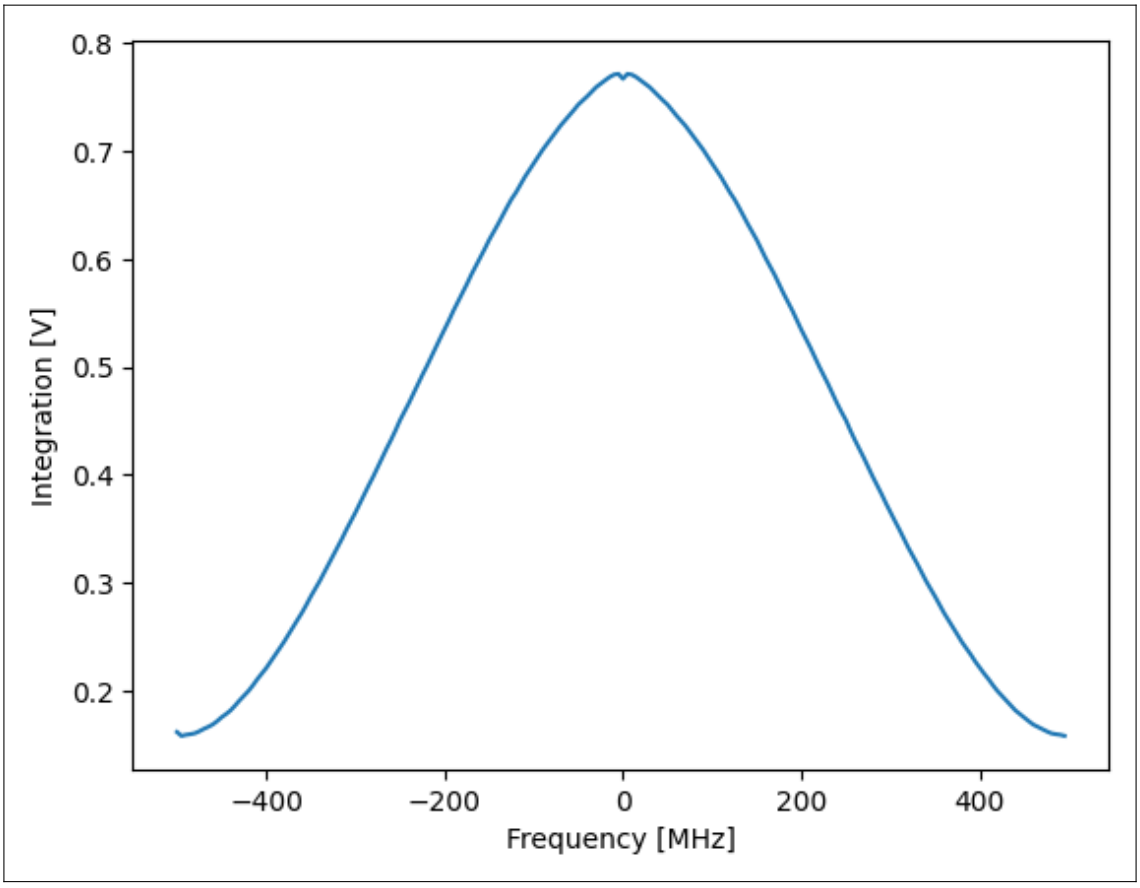
[18]:
```
qrm.sequencer0.sequence("sequence.json")
```

[19]:
```
%%time
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

data = qrm.get_acquisitions(0)["acq"]
```

```
CPU times: total: 156 ms
Wall time: 178 ms
```

Note that the same measurement as before is now two orders of magnitude faster. If we plot the integrated data, we get the same results as before.

[20]:
```
# For plotting, convert the NCO integer values back to frequencies
nco_sweep_range = (np.arange(n_steps)*nco_int_step_freq + nco_int_start_freq) /
↪ 4.0

I_data = (
    np.asarray(data["acquisition"]["bins"]["integration"]["path0"])
    / MAXIMUM_SCOPE_ACQUISITION_LENGTH
)
Q_data = (
    np.asarray(data["acquisition"]["bins"]["integration"]["path1"])
    / MAXIMUM_SCOPE_ACQUISITION_LENGTH
)
plot_amplitude(nco_sweep_range, I_data, Q_data)
```



NCO input delay compensation

By default, the input and output of the QRM are multiplied with the same NCO value. As the output path has a time of flight of about 146 ns between the NCO and playback, this means that there is a short time window after frequency/phase updates where demodulation is updated, but playback is still using the old value. There is also always a (fixed) relative phase between playback and demodulation. We can showcase this by using a similar program as before, but with less points, so that the frequency steps are more clearly visible.

[21]:
```
n_steps = 20
n_averages = 1000

step_freq = (stop_freq - start_freq) / n_steps
print(f"{n_steps} steps with step size {step_freq/1e6} MHz")

# Convert frequencies to multiples of 0.25 Hz
nco_int_start_freq = int(4 * start_freq)
nco_int_step_freq = int(4 * step_freq)

# For plotting, convert the NCO integer values back to frequencies
nco_sweep_range = np.arange(nco_int_start_freq, 4 * stop_freq, nco_int_step_
↪freq) / 4.0
```

```
20 steps with step size 50.0 MHz
```

To make the effect of NCO delay compensation more apparent, we modify the spectroscopy program for short integration time and acquire immediately after the frequency update, without waiting for time of flight. This means that the output at the new frequency only arrives at the input AFTER integration in the current loop iteration has finished. Without further modifications of the program this leads to an off-by-one error. Therefore, we increase the frequency as the first step in the loop.

[22]:
```
acquisitions = {"acq": {"num_bins": n_steps, "index": 0}}

setup = f"""
    move {n_averages}, R2

avg_loop:
    move    0, R0           # frequency
    move    0, R1           # step counter
"""

# To get a negative starting frequency, we substract a positive number from 0
if start_freq <= 0:
    setup += f"""
    sub R0, {-nco_int_start_freq}, R0
    """
else:
    setup += f"""
    add R0, {nco_int_start_freq}, R0
    """

spectroscopy = (
    setup
    + f"""
    reset_ph
    set_freq        R0
    upd_param       200

nco_set:
    # Due to time of flight, the new frequency will only arrive at the input␣
↪AFTER integration is done
    # Therefore, we already increase the frequency before the first␣
↪measurement.
    add             R0, {nco_int_step_freq}, R0
    nop
    set_freq        R0

    # we removed upd_param, so that acquisition starts the moment the␣
↪frequency is updated
    acquire         0, R1, 1200                 # Update the NCO and␣
↪immediately acquire
    add             R1, 1, R1
    nop
    jlt             R1, {n_steps}, @nco_set     # Loop over all frequencies

    loop            R2, @avg_loop

    stop                                # Stop
```

(continues on next page)

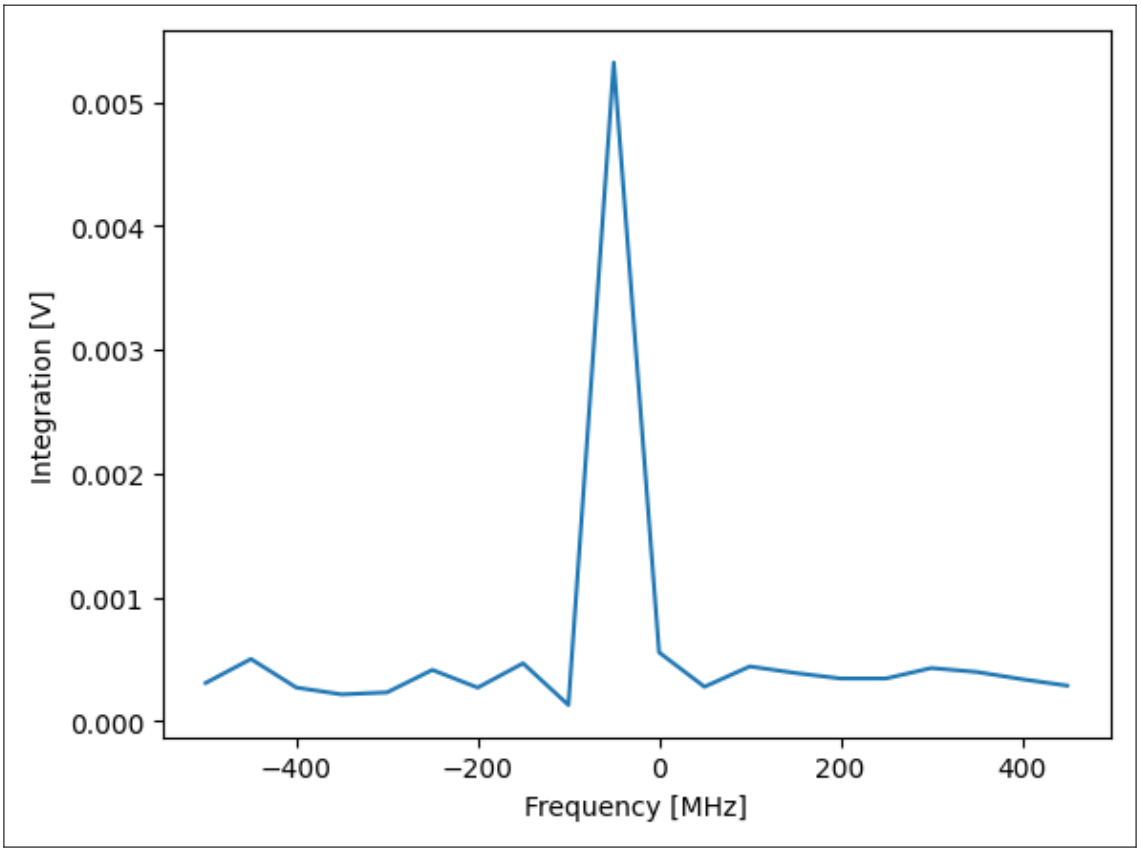
(continued from previous page)

```
"""
)

# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": {},
    "weights": {},
    "acquisitions": acquisitions,
    "program": spectroscopy,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
```

As a baseline, we will do a measurement with delay compensation disabled.

[23]:
```
qrm.sequencer0.sequence("sequence.json")
qrm.sequencer0.integration_length_acq(140)
qrm.sequencer0.nco_prop_delay_comp_en(False)
```

[24]:
```
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

data = qrm.get_acquisitions(0)["acq"]
I_data = (
    np.asarray(data["acquisition"]["bins"]["integration"]["path0"])
    / 140
)
Q_data = (
    np.asarray(data["acquisition"]["bins"]["integration"]["path1"])
    / 140
)
plot_amplitude(nco_sweep_range, I_data, Q_data)
```



Even though we only measured a small number of points, we can see that this is not compatible with the previous spectroscopy measurements. What happened is that `set_freq` updates the NCO frequency immediately. However, there is a time of flight of about 146 ns between the NCO and the output of the device. Thus, the signal will be demodulated at $f_0 + 100$ MHz immediately after `set_freq`, but the incoming signal is still modulated at f_0 for another 146 ns - longer than the integration time chosen above. The integrated signal will therefore be approximately zero.

Now we run the same experiment again, with delay compensation enabled.

[25]:
```
qrm.sequencer0.sequence("sequence.json")
qrm.sequencer0.nco_prop_delay_comp_en(True)
```

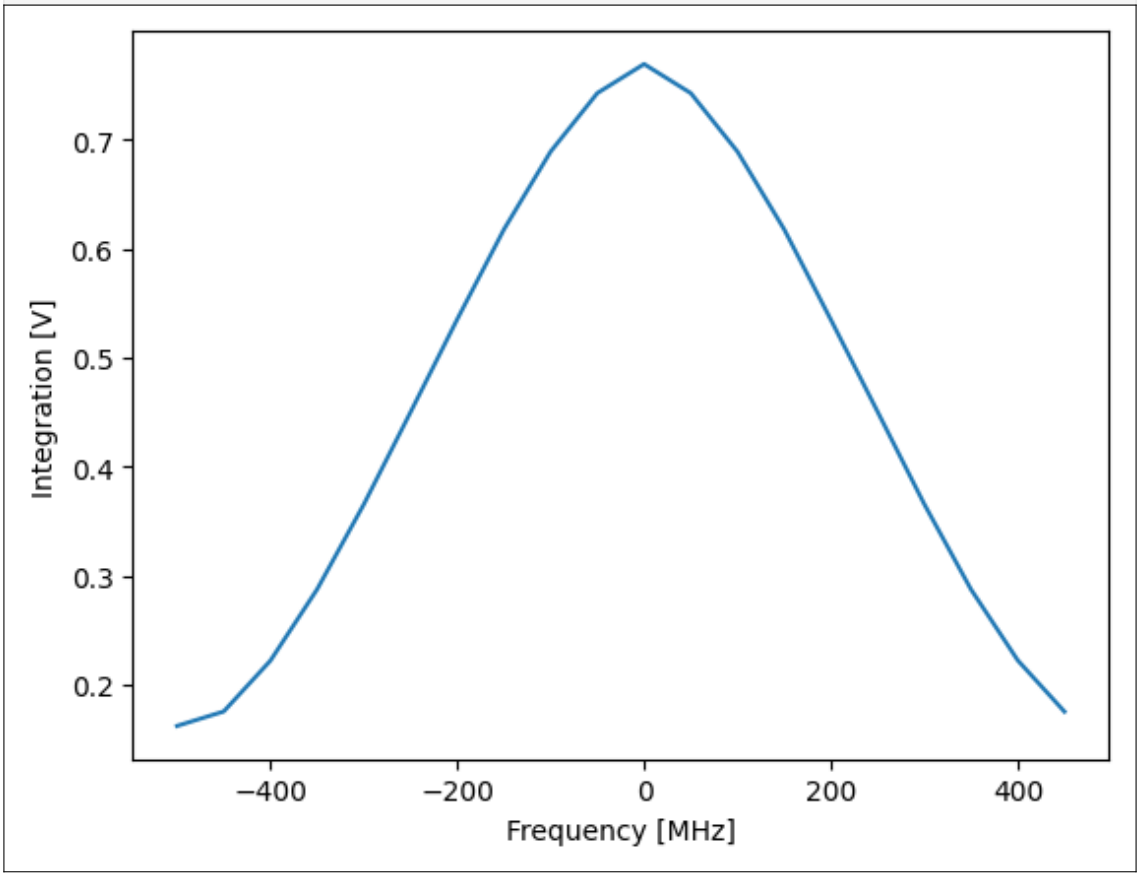
This ensures that, for demodulation, the NCO only updates after the time of flight, i.e. that frequency and phase of modulation and demodulation always match.

[26]:
```
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

data = qrm.get_acquisitions(0)["acq"]
I_data = (
    np.asarray(data["acquisition"]["bins"]["integration"]["path0"])
    / 140
)
Q_data = (
```

(continues on next page)

(continued from previous page)

```
    np.asarray(data["acquisition"]["bins"]["integration"]["path1"])
    / 140
)
amplitude = np.abs(I_data + 1j * Q_data)

plot_amplitude(nco_sweep_range, I_data, Q_data)
```



We can see that modulation and demodulation frequency now match, producing similar results as the spectroscopy measurements before.

Fast chirped pulses using Q1ASM

Now we will run a fast frequency sweep using Q1ASM to change the NCO frequency in real time during playback. This type of pulse is also called a chirp. First, we set up the QRM for continuous wave output and a single scope acquisition. This is significantly faster than spectroscopy, but also limits the maximum number of points we can measure in a single program. The total duration of the sweep must be 16384 ns or less, as that is the maximum length of a scope acquisition.

The sequencer program can fundamentally only support integer values. However, the NCO has a frequency resolution of 0.25 Hz and supports 10^9 phase values. Therefore, frequencies in the sequencer program must be given as an integer multiple of $1/4$ Hz, and phases as an integer multiple of $360/10^9$ degrees.

[27]:
```
step_time = 44  # Time per frequency step in ns. We can reduce this, but the␣
↪program needs to be changed. See next section
n_steps = int(16384 / step_time)
```

(continues on next page)

(continued from previous page)

```
n_averages = 10

step_freq = (stop_freq - start_freq) / n_steps
print(f"{n_steps} steps with step size {step_freq/1e6} MHz")

# Convert frequencies to multiples of 0.25 Hz
nco_int_start_freq = int(4 * start_freq)
nco_int_step_freq = int(4 * step_freq)

# For plotting, convert the NCO integer values back to frequencies
nco_sweep_range = np.arange(nco_int_start_freq, 4 * stop_freq, nco_int_step_
↪freq) / 4.0
```

```
372 steps with step size 2.6881720430107525 MHz
```

Now, we write a Q1ASM program that quickly changes the NCO's frequency, converting the continuous sine output into a chirp.

Internally, the processor stores negative values using two's complement. This has some implications for our program: - We cannot directly store a negative value in a register. Substracting a larger value from a smaller one works as expected though. - Immediate values are handled by the compiler, i.e. `set_freq -100` gives the expected result of -25 Hz. - Comparisons (`jlt, jge`) with registers storing a negative value do not work as expected, as the smallest negative number is **larger** than the largest positive number. To keep the program general we should therefore use `loop` instead.

[28]:
```
acquisitions = {"acq": {"num_bins": 1, "index": 0}}

setup = f"""
    move {n_averages}, R2

avg_loop:
    move    0, R0           # frequency
    move {n_steps}, R1      # n_steps
"""

# To get a negative starting frequency, we substract a positive number from 0
if start_freq <= 0:
    setup += f"""
    sub R0, {-nco_int_start_freq}, R0
    """
else:
    setup += f"""
    add R0, {nco_int_start_freq}, R0
    """

# Play a chirped pulse
chirp = (
    setup
    + f"""
    reset_ph
    set_freq 0
    upd_param 200
    acquire 0,0,4                          # Start acquisition. This is not␣
```

(continues on next page)

(continued from previous page)

```
↪blocking

nco_set:
    set_freq        R0                      # Set the frequency
    add             R0,{nco_int_step_freq}, R0  # Update the frequency register
    upd_param       {step_time}
    loop            R1, @nco_set            # Loop over all frequencies

    wait            10000
    loop            R2, @avg_loop

    stop                                    # Stop
"""
)

# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": {},
    "weights": {},
    "acquisitions": acquisitions,
    "program": chirp,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
```

[29]:
```
qrm.sequencer0.sequence("sequence.json")
qrm.sequencer0.integration_length_acq(MAXIMUM_SCOPE_ACQUISITION_LENGTH)
```

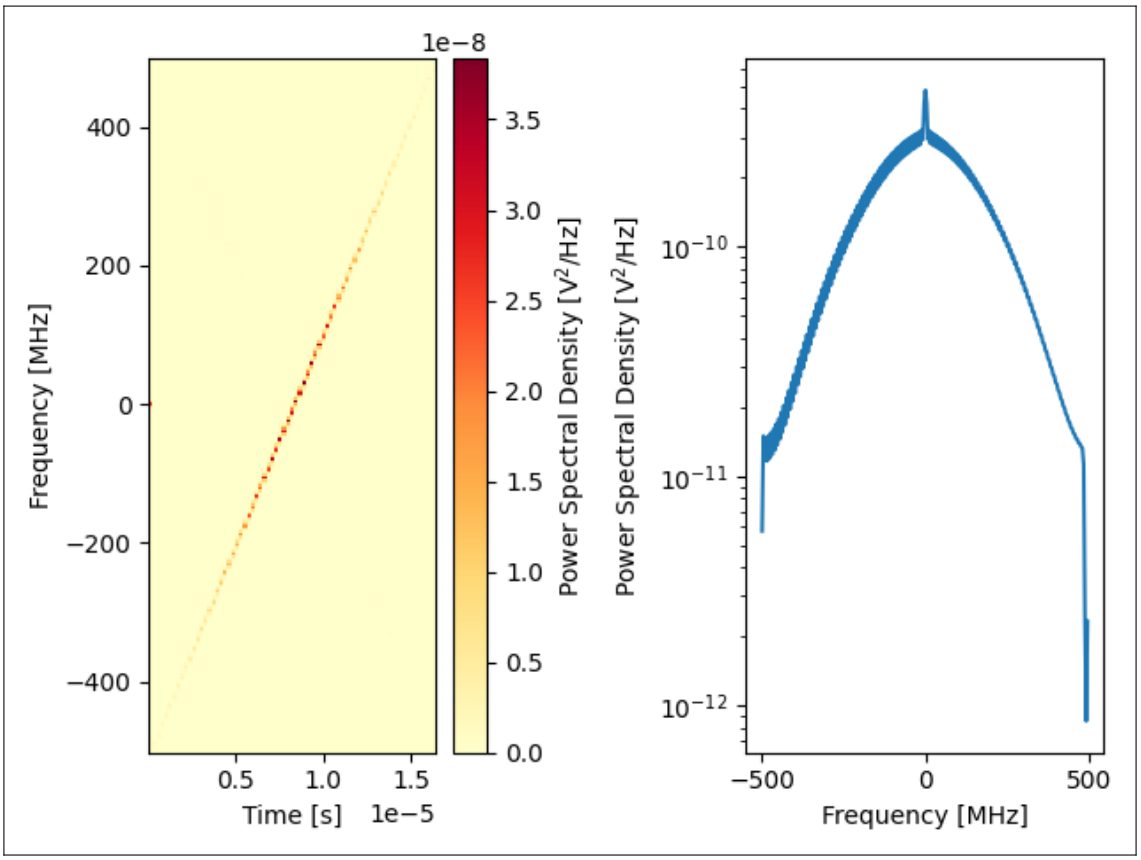
Run the program

[30]:
```
%%time
qrm.sequencer0.nco_freq(0)
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "acq")
data = qrm.get_acquisitions(0)["acq"]
```

```
CPU times: total: 15.6 ms
Wall time: 49.6 ms
```

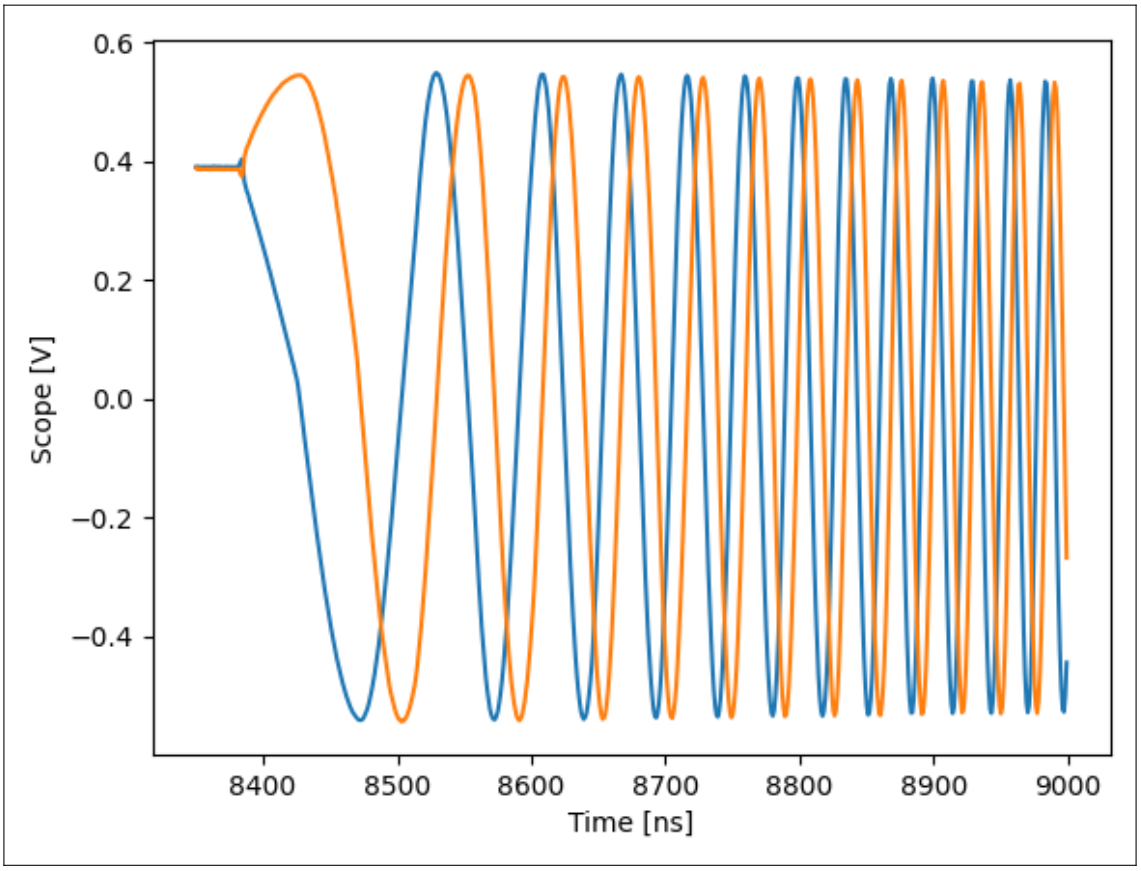
Note that this is significantly faster than standard spectroscopy with Q1ASM with a larger number of points. For this measurement, we only use the scope acquisition data. We can analyze it again with a spectrogram.

[31]:
```
trace = np.array(data["acquisition"]["scope"]["path0"]["data"]) + 1j * np.
↪array(
    data["acquisition"]["scope"]["path1"]["data"]
)
plot_spectrogram(trace)
```



Note the difference in timescale to before. In the spectrogram we can see the intended spectrum of the chirp. And finally, we can visualize the chirp pulse. For better clarity, we show the low frequency parts around 8400ns.

[32]:
```
plot_scope(trace, 8350, 9000)
```



Timings of the Q1 and realtime processors

The sequencer has a real-time pipeline that generates the output and a classical pipeline responsible for logic and filling the queue of the real-time pipeline. This queue is 32 instructions long and starts pre-filled when the sequencer is started. If the runtime of classical instructions is shorter than the corresponding real-time instructions, the sequencer will stop. See also in the documentation of the sequencer.

We can see this by running the same program as before, but with reduced time between frequency steps:

[33]:
```
step_time = 40  # this will cause stalling

n_steps = int(16384 / step_time)
step_freq = (stop_freq - start_freq) / n_steps
print(f"{n_steps} steps with step size {step_freq/1e6} MHz")

# Convert frequencies to multiples of 0.25 Hz
nco_int_start_freq = int(4 * start_freq)
nco_int_step_freq = int(4 * step_freq)

# For plotting, convert the NCO integer values back to frequencies
nco_sweep_range = np.arange(nco_int_start_freq, 4 * stop_freq, nco_int_step_
↪freq) / 4.0
```

```
409 steps with step size 2.4449877750611244 MHz
```

[34]:
```
acquisitions = {"acq": {"num_bins": 1, "index": 0}}

setup = f"""
    move {n_averages}, R2

avg_loop:
    move {n_steps}, R1  # n_steps
    move    0, R0          # frequency
    nop
"""

# To get a negative starting frequency, we substract a positive number from 0
if start_freq <= 0:
    setup += f"""
    sub R0, {-nco_int_start_freq}, R0
    """
else:
    setup += f"""
    add R0, {nco_int_start_freq}, R0
    """

# Play a chirped pulse
chirp = (
    setup
    + f"""
    reset_ph
    set_freq 0
    upd_param 200
    acquire 0,0,4                           # Start acquisition. This is not␣
↪blocking

nco_set:
    set_freq        R0                      # Set the frequency
    add             R0,{nco_int_step_freq}, R0  # Update the frequency register
    upd_param       {step_time}
    loop            R1, @nco_set            # Loop over all frequencies

    wait            10000
    loop            R2, @avg_loop

    stop                                    # Stop
"""
)

# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": {},
    "weights": {},
    "acquisitions": acquisitions,
    "program": chirp,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
```

(continues on next page)

(continued from previous page)

```
    file.close()
```

[35]:
```
qrm.sequencer0.sequence("sequence.json")

qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "acq")
data = qrm.get_acquisitions(0)["acq"]
```

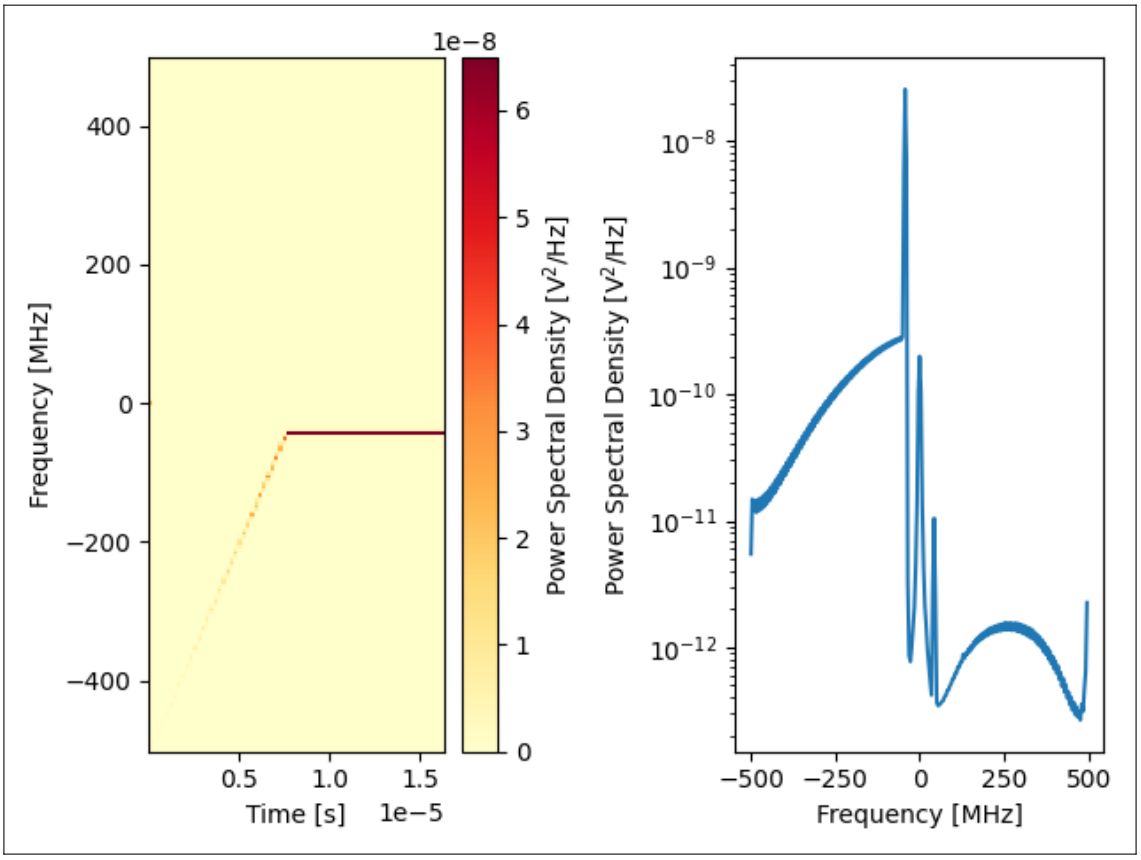
As can be seen from the red LEDs on the front of your Cluster/Pulsar, the Q1 processor has stalled, and the program stopped.

[36]:
```
print(qrm.sequencer0.get_sequencer_state())
```

```
Status: STOPPED, Flags: FORCED_STOP, SEQUENCE_PROCESSOR_RT_EXEC_COMMAND_
↪UNDERFLOW, ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_BINNING_DONE
```

We can also see on the scope that the chirp has been stopped prematurely.

[37]:
```
trace = np.array(data["acquisition"]["scope"][f"path0"]["data"]) + 1j * np.
↪array(
    data["acquisition"]["scope"][f"path1"]["data"]
)
plot_spectrogram(trace)
```



Before continuing, we clear the flags on the qrm:

[38]:
```
if 'pulsar' in device_name:
    qrm.clear()
else:
    cluster.clear()
```

If we want to the chirp to use as many updates as possible, we need to unroll the loop. This way, we can reduce the time per frequency step down to 24ns: `set_freq`, `upd_param` both take 4ns and `add` takes 12ns on the Q1 processor for a total of 20ns per frequency step.

[39]:
```
step_time = 20

n_steps = int(16384 / step_time)
step_freq = (stop_freq - start_freq) / n_steps
print(f"{n_steps} steps with step size {step_freq/1e6} MHz")

# Convert frequencies to multiples of 0.25 Hz
nco_int_start_freq = int(4 * start_freq)
nco_int_step_freq = int(4 * step_freq)

# For plotting, convert the NCO integer values back to frequencies
nco_sweep_range = np.arange(nco_int_start_freq, 4 * stop_freq, nco_int_step_
↪freq) / 4.0
```

```
819 steps with step size 1.221001221001221 MHz
```

This does not leave room for a loop, which would take an additional 24ns. However, we can use the instruction memory of the Q1 processor:

[40]:
```
chirp = f"""
    move {n_averages}, R2

avg_loop:
    move    0, R0           # frequency
    nop
"""

# To get a negative starting frequency, we substract a positive number from 0
if start_freq <= 0:
    chirp += f"""
    sub R0, {-nco_int_start_freq}, R0
    """
else:
    chirp += f"""
    add R0, {nco_int_start_freq}, R0
    """

chirp += f"""
    reset_ph
    set_freq 0
    upd_param 200
    acquire 0,0,4                       # Start acquisition. This is not␣
↪blocking

"""
# unroll the loop into individual commands
for _ in range(n_steps):
    chirp += f"""
    set_freq        R0                  # Set the frequency
    add             R0,{nco_int_step_freq}, R0  # Update the frequency register
    upd_param       {step_time}
"""
chirp += """

    wait            10000
    loop            R2, @avg_loop

    stop                                # Stop
"""
# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": {},
    "weights": {},
    "acquisitions": acquisitions,
    "program": chirp,
}
with open("sequence.json", "w", encoding="utf-8") as file:
```

(continues on next page)

(continued from previous page)

```
    json.dump(sequence, file, indent=4)
    file.close()
```

[41]:
```
qrm.sequencer0.sequence("sequence.json")
```

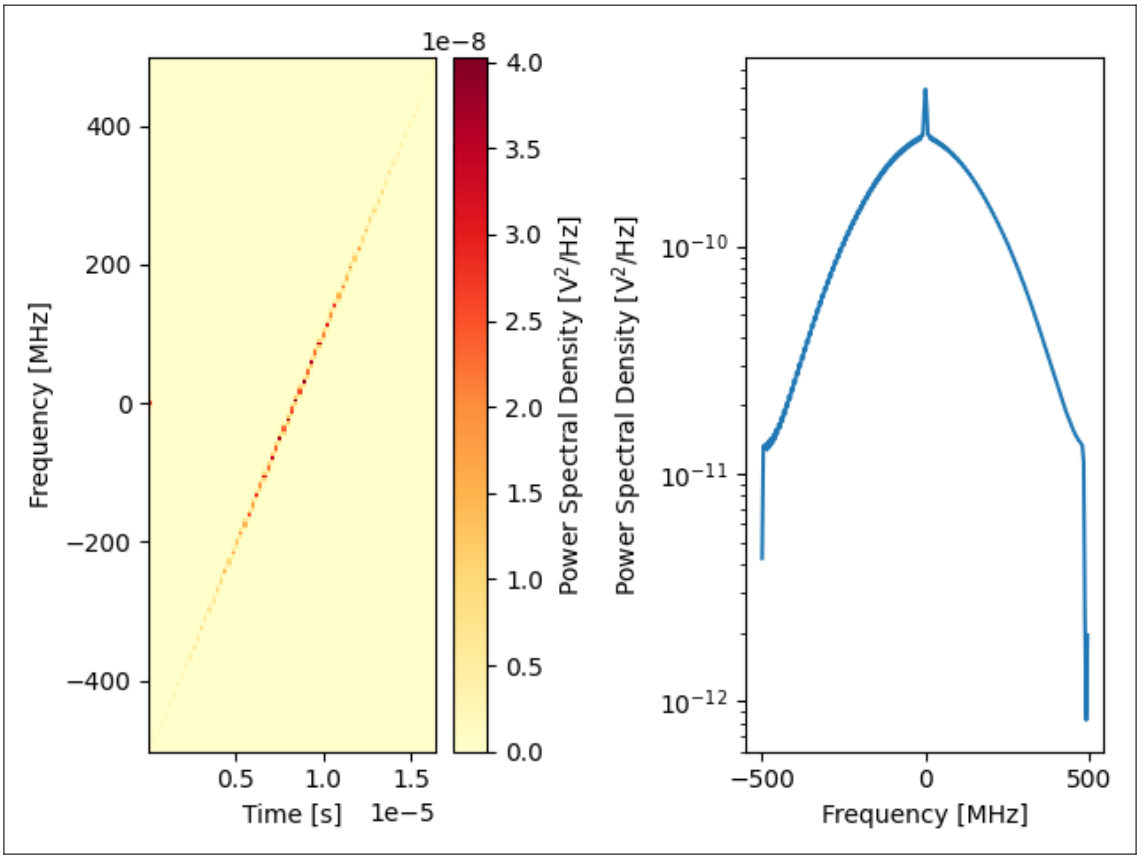
Now we execute the program and plot the spectrogram:

[42]:
```
%%time
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "acq")
data = qrm.get_acquisitions(0)["acq"]
```

```
CPU times: total: 15.6 ms
Wall time: 45.1 ms
```

Note that the execution time is comparable to the other chirped measurements, but with more than double the number of points.

[43]:
```
trace = np.array(data["acquisition"]["scope"][f"path0"]["data"]) + 1j * np.
↪array(
    data["acquisition"]["scope"][f"path1"]["data"]
)
plot_spectrogram(trace)
```



1.19.3 Phase updates

Virtual Z gates

In addition to fast frequency updates, the sequencer also supports real-time changes of the NCO phase. In particular for superconducting qubits, this can be used for a so-called virtual Z gate, see McKay et al. (2016). The virtual Z gate is a change of reference frame rather than a physical operation. Therefore, it is instantaneous and near perfect - the dominant error being that the NCO has a finite resolution of 10^9 different phases. Below, we will demonstrate how to to use a virtual Z to use the same pulse for both X and Y rotations.

As the sequencer internally only supports integer values, we must first convert the phase into an integer multiple of $360/10^9$ degree:

[44]:
```
int_90 = int(90 * (1e9 / 360))
int_270 = int(270 * (1e9 / 360))
```

[45]:
```
# Waveforms
waveform_len = 1000
waveforms = {
    "gaussian": {
        "data": gaussian(waveform_len, std=0.133 * waveform_len).tolist(),
        "index": 0,
    },
```

(continues on next page)

(continued from previous page)

```
}

# Acquisitions
acquisitions = {"scope": {"num_bins": 1, "index": 0}}

# Program
virtual_z = f"""
acquire         0,0,4
play            0,0,{waveform_len}      # X90

# This is equivalent to Y90, but uses the same waveform as X90
reset_ph
set_ph_delta    {int_90}     # Z90
play            0,0,{waveform_len}      # X90
set_ph_delta    {int_270}    # Z-90

stop
"""

# Write sequence to file.
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(
        {
            "waveforms": waveforms,
            "weights": {},
            "acquisitions": acquisitions,
            "program": virtual_z,
        },
        file,
        indent=4,
    )
    file.close()
```

For this experiment, we reset the QRM and enable modulation at a low frequency so that the effects are easily visible (without offsets).

[46]:
```
# Program sequencers.
if 'pulsar' in device_name:
    qrm.reset()
else:
    cluster.reset()

qrm.sequencer0.sequence("sequence.json")

# Configure sequencer
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)
qrm.sequencer0.nco_freq(3e6)

# Enable modulation
qrm.sequencer0.mod_en_awg(True)
qrm.sequencer0.demod_en_acq(True)
```

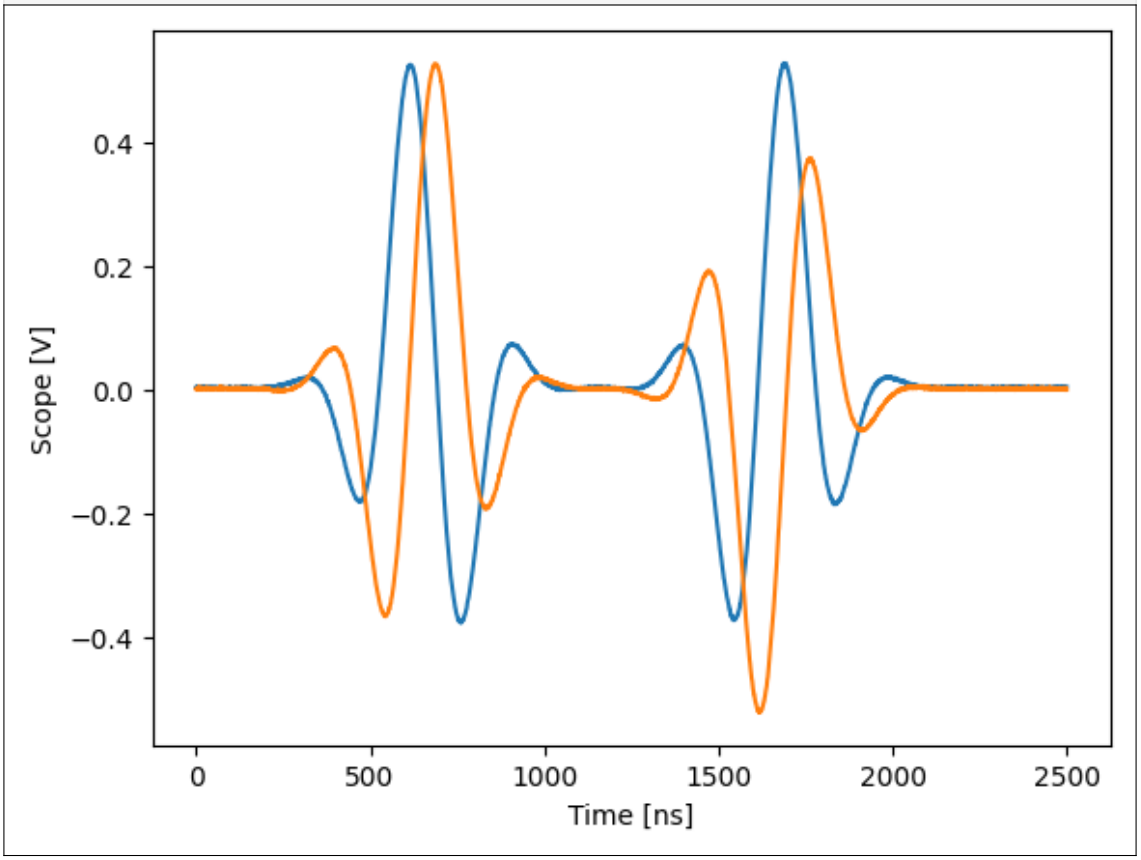
(continues on next page)

(continued from previous page)

```
# Enable hardware averaging for the scope
qrm.scope_acq_avg_mode_en_path0(True)
qrm.scope_acq_avg_mode_en_path1(True)

qrm.sequencer0.integration_length_acq(MAXIMUM_SCOPE_ACQUISITION_LENGTH)
```

Now we can run the program and look at the scope acquisition.

[47]:
```
# Start the sequence
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop
qrm.get_acquisition_state(0, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

trace = np.asarray(
    acq["scope"]["acquisition"]["scope"]["path0"]["data"]
) + 1j * np.asarray(acq["scope"]["acquisition"]["scope"]["path1"]["data"])

plot_scope(trace, 0, 2500)
```



Chirped pulses using the phase update

As an exercise combining the concepts from this notebook, we can also create a chirp using only phase updates. This is purely educational and should not be used in an experiment. Again, we do not use a loop, and use the instruction memory instead to make the step size smaller.

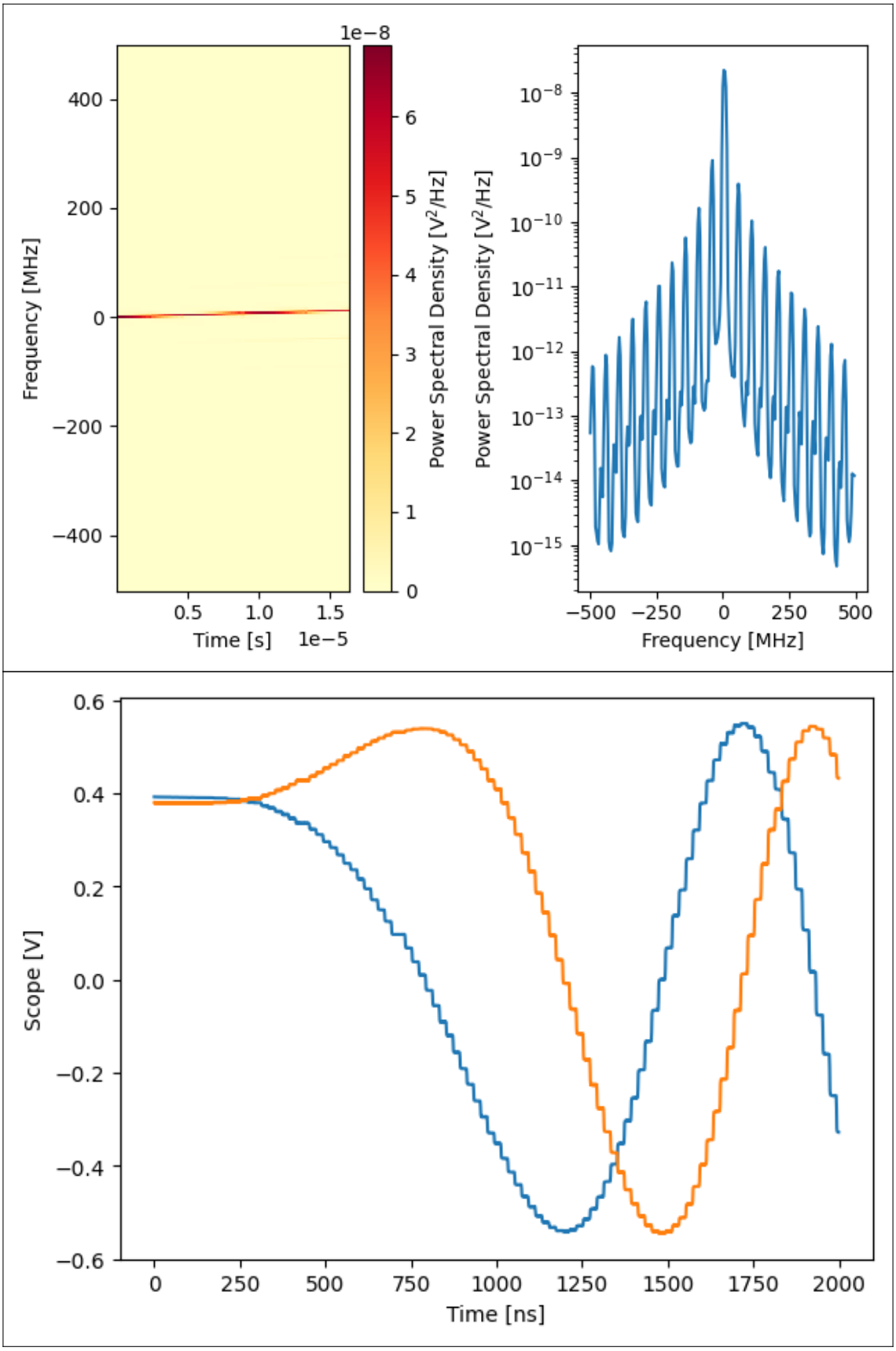
[48]:
```
n_averages = 100
n_steps = int(16384 / (20))

acquisitions = {"acq": {"num_bins": 1, "index": 0}}


phase_chirp = f"""
    move {n_averages}, R2

avg_loop:
    move    0, R0                           # phase update step size
    reset_ph
    set_freq 0
    upd_param 200
    acquire 0,0,4                           # Start acquisition. This is not␣
↪blocking

nco_set:
"""
# step the phase with increasing step size
for _ in range(n_steps):
    phase_chirp += f"""
    set_ph_delta    R0
    upd_param       20
    add             R0, {int(1e9/(4*n_steps))}, R0      # increase the
↪'frequency'
"""
phase_chirp += """
    wait            10000
    loop            R2, @avg_loop

    stop                                    # Stop
"""

# Write sequence to file.
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(
        {
            "waveforms": waveforms,
            "weights": {},
            "acquisitions": acquisitions,
            "program": phase_chirp,
        },
        file,
        indent=4,
    )
    file.close()
```

We can run this with the same settings as the chirps before:

[49]:
```
qrm.sequencer0.sequence("sequence.json")

# Set DC Offset
qrm.sequencer0.offset_awg_path0(1)
qrm.sequencer0.offset_awg_path1(1)

qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "acq")
data = qrm.get_acquisitions(0)["acq"]
```

In the spectrogram we can see a slow frequency sweep - plus high frequency components. The reason for those is easily visible in the scope, the "sine" wave is not smooth, but instead made up of many square pulses (as expected).

[50]:
```
trace = np.array(data["acquisition"]["scope"]["path0"]["data"]) + 1j * np.
↪array(
    data["acquisition"]["scope"]["path1"]["data"]
)
plot_spectrogram(trace)

plot_scope(trace, 0, 2000)
```


1e−8
400
200
0
−200
−400
Frequency [MHz]
0.5
1.0
1.5
Time [s]
1e−5
6
5
4
3
2
1
0
Power Spectral Density [V²/Hz]
10⁻⁸
10⁻⁹
10⁻¹⁰
10⁻¹¹
10⁻¹²
10⁻¹³
10⁻¹⁴
10⁻¹⁵
Power Spectral Density [V²/Hz]
−500
−250
0
250
500
Frequency [MHz]
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Scope [V]
0
250
500
750
1000
1250
1500
1750
2000
Time [ns]

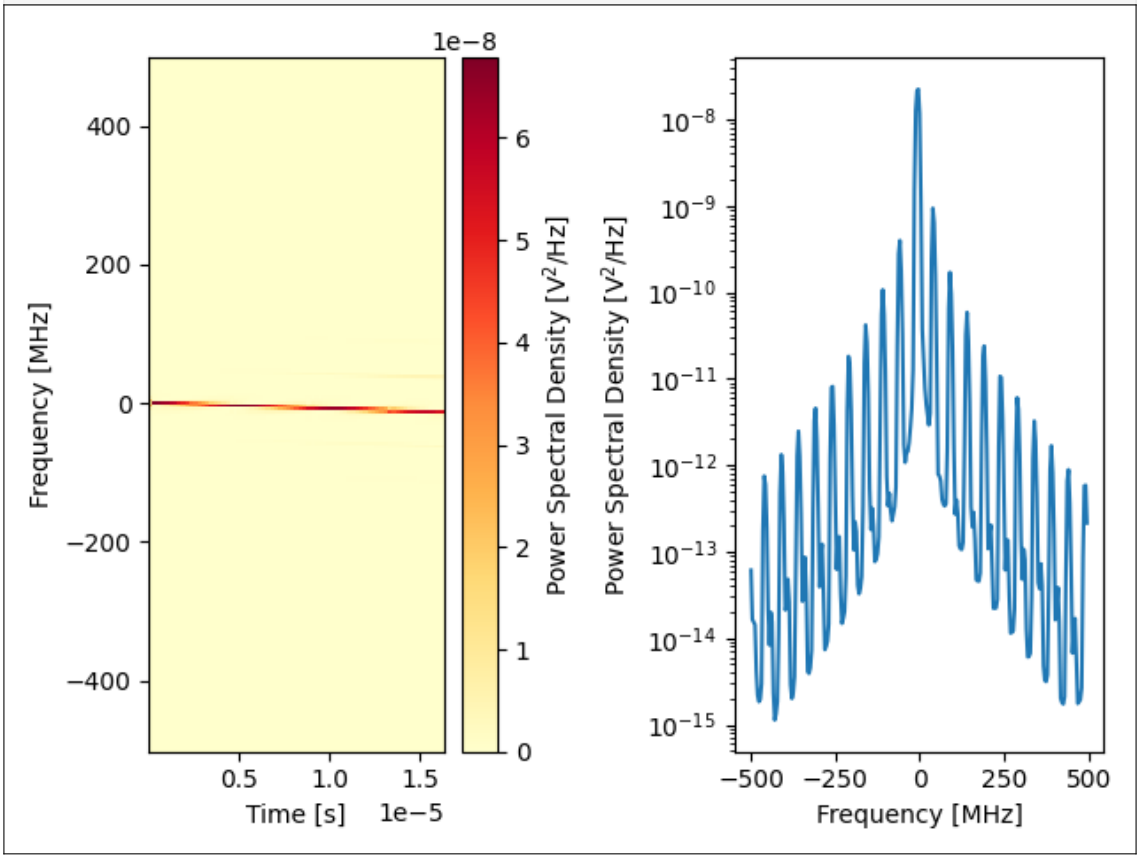
Using registers, we can also do negative phase steps:

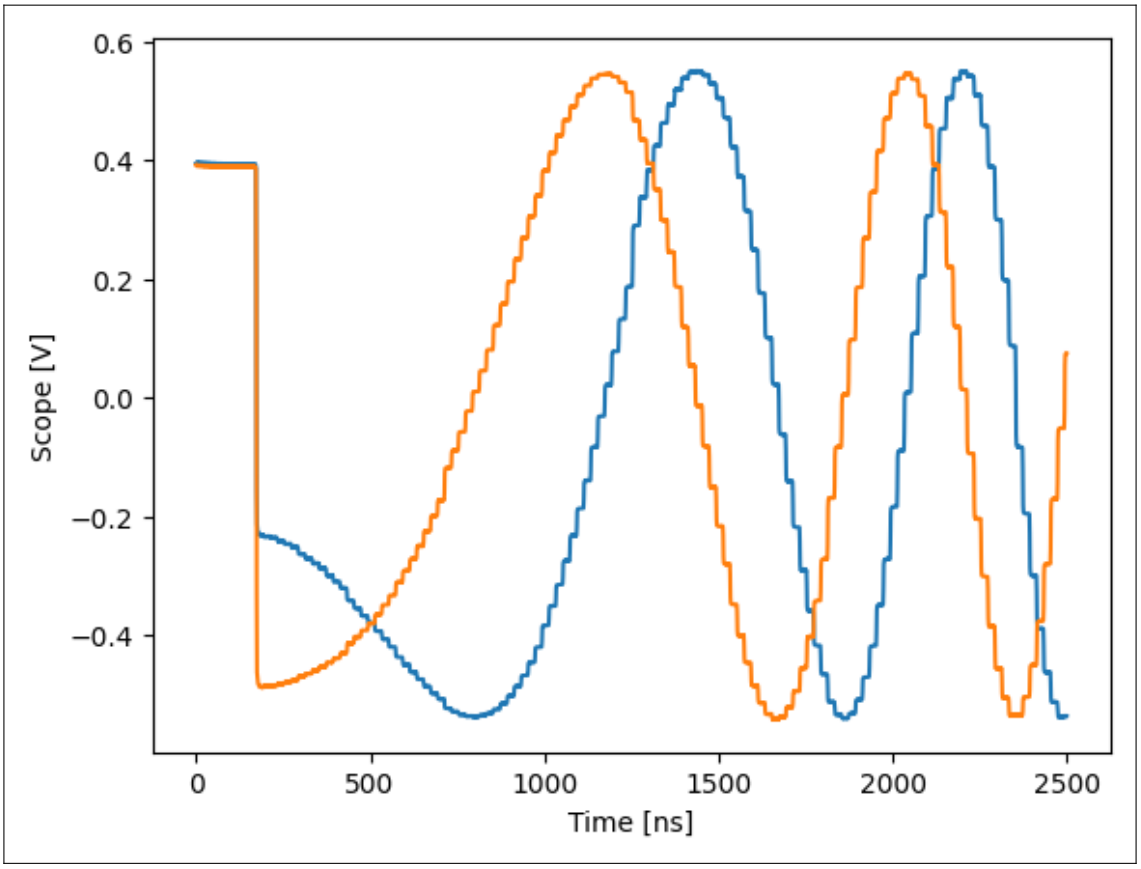
[51]:
```
n_averages = 100
n_steps = int(16384 / (20))

acquisitions = {"acq": {"num_bins": 1, "index": 0}}


phase_chirp = f"""
    move {n_averages}, R2

avg_loop:
    move    0, R0                           # phase update step size
    reset_ph
    upd_param 200
    set_freq 0
    acquire 0,0,4                           # Start acquisition. This is not␣
↪blocking

nco_set:
"""
# step the phase with increasing (negative) step size
for _ in range(n_steps):
    phase_chirp += f"""
    set_ph_delta    R0
    upd_param       20
    sub             R0, {int(1e9/(4*n_steps))}, R0      # increase the step␣
↪size ('frequency')
"""
phase_chirp += """
    wait            10000
    loop            R2, @avg_loop

    stop                                    # Stop
"""

# Write sequence to file.
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(
        {
            "waveforms": waveforms,
            "weights": {},
            "acquisitions": acquisitions,
            "program": phase_chirp,
        },
        file,
        indent=4,
    )
    file.close()

qrm.sequencer0.sequence("sequence.json")

qrm.arm_sequencer(0)
```

(continues on next page)

(continued from previous page)

```
qrm.start_sequencer()

# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "acq")
data = qrm.get_acquisitions(0)["acq"]

trace = np.array(data["acquisition"]["scope"]["path0"]["data"]) + 1j * np.
↪array(
    data["acquisition"]["scope"]["path1"]["data"]
)
plot_spectrogram(trace)

plot_scope(trace, 0, 2500)
```





1.19.4 Stop

Finally, let's stop the playback and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[52]:
```
# Stop both sequencers.
qrm.stop_sequencer()

# Print status of both sequencers (should now say it is stopped).
print(qrm.get_sequencer_state(0))
print(qrm.get_sequencer_state(1))
print()

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qxm.print_readable_snapshot(update=True)

# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

```
Status: STOPPED, Flags: FORCED_STOP, ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_
↪OVERWRITTEN_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_SCOPE_OVERWRITTEN_PATH_1, ACQ_
```

(continues on next page)

(continued from previous page)

```
↪BINNING_DONE
Status: STOPPED, Flags: FORCED_STOP
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`ttl_acquisition.ipynb`

1.20 TTL acquisition

In this tutorial we will demonstrate the sequencer based TTL (Transistor-Transistor-Logic) acquisition procedure. The TTL acquisition enables us to count trigger pulses, based on a settable threshold. The acquisition protocol allows us to save the number of triggers in separate bins, or average the triggers on the fly (see section TTL Acquisitions). We will showcase this functionality by using a QRM of which output $O^{[1]}$ is directly connected to input $I^{[1]}$, to both send pulses and acquire the resulting data.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.20.1 Setup

First, we are going to import the required packages.

[2]:
```
import math
import os

import matplotlib.pyplot as plt
import numpy as np
from numpy import random

# Set up the environment.
import scipy.signal
from IPython.display import display
import ipywidgets as widgets

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[3]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('cluster-mm',), value='cluster-
↪mm')
```

Pulsar QRM

Choose the Pulsar QRM, run the following cell. Skip to the *Cluster QRM section* if you selected a Cluster module.

[ ]:
```
# Close existing connections to the Pulsar modules
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qrm = Pulsar(f"{device_name}", ip_address, debug=True)
qrm.reset()
print(f"{device_name} connected at {ip_address}")
print(qrm.get_system_state())
```

Skip to *Generate Waveforms* if you have not selected a Cluster module.

Cluster QRM

First we connect to the Cluster using its IP address. Go to the *Pulsar QRM section* if you are using a Pulsar.

[4]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.2.150
```

We then find all available cluster modules to connect to them individually.

[5]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QRM module from the available modules in your Cluster:")
display(connect_qxm)
```

```
{'module8': 'QRM'}

Select the QRM module from the available modules in your Cluster:
```

```
Dropdown(options=('module8',), value='module8')
```

Finally, we connect to the selected Cluster module.

[6]:
```
# Connect to the cluster QRM
qrm = getattr(
```

(continues on next page)

(continued from previous page)

```
    cluster, connect_qxm.value
)  # Connect to the module that you have chosen above
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

```
QRM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

1.20.2 Generate waveforms

Next, we need to create the waveforms used by the sequence for playback on the outputs. Here, we define a single waveform consisting of a 16 ns block pulse with an amplitude of 0.5.

[7]:
```
# Waveform length parameter
waveform_length = 16  # nanoseconds

waveforms= {
     "block": {"data": [0.5 for i in range(0, waveform_length)], "index": 0},
}
```

1.20.3 Specify acquisitions

We need to specify the acquisitions so that the instrument can allocate the required memory for its acquisition list. Here, we create one acquisition that consists of 100 bins.

[8]:
```
# Acquisitions
acquisitions = {
    "ttl":   {"num_bins": 100, "index": 0},
}
```

1.20.4 Create Q1ASM programs

Now that we have the waveform and acquisition speficied, we define a simple Q1ASM program that sequences the waveforms and one that triggers the acquisitions. We will send 5 block pulses of 16 ns at 1 MHz (with 984 ns in between them). At the same time we will perform a 6000 ns TTL acquisition. Note that 1 MHz is the maximum continuous rate for a TTL acquisition.

The TTL acquisition is carried out with the `acquire_ttl` command that takes four arguments. The first argument is the index of what acquisition should be done, the second specifies in what bin index it is stored, the third toggles the acquisition on or off and finally the fourth argument is the amount of ns to wait. See the documentation for a more detailed overview of the sequencer instructions.

[9]:
```
# Sequence program for AWG.
seq_prog_awg = """
          wait_sync 4         #Wait for sequencers to synchronize and then␣
↪wait another 4ns.
          move      5,R0      #Loop iterator.
loop:
          play      0,0,16    #Play a block on output path 0 and wait 16ns.
```

(continues on next page)

(continued from previous page)

```
            wait      984      #Wait 984ns
            loop      R0, @loop #Repeat loop until R0 is 0

            stop               #Stop the sequence after the last iteration.
"""

# Sequence program for aqcuiring
seq_prog_acq = """
        wait_sync 4           #Wait for sequencers to synchronize and then␣
↪wait another 4ns.
        wait 140              #Approximate time of flight
        acquire_ttl 0,0,1,4   #Turn on TTL acquire on input path 0 and wait␣
↪4ns.
        wait 6000             #Wait 6000ns.
        acquire_ttl 0,0,0,4   #Turn off TTL acquire on input path 0 and wait␣
↪4ns.

        stop                   #Stop sequencer.
"""
```

1.20.5 Upload sequence

The sequences are uploaded to the sequencers. We will use sequencer 0 to send the pulses and sequencer 1 to acquire them.

[10]:
```
# Add sequence program, waveform and acquistitions to single dictionary.
sequence_awg = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": {},
    "program": seq_prog_awg,
    }
sequence_acq= {
       "waveforms": {},
       "weights": {},
       "acquisitions": acquisitions,
       "program": seq_prog_acq,
   }
```

[11]:
```
# Upload sequence.
qrm.sequencer0.sequence(sequence_awg)
qrm.sequencer1.sequence(sequence_acq)
```

1.20.6 Play sequence

Now we configure the sequencers, and play the sequence.

We will use sequencer 0 which will drive output O^1, and sequencer 1 which will acquire on input I^1, enabling syncing, and prepare the (scope) acquisition.

Then the TTL acquisition is configured by using `ttl_acq_input_select` to select I^1 as input. We set `ttl_acq_auto_bin_incr_en` to False such that our TTL count will be put in one bin. We set our TTL threshold to a value of 0.5 of our input range (corresponding to 0.5 V) using `ttl_acq_threshold`, and our input gain to 0 dB using `in0_gain`.

[13]:
```
# Map sequencer to specific outputs (but first disable all sequencer␣
↪connections)
for sequencer in qrm.sequencers:
    for out in range(0, 2):
        sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)

# Enable sync
qrm.sequencer0.sync_en(True)
qrm.sequencer1.sync_en(True)

# Delete previous acquisition.
qrm.delete_acquisition_data(1, "ttl")

# Configure scope mode
qrm.scope_acq_sequencer_select(1)

# Choose threshold and input gain
threshold = 0.5
input_gain = 0

# Configure the TTL acquisition
qrm.sequencer1.ttl_acq_input_select(0)
qrm.sequencer1.ttl_acq_auto_bin_incr_en(False)

#Set input gain and threshold
qrm.in0_gain(input_gain)
qrm.sequencer1.ttl_acq_threshold(threshold)
```

We start the sequence, and print the status flags of our sequencers.

[14]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.arm_sequencer(1)
qrm.start_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0, 1))
print(qrm.get_sequencer_state(1, 1))
```

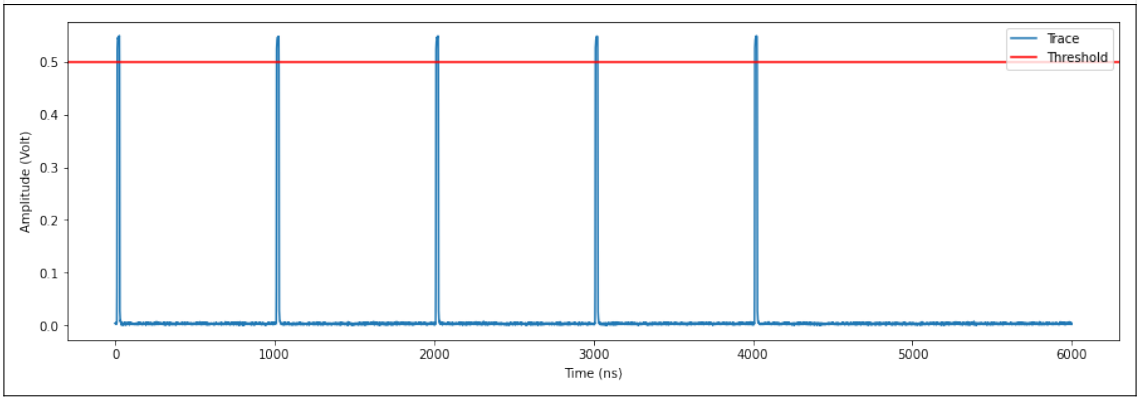
```
Status: STOPPED, Flags: NONE
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

1.20.7 Retrieve acquisition

We retrieve the acquisition data from sequencer 1. Then, both plot the scope data for the first 6000 ns and print the number of counted pulses, that is stored in `data['acquisition']["bins"]["avg_cnt"][0]`.

[15]:
```
# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(1, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(1, "ttl")

# Get acquisition list from instrument.
data = qrm.get_acquisitions(1)["ttl"]

# Plot acquired signal on both inputs.
print("pulses detected: " + str(data["acquisition"]["bins"]["avg_cnt"][0]))

fig, ax = plt.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(data["acquisition"]["scope"]["path0"]["data"][0:6000], label='Trace')
ax.axhline(y=threshold, color='r', label='Threshold')
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Amplitude (Volt)")
plt.legend(loc="upper right")
plt.show()
```

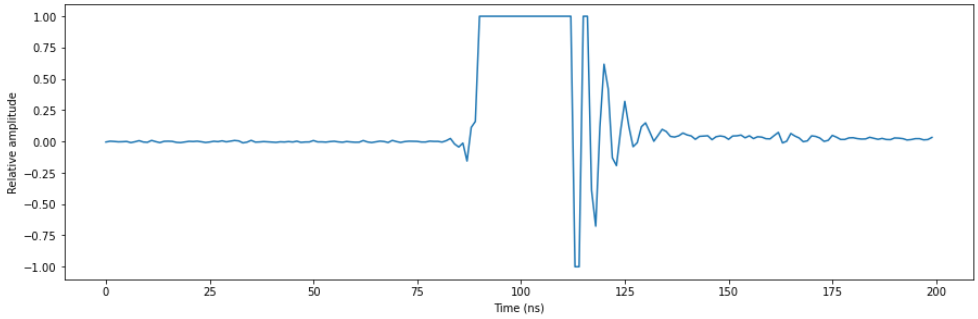
```
pulses detected: 5
```



We observe that we indeed do see five 16 ns pulses in the scope acquisition. Furthermore we observe that the amount of pulses counted matches the amount we of the scope trace. You are encouraged to change the threshold value and input gain yourself, and learn about th importance of calibrating these values.

Note

It is important to correctly calibrate your input gain and threshold. For example in a setup with noise and/or interferences, setting the input gain too high might result in these kind of pulses:



Such a single pulse results in multiple counts, as the threshold is passed multiple times. We therefore strongly recommend calibrating the TTL acquisition using the scope acquisition as is shown above.

1.20.8 Short pulse bursts

As the acquisition module in the sequencer has an internal buffer of 8 data entries, any pulse bursts of <8 will be handled correctly when exceeding the maximum continuous rate of 1 MHz. When exceeding the buffer limit, a `ACQ_BINNING_FIFO_ERROR` will be thrown. To illustrate this we will define a function `generate_pulse_program` that takes as input `num_pulses` and `wait_time`. We also define a function to upload the sequence to the AWG.

[16]:
```
# Sequence program for AWG.
def generate_pulse_program(num_pulses, wait_time):
    seq_prog_awg = f"""
                wait_sync 4         #Wait for sequencers to synchronize and then␣
↪wait another 4ns.
                move      {num_pulses},R0     #Loop iterator.
    loop:
                play      0,0,16    #Play a block on output path 0 and wait 16ns.
                wait      {wait_time}     #Wait 1000ns
                loop      R0, @loop #Repeat loop until R0 is 0

                stop                #Stop the sequence after the last iteration.
    """
    return seq_prog_awg

# Upload sequence to AWG
def upload_sequence(seq_prog_awg):
    sequence_awg = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": {},
    "program": seq_prog_awg,
    }

    qrm.sequencer0.sequence(sequence_awg)
```

We now generate the program, upload it and set the threshold and input gain.

[17]:
```
seq_prog_awg = generate_pulse_program(num_pulses=5, wait_time=20)
upload_sequence(seq_prog_awg)
```

(continues on next page)

(continued from previous page)

```
# Choose threshold and input gain
threshold = 0.5
input_gain = 0

# Delete previous acquisition.
qrm.delete_acquisition_data(1, "ttl")

#Set input gain and threshold
qrm.in0_gain(input_gain)
qrm.sequencer1.ttl_acq_threshold(threshold)
```

We then arm the sequencers and play the sequence.

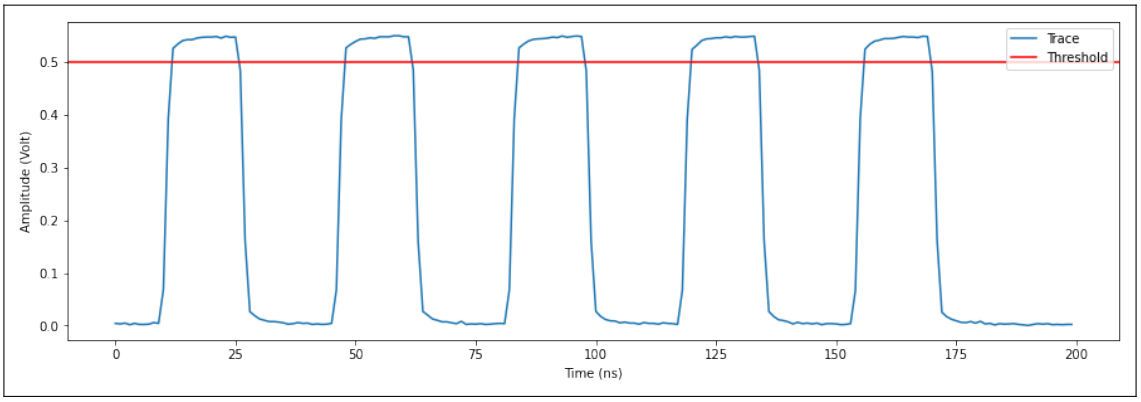
[18]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.arm_sequencer(1)
qrm.start_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0, 1))
print(qrm.get_sequencer_state(1, 1))
```

```
Status: STOPPED, Flags: NONE
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

We retrieve the acquisition and plot it as in the previous section.

[19]:
```
# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(1, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(1, "ttl")

# Get acquisition list from instrument.
data = qrm.get_acquisitions(1)["ttl"]

# Plot acquired signal on both inputs.
print("pulses detected: " + str(data["acquisition"]["bins"]["avg_cnt"][0]))

fig, ax = plt.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(data["acquisition"]["scope"]["path0"]["data"][0:200], label='Trace')
ax.axhline(y=threshold, color='r', label='Threshold')
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Amplitude (Volt)")
plt.legend(loc="upper right")
plt.show()
```

```
pulses detected: 5
```



We encourage you to play around with the `num_pulses`, `wait_time`, `input_gain` and `threshold` yourself.

1.20.9 Auto bin increment

In the *play sequence* section we set the `ttl_acq_auto_bin_incr_en` to False, meaning all of our pulses get counted within one bin. When it is set to True our data will be stored in separate bins, where the bin index is incremented by one for every detected pulse. The pulse count is therefore equal to the number of valid bins. When doing multiple measurements, this allows us to acquire a cumulative probability distribution of counted triggers.

To illustrate the usage we will define a function which returns a Q1ASM program that plays a `N` amount of pulses (at a 1 MHz rate), where `N` is a random number between 1 and 100 taken from a Poissonian distribution. This is meant to mock a stochastic physical process, e.g the amount of photons emitted by a laser. We will call the function a 1000 times and run it, without deleting the acquisition data between the runs. After this we will plot a histogram from our acquired data to inspect the result.

We will now define new Q1ASM programs. We define a function which returns a program that generates `num_pulses` number of pulses. These pulses are 16 ns long, and are send at a rate of 1 MHz again.

For the acquiring sequencer we will execute the same program as in the previous section, albeit with an acquisition window of 100.000 ns.

[20]:
```
# Sequence program for aqcuiring
seq_prog_acq = """
        move      10,R0       #Loop iterator.
        wait_sync 4           #Wait for sequencers to synchronize and then␣
↪wait another 4ns.
        wait 140              #Approximate time of flight
        acquire_ttl 0,0,1,4   #Turn on TTL acquire on input path 0 and wait␣
↪4ns.
loop:
        wait 10000            #Wait 10000 ns
        loop      R0, @loop #Repeat loop until R0 is 0
        acquire_ttl 0,0,0,4   #Turn off TTL acquire on input path 0 and wait␣
↪4ns.

        stop                   #Stop sequencer.
"""
```

We will first configure our sequencers. Then use `random` to generate an amount of pulses, and repeat this 10000 times.

[21]:
```
# Choose threshold and input gain
threshold = 0.5
input_gain = 0

# Delete previous acquisition.
qrm.delete_acquisition_data(1, "ttl")

# Configure the TTL acquisition
qrm.sequencer1.ttl_acq_input_select(0)
qrm.sequencer1.ttl_acq_auto_bin_incr_en(True)

# Enable sync
qrm.sequencer0.sync_en(True)
qrm.sequencer1.sync_en(True)

#Set input gain and threshold
qrm.in0_gain(input_gain)
qrm.sequencer1.ttl_acq_threshold(threshold)

sequence_acq= {
    "waveforms": {},
    "weights": {},
    "acquisitions": acquisitions,
    "program": seq_prog_acq,
}

#Upload acquire sequence.
qrm.sequencer1.sequence(sequence_acq)

# Map sequencer to specific outputs (but first disable all sequencer␣
↪connections)
for sequencer in qrm.sequencers:
    for out in range(0, 2):
        sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)
```

[22]:
```
num_pulses_list = random.poisson(lam=50, size=1000)
wait_time = 1000

for num_pulses in num_pulses_list:
    seq_prog_awg = generate_pulse_program(num_pulses, wait_time)
    upload_sequence(seq_prog_awg)
    # Arm and start sequencer.
    qrm.arm_sequencer(0)
    qrm.arm_sequencer(1)
    qrm.start_sequencer()
    qrm.get_acquisition_state(1, 1)
```

Create histogram

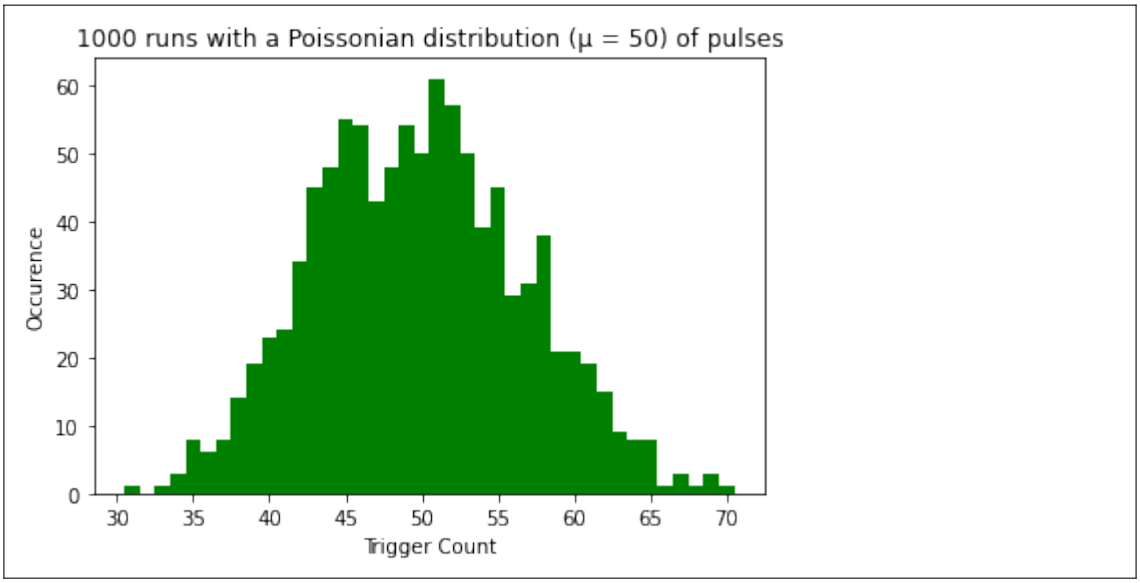
We retrieve the acquired data from the sequence to take a look at it.

[23]:
```
# Wait for the sequencer to stop with a timeout period of one minute.
qrm.get_acquisition_state(1, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(1, "ttl")

# Get acquisition list from instrument.
data = qrm.get_acquisitions(1)["ttl"]["acquisition"]["bins"]["avg_cnt"]

# Plot acquired signal on both inputs.
print(f"counts per bin: {data}")
```

```
counts per bin: [1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000,␣
↪1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000,
↪ 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 999, 999, 998, 995, 987,␣
↪981, 973, 959, 940, 917, 893, 859, 814, 766, 711, 657, 614, 566, 512, 462,␣
↪401, 344, 294, 255, 210, 181, 150, 112, 91, 70, 51, 36, 27, 19, 11, 10, 7, 6,
↪ 3, 2, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,␣
↪nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan]
```

Every time the acquisition_ttl is ran the sequencer starts over with incrementing the bins (from bin 0). In every bin, the counts are summed up for all 1000 runs. Therefore, the data is a cumulative probability distrubition of triggers counted. We now reorganize the acquired data into a probability distribution function (a histogram), and plot the result.

[24]:
```
def create_histogram(data):
    res = {}
    runs = data[0]
    old_temp = 0
    for count, value in enumerate(data):
        # Reached end of data
        if str(value) == 'nan': break
        new_temp = runs - value
        # Only add if something changed
        if new_temp == old_temp: continue
        num_pulses = new_temp - old_temp
        old_temp = new_temp
        res[count] = num_pulses
    return res

res = create_histogram(data)
plt.bar(res.keys(), res.values(), 1, color='g')
plt.title("1000 runs with a Poissonian distribution (μ = 50) of pulses")
plt.xlabel("Trigger Count")
plt.ylabel("Occurence")
plt.show()
```



1.20.10 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[25]:
```
# Stop sequencer.
qrm.stop_sequencer()

# Print status of sequencer.
print(qrm.get_sequencer_state(0))
print()

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qrm.print_readable_snapshot(update=True)

# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

```
Status: STOPPED, Flags: FORCED_STOP
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`synchronization.ipynb`

1.21 Synchronization

In this tutorial we will demonstrate how to synchronize two Qblox instruments using the SYNQ technology (see section Synchronization). For this tutorial we will use one QCM and one QRM and we will be acquiring waveforms sequenced by the QCM using the QRM. By synchronizing the two instruments using the SYNQ technology, timing the acquisition of the waveforms becomes trivial.

If we are using a Pulsar QCM and QRM we need to connect both instruments to the same network, connect the REF^{out} of the QCM to the REF^{in} of the QRM using a 50cm coaxial cable, connect their SYNQ ports using the SYNQ cable and finally connect $\text{O}^{[1-2]}$ of the QCM to $\text{I}^{[1-2]}$ of the QRM respectively.

The Cluster QCM and QRM are internally connected for SYNQ capability and therefore no new need connections need to be made if one is using a Cluster.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.21.1 Setup

First, we are going to import the required packages and connect to the instrument.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar

# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect_qcm = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select QCM",
)
connect_qrm = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
```

(continues on next page)

(continued from previous page)

```
↪keys()],
    description="Select QRM",
)
connect_cluster = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Cluster",
)
```

Pulsar

Select the Pulsar QCM and QRM modules by running the following cell. Skip to the *Cluster section* if you are using a Cluster.

[ ]:
```
display(connect_qcm)
display(connect_qrm)
```

[ ]:
```
# Close existing connections to Pulsar
Pulsar.close_all()

# Retrieve device name and IP address and Connect
# QCM
qcm_device_name = connect_qcm.value
qcm_device_number = connect_qcm.options.index(qcm_device_name)
qcm_ip_address = device_list[device_keys[qcm_device_number]]["identity"]["ip"]
qcm = Pulsar(f"{qcm_device_name}", qcm_ip_address)
qcm.reset()  # reset QCM
print(f"{qcm_device_name} connected at {qcm_ip_address}")
print(qcm.get_system_state())

# QRM
qrm_device_name = connect_qrm.value
qrm_device_number = connect_qrm.options.index(qrm_device_name)
qrm_ip_address = device_list[device_keys[qrm_device_number]]["identity"]["ip"]
qrm = Pulsar(f"{qrm_device_name}", qrm_ip_address)
qrm.reset()  # reset QRM
print(f"{qrm_device_name} connected at {qrm_ip_address}")
print(qrm.get_system_state())
```

[5]:
```
# Set reference clock source.
qrm.reference_source("external")
```

Skip to the next section *(Generate Waveform)* if you are not using a cluster.

Cluster

First we connect to the Cluster using its IP address. Go to the *Pulsar section* if you are using a Pulsar.

[2]:
```
display(connect_cluster)
```

```
Dropdown(description='Select Cluster', options=('cluster-mm',), value='cluster-
↪mm')
```

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect_cluster.value
device_number = connect_cluster.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster
cluster = Cluster(device_name, ip_address)
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.0.2
```

We then find all available cluster modules to connect to them individually.

[5]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

connect_qcm = widgets.Dropdown(
    options=[key for key in available_slots.keys()], description="Select QCM"
)
connect_qrm = widgets.Dropdown(
    options=[key for key in available_slots.keys()], description="Select QRM"
)
```

```
cluster-mm connected at 192.168.1.0
```

Select the QCM and QRM module from the available modules in your Cluster.

[6]:
```
print(available_slots)
print()
display(connect_qcm)
display(connect_qrm)
```

```
{'module2': 'QCM', 'module4': 'QRM'}
```

```
Dropdown(description='Select QCM', options=('module2', 'module4'), value=
↪'module2')
```

```
Dropdown(description='Select QRM', options=('module2', 'module4'), value=
↪'module2')
```

Finally, we connect to the selected Cluster modules.

[7]:
```
# Connect to QCM and QRM
qcm = getattr(
    cluster, connect_qcm.value
)  # Connect to the module that you have chosen above
print(f"{available_slots[connect_qcm.value]} connected")


qrm = getattr(cluster, connect_qrm.value)
print(f"{available_slots[connect_qrm.value]} connected")
print(cluster.get_system_state())
```

```
QCM connected
QCM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

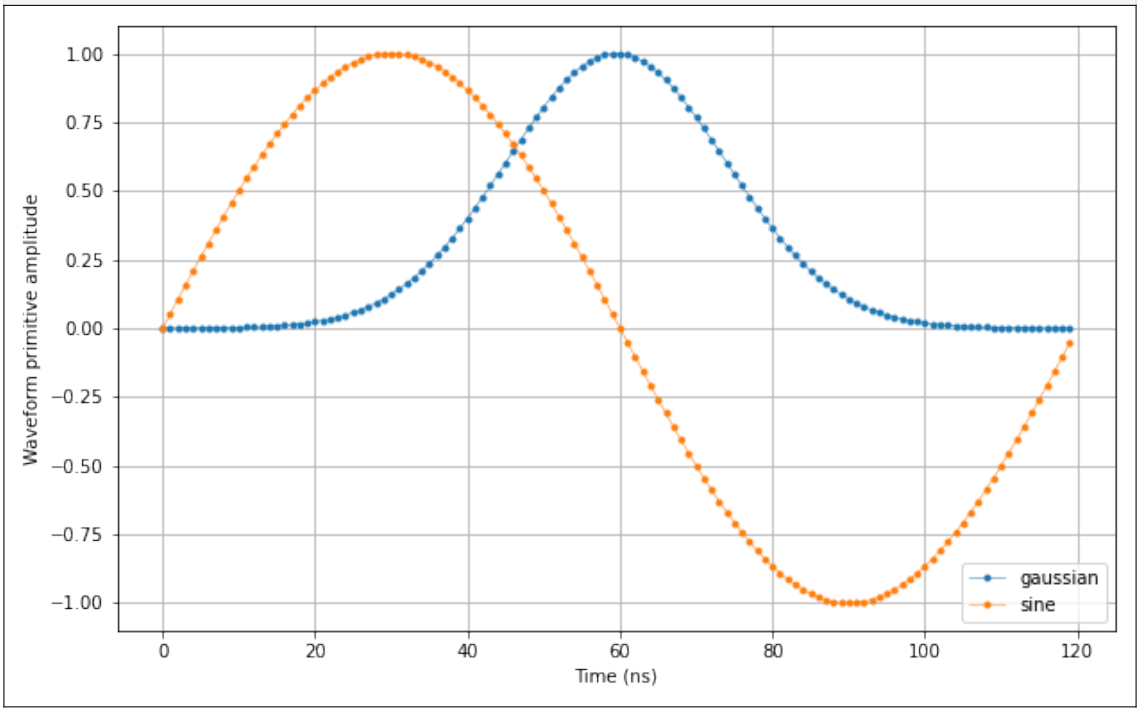
1.21.2 Generate waveforms

Next, we need to create the waveforms for the sequence.

[6]:
```
# Waveform parameters
waveform_length = 120  # nanoseconds

# Waveform dictionary (data will hold the samples and index will be used to␣
↪select the waveforms in the instrument).
waveforms = {
    "gaussian": {
        "data": scipy.signal.gaussian(
            waveform_length, std=0.12 * waveform_length
        ).tolist(),
        "index": 0,
    },
    "sine": {
        "data": [
            math.sin((2 * math.pi / waveform_length) * i)
            for i in range(0, waveform_length)
        ],
        "index": 1,
    },
}
```

Let's plot the waveforms to see what we have created.

[7]:
```
time = numpy.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())),␣
↪1)
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(10, 10 / 1.61))

for wf, d in waveforms.items():
    ax.plot(time[: len(d["data"])], d["data"], ".-", linewidth=0.5, label=wf)

ax.legend(loc=4)
ax.yaxis.grid()
ax.xaxis.grid()
ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

matplotlib.pyplot.draw()
matplotlib.pyplot.show()
```



1.21.3 Specify acquisitions

We also need to specify the acquisitions so that the instrument can allocate the required memory for it's acquisition list. In this case we will create one acquisition specification that creates a single bin. However, we will not be using the bin in this turorial.

[8]:
```
# Acquisitions
acquisitions = {"measurement": {"num_bins": 1, "index": 0}}
```

1.21.4 Create Q1ASM programs

Now that we have the waveforms and acquisition specifications for the sequence, we need a simple Q1ASM program that sequences the waveforms in the QCM and acquires the waveforms in the QRM.

[9]:
```
# QCM sequence program.
qcm_seq_prog = """
wait_sync 4         #Synchronize sequencers over multiple instruments.
play      0,1,16384 #Play waveforms and wait remaining duration of scope␣
↪acquisition.
stop                #Stop.
"""

# QRM sequence program.
qrm_seq_prog = """
wait_sync 4         #Synchronize sequencers over multiple instruments.
acquire   0,0,16384 #Acquire waveforms and wait remaining duration of scope␣
↪acquisition.
stop                #Stop.
"""
```

1.21.5 Upload sequences

Now that we have the waveforms and Q1ASM programs, we can combine them in the sequences stored in JSON files.

[10]:
```
# Add QCM sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": acquisitions,
    "program": qcm_seq_prog,
}
with open("qcm_sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()

# Add QRM sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": acquisitions,
    "program": qrm_seq_prog,
}
with open("qrm_sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

Let's write the JSON file to the instruments. We will use sequencer 0 of both QCM and QRM, which will drive outputs $O^{[1-2]}$ of the QCM and acquire on inputs $I^{[1-2]}$ of the QRM.

[11]:
```
# Upload waveforms and programs to QCM.
qcm.sequencer0.sequence("qcm_sequence.json")

# Upload waveforms and programs to QRM.
qrm.sequencer0.sequence("qrm_sequence.json")
```

1.21.6 Play sequences

The sequence has been uploaded to the instruments. Now we need to configure the sequencers of both the QCM and QRM to use the `wait_sync` instruction to synchronize and we need to configure the sequencer of the QRM to trigger the acquisition with the `acquire` instruction. Furthermore we also need to attenuate the QCM's outputs to 40% to be able to capture the full range of the waveforms on the QRM's inputs.

Attenuation $= Input/Output = 2V/5V = 0.4$

[12]:
```
# Configure the sequencer of the QCM.
qcm.sequencer0.sync_en(True)
qcm.sequencer0.gain_awg_path0(0.35)  # Adding a bit of margin to the 0.4
qcm.sequencer0.gain_awg_path1(0.35)

# Map sequencer of the QCM to specific outputs (but first disable all␣
↪sequencer connections)
for sequencer in qcm.sequencers:
    for out in range(0, 2):
        sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)
qcm.sequencer0.channel_map_path0_out0_en(True)
qcm.sequencer0.channel_map_path1_out1_en(True)

# Configure the scope acquisition of the QRM.
qrm.scope_acq_sequencer_select(0)
qrm.scope_acq_trigger_mode_path0("sequencer")
qrm.scope_acq_trigger_mode_path1("sequencer")

# Configure the sequencer of the QRM.
qrm.sequencer0.sync_en(True)
```

Now let's start the sequences.

[13]:
```
# Arm and start sequencer of the QCM (only sequencer 0).
qcm.arm_sequencer(0)
qcm.start_sequencer()

# Print status of sequencer of the QCM.
print("QCM status:")
print(qcm.get_sequencer_state(0))
print()

# Arm and start sequencer of the QRM (only sequencer 0).
qrm.arm_sequencer(0)
qrm.start_sequencer()
```

(continues on next page)

(continued from previous page)

```
# Print status of sequencer of the QRM.
print("QRM status:")
print(qrm.get_sequencer_state(0))
```

```
QCM status:
Status: Q1_STOPPED, Flags: NONE

QRM status:
Status: STOPPED, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_
↪BINNING_DONE
```

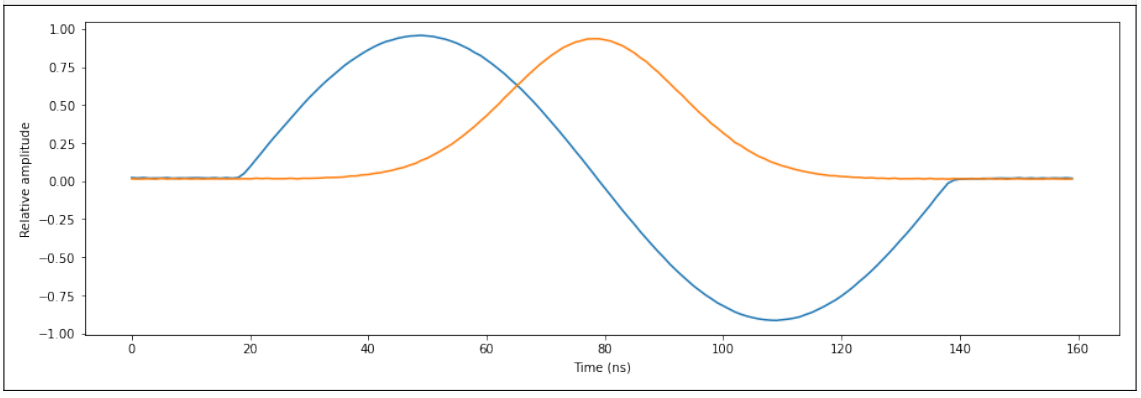
1.21.7 Retrieve acquisition

The waveforms have now been sequenced on the outputs and acquired on the inputs by both instruments. And as you might have noticed, timing these operations was simplified significantly by the SYNQ technology. Lets retrieve the resulting data, but first let's make sure the sequencers have finished.

[14]:
```
# Wait for the QCM sequencer to stop with a timeout period of one minute.
qcm.get_sequencer_state(0, 1)

# Wait for the QRM acquisition to finish with a timeout period of one minute.
qrm.get_acquisition_state(0, 1)

# Move acquisition data from temporary memory to acquisition list.
qrm.store_scope_acquisition(0, "measurement")

# Get acquisition list from instrument.
acq = qrm.get_acquisitions(0)
```

Let's plot the result.

[15]:
```
# Plot acquired signal on both inputs.
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(acq["measurement"]["acquisition"]["scope"]["path0"]["data"][130:290])
ax.plot(acq["measurement"]["acquisition"]["scope"]["path1"]["data"][130:290])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



1.21.8 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Stop sequencers.
qcm.stop_sequencer()
qrm.stop_sequencer()

# Print status of sequencers.
print("QCM:")
print(qcm.get_sequencer_state(0))
print()

print("QRM:")
print(qrm.get_sequencer_state(0))
print()

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of instrument parameters.
# print("QCM snapshot:")
# qcm.print_readable_snapshot(update=True)
# print()

# print("QRM snapshot:")
# qrm.print_readable_snapshot(update=True)

# Close the instrument connections.
Pulsar.close_all()
Cluster.close_all()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`multiplexed_sequencing.ipynb`

1.22 Multiplexed sequencing

The Pulsar/Cluster QRM/QCM supports six sequencers. The output of each sequencer is multiplexed, accumulated and connected to the output ports of the instrument as shown in the figure below. Furthermore, input paths 0 and 1 of every sequencer are connected to inputs I^1 and I^2 respectively. In the first part of the tutorial we will connect outputs of multiple sequencers to a single pair of output ports and demonstrate how to multiplex output paths 0 and 1 of these sequencers as well as show how these multiplexed signals are accumulated. In the second part of the tutorial we will demonstrate firstly, how to frequency multiplex the output paths of different sequencers at different unique carrier frequencies on a single pair of output ports and secondly, how to aquire, demodulated and integrate the recieved frequency multiplexed waveforms. In the third part of the tutorial we will demonstrate "real mode", where we will control various sequencers to independantly output real signals on each output port of the instrument.

We will show this by using a QRM and directly connecting outputs $O^{[1-2]}$ to inputs $I^{[1-2]}$ respectively. We will then use the QRM's sequencers to sequence waveforms on the outputs and simultaneously acquire the resulting waveforms on the inputs.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.22.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive
from scipy.fft import rfft, rfftfreq

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('pulsar-qrm', 'pulsar-qcm'),␣
↪value='pulsar-qrm')
```

Pulsar QRM

If you chose the Pulsar QRM, run the following cell. Skip to the *Cluster QRM section* if you selected a Cluster module.

[3]:
```
# Close existing connections to the Pulsar modules
Pulsar.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# Connect to device and reset
qrm = Pulsar(f"{device_name}", ip_address)
qrm.reset()
cluster = None  # In absence of a cluster
print(f"{device_name} connected at {ip_address}")
print(qrm.get_system_state())
```

```
pulsar-qrm connected at 192.168.0.4
Status: OKAY, Flags: NONE, Slot flags: NONE
```

Skip to the next section *(Generate Waveform)* if you are not using a cluster.

Cluster QRM

First we connect to the Cluster using its IP address. Go to the *Pulsar QRM section* if you are using a Pulsar.

[ ]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

We then find all available cluster modules to connect to them individually.

```
[ ]: # Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qxm = widgets.Dropdown(options=[key for key in available_slots.keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QRM module from the available modules in your Cluster:")
display(connect_qxm)
```

Finally, we connect to the selected Cluster module.

```
[ ]: # Connect to the cluster QRM
qrm = getattr(
    cluster, connect_qxm.value
)  # Connect to the module that you have chosen above
print(f"{available_slots[connect_qxm.value]} connected")
print(cluster.get_system_state())
```

1.22.2 Generate waveforms

Next, we need to create the gaussian and block waveforms for the sequence.

```
[4]: # Waveforms
waveform_len = 1000
waveforms = {
    "gaussian": {
        "data": scipy.signal.gaussian(waveform_len, std=0.133 * waveform_len).
↪tolist(),
        "index": 0,
    },
    "sine": {
        "data": [
            math.sin((2 * math.pi / waveform_len) * i) for i in range(0,␣
↪waveform_len)
        ],
        "index": 1,
    },
```

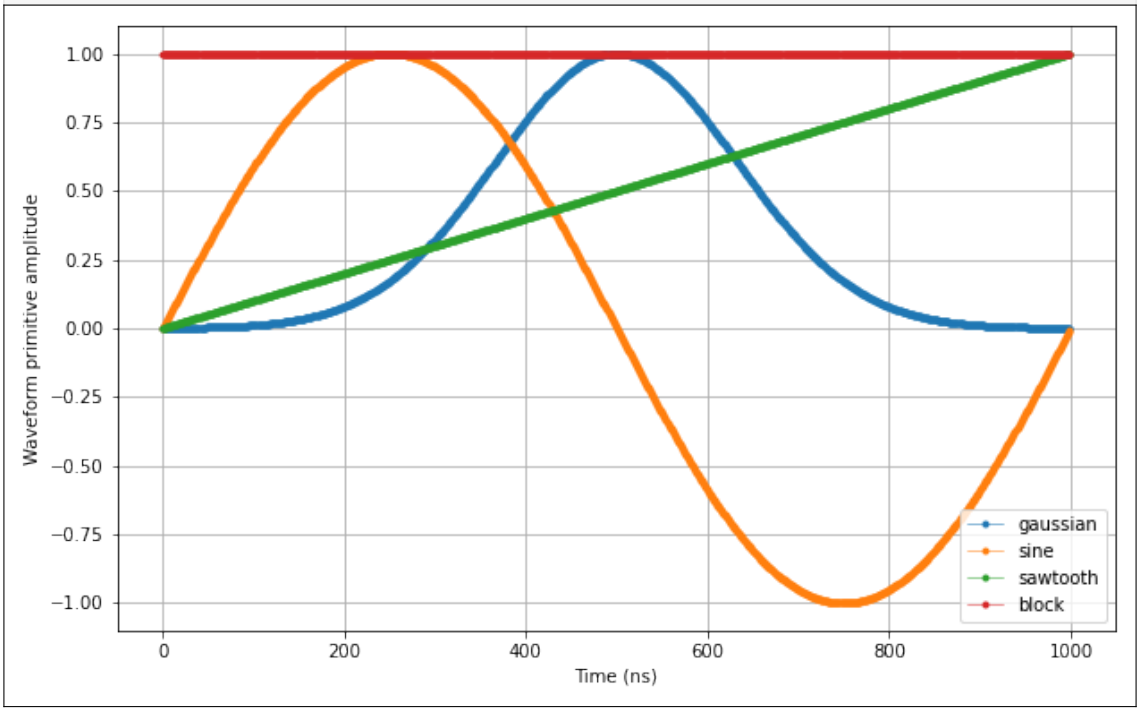
(continues on next page)

(continued from previous page)

```
    "sawtooth": {
        "data": [(1.0 / waveform_len) * i for i in range(0, waveform_len)],
        "index": 2,
    },
    "block": {"data": [1.0 for i in range(0, waveform_len)], "index": 3},
}
```

Let's plot the waveforms to see what we have created.

[5]:
```
time = numpy.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())),␣
↪1)
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(10, 10 / 1.61))

for wf, d in waveforms.items():
    ax.plot(time[: len(d["data"])], d["data"], ".-", linewidth=0.5, label=wf)

ax.legend(loc=4)
ax.yaxis.grid()
ax.xaxis.grid()
ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

matplotlib.pyplot.draw()
matplotlib.pyplot.show()
```



1.22.3 Specify the acquisitions

We will only use a single bin in this tutorial, so we can keep it simple

[6]:
```
# Acquisitions
acquisitions = {"scope": {"num_bins": 1, "index": 0}}
```

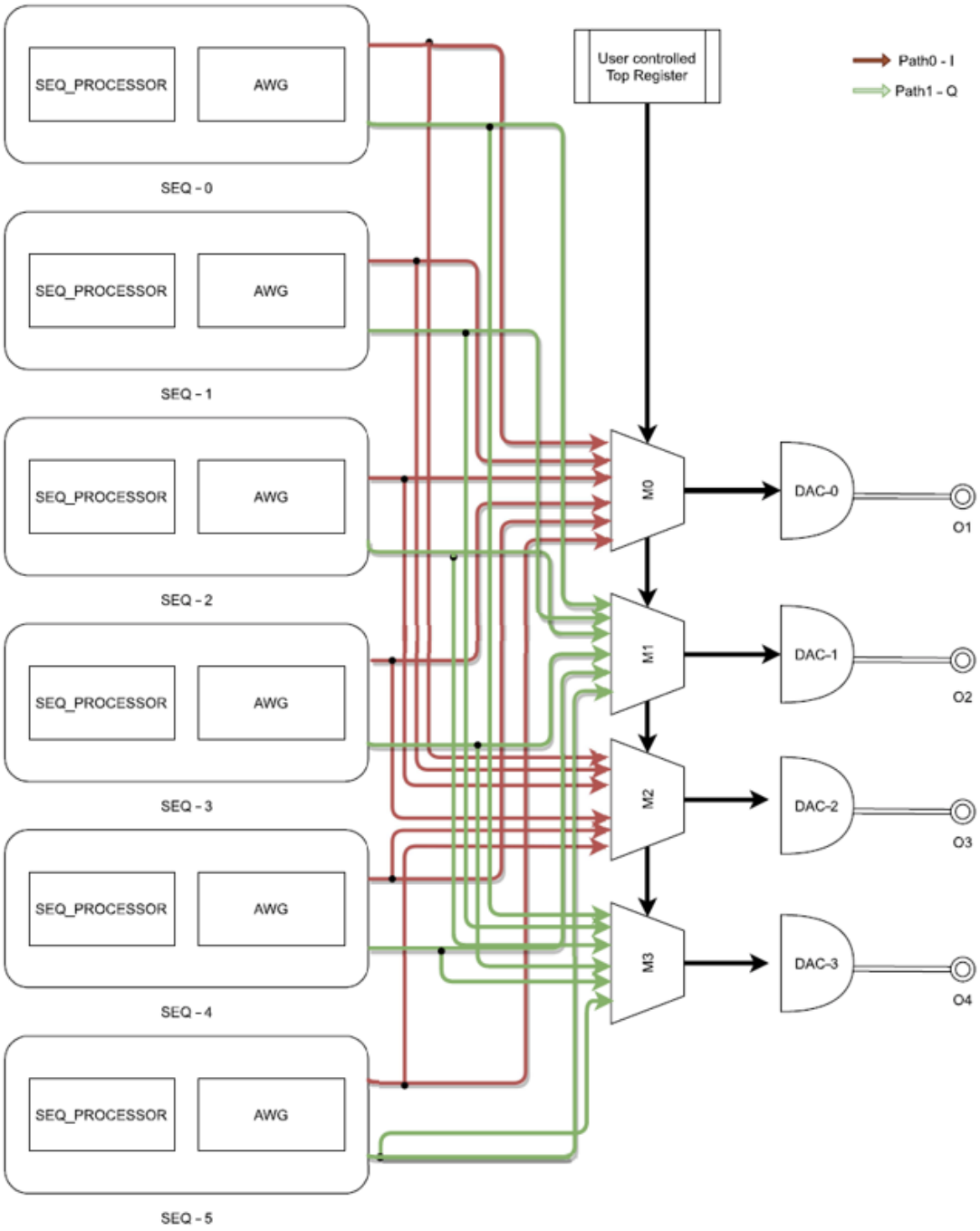
1.22.4 Create Q1ASM program and upload the sequence

Now that we have the waveform and acquisition specifications for the sequence, we need a simple Q1ASM program that sequences the waveforms and triggers the acquisitions. In this case we will play a gaussian and a sinosoid wave for path 0 and 1 respectively per sequencer.

[7]:
```
# Number of sequencers per instrument
num_seq = 6

# Program
program = """
wait_sync 4
play      0,1,4
wait      140
acquire   0,0,16380
stop
"""

# Write sequence to file.
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(
        {
            "waveforms": waveforms,
            "weights": waveforms,
            "acquisitions": acquisitions,
            "program": program,
        },
        file,
        indent=4,
    )
    file.close()

# Program sequencers.
for sequencer in qrm.sequencers:
    sequencer.sequence("sequence.json")
```

1.22.5 Multiplexed sequencer output control

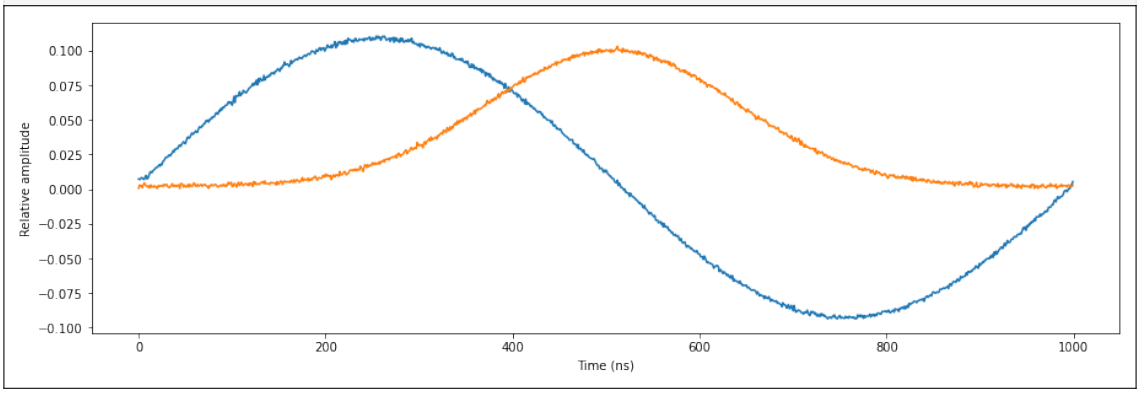
The output paths of each sequencer are connected to the instrument's outputs as shown in the figure below. For a QRM, path 0 of each sequencer can only be connected to output O^1 and path 1 only to output O^2 (for a QCM, path 0 can be connected to output O^1 and O^3 and path 1 to O^2 and O^4), . In order to connect the sequencer output paths to corresponding output ports, we need to configure the sequencer's channel map by calling sequencer`X`_channel_map_path`Y`_out`Z`_en where `X`, `Y` and `Z` represents sequencer ID, path ID and output port number respectively.



Now lets configure the first sequencer to output its paths on O^1 and O^2. We will scale the amplitude of

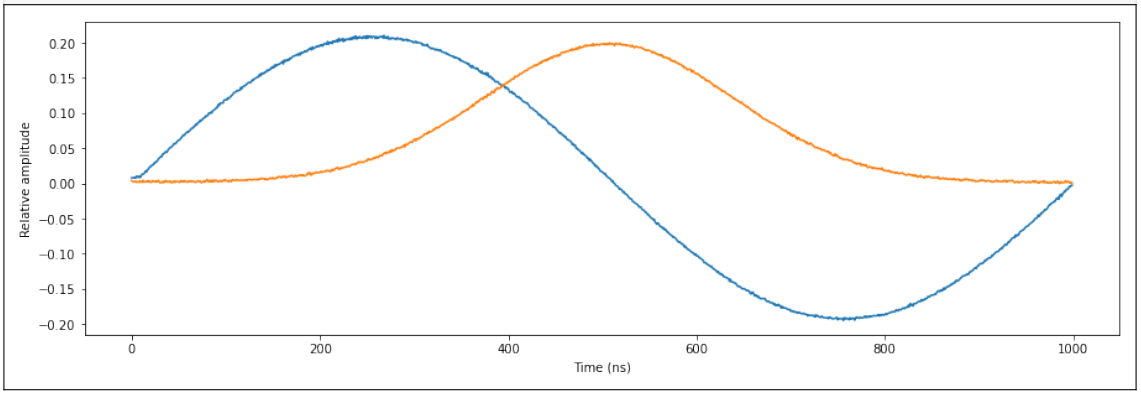
the signal such that we are able the show what happens when other sequencers are added and eventually output overflow occurs.

[8]:
```
# Configure the sequencer to trigger the scope acquisition.
qrm.scope_acq_sequencer_select(0)
qrm.scope_acq_trigger_mode_path0("sequencer")
qrm.scope_acq_trigger_mode_path1("sequencer")

# Configure sequencer
qrm.sequencer0.sync_en(True)
qrm.sequencer0.gain_awg_path0(
    1.1 / num_seq
)  # The output range is 1.0 to -1.0, but we want to show what happens when␣
↪the signals go out of range.
qrm.sequencer0.gain_awg_path1(1.1 / num_seq)
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)

# Start the sequence
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer to stop
qrm.get_acquisition_state(0, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

# Plot the results
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))

ax.plot(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_len])
ax.plot(acq["scope"]["acquisition"]["scope"]["path1"]["data"][0:waveform_len])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")

matplotlib.pyplot.show()
```



Let's add the second sequencer.

[9]:
```
# Configure the sequencer
qrm.sequencer1.sync_en(True)
qrm.sequencer1.gain_awg_path0(1.1 / num_seq)
qrm.sequencer1.gain_awg_path1(1.1 / num_seq)
qrm.sequencer1.channel_map_path0_out0_en(True)
qrm.sequencer1.channel_map_path1_out1_en(True)

# Start the sequencers
for seq in range(0, 2):
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop
for seq in range(0, 2):
    qrm.get_acquisition_state(seq, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

# Plot the results
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))

ax.plot(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_len])
ax.plot(acq["scope"]["acquisition"]["scope"]["path1"]["data"][0:waveform_len])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")

matplotlib.pyplot.show()
```



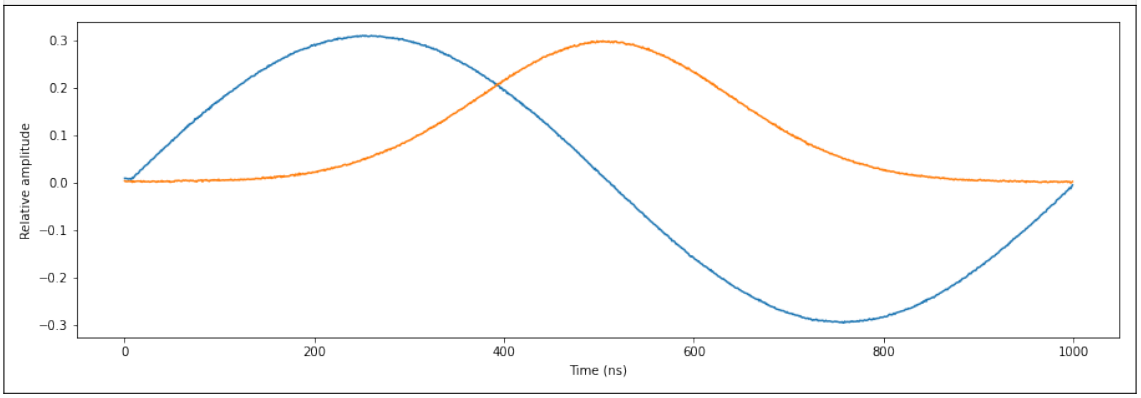
Let's add the third sequencer.

[10]:
```
# Configure the sequencer
qrm.sequencer2.sync_en(True)
qrm.sequencer2.gain_awg_path0(1.1 / num_seq)
qrm.sequencer2.gain_awg_path1(1.1 / num_seq)
qrm.sequencer2.channel_map_path0_out0_en(True)
qrm.sequencer2.channel_map_path1_out1_en(True)

# Start the sequencers
for seq in range(0, 3):
```

(continues on next page)

(continued from previous page)

```
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop
for seq in range(0, 3):
    qrm.get_acquisition_state(seq, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

# Plot the results
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))

ax.plot(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_len])
ax.plot(acq["scope"]["acquisition"]["scope"]["path1"]["data"][0:waveform_len])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")

matplotlib.pyplot.show()
```



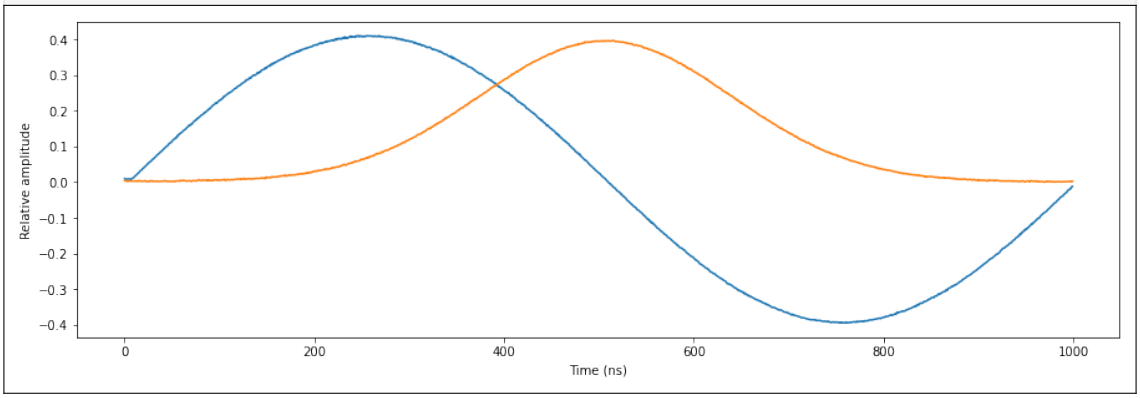
Let's add the fourth sequencer.

[11]:
```
# Configure the sequencer
qrm.sequencer3.sync_en(True)
qrm.sequencer3.gain_awg_path0(1.1 / num_seq)
qrm.sequencer3.gain_awg_path1(1.1 / num_seq)
qrm.sequencer3.channel_map_path0_out0_en(True)
qrm.sequencer3.channel_map_path1_out1_en(True)

# Start the sequencers
for seq in range(0, 4):
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop
for seq in range(0, 4):
    qrm.get_acquisition_state(seq, 1)

# Get acquisition data
```

(continues on next page)

(continued from previous page)

```
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

# Plot the results
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))

ax.plot(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_len])
ax.plot(acq["scope"]["acquisition"]["scope"]["path1"]["data"][0:waveform_len])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")

matplotlib.pyplot.show()
```



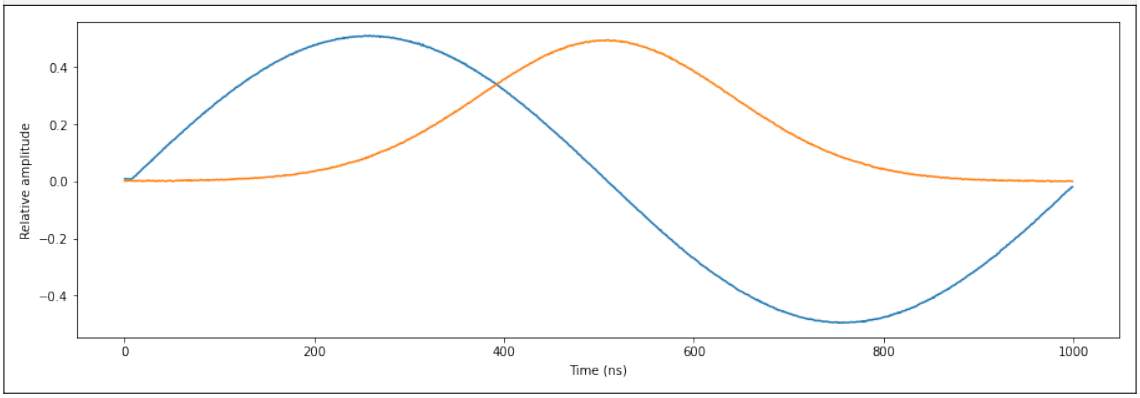
Let's add the fifth sequencer.

[12]:
```
# Configure the sequencer
qrm.sequencer4.sync_en(True)
qrm.sequencer4.gain_awg_path0(1.1 / num_seq)
qrm.sequencer4.gain_awg_path1(1.1 / num_seq)
qrm.sequencer4.channel_map_path0_out0_en(True)
qrm.sequencer4.channel_map_path1_out1_en(True)

# Start the sequencers
for seq in range(0, 5):
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop
for seq in range(0, 5):
    qrm.get_acquisition_state(seq, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

# Plot the results
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))

ax.plot(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_len])
ax.plot(acq["scope"]["acquisition"]["scope"]["path1"]["data"][0:waveform_len])
```

(continues on next page)

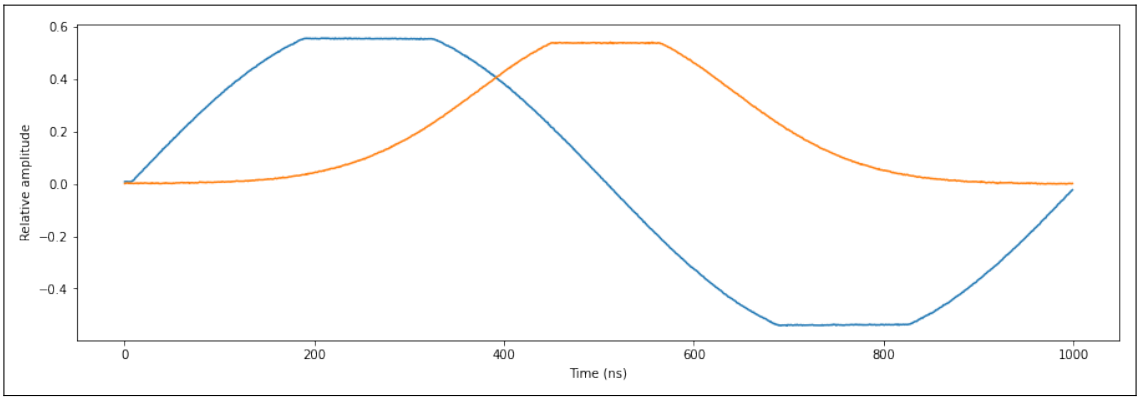
(continued from previous page)

```
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")

matplotlib.pyplot.show()
```



Let's add the sixth sequencer.

[13]:
```
# Configure the sequencer
qrm.sequencer5.sync_en(True)
qrm.sequencer5.gain_awg_path0(1.1 / num_seq)
qrm.sequencer5.gain_awg_path1(1.1 / num_seq)
qrm.sequencer5.channel_map_path0_out0_en(True)
qrm.sequencer5.channel_map_path1_out1_en(True)

# Start the sequencers
for seq in range(0, 6):
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop
for seq in range(0, 6):
    qrm.get_acquisition_state(seq, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

# Plot the results
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))

ax.plot(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_len])
ax.plot(acq["scope"]["acquisition"]["scope"]["path1"]["data"][0:waveform_len])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")

matplotlib.pyplot.show()
```



Note that now the outputs overflow and the output signal is clipped as intended. Also note that the output range of a QRM is 1 Vpp, while it's input range is 2 Vpp. This causes the signal in the figure to be clipped at 0.5 and -0.5.

1.22.6 Frequency multiplexing

Next, we will show frequency multiplexing. We will connect the outputs of various sequencers to a single output port pair, modulate their waveforms on unique carrier frequencies and in turn acquire, demodulate and integrate the results fed back into the inputs to validate the acquired signals. In this case, for simplicity, we will modulate a square wave pulse on each sequencer and we will play with the output gain. In order to visualise the frequency multiplexing, we will preform an FFT over the acquired scope acquisitions.

[14]:
```
# Reset
if cluster:
    cluster.reset()
else:
    qrm.reset()
# Program
program = """
      wait_sync 4
loop: play      3,3,4
      wait      140
      acquire   0,0,16380
      stop
"""

# Write sequence to file
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(
        {
            "waveforms": waveforms,
            "weights": waveforms,
            "acquisitions": acquisitions,
            "program": program,
        },
        file,
        indent=4,
    )
    file.close()
```

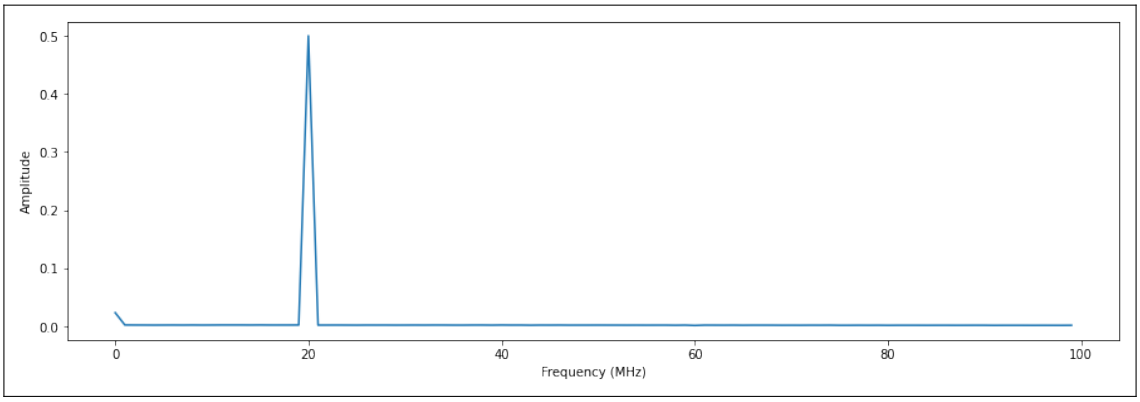
Lets start with a single sequencer with an AWG gain of 1.0 (only on path 0 to create a "real" pulse). Let's modulate it's output with a carrier frequency of 20MHz.

[15]:
```
# Program sequencer
qrm.sequencer0.sequence("sequence.json")

# Configure the channel map
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)

# Configure sequencer
qrm.sequencer0.sync_en(True)
qrm.sequencer0.gain_awg_path0(1.0)
qrm.sequencer0.gain_awg_path1(0.0)
qrm.sequencer0.nco_freq(20e6)
qrm.sequencer0.mod_en_awg(True)
qrm.sequencer0.demod_en_acq(True)
qrm.sequencer0.integration_length_acq(waveform_len)
```

[16]:
```
# Start the sequencer
qrm.arm_sequencer(0)
qrm.start_sequencer(0)

# Wait for the sequencer to stop
qrm.get_acquisition_state(0, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)
```

Now lets have a look at the FFT of the scope acquisition to verify the presence of one tone at 20MHz.

[17]:
```
# Plot the FFT
fig, bx = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
yf = abs(rfft(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_
↪len]))
xf = rfftfreq(1000, 1 / 1e3)
norm_fact = qrm.sequencer0.gain_awg_path0() / 2 / numpy.max(yf)
bx.plot(xf[0:100], yf[0:100] * norm_fact)
bx.set_xlabel("Frequency (MHz)")
bx.set_ylabel("Amplitude")
matplotlib.pyplot.show()
```



Now let's have a look at the hardware demodulated and integrated results and check if it matches our expectations. Don't forget that we need to divide the integration results by the integration length. In this case, the integration length is the same as the waveform length.

[18]:
```
bins = acq["scope"]["acquisition"]["bins"]
I = bins["integration"]["path0"][0] / waveform_len
Q = bins["integration"]["path1"][0] / waveform_len
print("Integeration results:")
print("I = {}".format(I))
print("Q = {}".format(Q))
print("R = sqrt(I^2 + Q^2) = {}".format(math.sqrt(I**2 + Q**2)))
```

```
Integeration results:
I = -0.008239863214460186
Q = -0.00370200293111871
R = sqrt(I^2 + Q^2) = 0.009033281324913206
```

The pulse acquired at the inputs is automatically demodulated at 20MHz, but due to phase rotation caused by output-to-input latency the result is not purely real. However the amplitude of the IQ vector is 0.5 as expected because: 1Vpp output range / 2Vpp input range = 0.5.

Now lets increase the number of sequencers to three, each with a slightly different AWG gain. We will modulate the signals of sequencer 0 to 2 with a carrier frequencies at 20MHz, 30MHz and 40MHz respectively.

[19]:
```
num_seq = 3
for seq in range(0, num_seq):
    # Program sequencers
    qrm.sequencers[seq].sequence("sequence.json")

    # Configure the channel map
    qrm.sequencers[seq].channel_map_path0_out0_en(True)
    qrm.sequencers[seq].channel_map_path1_out1_en(True)

    # Configure the sequencers
    qrm.sequencers[seq].sync_en(True)
    qrm.sequencers[seq].mod_en_awg(True)
    qrm.sequencers[seq].demod_en_acq(True)
    qrm.sequencers[seq].integration_length_acq(waveform_len)

# Set the gains
```

(continues on next page)

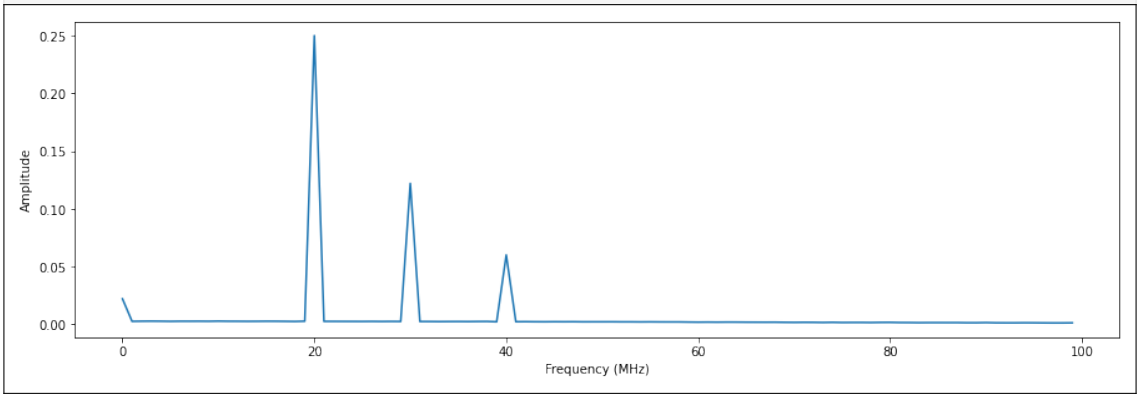
(continued from previous page)

```
qrm.sequencer0.gain_awg_path0(0.5)
qrm.sequencer0.gain_awg_path1(0.0)
qrm.sequencer1.gain_awg_path0(0.25)
qrm.sequencer1.gain_awg_path1(0.0)
qrm.sequencer2.gain_awg_path0(0.125)
qrm.sequencer2.gain_awg_path1(0.0)

# Set the frequencies
qrm.sequencer0.nco_freq(20e6)
qrm.sequencer1.nco_freq(30e6)
qrm.sequencer2.nco_freq(40e6)
```

[20]:
```
# Start the sequencers
for seq in range(0, 3):
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop
for seq in range(0, 3):
    qrm.get_acquisition_state(seq, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)
```

Now lets have a look at the FFT of the scope acquisition to verify the presence of three tones at 20MHz, 30Mhz and 40MHz.

[21]:
```
# Plot the FFT
fig, bx = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
yf = abs(rfft(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_
↪len]))
xf = rfftfreq(1000, 1 / 1e3)
bx.plot(xf[0:100], yf[0:100] * norm_fact)
bx.set_xlabel("Frequency (MHz)")
bx.set_ylabel("Amplitude")
matplotlib.pyplot.show()
```



Now let's check if the hardware demodulated and integrated results match our expectations.

[22]:
```
for seq in range(0, num_seq):
    qrm.store_scope_acquisition(seq, "scope")
    acq = qrm.get_acquisitions(seq)
    bins = acq["scope"]["acquisition"]["bins"]
    I = bins["integration"]["path0"][0] / waveform_len
    Q = bins["integration"]["path1"][0] / waveform_len
    print("Sequencer {}".format(seq))
    print("Integeration results:")
    print("I = {}".format(I))
    print("Q = {}".format(Q))
    print("R = sqrt(I^2 + Q^2) = {}".format(math.sqrt(I**2 + Q**2)))
    print(
        "------------------------------------------------------------------------
↪-----------"
    )
```

```
Sequencer 0
Integeration results:
I = -0.0046311675622862725
Q = -0.0015676599902296042
R = sqrt(I^2 + Q^2) = 0.0048893016715006715
--------------------------------------------------------------------------------
↪--
Sequencer 1
Integeration results:
I = -0.0034816805080605767
Q = 0.00247923790913532
R = sqrt(I^2 + Q^2) = 0.0042741922944929175
--------------------------------------------------------------------------------
↪--
Sequencer 2
Integeration results:
I = -0.002084513922813874
Q = 0.0004640937957987298
R = sqrt(I^2 + Q^2) = 0.002135551766102559
--------------------------------------------------------------------------------
↪--
```

Again, the acquired signals on the input are automatically demodulated at 20MHz, 30MHz and 40MHz and we see that the amplitude of the IQ vectors match the gain values we set divided by two, which matches our expectations.

Now, let's try it one final time with six sequencers, each with 0.15 AWG gain. We will modulate the outputs of sequencer 0 to 5 with carrier frequencies at 20MHz, 30Mhz, 40MHz, 50MHz, 60MHz and 70MHz respectivey.

[23]:
```
num_seq = 6
for seq in range(0, num_seq):
    # Program sequencers
    qrm.sequencers[seq].sequence("sequence.json")

    # Configure the channel map
    qrm.sequencers[seq].channel_map_path0_out0_en(True)
    qrm.sequencers[seq].channel_map_path1_out1_en(True)
```

(continues on next page)

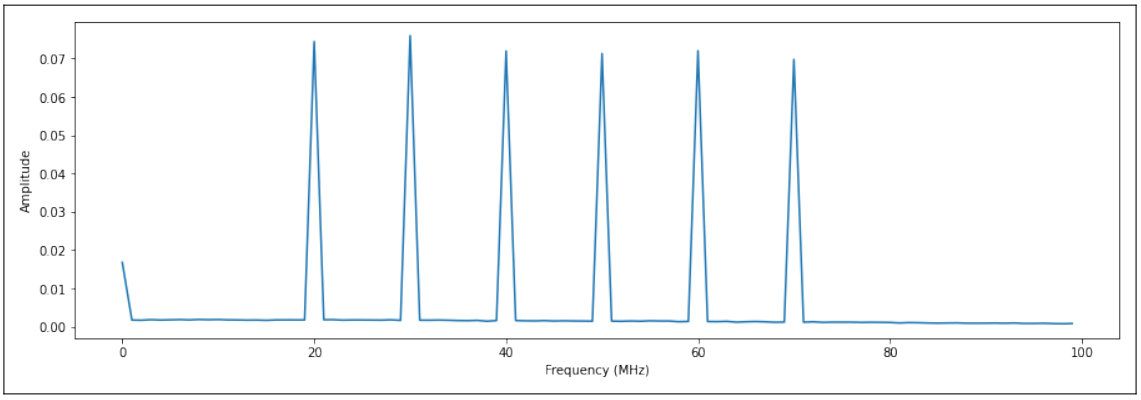
(continued from previous page)

```
    # Configure the sequencers
    qrm.sequencers[seq].sync_en(True)
    qrm.sequencers[seq].gain_awg_path0(0.15)
    qrm.sequencers[seq].gain_awg_path1(0.0)
    qrm.sequencers[seq].mod_en_awg(True)
    qrm.sequencers[seq].demod_en_acq(True)
    qrm.sequencers[seq].integration_length_acq(waveform_len)

# Set the frequencies
qrm.sequencer0.nco_freq(20e6)
qrm.sequencer1.nco_freq(30e6)
qrm.sequencer2.nco_freq(40e6)
qrm.sequencer3.nco_freq(50e6)
qrm.sequencer4.nco_freq(60e6)
qrm.sequencer5.nco_freq(70e6)
```

[24]:
```
# Start the sequencers
for seq in range(0, 6):
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop
for seq in range(0, 6):
    qrm.get_acquisition_state(seq, 1)

# Get acquisition data
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)
```

Now lets have a look at the FFT of the scope acquisition to verify the presence of six tones at 20MHz, 30Mhz, 40MHz, 50MHz, 60MHz and 70MHz

[25]:
```
# Plot the FFT
fig, bx = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
yf = abs(rfft(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:waveform_
↪len]))
xf = rfftfreq(1000, 1 / 1e3)
bx.plot(xf[0:100], yf[0:100] * norm_fact)
bx.set_xlabel("Frequency (MHz)")
bx.set_ylabel("Amplitude")
matplotlib.pyplot.show()
```



Note that we lose a little bit of power over the frequency range, but that is to be expected due to frequency dependant components in the output and input path. Now let's check if the hardware demodulated and integrated results match our expectations.

[26]:
```
for seq in range(0, num_seq):
    qrm.store_scope_acquisition(seq, "scope")
    acq = qrm.get_acquisitions(seq)
    bins = acq["scope"]["acquisition"]["bins"]
    I = bins["integration"]["path0"][0] / waveform_len
    Q = bins["integration"]["path1"][0] / waveform_len
    print("Sequencer {}".format(seq))
    print("Integeration results:")
    print("I = {}".format(I))
    print("Q = {}".format(Q))
    print("R = sqrt(I^2 + Q^2) = {}".format(math.sqrt(I**2 + Q**2)))
    print(
        "------------------------------------------------------------------------
↪-----------"
    )
```

```
Sequencer 0
Integeration results:
I = -0.001826086956521739
Q = -0.0006306790425012213
R = sqrt(I^2 + Q^2) = 0.0019319289913009444
------------------------------------------------------------------------------
↪--
Sequencer 1
Integeration results:
I = -0.0015329750854909622
Q = 0.0002892037127503664
R = sqrt(I^2 + Q^2) = 0.001560016474337569
------------------------------------------------------------------------------
↪--
Sequencer 2
Integeration results:
I = -0.0012208109428431852
Q = 0.0026722032242305813
R = sqrt(I^2 + Q^2) = 0.0029378647739053583
------------------------------------------------------------------------------
↪--
```

(continues on next page)

(continued from previous page)

```
Sequencer 3
Integeration results:
I = 0.001172447484123107
Q = 0.002017098192476795
R = sqrt(I^2 + Q^2) = 0.002333091987282061
--------------------------------------------------------------------------------
--
Sequencer 4
Integeration results:
I = 0.003261846604787494
Q = 0.0005994137762579385
R = sqrt(I^2 + Q^2) = 0.0033164650078557293
--------------------------------------------------------------------------------
--
Sequencer 5
Integeration results:
I = 0.0027655105031753786
Q = -0.0032325354176844165
R = sqrt(I^2 + Q^2) = 0.0042540960931504
--------------------------------------------------------------------------------
--
```

Taking the power loss over frequency into account, the amplitudes of the IQ vectors match our expectations again.

1.22.7 Real mode

Many applications require multiple outputs to be controlled independantly and only output real signals instead of a modulated IQ pair. To achieve this we will connect one sequencer to each output and only use path 0 to control an even numbered output and path 1 to control an odd numbered output. To demonstrate this, we will simply output an independantly timed sine on output O^1 and a sawtooth on output O^2, which we will then acquire on the inputs.

Lets create a Q1ASM program to sequence the waveforms for sequencer 0.

[27]:
```
# Reset
if cluster:
    # reset entire cluster
    cluster.reset()
else:
    # reset the pulsar module
    qrm.reset()
# Program
program = """
wait_sync 4
play      1,1,4
wait      140
acquire   0,0,16380
stop
"""

# Write sequence to file
```

(continues on next page)

(continued from previous page)

```
with open("sequence0.json", "w", encoding="utf-8") as file:
    json.dump(
        {
            "waveforms": waveforms,
            "weights": waveforms,
            "acquisitions": acquisitions,
            "program": program,
        },
        file,
        indent=4,
    )
    file.close()
```

Lets create a Q1ASM program to sequence the waveforms for sequencer 1.

[28]:
```
# Program
program = """
wait_sync 4
wait      500
play      2,2,1000
stop
"""

# Write sequence to file
with open("sequence1.json", "w", encoding="utf-8") as file:
    json.dump(
        {
            "waveforms": waveforms,
            "weights": waveforms,
            "acquisitions": acquisitions,
            "program": program,
        },
        file,
        indent=4,
    )
    file.close()
```

Let's configure both sequencers and connect them to their respective outputs.

[29]:
```
# Configure scope mode
qrm.scope_acq_sequencer_select(0)
qrm.scope_acq_trigger_mode_path0("sequencer")
qrm.scope_acq_trigger_mode_path1("sequencer")

# Program sequencer
num_seq = 2
for seq in range(0, num_seq):
    qrm.sequencers[seq].sequence("sequence{}.json".format(seq))

# Configure the channel map
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer1.channel_map_path1_out1_en(True)
```

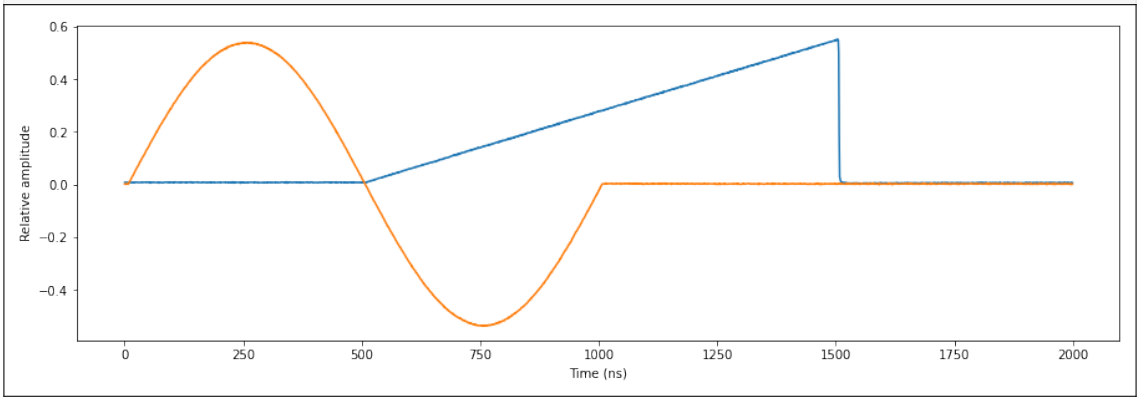
(continues on next page)

(continued from previous page)

```
# Configure sequencer
for seq in range(0, num_seq):
    qrm.sequencers[seq].sync_en(True)
    qrm.sequencers[seq].mod_en_awg(False)

qrm.sequencer0.gain_awg_path1(
    0.0
)  # Disable sequencer 0 path 1, because we will not use it.
qrm.sequencer1.gain_awg_path0(
    0.0
)  # Disable sequencer 1 path 0, because we will not use it.
```

Now, let start the sequencers and visualise the resulting sequence.

[30]:
```
# Start sequencers
for seq in range(0, num_seq):
    qrm.arm_sequencer(seq)
qrm.start_sequencer()

# Wait for sequencers to stop (only sequencer 0 will acquire)
for seq in range(0, num_seq):
    qrm.get_sequencer_state(seq, 1)
qrm.get_acquisition_state(0, 1)

# Get acquisition
qrm.store_scope_acquisition(0, "scope")
acq = qrm.get_acquisitions(0)

# Plot result
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15 / 2 / 1.61))
ax.plot(acq["scope"]["acquisition"]["scope"]["path0"]["data"][0:2000])
ax.plot(acq["scope"]["acquisition"]["scope"]["path1"]["data"][0:2000])
ax.set_xlabel("Time (ns)")
ax.set_ylabel("Relative amplitude")
matplotlib.pyplot.show()
```



As expected, we see a sine and sawtooth that are independantly sequenced on the outputs.

1.22.8 Stop

Finally, let's close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# qrm.print_readable_snapshot(update=True)

# Close the instrument connection.
# Close the instrument connection.
Pulsar.close_all()
Cluster.close_all()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`rf_control.ipynb`

1.23 RF control

In this tutorial we will demonstrate control and calibration of RF modules. We'll start of by performing spectroscopy: sweeping the LO frequency over a frequency range and measuring the through response (S21) of the device under test. In the second part of the tutorial we'll take a look at the spectrum of the output signal at a fixed frequency. We will see some unwanted signals and remove these by calibrating the internal upconverting mixer. To do this, we will connect a spectrum analyser to output O^1 and sweep the relevant parameters (I/Q DC offsets and IF phase/amplitude) by hand.

To run this tutorial, you will need: * QRM-RF module * Spectrum analyser * Device under test: a T-splitter * Two SMA-cables * Installation and enabling of ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.23.1 Setup

First, we are going to import the required packages.

[1]:
```
# Import ipython widgets
import json
import math
import os

import ipywidgets as widgets
import matplotlib.pyplot
import numpy as np

# Set up the environment.
import scipy.signal
from IPython.display import display
```

(continues on next page)

(continued from previous page)

```
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar
```

Scan For Devices

We scan for the available devices connected via ethernet using the Plug & Play functionality of the Qblox Instruments package (see Plug & Play for more info).

[2]:
```
# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Device",
)
print(
    "The following widget displays all the existing modules that are connected␣
↪to your PC which includes the Pulsar modules as well as a Cluster. Select␣
↪the device you want to run the notebook on."
)
display(connect)
```

```
The following widget displays all the existing modules that are connected to␣
↪your PC which includes the Pulsar modules as well as a Cluster. Select the␣
↪device you want to run the notebook on.
```

```
Dropdown(description='Select Device', options=('cluster-mm',), value='cluster-
↪mm')
```

Cluster QRM-RF

First we connect to the Cluster using its IP address.

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect.value
device_number = connect.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]


# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
```

(continues on next page)

(continued from previous page)

```
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.0.2
```

We then find all available cluster modules to connect to them individually.

[4]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

# List of all QxM modules present
connect_qrm_rf = widgets.Dropdown(options=[key for key in available_slots.
↪keys()])

print(available_slots)
# display widget with cluster modules
print()
print("Select the QRM-RF module from the available modules in your Cluster:")
display(connect_qrm_rf)
```

```
{'module3': 'QRM', 'module5': 'QCM-RF', 'module8': 'QCM', 'module16': 'QRM-RF'}

Select the QRM-RF module from the available modules in your Cluster:
```

```
Dropdown(options=('module3', 'module5', 'module8', 'module16'), value='module3
↪')
```

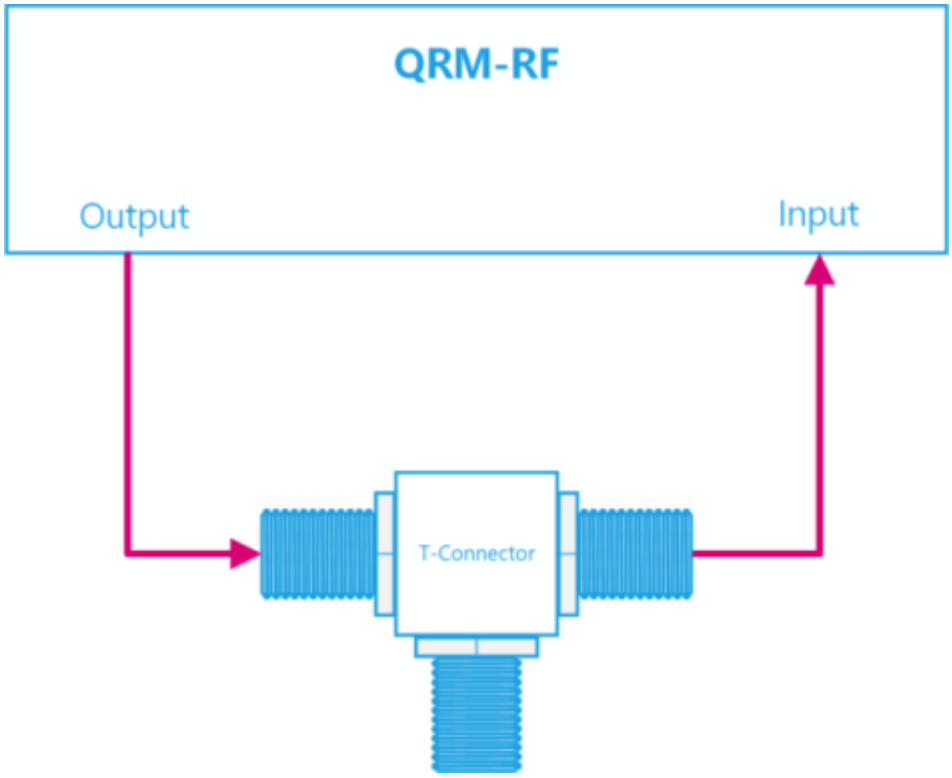
Finally, we connect to the selected Cluster module.

[5]:
```
# Connect to the cluster QxM module
module = connect_qrm_rf.value
qrm_rf = getattr(cluster, module)
print(f"{available_slots[connect_qrm_rf.value]} connected")

print(cluster.get_system_state())
```

```
QRM-RF connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

1.23.2 Spectroscopy

We will sweep the local oscillator across the full range and plot the frequency response. Connect the output of QRM-RF to its input via a T-connecter (one end of the T-connecter should be left open so as to produce a resonator) as shown in the image below. Be aware: this will not work if a splitter is used.



First we need to define our waveforms and acquisition memory. As we are using the NCO to generate an IF signal, we can use a constant (DC) waveform. We need to make sure that the waveform is long enough to run during the whole acquisition integration length and to compensate for the delay of output to input (the `holdoff_length`). Finally we'll also add some averaging to increase the quality of the result.

[6]:
```
# Parameters
no_averages = 10
integration_length = 1000
holdoff_length = 200
waveform_length = integration_length + holdoff_length

# Create DC waveform
waveforms = {"dc": {"data": [0.5 for i in range(0, waveform_length)], "index":␣
↪0}}
```

We will only use a single bin in this tutorial, so we can keep it simple.

[7]:
```
# Acquisitions
acquisitions = {"acq": {"num_bins": 1, "index": 0}}
```

Now that we have the waveform and acquisition specifications for the sequence, we need a simple Q1ASM program that sequences the waveforms and triggers the acquisitions. This program plays a DC wave, waits for the specified hold-off time and then acquires the signal. It repeats this process for the amount of averages specified.

[8]:
```
# Sequence program.
seq_prog = """
      wait_sync 4
      move    0,R1         #Average iterator.
      nop

loop: play    0,0,4
      wait    {}           #Wait the hold-off time
      acquire 0,0,{}       #Acquire bins and store them in "avg" acquisition.
      add     R1,1,R1      #Increment avg iterator
      nop                  #Wait a cycle for R1 to be available.
      jlt     R1,{},@loop  #Run until number of average iterations is done.

      stop                 #Stop.
""".format(
    holdoff_length, integration_length, no_averages
)

# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
```

Now lets configure the device and the sequencer. In the RF modules there is a switch directly before the output connector, we need to turn this on to get any signal out of the device. It is connected to a marker, so for now we override the marker values and fix them. This can of course also be done in the sequence program.

Additionally we'll also set the output path DC offset to a known 'decent' value. We'll go into this parameter a bit further on. All the other parameters are to set the NCO frequency and to make sure the acquisition is configured correctly. For more information on these parameters, see their respective tutorials.

[9]:
```
qrm_rf.sequencer0.marker_ovr_en(True)
qrm_rf.sequencer0.marker_ovr_value(15)  # Enables output on QRM-RF

qrm_rf.out0_offset_path0(5.5)
qrm_rf.out0_offset_path1(5.5)

# Configure scope mode
qrm_rf.scope_acq_sequencer_select(0)
qrm_rf.scope_acq_trigger_mode_path0("sequencer")
qrm_rf.scope_acq_trigger_mode_path1("sequencer")

# Configure the sequencer
qrm_rf.sequencer0.mod_en_awg(True)
qrm_rf.sequencer0.demod_en_acq(True)
qrm_rf.sequencer0.nco_freq(100e6)
```

(continues on next page)

(continued from previous page)

```
qrm_rf.sequencer0.integration_length_acq(integration_length)
qrm_rf.sequencer0.sync_en(True)
```

Now we are ready to start the actual spectroscopy.

[10]:
```
lo_sweep_range = np.linspace(2e9, 18e9, 200)

lo_data_0 = []
lo_data_1 = []

for lo_val in lo_sweep_range:
    # Update the LO frequency.
    qrm_rf.out0_in0_lo_freq(lo_val)

    # Upload sequence. This clears the acquisitions.
    qrm_rf.sequencer0.sequence("sequence.json")

    qrm_rf.arm_sequencer(0)
    qrm_rf.start_sequencer()

    # Wait for the sequencer to stop with a timeout period of one minute.
    qrm_rf.get_acquisition_state(0, 1)

    # Move acquisition data from temporary memory to acquisition list.
    qrm_rf.store_scope_acquisition(0, "acq")

    # Get acquisition list from instrument.
    data = qrm_rf.get_acquisitions(0)["acq"]

    # Store the acquisition data.
    lo_data_0.append(data["acquisition"]["bins"]["integration"]["path0"][0])
    lo_data_1.append(data["acquisition"]["bins"]["integration"]["path1"][0])

# The result still needs to be divided by the integration length to make sure
# the units are correct.
lo_data_0 = np.asarray(lo_data_0) / integration_length
lo_data_1 = np.asarray(lo_data_1) / integration_length
```

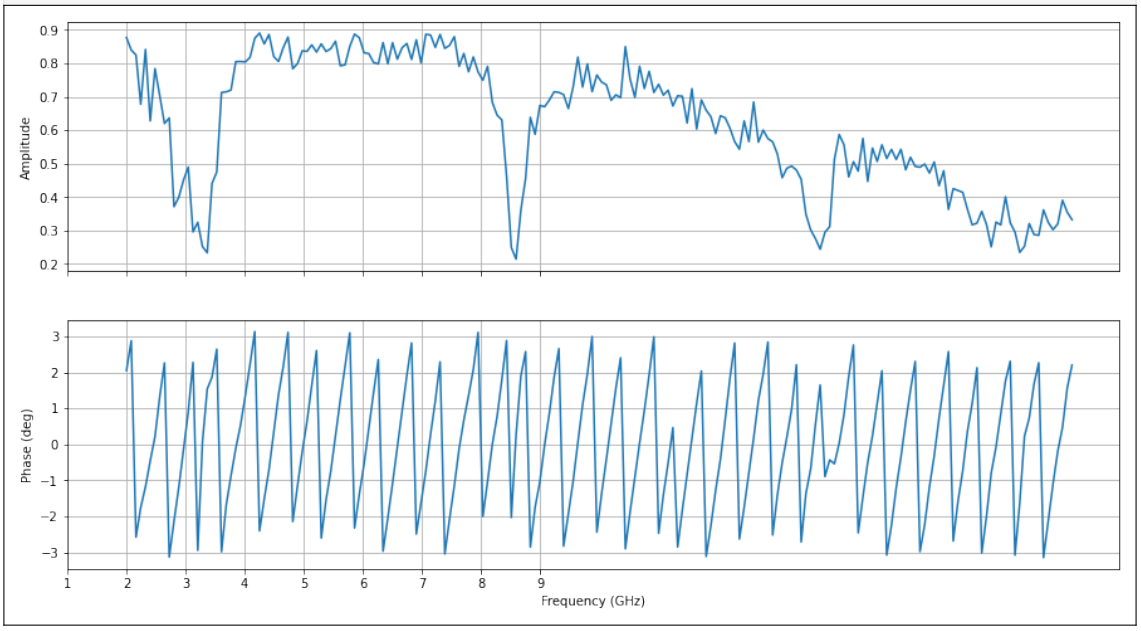
We can now plot the results. For ease of viewing we'll first convert them to amplitude and phase.

[11]:
```
amplitude = np.sqrt(lo_data_0**2 + lo_data_1**2)
phase = np.arctan2(lo_data_1, lo_data_0)

fig, [ax1, ax2] = matplotlib.pyplot.subplots(2, 1, sharex=True, figsize=(15,␣
↪8))

ax1.plot(lo_sweep_range / 1e9, amplitude)
ax1.grid(True)
ax1.set_ylabel("Amplitude")

ax2.plot(lo_sweep_range / 1e9, phase)
ax2.grid(True)
```

(continues on next page)

(continued from previous page)

```
ax2.set_ylabel("Phase (deg)")
ax2.set_xlabel("Frequency (GHz)")
ax2.set_xticks(np.arange(1, 10, 1))
matplotlib.pyplot.show()
```



From the spectroscopy we can see that the first resonance dip of the resonator is at roughly 3.5 GHz.

1.23.3 Mixer calibration

For this section, we are going to look at the output spectrum of the QRM at a fixed (output) frequency of 5 GHz. We start by resetting the device to make sure it's in a known state and then upload a simple sequence program that keeps playing the DC waveform. This will be modulated and upconverted within the QRM-RF before outputting.

[24]:
```
cluster.reset()

# Sequence program.
seq_prog = """
      wait_sync 4

loop: play    0,0,1200
      jmp     @loop
"""

# Add sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": waveforms,
    "weights": {},
    "acquisitions": acquisitions,
    "program": seq_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
```

(continues on next page)

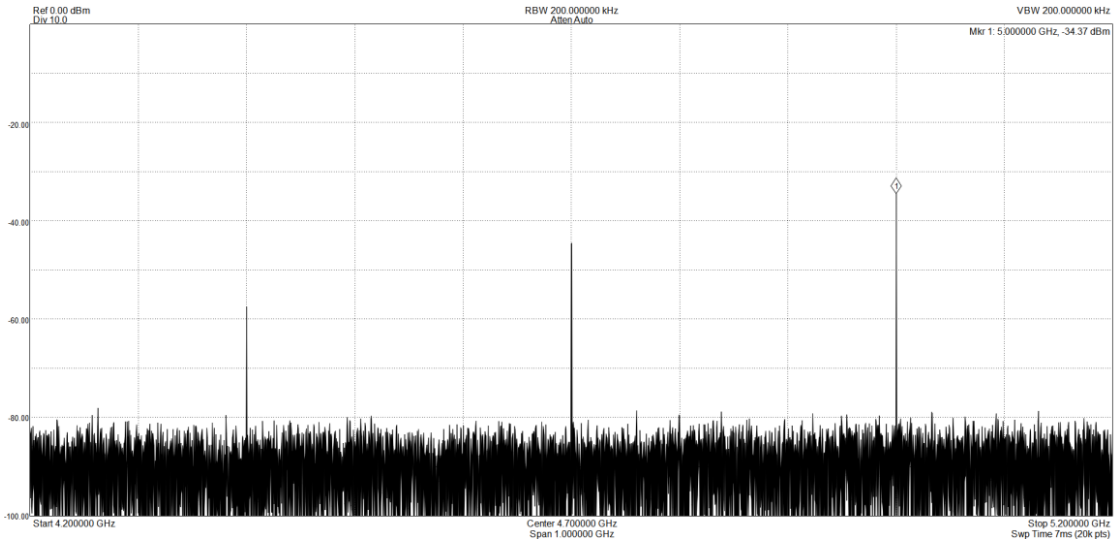
(continued from previous page)

```
    json.dump(sequence, file, indent=4)
    file.close()

qrm_rf.sequencer0.sequence("sequence.json")
```

Let's configure the sequencer to generate an IF frequency of 300 MHz. To get an output frequency of 5 GHz, we then have to configure the LO to run at 4.7 GHz.

[ ]:
```
# Configure the Local oscillator
qrm_rf.out0_in0_lo_freq(5e9 - 300e6)

qrm_rf.sequencer0.marker_ovr_en(True)
qrm_rf.sequencer0.marker_ovr_value(15)  # Enables output on QRM-RF

# Configure the sequencer
qrm_rf.sequencer0.mod_en_awg(True)
qrm_rf.sequencer0.nco_freq(300e6)
qrm_rf.sequencer0.sync_en(True)

qrm_rf.arm_sequencer(0)
qrm_rf.start_sequencer(0)

print("Status:")
print(qrm_rf.get_sequencer_state(0))
```

Connect the output of the QRM-RF (O1) to the spectrum analyzer. This is what the output looks like on the spectrum analyser (center frequency at 4.7 GHz with 1 GHz bandwidth).



As we can see from this image, the output is not exactly the single peak at 5 GHz that we want. We seem to have some LO leakage (at 4.7 GHz) and an unwanted sideband (4.4 GHz). This is due to mixer imperfections. We can suppress these by calibrating the mixer:

- Using DC offsets we can lower the LO leakage.
- By changing the gain ratio and phase of the IF signal we can cancel the unwanted sideband.

Create control sliders for theses parameters. Each time the value of a parameter is updated, the sequencer

is automatically stopped from the embedded firmware for safety reasons and has to be manually restarted. The sliders cover the valid parameter range. If the code below is modified to input invalid values, the Pulsar firmware will not program the values.

Execute the code below, move the sliders and observe the result on the spectrum analyser.

[ ]:
```
def set_offset0(offset0):
    qrm_rf.out0_offset_path0(offset0)


def set_offset1(offset1):
    qrm_rf.out0_offset_path1(offset1)


def set_gain_ratio(gain_ratio):
    qrm_rf.sequencer0.mixer_corr_gain_ratio(gain_ratio)
    # Start
    qrm_rf.arm_sequencer(0)
    qrm_rf.start_sequencer(0)


def set_phase_offset(phase_offset):
    qrm_rf.sequencer0.mixer_corr_phase_offset_degree(phase_offset)
    # Start
    qrm_rf.arm_sequencer(0)
    qrm_rf.start_sequencer(0)


interact(
    set_offset0,
    offset0=widgets.FloatSlider(
        min=-14.0,
        max=14.0,
        step=0.001,
        start=0.0,
        layout=widgets.Layout(width="1200px"),
    ),
)
interact(
    set_offset1,
    offset1=widgets.FloatSlider(
        min=-14.0,
        max=14.0,
        step=0.001,
        start=0.0,
        layout=widgets.Layout(width="1200px"),
    ),
)
interact(
    set_gain_ratio,
    gain_ratio=widgets.FloatSlider(
        min=0.9, max=1.1, step=0.001, start=1.0, layout=widgets.Layout(width=
↪"1200px")
    ),
```

(continues on next page)

(continued from previous page)

```
)
interact(
    set_phase_offset,
    phase_offset=widgets.FloatSlider(
        min=-45.0,
        max=45.0,
        step=0.001,
        start=0.0,
        layout=widgets.Layout(width="1200px"),
    ),
)
```

1.23.4 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[ ]:
```
# Stop sequencer.
qrm_rf.stop_sequencer()

# Print status of sequencer.
print(qrm_rf.get_sequencer_state(0))
print()

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of the instrument parameters.
# print("Snapshot:")
# cluster.print_readable_snapshot(update=True)

# Close the instrument connection.
cluster.close()
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`rabi_experiment.ipynb`

1.24 Rabi experiment

In this tutorial we will combine the techniques explained in the other tutorials and show how to perform a Rabi experiment. For this tutorial we will need one QCM to generate the Rabi pulses and one QRM to perform the readout, although the QCM could be replaced by another QRM if needed.

Ports $O^{[1-2]}$ of the QCM are used for the driving pulse, while $O^{[1-2]}$ of the QRM are used for the readout pulse. Finally, ports $I^{[1-2]}$ are used for the acquisition of the readout tone. In this tutorial it is assumed O^1 of the QRM is connected to I^1 of the QRM for time of flight calibration. Furthermore we assume that O^1 of the QCM and O^2 of the QRM are connected to an external oscilloscope to view the Rabi experiment pattern. The scope can be triggered of marker 1 of the QCM.

As demonstrated in the synchronization tutorial, the SYNQ technology synchronizes the programs in the two modules. If you are using Pulsar modules, to ensure synchronization between the modules, connect the REF^{out} of the Pulsar QCM to the REF^{in} of the Pulsar QRM using a coaxial cable, and connect their SYNQ ports using the SYNQ cable.

If you are using a Cluster, then no extra connection needs to be made as the modules are internally connected with SYNQ capability.

To run this tutorial please make sure you have installed and enabled ipywidgets:

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

1.24.1 Setup

First, we are going to import the required packages and connect to the instrument.

[1]:
```
# Import ipython widgets
import json
import math

import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np

# Set up the environment.
import scipy.signal
from IPython.display import display
from ipywidgets import fixed, interact, interact_manual, interactive

from qblox_instruments import Cluster, PlugAndPlay, Pulsar

# Scan for available devices and display
with PlugAndPlay() as p:
    # get info of all devices
    device_list = p.list_devices()
    device_keys = list(device_list.keys())

# create widget for names and ip addresses
connect_qcm = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select QCM",
)
connect_qrm = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select QRM",
)
connect_cluster = widgets.Dropdown(
    options=[(device_list[key]["description"]["name"]) for key in device_list.
↪keys()],
    description="Select Cluster",
)
```

Pulsar

Select the Pulsar QCM and QRM modules by running the following cell. Skip to the *Cluster* if you are using a Cluster.

[2]:
```
display(connect_qcm)
display(connect_qrm)
```

```
Dropdown(description='Select QCM', options=('pulsar-qcm', 'pulsar-qrm'), value=
↪'pulsar-qcm')
```

```
Dropdown(description='Select QRM', options=('pulsar-qcm', 'pulsar-qrm'), value=
↪'pulsar-qcm')
```

[3]:
```
# Close existing connections to Pulsar
Pulsar.close_all()

# Retrieve device name and IP address and Connect
# QCM
qcm_device_name = connect_qcm.value
qcm_device_number = connect_qcm.options.index(qcm_device_name)
qcm_ip_address = device_list[device_keys[qcm_device_number]]["identity"]["ip"]
qcm = Pulsar(f"{qcm_device_name}", qcm_ip_address)
qcm.reset()  # reset all params
print(f"{qcm_device_name} connected at {qcm_ip_address}")
print(qcm.get_system_state())

# QRM
qrm_device_name = connect_qrm.value
qrm_device_number = connect_qrm.options.index(qrm_device_name)
qrm_ip_address = device_list[device_keys[qrm_device_number]]["identity"]["ip"]
qrm = Pulsar(f"{qrm_device_name}", qrm_ip_address)
qrm.reset()  # reset all params
print(f"{qrm_device_name} connected at {qrm_ip_address}")
print(qrm.get_system_state())
```

```
pulsar-qcm connected at 192.168.0.3
Status: OKAY, Flags: NONE, Slot flags: NONE
pulsar-qrm connected at 192.168.0.4
Status: OKAY, Flags: NONE, Slot flags: NONE
```

[4]:
```
# Set reference clock source.
qrm.reference_source("external")
```

Skip to the next section *(Sequencer Setup)* if you are not using a cluster.

Cluster

First we connect to the Cluster using its IP address. Go to the *Pulsar section* if you are using a Pulsar.

[2]:
```
display(connect_cluster)
```

```
Dropdown(description='Select Cluster', options=('cluster-mm',), value='cluster-
↪mm')
```

[3]:
```
# close all previous connections to the cluster
Cluster.close_all()

# Retrieve device name and IP address
device_name = connect_cluster.value
device_number = connect_cluster.options.index(device_name)
ip_address = device_list[device_keys[device_number]]["identity"]["ip"]

# connect to the cluster and reset
cluster = Cluster(device_name, ip_address)
cluster.reset()
print(f"{device_name} connected at {ip_address}")
```

```
cluster-mm connected at 192.168.0.2
```

We then find all available cluster modules to connect to them individually.

[4]:
```
# Find all QRM/QCM modules
available_slots = {}
for module in cluster.modules:
    # if module is currently present in stack
    if cluster._get_modules_present(module.slot_idx):
        # check if QxM is RF or baseband
        if module.is_rf_type:
            available_slots[f"module{module.slot_idx}"] = ["QCM-RF", "QRM-RF"][
                module.is_qrm_type
            ]
        else:
            available_slots[f"module{module.slot_idx}"] = ["QCM", "QRM"][
                module.is_qrm_type
            ]

connect_qcm = widgets.Dropdown(
    options=[key for key in available_slots.keys()], description="Select QCM"
)
connect_qrm = widgets.Dropdown(
    options=[key for key in available_slots.keys()], description="Select QRM"
)
```

Select the QCM and QRM module from the available modules in your Cluster.

[5]:
```
print(available_slots)
print()
display(connect_qcm)
display(connect_qrm)
```

```
{'module2': 'QCM', 'module4': 'QCM', 'module6': 'QRM', 'module8': 'QCM-RF'}
```

```
Dropdown(description='Select QCM', options=('module2', 'module4', 'module6',
↪'module8'), value='module2')
```

```
Dropdown(description='Select QRM', options=('module2', 'module4', 'module6',
↪'module8'), value='module2')
```

Finally, we connect to the selected Cluster module.

[6]:
```
# Connect to QCM and QRM
qcm = getattr(
    cluster, connect_qcm.value
)  # Connect to the module that you have chosen above
print(f"{available_slots[connect_qcm.value]} connected")

qrm = getattr(cluster, connect_qrm.value)
print(f"{available_slots[connect_qrm.value]} connected")
print(cluster.get_system_state())
```

```
QCM connected
QRM connected
Status: OKAY, Flags: NONE, Slot flags: NONE
```

1.24.2 Sequencer Setup

Set sync_en to synchronize across modules.

[7]:
```
# Set sync_en
qrm.sequencer0.sync_en(True)
qcm.sequencer0.sync_en(True)
```

Configure the NCO of both the QRM and QCM to 100 MHz and enable the up- and down-conversion in the sequencers

[8]:
```
qrm.sequencer0.nco_freq(100e6)
qrm.sequencer0.mod_en_awg(True)
qrm.sequencer0.demod_en_acq(True)

qcm.sequencer0.nco_freq(100e6)
qcm.sequencer0.mod_en_awg(True)
```

Configure the outputs of the QRM and QCM such that sequencer0 is the only enabled sequencer and maps to $O^{[1-2]}$

[9]:
```
# Map sequencer of the QCM to specific outputs (but first disable all␣
↪sequencer connections)
for sequencer in qcm.sequencers:
    for out in range(0, 2):
        sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)
qcm.sequencer0.channel_map_path0_out0_en(True)
```

(continues on next page)

(continued from previous page)

```
qcm.sequencer0.channel_map_path1_out1_en(True)

# Map sequencer of the QRM to specific outputs (but first disable all␣
↪sequencer connections)
for sequencer in qrm.sequencers:
    for out in range(0, 2):
        sequencer.set("channel_map_path{}_out{}_en".format(out % 2, out),␣
↪False)
qrm.sequencer0.channel_map_path0_out0_en(True)
qrm.sequencer0.channel_map_path1_out1_en(True)
```

1.24.3 Define waveforms

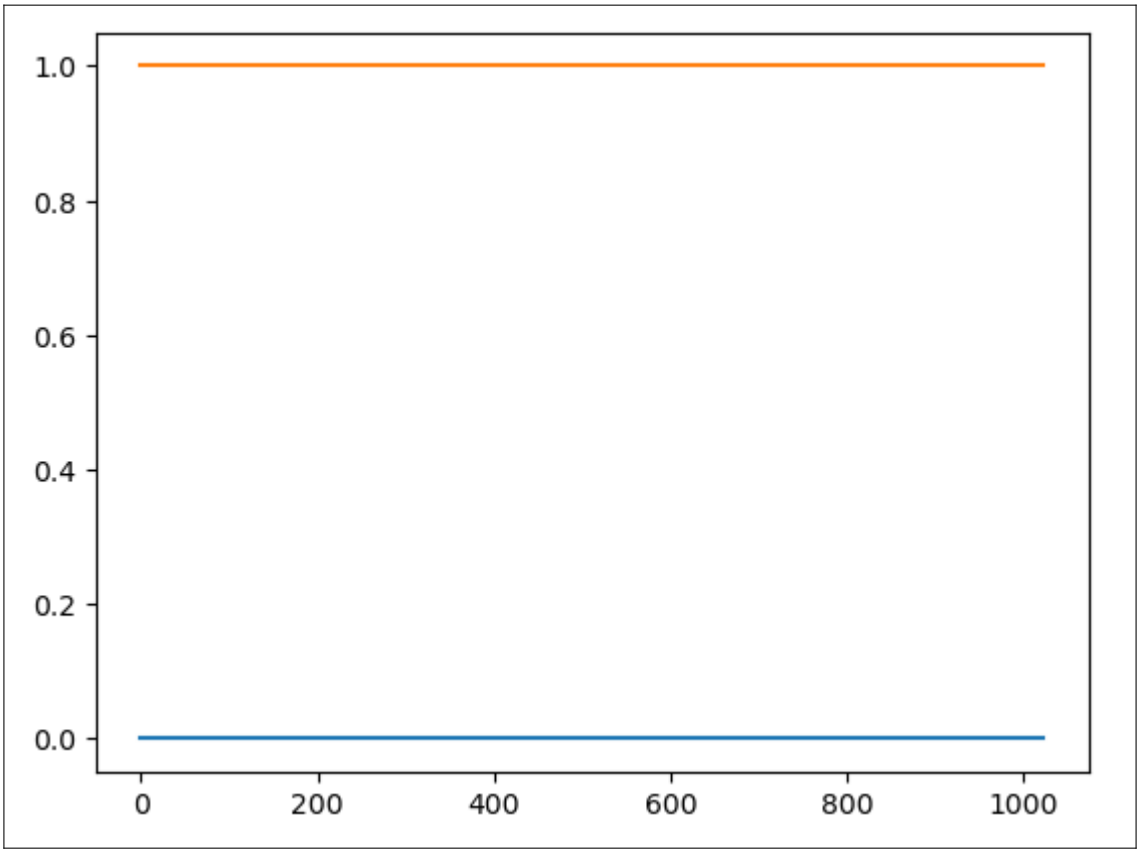
To readout the systems we define constant pulses `one` and `zero` which will be up converted by the `NCO` to create the appropriate tones for an IQ mixer. Similarly for driving the qubit, we define a Gaussian pulse, together with a zero pulse of equal length to serve as inputs for an IQ mixer. In this tutorial we do not assume mixers to be connected to the inputs and outputs of the QCM/QRM.

[10]:
```
t = np.arange(-80, 81, 1)
sigma = 20
wfs = {
    "zero": {"index": 0, "data": [0.0] * 1024},
    "one": {"index": 1, "data": [1.0] * 1024},
    "gauss": {"index": 2, "data": list(np.exp(-(0.5 * t**2 / sigma**2)))},
    "empty": {"index": 3, "data": list(0.0 * t)},
}
```

Hence we obtain the following waveforms for readout:

[11]:
```
plt.plot(wfs["zero"]["data"])
plt.plot(wfs["one"]["data"])
```

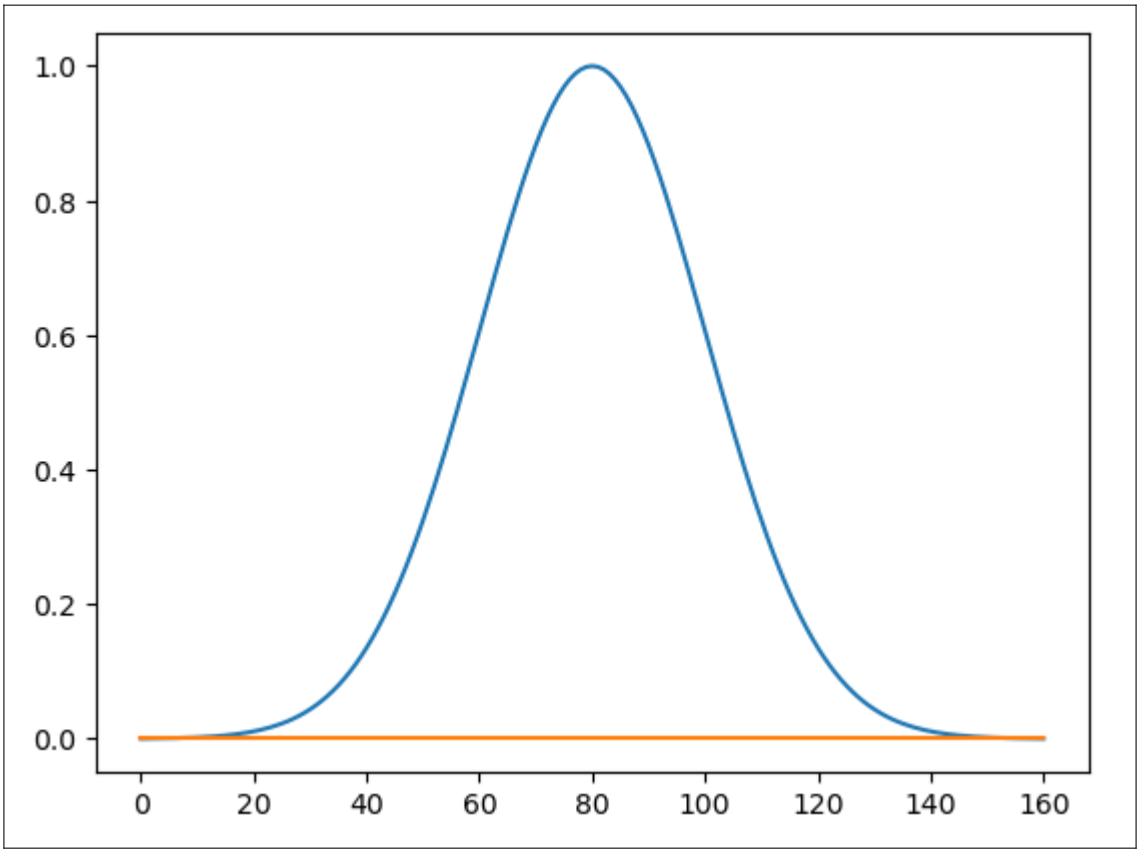
[11]:
```
[<matplotlib.lines.Line2D at 0x29bf811b850>]
```



And for drive:

[12]:
```
plt.plot(wfs["gauss"]["data"])
plt.plot(wfs["empty"]["data"])
```

[12]:
```
[<matplotlib.lines.Line2D at 0x29bf8361bb0>]
```



Finally, we define two acquisitions. A single readout to perform the calibration measurements. Secondly we create a `rabi` readout sequence that contains 50 different bins for saving the results, one for each of the different amplitudes used for the drive tone.

[13]:
```
num_bins = 50  # Number of amplitudes to be measured
acquisitions = {
    "single": {"num_bins": 1, "index": 0},
    "rabi": {"num_bins": num_bins, "index": 1},
}
```

1.24.4 Calibration experiments

TOF calibration

As a first step, we calibrate the time of flight (tof) for the QRM module. In order to do so, we play a readout pulse and analyze the obtained signal on the oscilloscope to find the travel time of the pulse through the system.

[14]:
```
qrm_prog = f"""
play    1, 0, 4     # start readout pulse
acquire 0, 0, 16384 # start the 'single' acquisition sequence and wait for the␣
↪length of the scope acquisition window
stop
"""
```

Upload the program, together with the waveforms and acquisitions to the QRM

[15]:
```
sequence = {
    "waveforms": wfs,
    "weights": {},
    "acquisitions": acquisitions,
    "program": qrm_prog,
}
with open("sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()
# Upload sequence.
qrm.sequencer0.sequence("sequence.json")
```

Perform the calibration experiment

[16]:
```
# Arm and start sequencer.
qrm.arm_sequencer(0)
qrm.start_sequencer()

# Wait for the sequencer and acquisition to finish with a timeout period of␣
↪one minute.
qrm.get_acquisition_state(0, 1)
qrm.store_scope_acquisition(0, "single")
# Print status of sequencer.
print(qrm.get_sequencer_state(0))
```

```
Status: STOPPED, Flags: FORCED_STOP, ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_DONE_
↪PATH_1, ACQ_BINNING_DONE
```

Analyze the resulting signal on the scope to find the tof

[17]:
```
p0 = np.array(
    qrm.get_acquisitions(0)["single"]["acquisition"]["scope"]["path0"]["data"]
)
p1 = np.array(
    qrm.get_acquisitions(0)["single"]["acquisition"]["scope"]["path1"]["data"]
)
# Determine when the signal crosses half-max for the first time (in ns)
t_halfmax = np.where(np.abs(p0) > np.max(p0) / 2)[0][0]

# The time it takes for a sine wave to reach its half-max value is (in ns)
correction = 1 / qrm.sequencer0.nco_freq() * 1e9 / 12

tof_measured = t_halfmax - correction
```

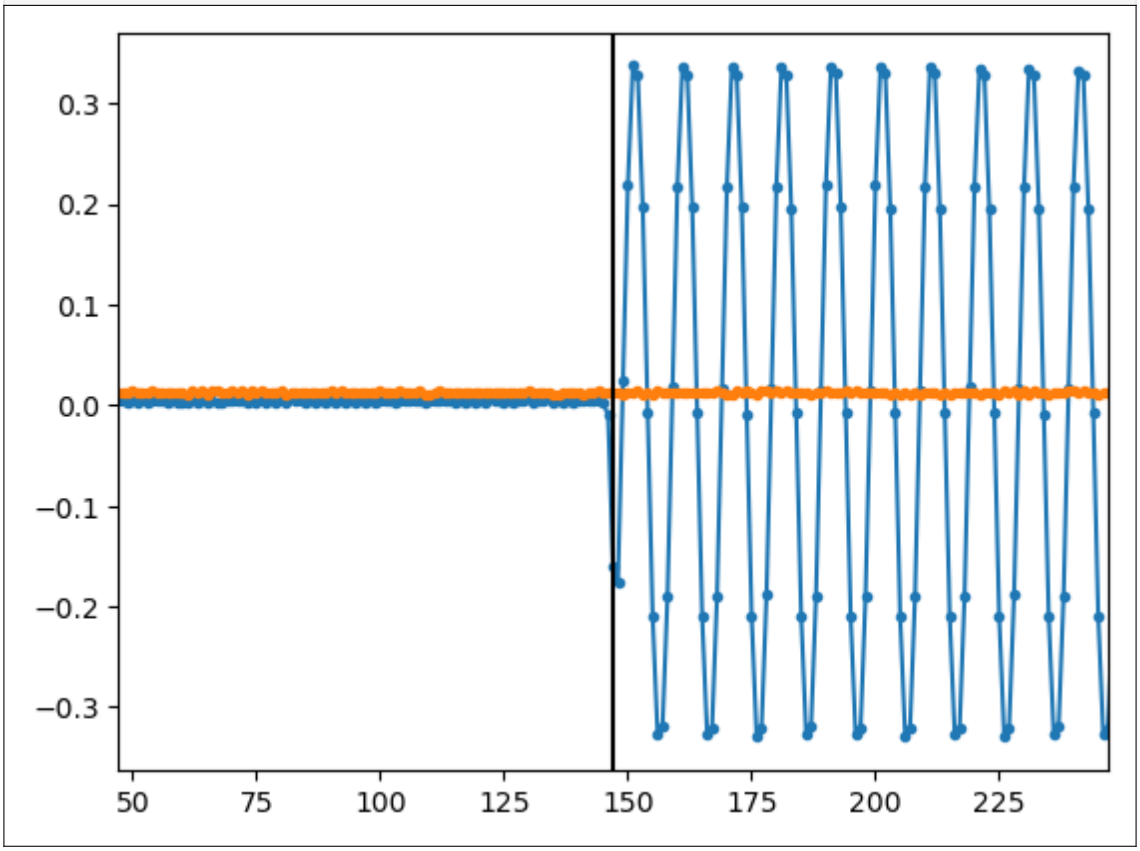
Plot the signal on the scope, around the rising and falling edge of the acquisition signal, as determined by the tof analysis above:

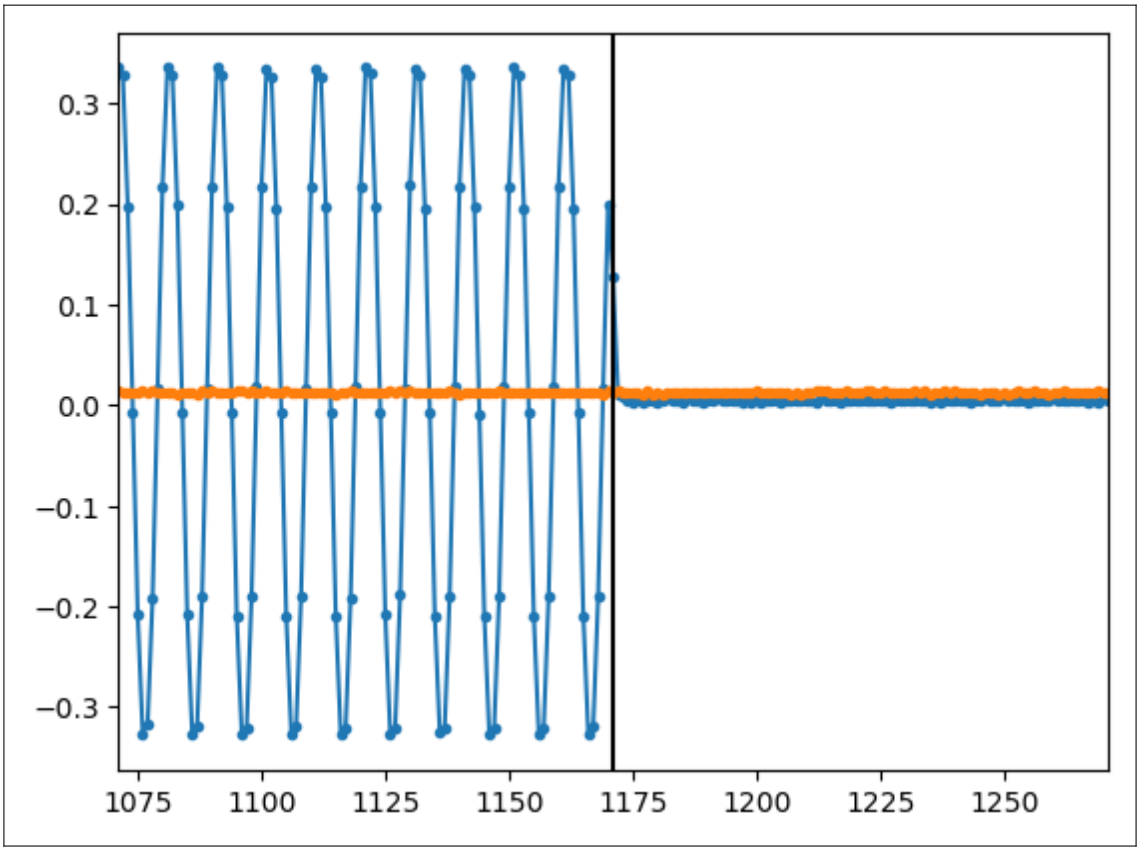
[18]:
```
r = qrm.get_acquisitions(0)["single"]["acquisition"]["scope"]
plt.plot(r["path0"]["data"], ".-")
plt.plot(r["path1"]["data"], ".-")
plt.axvline(tof_measured, c="k")
plt.xlim(
    tof_measured - 10 / qrm.sequencer0.nco_freq() * 1e9,
```

(continues on next page)

(continued from previous page)

```
    tof_measured + 10 / qrm.sequencer0.nco_freq() * 1e9,
)
plt.show()

plt.plot(r["path0"]["data"], ".-")
plt.plot(r["path1"]["data"], ".-")
plt.axvline(1024 + tof_measured, c="k")
plt.xlim(
    1024 + tof_measured - 10 / qrm.sequencer0.nco_freq() * 1e9,
    1024 + tof_measured + 10 / qrm.sequencer0.nco_freq() * 1e9,
)
plt.show()
```





1.24.5 Parameters

Set the parameters for the Rabi experiment

[19]:
```
# all times must be divisible by 4
reset_time = 200  # reset time for the qubit in microseconds
tof = int(tof_measured / 4) * 4  # time of flight must be divisible by 4
readout_delay = (
    164  # time to delay the readout pulse after the start of the rotation␣
↪pulse
)

navg = 1000  # number of averages
stepsize = int(65535 / 100)
```

1.24.6 Rabi

Normally, a Rabi experiment would be performed by changing the amplitude in the inner loop, and averaging in the outer loop. To make the resulting experiment visible on an oscilloscope however, in this tutorial we swapped these two loops

[20]:
```
# QCM sequence program.
qcm_seq_prog = f"""
# Registers used:
# R0 loops over the different awg amplitudes used for the rabi driving pulse
# R2 is used to count the averages needed for a single amplitude
# R3 contains the qubit reset time in microseconds

          move          0, R0                       # start with awg␣
↪amplitude 0
          wait_sync     4                           # Synchronize the QRM␣
↪with the QCM


ampl_loop: add          R0, {stepsize}, R0          # increase the pulse␣
↪amplitude by the stepsize
          move          {navg}, R2                  # reset the number of␣
↪averages and save in the R2 register

          # let the qubit relax to its groundstate
navg_loop: move         {reset_time}, R3            # reset the number of␣
↪microseconds to wait and save in the R3 register
rst_loop:  wait         1000                        # wait 1 microsecond
          loop          R3,@rst_loop                # repeat the 1␣
↪microsecond wait as much as needed to let the qubit relax

          set_awg_gain  R0, R0                      # Set the new amplitude␣
↪used for the drive pulse
          set_mrk       1                           # Set marker 1 high for␣
↪to enable synchronization with external oscilloscope
          wait_sync     4                           # Synchronize with the␣
↪qrm to signify a measurement is coming
          play          2,3,16384                   # Play waveforms and wait␣
↪remaining duration of scope acquisition

          set_mrk       0                           # Reset marker 1
          upd_param     4

          loop          R2,@navg_loop                       # Repeat the␣
↪experiment to average, until R2 becomes 0
          jlt           R0,{num_bins*stepsize},@ampl_loop   # Repeat the␣
↪experiment for different pulse amplitudes from 0 to num_bins
          stop                                              # Stop.

"""

# QRM sequence program.
qrm_seq_prog = f"""
```

(continues on next page)

(continued from previous page)

```
# Registers used:
# R0 counts which bin to acquire into, a new bin for every new amplitude
# R2 is used to count the averages needed for a single amplitude

          wait_sync     4                         # Synchronize the QRM␣
↪with the QCM.
          move          0, R0                     # the first acquisition␣
↪uses bin 0
ampl_loop: move         {navg}, R2                # reset the amount of␣
↪averages to be taken to the initial value

navg_loop: wait_sync    {readout_delay}           # wait for the QCM to␣
↪signal a pulse is coming and wait the readout_delay
          play          1,0,{tof}                 # play readout pulse and␣
↪wait for the tof
          acquire       1,R0,16384                # Acquire waveforms and␣
↪wait remaining duration of scope acquisition.

          loop          R2, @navg_loop            # Repeat this measurement␣
↪for every average
          add           R0,1,R0                   # Increment the bin into␣
↪which we are measuring
          jmp           @ampl_loop                # repeat
"""
```

Upload programs and waveforms to QRM and QCM

[21]:
```
# Add QCM sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": wfs,
    "weights": {},
    "acquisitions": acquisitions,
    "program": qcm_seq_prog,
}
with open("qcm_sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()

# Add QRM sequence to single dictionary and write to JSON file.
sequence = {
    "waveforms": wfs,
    "weights": {},
    "acquisitions": acquisitions,
    "program": qrm_seq_prog,
}
with open("qrm_sequence.json", "w", encoding="utf-8") as file:
    json.dump(sequence, file, indent=4)
    file.close()

# Upload sequence to QCM.
qcm.sequencer0.sequence("qcm_sequence.json")
```

(continues on next page)

(continued from previous page)

```
# Upload sequence to QRM.
qrm.sequencer0.sequence("qrm_sequence.json")
```

Arm and start sequencer0 of both the QCM and QRM. The `wait_sync` command together with the SYNQ technology ensures both modules start simultaneously.

[22]:
```
# Arm and start sequencer of the QCM (only sequencer 0).
qcm.arm_sequencer(0)
qcm.start_sequencer(0)

# Print status of sequencer of the QCM.
print("QCM:")
print(qcm.get_sequencer_state(0))
print()

# Arm and start sequencer of the QRM (only sequencer 0).
qrm.arm_sequencer(0)
qrm.start_sequencer(0)

# Print status of sequencer of the QRM.
print("QRM:")
print(qrm.get_sequencer_state(0))
print("QCM:")
print(qcm.get_sequencer_state(0, 1))
qrm.stop_sequencer(
    0
)  # We didn't tell the QRM how many different amplitudes would be measured, so␣
↪here we tell it to stop.
print("QRM:")
print(qrm.get_sequencer_state(0, 1))
```

```
QCM:
Status: RUNNING, Flags: NONE

QRM:
Status: RUNNING, Flags: ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_OVERWRITTEN_PATH_0,␣
↪ACQ_SCOPE_DONE_PATH_1, ACQ_SCOPE_OVERWRITTEN_PATH_1, ACQ_BINNING_DONE
QCM:
Status: STOPPED, Flags: NONE
QRM:
Status: STOPPED, Flags: FORCED_STOP, ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_
↪OVERWRITTEN_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_SCOPE_OVERWRITTEN_PATH_1, ACQ_
↪BINNING_DONE
```

1.24.7 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection. One can also display a detailed snapshot containing the instrument parameters before closing the connection by uncommenting the corresponding lines.

[23]:
```
# Stop sequencers.
qcm.stop_sequencer()
qrm.stop_sequencer()

# Print status of sequencers.
print("QCM :")
print(qcm.get_sequencer_state(0))
print()

print("QRM :")
print(qrm.get_sequencer_state(0))
print()

# Uncomment the following to print an overview of the instrument parameters.
# Print an overview of instrument parameters.
# print("QCM snapshot:")
# qcm.print_readable_snapshot(update=True)
print()

# print("QRM snapshot:")
# qrm.print_readable_snapshot(update=True)

# Close the instrument connections.
Pulsar.close_all()
Cluster.close_all()
```

```
QCM :
Status: STOPPED, Flags: FORCED_STOP

QRM :
Status: STOPPED, Flags: FORCED_STOP, ACQ_SCOPE_DONE_PATH_0, ACQ_SCOPE_
↪OVERWRITTEN_PATH_0, ACQ_SCOPE_DONE_PATH_1, ACQ_SCOPE_OVERWRITTEN_PATH_1, ACQ_
↪BINNING_DONE
```

[ ]:
```
```

See also:

An IPython notebook version of this tutorial can be downloaded here:

`spi_rack.ipynb`

1.25 SPI Rack driver

In this tutorial we explain basic usage of the modular SPI Rack drivers provided by the Qblox instruments package. The driver is based on the programming interface provided by SPI-rack.

In order to connect to the SPI Rack driver:

[1]:
```
from qblox_instruments import SpiRack

# In our case the SPI Rack is connected to COM port 4
spi = SpiRack('SPI Rack', 'COM4')
```

```
Connected to: Qblox SPI Rack (serial:None, firmware:{'device': 'v1.6 - May 10␣
↪2019 - mt', 'driver': {'version': '0.3.2', 'date': '21/04/2021-16:38:33',
↪'hash': '0x94E811E5', 'dirty': False}}) in 0.00s
```

To verify everything is working correctly we can read out the temperature of the C1b module

[2]:
```
spi.temperature()
```

[2]:
```
23.75
```

1.25.1 Connecting to S4g

Next, we need to add the modules to our driver. We will be using a S4g module for this tutorial.

[3]:
```
spi.add_spi_module(2, "S4g")
```

The S4g module we added can now be accessed through `spi.module2`. Let's try setting an output current:

[4]:
```
spi.module2.dac0.current(1e-3)
```

This sets the output current of dac0 (output 1) to 1 mA. We can read this current back through:

[5]:
```
spi.module2.dac0.current()
```

[5]:
```
0.0009998321533203139
```

Now we will change the output range.

[6]:
```
spi.module2.dac1.span()
```

[6]:
```
'range_max_bi'
```

We see that the span of DAC channel 1 is set to the default `'range_max_bi'` (corresponding to -40 to +40 mA), changing this works similar to how we change the current.

[7]:
```
spi.module2.dac1.span('range_min_bi')
spi.module2.dac1.span()
```

[7]:
```
'range_min_bi'
```

Now we updated the range to the smaller -20 to +20 mA range. We will now set all output values back to zero using the convenient `set_dacs_zero` function.

```
[8]: spi.module2.dac0.current()
[8]: 0.0009998321533203139

[9]: spi.set_dacs_zero()

[10]: spi.module2.dac0.current()
[10]: 0.0
```

1.25.2 Connecting to D5a

Connecting to the D5a module works the same way as the S4g. Instead of the default name module1, let's give this D5a an alias and call it `alice`

```
[11]: spi.add_spi_module(1, "D5a", "alice")

[12]: spi.alice.dac0.voltage(1.0)
      spi.alice.dac0.voltage()
[12]: 1.0

[13]: spi.set_dacs_zero()
      spi.alice.dac0.voltage()
[13]: 0.0
```

Similar to the S4g, the D5a also has a span that can be set.

Supported settings are: `'range_4V_uni'`, `'range_4V_bi'` and `'range_2V_bi'`.

```
[14]: spi.alice.dac0.span()
[14]: 'range_4V_bi'
```

1.25.3 Extending the SPI module drivers

It is possible to use custom drivers for the SPI modules within the SPI rack driver provided by `qblox-instruments`. Any class that inherits from `SpiModuleBase` is accepted by the spi rack driver. The driver can be added as a module by passing a reference to the class instead of the usual string.

As an example, we will add the `DummySpiModule` to the SPI Rack.

```
[15]: from qblox_instruments.qcodes_drivers.spi_rack_modules import DummySpiModule

      spi.add_spi_module(3, DummySpiModule, "bob")
```

1.25.4 Closing the instrument

Finally, we will close the connection to the instrument.

```
[16]: spi.close()
```

1.26 Pulsar

The Pulsar driver is separated into three layers:

- *QCoDeS driver*: Instrument driver based on QCoDeS and the instrument's native interface.
- *Native interface*: Instrument API that provides control over the instrument and is an extension of the the SCPI interface.
- *SCPI interface*: Instrument API based on the SCPI standard which in turn is based on IEEE488.2.

1.26.1 QCoDeS driver

class qblox_instruments.**Pulsar**(**args: Any*, ***kwargs: Any*)

Bases: `Pulsar`, `Instrument`

This class connects QCoDeS to the Pulsar native interface.

__init__(*name: str*, *identifier: Optional[str] = None*, *port: Optional[int] = None*, *debug: Optional[int] = None*, *dummy_type: Optional[*PulsarType*] = None*)

Creates Pulsar QCoDeS class and adds all relevant instrument parameters. These instrument parameters call the associated methods provided by the native interface.

Parameters

- **name** (*str*) – Instrument name.
- **identifier** (*Optional[str]*) – Instrument identifier. See *resolve()*. If None, the instrument is identified by name.
- **port** (*Optional[int]*) – Override for the TCP port through which we should connect.
- **debug** (*Optional[int]*) – Debug level (0 | None = normal, 1 = no version check, >1 = no version or error checking).
- **dummy_type** (*Optional[PulsarType]*) – Configure as dummy module of specified type.

property sequencers: List

Get list of sequencers submodules.

Returns

List of sequencer submodules.

Return type

list

reset() → None

Resets device, invalidates QCoDeS parameter cache and clears all status and event registers (see SCPI).

QCoDeS instrument parameters

QCoDeS parameters generated by *Pulsar*. The parameters are dscribed in *QCM-QRM*.

1.26.2 Native interface

class qblox_instruments.native.**Pulsar**(*identifier: str*, *port: Optional[int] = None*, *debug: Optional[int] = None*, *dummy_type: Optional[PulsarType] = None*)

Bases: object

Class that provides the native API for the Pulsar. It provides methods to control all functions and features provided by the Pulsar, like sequencer, waveform and acquistion handling.

This class is build upon the *PulsarQcm* and *PulsarQrm* classes through composition. This allows us to create a generic native Pulsar class that supports both Pulsar types, but only exposes the relevant interfaces, while also keeping the SCPI classes automatically generated for fast prototyping and development. On instantiation, this class probes the connected device to see which Pulsar SCPI interface to support and adds all respective attributes to this class. Afterwards, this class can be used as if it inherits from it's respective Pulsar SCPI interface with all features and functionality available.

Note that the bulk of the functionality of this class is contained in the *generic_func* module so that this functionality can be shared between instruments.

__init__(*identifier: str*, *port: Optional[int] = None*, *debug: Optional[int] = None*, *dummy_type: Optional[PulsarType] = None*)

Creates Pulsar native interface object.

Parameters

- **identifier** (*str*) – Instrument identifier. See *resolve()* for more information.
- **port** (*Optional[int]*) – Instrument port. If None, this will be determined automatically.
- **debug** (*Optional[int]*) – Debug level (0 | None = normal, 1 = no version check, >1 = no version or error checking).
- **dummy_type** (*Optional[PulsarType]*) – Configure as dummy module of specified type.

Raises

- **RuntimeError** – Instrument cannot be reached due to invalid IP configuration.
- **ConnectionError** – Instrument type is not supported.

property instrument_class: *InstrumentClass*

Get instrument class (e.g. Pulsar, Cluster).

Returns

Instrument class

Return type

InstrumentClass

property instrument_type: *InstrumentType*

Get instrument type (e.g. QRM, QCM).

Returns

Instrument type

Return type

InstrumentType

property is_qcm_type: bool

Return if module is of type QCM.

Returns
True if module is of type QCM.
Return type
bool

property is_qrm_type: bool

Return if module is of type QRM.
Returns
True if module is of type QRM.
Return type
bool

property is_rf_type: bool

Return if module is of type QCM-RF or QRM-RF.
Returns
True if module is of type QCM-RF or QRM-RF.
Return type
bool

get_idn() → Dict

Get device identity and build information and convert them to a dictionary.
Returns
Dictionary containing manufacturer, model, serial number and build information. The build information is subdivided into FPGA firmware, kernel module software, application software and driver software build information. Each of those consist of the version, build date, build Git hash and Git build dirty indication.
Return type
dict

get_system_state() → *SystemState*

Get general system state and convert it to a `SystemState`.
Returns
Tuple containing general system status and corresponding flags.
Return type
SystemStatus

arm_sequencer(*sequencer: Optional[int] = None*) → None

Prepare the indexed sequencer to start by putting it in the armed state. If no sequencer index is given, all sequencers are armed. Any sequencer that was already running is stopped and rearmed. If an invalid sequencer index is given, an error is set in system error.
Parameters
sequencer (*Optional[int]*) – Sequencer index.
Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

start_sequencer(*sequencer: Optional[int] = None*) → None

Start the indexed sequencer, thereby putting it in the running state. If an invalid sequencer index is given or the indexed sequencer was not yet armed, an error is set in system error. If no sequencer index is given, all armed sequencers are started and any sequencer not in the armed state is ignored. However, if no sequencer index is given and no sequencers are armed, and error is set in system error.
Parameters
sequencer (*Optional[int]*) – Sequencer index.

Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`stop_sequencer`(*sequencer: Optional[int] = None*) → None

Stop the indexed sequencer, thereby putting it in the stopped state. If an invalid sequencer index is given, an error is set in system error. If no sequencer index is given, all sequencers are stopped.

Parameters
sequencer (*Optional[int]*) – Sequencer index.

Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_sequencer_state`(*sequencer: int*, *timeout: int = 0*, *timeout_poll_res: float = 0.02*) → *SequencerState*

Get the sequencer state. If an invalid sequencer index is given, an error is set in system error. If the timeout is set to zero, the function returns the state immediately. If a positive non-zero timeout is set, the function blocks until the sequencer completes. If the sequencer hasn't stopped before the timeout expires, a TimeoutError is thrown.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.

Returns
Tuple containing sequencer status and corresponding flags.

Return type
SequencerState

Raises
TimeoutError – Timeout

`get_waveforms`(*sequencer: int*) → Dict

Get all waveforms and weigths in the AWG waveform list of indexed sequencer. The returned dictionary is structured as follows:

- name: waveform name.
  - data: waveform samples in a range of 1.0 to -1.0.
  - **index: waveform index used by the sequencer Q1ASM program to refer** to the waveform.

Parameters
sequencer (*int*) – Sequencer index.

Returns
Dictionary with waveforms.

Return type
dict

`get_weights`(*sequencer: int*) → Dict

Get all weigths in the acquisition weight lists of indexed sequencer. The returned dictionary is structured as follows:

-name : weight name.
- data: weight samples in a range of 1.0 to -1.0.
- **index: weight index used by the sequencer Q1ASM program to refer** to the weight.

Parameters
sequencer (*int*) – Sequencer index.

Returns
Dictionary with weights.
Return type
dict
Raises
NotImplementedError – Functionality not available on this module.

get_acquisition_state(*sequencer: int*, *timeout: int = 0*, *timeout_poll_res: float = 0.02*) → bool

Return acquisition binning completion state of the indexed sequencer. If an invalid sequencer is given, an error is set in system error. If the timeout is set to zero, the function returns the state immediately. If a positive non-zero timeout is set, the function blocks until the acquisition binning completes. If the acquisition hasn't completed before the timeout expires, a TimeoutError is thrown. Note that when sequencer state checking is enabled, the sequencer state is checked using get_sequencer_state with the selected timeout period first and then the acquisition state is checked with the same timeout period. This means that the total timeout period is two times the set timeout period.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.
- **check_seq_state** (*bool*) – Check if sequencer is done before checking acquisition state.

Returns
Indicates the acquisition binning completion state (False = uncompleted, True = completed).
Return type
bool
Raises
- **TimeoutError** – Timeout
- **NotImplementedError** – Functionality not available on this module.

delete_acquisition_data(*sequencer: int*, *name: str = ''*, *all: bool = False*) → None

Delete data from an acquisition specified by name in the acquisition list of indexed sequencer or delete data in all acquisitions if *all* is True.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Weight name

Raises
NotImplementedError – Functionality not available on this module.

store_scope_acquisition(*sequencer: int*, *name: str*) → None

After an acquisition has completed, store the scope acquisition results in the acquisition specified by name of the indexed sequencers. If an invalid sequencer index is given an error is set in system error. To get access to the acquisition results, the sequencer will be stopped when calling this function.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Acquisition name.

Raises
NotImplementedError – Functionality not available on this module.

get_acquisitions(*sequencer: int*) → Dict

Get all acquisitions in acquisition lists of indexed sequencer. The acquisition scope and bin data is normalized to a range of -1.0 to 1.0 taking both the bit widths of the processing path and

average count into considaration. For the binned integration results, the integration length is not handled during normalization and therefore these values have to be divided by their respective integration lenghts. The returned dictionary is structured as follows:

- name: acquisition name
  - **index: acquisition index used by the sequencer Q1ASM program to** refer to the acquisition.
  - acquisition: acquisition dictionary
    - scope: Scope data
      - path0: input path 0
      - data: acquisition samples in a range of 1.0 to -1.0.
      - **out-of-range: out-of-range indication for the entire** acquisition (False = in-range, True = out-of-range).
      - avg_cnt: number of averages.
      - path1: input path 1
      - data: acquisition samples in a range of 1.0 to -1.0.
      - **out-of-range: out-of-range indication for the entire** acquisition (False = in-range, True = out-of-range).
      - avg_cnt: number of averages.
    - bins: bin data
      - integration: integration data
      - path_0: input path 0 integration result bin list
      - path_1: input path 1 integration result bin list
      - threshold: threshold result bin list
      - valid: list of valid indications per bin
      - avg_cnt: list of number of averages per bin

Parameters
sequencer (*int*) – Sequencer index.
Returns
Dictionary with acquisitions.
Return type
dict
Raises
NotImplementedError – Functionality not available on this module.

set_dummy_binned_acquisition_data(*sequencer: int*, *acq_index_name: str*, *data: Iterable[Optional[DummyBinnedAcquisitionData]]*)

Set dummy binned acquisition data for the dummy.

Parameters

- **sequencer** (*int*) – Sequencer.
- **acq_index_name** (*str*) – Acquisition index name.
- **data** (*Iterable[Union[DummyBinnedAcquisitionData, None]]*) – Dummy data for the binned acquisition. An iterable of all the bin values.
- **slot_idx** (*Union[int, None]*) – Slot of the hardware you want to set the data to on a cluster.

set_dummy_scope_acquisition_data(*sequencer: Optional[int]*, *data: DummyScopeAcquisitionData*)

Set dummy scope acquisition data for the dummy.

Parameters

- **sequencer** (*int*) – Sequencer.
- **data** (DummyScopeAcquisitionData) – Dummy data for the scope acquisition.

1.26.3 SCPI interface

Based on the used instrument, one of the two following interfaces is selected.

QCM interface

class qblox_instruments.scpi.**PulsarQcm**(*transport:* Transport, *debug:* *int* = *0*)

Bases: *Ieee488_2*

This interface provides an API for the mandatory and required SCPI calls and adds Pulsar related functionality (see SCPI).

__init__(*transport:* Transport, *debug:* *int* = *0*)

Creates SCPI interface object.

Parameters

- **transport** (Transport) – Transport class responsible for the lowest level of communication (e.g. Ethernet).
- **debug** (*int*) – Debug level (0 = normal, 1 = no version check, >1 = no version or error checking).

Raises

ConnectionError – Debug level is 0 and there is a device or version mismatch.

get_system_error() → str

Get system error from queue (see SCPI).

Parameters

None –

Returns

System error description string.

Return type

str

get_num_system_error() → int

Get number of system errors (see SCPI).

Parameters

None –

Returns

Current number of system errors.

Return type

int

get_system_version() → str

Get SCPI system version (see SCPI).

Parameters

None –

Returns

SCPI system version.

Return type
str
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_name(*name: str*) → None

Sets the customer-specified name of the instrument. The name must not contain any newlines, backslashes, or double quotes.
Parameters
name (*str*) – The new name for the device.
Return type
None
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_name() → str

Returns the customer-specified name of the instrument.
Parameters
None –
Returns
The name of the device.
Return type
str
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_ip_config(*config: str*) → None

Reconfigures the IP address of this device. The configuration will not go into effect until reboot() is called or the device is power-cycled.
Parameters
config (*str*) –

IP configuration. May be one of the following things:

- an IPv4 address including prefix length, for example 192.168.0.2/24, - the string *dhcp* to enable IPv4 DHCP, - an IPv6 address including prefix length, for example 1:2::3:4/64, or - a semicolon-separated combination of an IPv4 configuration (IP address or *dhcp*) and an IPv6 address.

Return type
None
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_ip_config() → str

Returns the IP address configuration that will go into effect when the device reboots.

Parameters
None –
Returns

IP configuration. Can be one of the following things:

- an IPv4 address including prefix length, for example 192.168.0.2/24, - the string *dhcp* to enable IPv4 DHCP, - an IPv6 address including prefix length, for example 1:2::3:4/64, or - a semicolon-separated combination of an IPv4 configuration (IP address or *dhcp*) and an IPv6 address.

Return type
str
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

reboot() → None

Reboots the instrument.
Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

clear() → None

Clear all status and event registers (see SCPI).
Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_status_byte() → int

Get status byte register. Register is only cleared when feeding registers are cleared (see SCPI).
Parameters
None –
Returns
Status byte register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_service_request_enable(*reg: int*) → None

Set service request enable register (see SCPI).
Parameters
reg (*int*) – Service request enable register.
Return type
None
Raises

- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_service_request_enable() → int

Get service request enable register. The register is cleared after reading it (see SCPI).

Parameters
None –

Returns
Service request enable register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_standard_event_status_enable(*reg: int*) → None

Set standard event status enable register (see SCPI).

Parameters
reg (*int*) – Standard event status enable register.

Return type
None

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_standard_event_status_enable() → int

Get standard event status enable register. The register is cleared after reading it (see SCPI).

Parameters
None –

Returns
Standard event status enable register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_standard_event_status() → int

Get standard event status register. The register is cleared after reading it (see SCPI).

Parameters
None –

Returns
Standard event status register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_operation_complete() → None

Set device in operation complete query active state (see SCPI).

Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_complete() → bool
Get operation complete state (see SCPI).
Parameters
None –
Returns
Operation complete state (False = running, True = completed).
Return type
bool
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

test() → bool
Run self-test. Currently not implemented (see SCPI).
Parameters
None –
Returns
Test result (False = failed, True = success).
Return type
bool
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

wait() → None
Wait until operations completed before continuing (see SCPI).
Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

preset_system_status() → None
Preset system status registers. Connects general system status flags for PLL unlock and temperature out-of-range indications to event status enable, status questionable temperature and status questionable frequency registers respectively (see SCPI).
Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_questionable_condition() → int
Get status questionable condition register (see SCPI).

Parameters
None –
Returns
Status questionable condition register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_questionable_event()` → int

Get status questionable event register (see SCPI).
Parameters
None –
Returns
Status questionable event register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`set_questionable_enable(reg: int)` → None

Set status questionable enable register (see SCPI).
Parameters
reg (*int*) – Status questionable enable register.
Return type
None
Raises
- Exception – Invalid input parameter type.
- Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_questionable_enable()` → int

Get status questionable enable register (see SCPI).
Parameters
None –
Returns
Status questionable enable register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_operation_condition()` → int

Get status operation condition register (see SCPI).
Parameters
None –
Returns
Status operation condition register.
Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_events() → int

Get status operation event register (see SCPI).

Parameters
None –
Returns
Status operation event register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_operation_enable(*reg: int*) → None

Set status operation enable register (see SCPI).

Parameters
reg (*int*) – Status operation enable register.
Return type
None
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_enable() → int

Get status operation enable register (see SCPI).

Parameters
None –
Returns
Status operation enable register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

identify() → None

Toggle frontpanel LEDs to visually identify the instrument.

Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_afe_temperature() → float

Get current analog frontend board temperature (inside device).

Parameters
None –
Returns
Current analog frontend board temperature.

Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_afe_temperature()` → float

Get maximum analog frontend board temperature since boot or clear (inside device).
Parameters
None –
Returns
Maximum analog frontend board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_current_lo_temperature()` → float

Get current local oscillator board temperature (inside device).
Parameters
None –
Returns
Current local oscillator board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_lo_temperature()` → float

Get maximum local oscillator board temperature since boot or clear (inside device).
Parameters
None –
Returns
Maximum local oscillator board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_current_fpga_temperature()` → float

Get current FPGA junction temperature (inside device).
Parameters
None –
Returns
Current FPGA junction temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_fpga_temperature()` → float

Get maximum FPGA junction temperature since boot or clear (inside device).

Parameters
None –
Returns
Maximum FPGA junction temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_current_carrier_temperature()` → float

Get current carrier board temperature (inside device).
Parameters
None –
Returns
Current carrier board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_carrier_temperature()` → float

Get maximum carrier board temperature since boot or clear (inside device).
Parameters
None –
Returns
Maximum carrier board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_assembler_status()` → bool

Get assembler status. Refer to the assembler log to get more information regarding the assembler result.
Parameters
None –
Returns
Assembler status (False = failed, True = success).
Return type
bool
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_assembler_log()` → str

Get assembler log.
Parameters
None –
Returns
Assembler log.
Return type
str

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

QRM interface

class qblox_instruments.scpi.**PulsarQrm**(*transport:* Transport, *debug:* *int* = *0*)

Bases: *Ieee488_2*

This interface provides an API for the mandatory and required SCPI calls and adds Pulsar related functionality (see SCPI).

__init__(*transport:* Transport, *debug:* *int* = *0*)

Creates SCPI interface object.
Parameters
- **transport** (Transport) – Transport class responsible for the lowest level of communication (e.g. Ethernet).
- **debug** (*int*) – Debug level (0 = normal, 1 = no version check, >1 = no version or error checking).

Raises
ConnectionError – Debug level is 0 and there is a device or version mismatch.

get_system_error() → str

Get system error from queue (see SCPI).
Parameters
None –
Returns
System error description string.
Return type
str

get_num_system_error() → int

Get number of system errors (see SCPI).
Parameters
None –
Returns
Current number of system errors.
Return type
int

get_system_version() → str

Get SCPI system version (see SCPI).
Parameters
None –
Returns
SCPI system version.
Return type
str
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_name(*name: str*) → None

Sets the customer-specified name of the instrument. The name must not contain any newlines, backslashes, or double quotes.

Parameters

name (*str*) – The new name for the device.

Return type

None

Raises

- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_name() → str

Returns the customer-specified name of the instrument.

Parameters

None –

Returns

The name of the device.

Return type

str

Raises

- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_ip_config(*config: str*) → None

Reconfigures the IP address of this device. The configuration will not go into effect until reboot() is called or the device is power-cycled.

Parameters

config (*str*) –

IP configuration. May be one of the following things:

- an IPv4 address including prefix length, for example 192.168.0.2/24, - the string *dhcp* to enable IPv4 DHCP, - an IPv6 address including prefix length, for example 1:2::3:4/64, or - a semicolon-separated combination of an IPv4 configuration (IP address or *dhcp*) and an IPv6 address.

Return type

None

Raises

- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_ip_config() → str

Returns the IP address configuration that will go into effect when the device reboots.

Parameters

None –

Returns

IP configuration. Can be one of the following things:

- an IPv4 address including prefix length, for example 192.168.0.2/24, - the string *dhcp* to enable IPv4 DHCP, - an IPv6 address including prefix length, for example 1:2::3:4/64, or - a semicolon-separated combination of an IPv4 configuration (IP address or *dhcp*) and an IPv6 address.

Return type
str

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

reboot() → None

Reboots the instrument.

Parameters
None –

Return type
None

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

clear() → None

Clear all status and event registers (see SCPI).

Parameters
None –

Return type
None

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_status_byte() → int

Get status byte register. Register is only cleared when feeding registers are cleared (see SCPI).

Parameters
None –

Returns
Status byte register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_service_request_enable(*reg: int*) → None

Set service request enable register (see SCPI).

Parameters
reg (*int*) – Service request enable register.

Return type
None

Raises

- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_service_request_enable() → int

Get service request enable register. The register is cleared after reading it (see SCPI).

Parameters
None –

Returns
Service request enable register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_standard_event_status_enable(*reg: int*) → None

Set standard event status enable register (see SCPI).

Parameters
reg (*int*) – Standard event status enable register.

Return type
None

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_standard_event_status_enable() → int

Get standard event status enable register. The register is cleared after reading it (see SCPI).

Parameters
None –

Returns
Standard event status enable register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_standard_event_status() → int

Get standard event status register. The register is cleared after reading it (see SCPI).

Parameters
None –

Returns
Standard event status register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_operation_complete() → None

Set device in operation complete query active state (see SCPI).

Parameters
None –

Return type
None

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_complete() → bool

Get operation complete state (see SCPI).

Parameters
None –
Returns
Operation complete state (False = running, True = completed).
Return type
bool
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

test() → bool

Run self-test. Currently not implemented (see SCPI).

Parameters
None –
Returns
Test result (False = failed, True = success).
Return type
bool
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

wait() → None

Wait until operations completed before continuing (see SCPI).

Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

preset_system_status() → None

Preset system status registers. Connects general system status flags for PLL unlock and temperature out-of-range indications to event status enable, status questionable temperature and status questionable frequency registers respectively (see SCPI).

Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_questionable_condition() → int

Get status questionable condition register (see SCPI).

Parameters
None –
Returns
Status questionable condition register.

Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_questionable_event() → int
Get status questionable event register (see SCPI).
Parameters
None –
Returns
Status questionable event register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_questionable_enable(*reg: int*) → None
Set status questionable enable register (see SCPI).
Parameters
reg (*int*) – Status questionable enable register.
Return type
None
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_questionable_enable() → int
Get status questionable enable register (see SCPI).
Parameters
None –
Returns
Status questionable enable register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_condition() → int
Get status operation condition register (see SCPI).
Parameters
None –
Returns
Status operation condition register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_events() → int
Get status operation event register (see SCPI).

Parameters
None –
Returns
Status operation event register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_operation_enable(*reg: int*) → None

Set status operation enable register (see SCPI).
Parameters
reg (*int*) – Status operation enable register.
Return type
None
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_enable() → int

Get status operation enable register (see SCPI).
Parameters
None –
Returns
Status operation enable register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

identify() → None

Toggle frontpanel LEDs to visually identify the instrument.
Parameters
None –
Return type
None
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_afe_temperature() → float

Get current analog frontend board temperature (inside device).
Parameters
None –
Returns
Current analog frontend board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_afe_temperature()` → float

Get maximum analog frontend board temperature since boot or clear (inside device).

Parameters
`None` –

Returns
Maximum analog frontend board temperature.

Return type
float

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_current_lo_temperature()` → float

Get current local oscillator board temperature (inside device).

Parameters
`None` –

Returns
Current local oscillator board temperature.

Return type
float

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_lo_temperature()` → float

Get maximum local oscillator board temperature since boot or clear (inside device).

Parameters
`None` –

Returns
Maximum local oscillator board temperature.

Return type
float

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_current_fpga_temperature()` → float

Get current FPGA junction temperature (inside device).

Parameters
`None` –

Returns
Current FPGA junction temperature.

Return type
float

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_fpga_temperature()` → float

Get maximum FPGA junction temperature since boot or clear (inside device).

Parameters
`None` –

Returns
Maximum FPGA junction temperature.

Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_current_carrier_temperature()` → float

Get current carrier board temperature (inside device).
Parameters
None –
Returns
Current carrier board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_carrier_temperature()` → float

Get maximum carrier board temperature since boot or clear (inside device).
Parameters
None –
Returns
Maximum carrier board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_assembler_status()` → bool

Get assembler status. Refer to the assembler log to get more information regarding the assembler result.
Parameters
None –
Returns
Assembler status (False = failed, True = success).
Return type
bool
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_assembler_log()` → str

Get assembler log.
Parameters
None –
Returns
Assembler log.
Return type
str
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

start_adc_calib() → None

Calibrates ADC delay and offset values. This method sets the correct delay values for every input data index (IO data lane) in order to avoid timing violations which occur while sampling ADC data. It also calibrates offsets internal to the ADC.

Parameters

None –

Return type

None

Raises

- Exception – Invalid input parameter type.
- Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

1.26.4 Supporting classes and functions

Please have a look at:

- *QCM-QRM*
- *Sequencer*
- *Generic native interface functions*
- *IEEE488.2*
- *Instrument types*

Instrument types

Instrument type enums and classes.

class qblox_instruments.types.**TypeEnum**(*value*)

Bases: Enum

Type base class that arranges child enum string representations.

class qblox_instruments.**InstrumentClass**(*value*)

Bases: *TypeEnum*

Instrument class enum.

PULSAR = 'Pulsar'

CLUSTER = 'Cluster'

class qblox_instruments.**InstrumentType**(*value*)

Bases: *TypeEnum*

Instrument/module type enum.

MM = 'MM'

QCM = 'QCM'

QRM = 'QRM'

class qblox_instruments.**PulsarType**(*value*)

Bases: *TypeEnum*

Pulsar module type enum.

PULSAR_QCM = 'Pulsar QCM'

PULSAR_QRM = 'Pulsar QRM'

class qblox_instruments.**ClusterType**(*value*)

Bases: *TypeEnum*

Cluster module type enum.

CLUSTER_QCM = 'Cluster QCM'

CLUSTER_QCM_RF = 'Cluster QCM-RF'

CLUSTER_QRM = 'Cluster QRM'

CLUSTER_QRM_RF = 'Cluster QRM-RF'

class qblox_instruments.**TypeHandle**(*instrument: Union[*PulsarType, ClusterType, *str]*)

Bases: object

Instrument type handler class.

__init__(*instrument: Union[*PulsarType, ClusterType, *str]*)

Create instrument type handler object.

Parameters

instrument (*Union[*PulsarType, ClusterType, *str]*) – Instrument/module type specification.

property instrument_class: *InstrumentClass*

Get instrument class (e.g. Pulsar, Cluster).

Returns

Instrument class

Return type

InstrumentClass

property instrument_type: *InstrumentType*

Get instrument type (e.g. MM, QRM, QCM).

Returns

Instrument type

Return type

InstrumentType

property is_mm_type: bool

Return if module is of type MM.

Returns

True if module is of type MM.

Return type

bool

property is_qcm_type: bool

Return if module is of type QCM.

Returns

True if module is of type QCM.

Return type
bool

property is_qrm_type: bool

Return if module is of type QRM.

Returns
True if module is of type QRM.

Return type
bool

property is_rf_type: bool

Return if module is of type QCM-RF or QRM-RF.

Returns
True if module is of type QCM-RF or QRM-RF.

Return type
bool

IEEE488.2

Every SCPI interface is based on the IEEE488.2 protocol. This Python implementation separates the protocol into two layers:

- *IEEE488.2 layer*: IEEE488.2 layer that implements the protocol based on the transport layer.
- *Transport layer*: Transport layers responsible for lowest level of communcation (e.g. Ethernet or dummy)."

IEEE488.2 layer

class qblox_instruments.ieee488_2.**Ieee488_2**(*transport:* Transport)

Bases: object

Class that implements the IEEE488.2 interface.

__init__(*transport:* Transport)

Creates IEEE488.2 interface object.

Parameters
transport (Transport) – Transport class responsible for the lowest level of communication (e.g. ethernet).

Transport layer

class qblox_instruments.ieee488_2.**IpTransport**(*host: str*, *port: int = 5025*, *timeout: float = 60.0*, *snd_buf_size: int = 524288*)

Bases: *Transport*

Class for data transport of IP socket.

__init__(*host: str*, *port: int = 5025*, *timeout: float = 60.0*, *snd_buf_size: int = 524288*)

Create IP socket transport class.

Parameters

- **host** (*str*) – Instrument IP address.
- **port** (*int*) – Instrument port.

- **timeout** (*float*) – Instrument call timeout in seconds.
- **snd_buf_size** (*int*) – Instrument buffer size for transmissions to instrument.

close() → None

Close IP socket.

write(*cmd_str: str*) → None

Write command to instrument over IP socket.

Parameters

cmd_str (*str*) – Command

write_binary(*data: bytes*) → None

Write binary data to instrument over IP socket.

Parameters

data (*bytes*) – Binary data

read_binary(*size: int*) → bytes

Read binary data from instrument over IP socket.

Parameters

size (*int*) – Number of bytes

Returns

Binary data array of length "size".

Return type

bytes

readline() → str

Read data from instrument over IP socket.

Returns

String with data.

Return type

str

class qblox_instruments.ieee488_2.**PulsarDummyTransport**(*dummy_type: Union[PulsarType, ClusterType], scope_acq_cfg_format: str, qcm_seq_cfg_format: str, qrm_seq_cfg_format: str*)

Bases: *QcmQrmDummyTransport*

Class to replace Pulsar device with dummy device to support software stack testing without hardware. The class implements all mandatory, required and Pulsar specific SCPI calls. Call reponses are largely artifically constructed to be inline with the call's functionality (e.g. **IDN?* returns valid, but artificial IDN data.) To assist development, the Q1ASM assembler has been completely implemented. Please have a look at the call's implentation to know what to expect from its response.

class qblox_instruments.ieee488_2.**ClusterDummyTransport**(*dummy_cfg: Dict[Union[str, int], ClusterType], scope_acq_cfg_format: str, qcm_seq_cfg_format: str, qrm_seq_cfg_format: str*)

Bases: *DummyTransport*

Class to replace Cluster device with dummy device to support software stack testing without hardware. The class implements all mandatory, required and Cluster specific SCPI calls. Call reponses

are largely artifically constructed to be inline with the call's functionality (e.g. **IDN?* returns valid, but artificial IDN data.) To assist development, the Q1ASM assembler has been completely implemented. Please have a look at the call's implentation to know what to expect from its response.

__init__(*dummy_cfg: Dict[Union[str, int], ClusterType]*, *scope_acq_cfg_format: str*, *qcm_seq_cfg_format: str*, *qrm_seq_cfg_format: str*)

Create Cluster dummy transport class.

Parameters

- **dummy_cfg** (*Dict*) – Dictionary of dummy module types (e.g. Cluster QCM, Cluster QRM). Each key of the dictionary is a slot index with a dummy type specification of type *ClusterType*.
- **scope_acq_cfg_format** (*str*) – Configuration format based on struct.pack format used to calculate scope acquisition configuration transaction size.
- **qcm_seq_cfg_format** (*str*) – QCM sequencer configuration format based on struct.pack format used to calculate scope acquisition configuration transaction size.
- **qrm_seq_cfg_format** (*str*) – QRM sequencer configuration format based on struct.pack format used to calculate scope acquisition configuration transaction size.

set_dummy_binned_acquisition_data(*slot_idx: int*, *sequencer: int*, *acq_index_name: str*, *data: Iterable[Optional[DummyBinnedAcquisitionData]]*)

Set dummy binned acquisition data for the dummy.

Parameters

- **slot_idx** (*int*) – Slot of the hardware you want to set the data to on a cluster.
- **sequencer** (*int*) – Sequencer.
- **acq_index_name** (*str*) – Acquisition index name.
- **data** (*Iterable[Union[DummyBinnedAcquisitionData, None]]*) – Dummy data for the binned acquisition. An iterable of all the bin values.

set_dummy_scope_acquisition_data(*slot_idx: int*, *data: DummyScopeAcquisitionData*)

Set dummy scope acquisition data for the dummy.

Parameters

- **slot_idx** (*int*) – Slot of the hardware you want to set the data to on a cluster.
- **data** (*DummyScopeAcquisitionData*) – Dummy data for the scope acquisition.

Supporting classes and functions

class qblox_instruments.ieee488_2.**QcmQrmDummyTransport**(*dummy_type: Union[PulsarType, ClusterType]*, *scope_acq_cfg_format: str*, *qcm_seq_cfg_format: str*, *qrm_seq_cfg_format: str*)

Bases: *DummyTransport*

Class to replace a QCM/QRM module with a dummy device to support software stack testing without hardware. The class implements all mandatory, required and QCM/QRM specific SCPI calls. Call reponses are largely artifically constructed to be inline with the call's functionality (e.g. **IDN?* returns valid, but artificial IDN data.) To assist development, the Q1ASM assembler has been completely implemented. Please have a look at the call's implentation to know what to expect from its response.

__init__(*dummy_type: Union[PulsarType, ClusterType]*, *scope_acq_cfg_format: str*, *qcm_seq_cfg_format: str*, *qrm_seq_cfg_format: str*)

Create QCM/QRM dummy transport class.

Parameters

- **dummy_type** (*Union[PulsarType, ClusterType]*) – Dummy module type (e.g. Pulsar QCM, Pulsar QRM)
- **scope_acq_cfg_format** (*str*) – Configuration format based on struct.pack format used to calculate scope acquisition configuration transaction size.
- **qcm_seq_cfg_format** (*str*) – QCM sequencer configuration format based on struct.pack format used to calculate scope acquisition configuration transaction size.
- **qrm_seq_cfg_format** (*str*) – QRM sequencer configuration format based on struct.pack format used to calculate scope acquisition configuration transaction size.

property is_qcm_type: bool

Return if module is of type QCM.

Returns

True if module is of type QCM.

Return type

bool

property is_qrm_type: bool

Return if module is of type QRM.

Returns

True if module is of type QRM.

Return type

bool

property is_rf_type: bool

Return if module is of type QCM-RF or QRM-RF.

Returns

True if module is of type QCM-RF or QRM-RF.

Return type

bool

set_dummy_binned_acquisition_data(*sequencer: int*, *acq_index_name: str*, *data: Iterable[Optional[DummyBinnedAcquisitionData]]*)

Set dummy binned acquisition data for the dummy.

Parameters

- **sequencer** (*int*) – Sequencer.
- **acq_index_name** (*str*) – Acquisition index name.
- **data** (*Iterable[Union[DummyBinnedAcquisitionData, None]]*) – Dummy data for the binned acquisition. An iterable of all the bin values.

set_dummy_scope_acquisition_data(*data: DummyScopeAcquisitionData*)

Set dummy scope acquisition data for the dummy.

Parameters

data (DummyScopeAcquisitionData) – Dummy data for the scope acquisition.

class qblox_instruments.ieee488_2.**DummyTransport**(*dummy_type: Union[PulsarType, ClusterType]*)

Bases: ABC, *Transport*

Class to replace device with dummy device to support software stack testing without hardware. The class implements all mandatory and required SCPI calls. Call reponses are largely artifically constructed to be inline with the call's functionality (e.g. **IDN?* returns valid, but artificial IDN data.)

__init__(*dummy_type: Union[PulsarType, ClusterType]*)

Create dummy transport class.

Parameters

dummy_type (*Union[PulsarType, ClusterType]*) – Dummy instrument type (e.g. Pulsar QCM, Pulsar QRM)

property instrument_class: str

Get instrument class (e.g. Pulsar, Cluster).

Returns

Instrument class

Return type

str

property instrument_type: str

Get instrument type (e.g. MM, QRM, QCM).

Returns

Instrument type

Return type

str

close() → None

Close and resets base dummy transport class.

write(*cmd_str: str*) → None

Write command to dummy. Stores command in command history.

Parameters

cmd_str (*str*) – Command

write_binary(*data: bytes*) → None

Write binary data to dummy. Stores command in command history.

Parameters

data (*bytes*) – Binary data

read_binary(*size: int*) → bytes

Read binary data from dummy.

Parameters

size (*int*) – Number of bytes

Returns

Binary data array of length "size".

Return type

bytes

readline() → str

Read data from dummy.

Returns

String with data.

Return type

str

abstract set_dummy_binned_acquisition_data(*sequencer: int*, *acq_index_name: str*, *data: Iterable[Optional[DummyBinnedAcquisitionData]]*)

Set dummy binned acquisition data for the dummy.

Parameters

- **sequencer** (*int*) – Sequencer.
- **acq_index_name** (*str*) – Acquisition index name.
- **data** (*Iterable[Union[DummyBinnedAcquisitionData, None]]*) – Dummy data for the binned acquisition. An iterable of all the bin values.

Raises

ValueError – If the slot_idx doesn't make sense for the transport.

abstract set_dummy_scope_acquisition_data(*data: DummyScopeAcquisitionData*)

Set dummy scope acquisition data for the dummy.

Parameters

data (DummyScopeAcquisitionData) – Dummy data for the scope acquisition.

Raises

ValueError – If the slot_idx doesn't make sense for the transport.

get_cmd_hist() → list

Get list of every executed command since the initialization or reset of the class.

Returns

List of executed command strings including arguments (does not include binary data argument).

Return type

list

class qblox_instruments.ieee488_2.**Transport**

Bases: object

Abstract base class for data transport to instruments.

abstract close() → None

Abstract method to close instrument.

abstract write(*cmd_str: str*) → None

Abstract method to write command to instrument.

Parameters

cmd_str (*str*) – Command

abstract write_binary(*data: bytes*) → None

Abstract method to write binary data to instrument.

Parameters

data (*bytes*) – Binary data

abstract read_binary(*size: int*) → bytes

Abstract method to read binary data from instrument.

Parameters

size (*int*) – Number of bytes

Returns

Binary data array of length "size".

Return type

bytes

abstract readline() → str

Abstract method to read data from instrument.

Returns

String with data.

Return type

str

class qblox_instruments.ieee488_2.**DummyBinnedAcquisitionData**(*data: Tuple[float, float]*, *thres: int*, *avg_cnt: int*)

Bases: object

Class to hold data for the dummy hardware for the binned acquisition. This class contains all values for one bin.

data: Tuple[float, float]

thres: int

avg_cnt: int

__init__(*data: Tuple[float, float]*, *thres: int*, *avg_cnt: int*) → None

class qblox_instruments.ieee488_2.**DummyScopeAcquisitionData**(*data: Iterable[Tuple[float, float]]*, *out_of_range: Tuple[bool, bool]*, *avg_cnt: Tuple[int, int]*)

Bases: object

Class to hold data for the dummy hardware for the scope acquisition. This class contains all values for the scope acquisition on one module.

data: Iterable[Tuple[float, float]]

out_of_range: Tuple[bool, bool]

avg_cnt: Tuple[int, int]

__init__(*data: Iterable[Tuple[float, float]]*, *out_of_range: Tuple[bool, bool]*, *avg_cnt: Tuple[int, int]*) → None

Generic native interface functions

The native interface *generic_func* module contains the bulk of the functional code for the native interfaces of the Pulsar and Cluster so that this code can be reused between instruments. The functions can be parametrized using the *FuncRefs* class that contains references to functions of the native interfaces parent class.

class qblox_instruments.native.generic_func.**StateEnum**(*value*)

Bases: Enum

State enum base class that arranges child enum string representations.

class qblox_instruments.native.generic_func.**StateTuple**

Bases: object

State tuple base class that arranges child tuple string representations.

class qblox_instruments.native.generic_func.**SystemStatus**(*value*)

Bases: *StateEnum*

System status enum.

BOOTING = 'System is booting.'

OKAY = 'System is okay.'

CRITICAL = 'An error indicated by the flags occured, but has been resolved.'

ERROR = 'An error indicated by the flags is occuring.'

class qblox_instruments.native.generic_func.**SystemStatusFlags**(*value*)

Bases: *StateEnum*

System status flags enum.

CARRIER_PLL_UNLOCKED = 'Carrier board PLL is unlocked.'

FPGA_PLL_UNLOCKED = 'FPGA PLL is unlocked.'

LO_PLL_UNLOCKED = 'Local oscillator PLL is unlocked.'

FPGA_TEMPERATURE_OUT_OF_RANGE = 'FPGA temperature is out-of-range.'

CARRIER_TEMPERATURE_OUT_OF_RANGE = 'Carrier board temperature is out-of-range.'

AFE_TEMPERATURE_OUT_OF_RANGE = 'Analog-frontend board temperature is out-of-range.'

LO_TEMPERATURE_OUT_OF_RANGE = 'Local oscillator board temperature is out-of-range.'

BACKPLANE_TEMPERATURE_OUT_OF_RANGE = 'Backplane board temperature is out-of-range.'

MODULE_NOT_CONNECTED = 'Module is not connected.'

MODULE_FIRMWARE_INCOMPATIBLE = 'Module firmware is incompatible with the rest of the system.'

class qblox_instruments.native.generic_func.**SystemStatusSlotFlags**(*slot_flags: Dict = {}*)

Bases: *SystemStatusSlotFlags*

Tuple containing lists of Cluster slot status flag enums of type `SystemStatusFlags`. Each Cluster slot has its own status flag list attribute named *slot<X>*.

class qblox_instruments.native.generic_func.**SystemState**(*status*, *flags*, *slot_flags*)

Bases: *SystemState*, *StateTuple*

System status tuple returned by `get_system_state()`. The tuple contains a system status enum of type `SystemStatus`, a list of associated system status flag enums of type `SystemStatusFlags` and a tuple of type `SystemStatusSlotFlags` containing Cluster slot status flags.

class qblox_instruments.native.generic_func.**SequencerStatus**(*value*)

Bases: *StateEnum*

Sequencer status enum.

IDLE = 'Sequencer waiting to be armed and started.'

ARMED = 'Sequencer is armed and ready to start.'

RUNNING = 'Sequencer is running.'

Q1_STOPPED = 'Classical part of the sequencer has stopped; waiting for real-time part to stop.'

STOPPED = 'Sequencer has completely stopped.'

class qblox_instruments.native.generic_func.**SequencerStatusFlags**(*value*)

Bases: *StateEnum*

Sequencer status flags enum.

DISARMED = 'Sequencer was disarmed.'

FORCED_STOP = 'Sequencer was stopped while still running.'

SEQUENCE_PROCESSOR_Q1_ILLEGAL_INSTRUCTION = 'Classical sequencer part executed an unknown instruction.'

SEQUENCE_PROCESSOR_RT_EXEC_ILLEGAL_INSTRUCTION = 'Real-time sequencer part executed an unknown instruction.'

SEQUENCE_PROCESSOR_RT_EXEC_COMMAND_UNDERFLOW = 'Real-time sequencer part command queue underflow.'

AWG_WAVE_PLAYBACK_INDEX_INVALID_PATH_0 = 'AWG path 0 tried to play an unknown waveform.'

AWG_WAVE_PLAYBACK_INDEX_INVALID_PATH_1 = 'AWG path 1 tried to play an unknown waveform.'

ACQ_WEIGHT_PLAYBACK_INDEX_INVALID_PATH_0 = 'Acquisition path 0 tried to play an unknown weight.'

ACQ_WEIGHT_PLAYBACK_INDEX_INVALID_PATH_1 = 'Acquisition path 1 tried to play an unknown weight.'

ACQ_SCOPE_DONE_PATH_0 = 'Scope acquisition for path 0 has finished.'

ACQ_SCOPE_OUT_OF_RANGE_PATH_0 = 'Scope acquisition data for path 0 was out-of-range.'

ACQ_SCOPE_OVERWRITTEN_PATH_0 = 'Scope acquisition data for path 0 was overwritten.'

ACQ_SCOPE_DONE_PATH_1 = 'Scope acquisition for path 1 has finished.'

ACQ_SCOPE_OUT_OF_RANGE_PATH_1 = 'Scope acquisition data for path 1 was out-of-range.'

ACQ_SCOPE_OVERWRITTEN_PATH_1 = 'Scope acquisition data for path 1 was overwritten.'

ACQ_BINNING_DONE = 'Acquisition binning completed.'

ACQ_BINNING_FIFO_ERROR = 'Acqusition binning encountered internal FIFO error.'

ACQ_BINNING_COMM_ERROR = 'Acqusition binning encountered internal communication error.'

ACQ_BINNING_OUT_OF_RANGE = 'Acquisition binning data out-of-range.'

ACQ_INDEX_INVALID = 'Acquisition tried to process an invalid acquisition.'

ACQ_BIN_INDEX_INVALID = 'Acquisition tried to process an invalid bin.'

OUTPUT_OVERFLOW = 'Output overflow.'

CLOCK_INSTABILITY = 'Clock source instability occurred.'

class qblox_instruments.native.generic_func.**SequencerState**(*status*, *flags*)

Bases: *SequencerState*, *StateTuple*

Sequencer status tuple returned by `get_sequencer_state()`. The tuple contains a sequencer status enum of type `SequencerStatus` and a list of associated sequencer status flag enums of type `SequencerStatusFlags`.

class qblox_instruments.native.generic_func.**FuncRefs**(*instrument: Optional[Any] = None*)

Bases: `object`

Function reference container intended to hold references to methods of the instrument's SCPI and native interfaces that are called by methods in *generic_func*. In effect, this class enables passing parametrized methods to the *generic_func* functions so that those functions can be reused between different instruments.

__init__(*instrument: Optional[Any] = None*)

Create function reference container.

Parameters

instrument (*Any*) – Instrument parent object of the function references.

property instrument: Any

Return function references parent object.

Returns

Instrument parent object of the function references.

Return type

Any

property funcs: Dict

Return dictionary of instrument function names and their associate references, referenced in this module's functions so that the referenced functions can be registered to this object using the register method.

Returns

Dictionary of required instrument function names and associated references.

Return type

dict

register(*ref: Callable[[Any], Any], attr_name: Optional[str] = None*) → None

Register function reference as attribute to object.

Parameters

- **ref** (*Callable[[Any], Any]*) – Function reference to register.
- **attr_name** (*Optional[str]*) – Attribute name to register function to. If attribute name is not provided. The function is registered to the name of the reference argument.

Raises

- **AttributeError** – Could not get name of reference.
- **KeyError** – Attribute name is not found in function name list.

qblox_instruments.native.generic_func.**copy_docstr**(*src_func*)

Decorator that copies the docstring from the provided function to the decorated function.

Parameters

src_func – Function from which to copy the docstring.

qblox_instruments.native.generic_func.**check_sequencer_index**(*sequencer: int*) → None

Check if sequencer index is within range. We just check if the index is a positive integer here, because sending a negative number breaks the underlying SCPI command. The upperbound is checked by the instrument.

Parameters

sequencer (*int*) – Sequencer index.

Raises

ValueError – Sequencer index is out-of-range (i.e. < 1).

qblox_instruments.native.generic_func.**check_qrm_type**(*is_qrm_type: bool*) → None

Check if module is of type QRM. If not throw a NotImplemented exception. This helper function can be used to catch execution of QRM functionality on non-QRM type modules.

Parameters

is_qrm_type (*bool*) – Is QRM module type.

Raises

NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**create_read_bin**(*read_bin_func: Callable[[str, bool], bytes]*, *cmd: str*) → Callable[[Optional[int], Optional[str]], bytes]

Create binary read function that can provide a binary read with a preconfigured command. This is usefull for functions like *_get_awg_waveforms*, that need a specific binary read command to kick off a stream of binary blocks.

Parameters

- **read_bin_func** (*Callable[[str, bool], bytes]*) – SCPI layer binary read method.
- **cmd** (*str*) – Unformated command string.

Returns

Binary read function with preconfigured command that takes the optional sequencer index and optional name string as arguments.

Return type

Callable[[Optional[int], Optional[str]], bytes]

qblox_instruments.native.generic_func.**get_scpi_commands**(*funcs:* FuncRefs) → Dict

Get SCPI commands and convert to dictionary.

Returns

Dictionary containing all available SCPI commands, corresponding parameters, arguments and Python methods and finally a descriptive comment.

Return type

dict

qblox_instruments.native.generic_func.**get_idn**(*funcs:* FuncRefs) → Dict

Get device identity and build information and convert them to a dictionary.

Returns

Dictionary containing manufacturer, model, serial number and build information. The build information is subdivided into FPGA firmware, kernel module software, application software and driver software build information. Each of those consist of the version, build date, build Git hash and Git build dirty indication.

Return type

dict

qblox_instruments.native.generic_func.**get_system_state**(*funcs:* FuncRefs) → *SystemState*

Get general system state and convert it to a `SystemState`.

Returns

Tuple containing general system status and corresponding flags.

Return type

SystemStatus

qblox_instruments.native.generic_func.**get_acq_scope_config_format**() → str

Get format for converting the configuration dictionary to a C struct.

Returns

String compatible with struct package.

Return type

str

qblox_instruments.native.generic_func.**set_acq_scope_config**(*funcs:* FuncRefs, *config: Dict*) → None

Set configuration of the scope acquisition. The configuration consists of multiple parameters in a C struct format. If an invalid sequencer index is given or the configation struct does not have the correct format, an error is set in system error.

Parameters
config (*dict*) – Configuration dictionary.
Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**get_acq_scope_config**(*funcs:* FuncRefs) → Dict

Get configuration of the scope acquisition. The configuration consists of multiple parameters in a C struct format. If an invalid sequencer index is given, an error is set in system error.
Returns
Configuration dictionary.
Return type
dict
Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**set_acq_scope_config_val**(*funcs:* FuncRefs, *param: str*, *val: Any*) → None

Set value of specific scope acquisition parameter.
Parameters
- **param** (*str*) – Parameter name.
- **val** (*Any*) – Value to set parameter to.

qblox_instruments.native.generic_func.**get_acq_scope_config_val**(*funcs:* FuncRefs, *param: str*) → Any

Get value of specific scope acquisition parameter.
Parameters
param (*str*) – Parameter name.
Returns
Parameter value.
Return type
Any

qblox_instruments.native.generic_func.**set_sequencer_program**(*funcs:* FuncRefs, *sequencer: int*, *program: str*) → None

Assemble and set Q1ASM program for the indexed sequencer. If assembling failes, an RuntimeError is thrown with the assembler log attached.
Parameters
- **sequencer** (*int*) – Sequencer index.
- **program** (*str*) – Q1ASM program.

Raises
RuntimeError – Assembly failed.

qblox_instruments.native.generic_func.**get_sequencer_config_format**(*is_qrm_type: bool*) → str

Get format for converting the configuration dictionary to a C struct.
Returns
String compatible with struct package.
Return type
str

qblox_instruments.native.generic_func.**set_sequencer_config**(*funcs:* FuncRefs, *sequencer: int*, *config: Dict*) → None

Set configuration of the indexed sequencer. The configuration consists dictionary containing multiple parameters that will be converted into a C struct supported by the Pulsar QRM.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **config** (*dict*) – Configuration dictionary.

qblox_instruments.native.generic_func.**get_sequencer_config**(*funcs:* FuncRefs, *sequencer: int*) → Dict

Get configuration of the indexed sequencer. The configuration consists dictionary containing multiple parameters that will be converted from a C struct provided by the Pulsar QRM.

Parameters

sequencer (*int*) – Sequencer index.

Returns

Configuration dictionary.

Return type

dict

qblox_instruments.native.generic_func.**set_sequencer_config_val**(*funcs:* FuncRefs, *sequencer: int*, *param: str*, *val: Any*) → None

Set value of specific sequencer parameter.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **param** (*str*) – Parameter name.
- **val** (*Any*) – Value to set parameter to.

qblox_instruments.native.generic_func.**get_sequencer_config_val**(*funcs:* FuncRefs, *sequencer: int*, *param: str*) → Any

Get value of specific sequencer parameter.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **param** (*str*) – Parameter name.

Returns

Parameter value.

Return type

Any

qblox_instruments.native.generic_func.**set_sequencer_config_rotation_matrix**(*funcs:* FuncRefs, *sequencer: int*, *phase_incr: float*) → None

Sets the integration result phase rotation matrix in the acquisition path.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **phase_incr** (*float*) – Phase increment in degrees.

Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**get_sequencer_config_rotation_matrix**(*funcs:* FuncRefs, *sequencer:* *int*) → float

Gets the integration result phase rotation matrix in the acquisition path.

Parameters
sequencer (*int*) – Sequencer index.

Returns
Phase increment in degrees.

Return type
float

Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**set_sequencer_channel_map**(*funcs:* FuncRefs, *sequencer:* *int*, *output:* *int*, *enable:* *bool*) → None

Set enable of the indexed sequencer's path to output. If an invalid sequencer index is given or the channel map is not valid, an error is set in system error.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **output** (*int*) – Output index.
- **enable** (*bool*) – Sequencer path to output enable

qblox_instruments.native.generic_func.**get_sequencer_channel_map**(*funcs:* FuncRefs, *sequencer:* *int*, *output:* *int*) → bool

Get enable of the indexed sequencer's path to output. If an invalid sequencer index is given or the channel map is not valid, an error is set in system error.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **output** (*int*) – Output index.

Returns
Sequencer path to output enable.

Return type
bool

qblox_instruments.native.generic_func.**arm_sequencer**(*funcs:* FuncRefs, *scpi_cmd_prefix:* *str*) → None

Prepare the indexed sequencer to start by putting it in the armed state. If no sequencer index is given, all sequencers are armed. Any sequencer that was already running is stopped and rearmed. If an invalid sequencer index is given, an error is set in system error.

Parameters
sequencer (*Optional[int]*) – Sequencer index.

Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

qblox_instruments.native.generic_func.**start_sequencer**(*funcs:* FuncRefs, *scpi_cmd_prefix: str*) → None

Start the indexed sequencer, thereby putting it in the running state. If an invalid sequencer index is given or the indexed sequencer was not yet armed, an error is set in system error. If no sequencer index is given, all armed sequencers are started and any sequencer not in the armed state is ignored. However, if no sequencer index is given and no sequencers are armed, and error is set in system error.

Parameters

sequencer (*Optional[int]*) – Sequencer index.

Raises

RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

qblox_instruments.native.generic_func.**stop_sequencer**(*funcs:* FuncRefs, *scpi_cmd_prefix: str*) → None

Stop the indexed sequencer, thereby putting it in the stopped state. If an invalid sequencer index is given, an error is set in system error. If no sequencer index is given, all sequencers are stopped.

Parameters

sequencer (*Optional[int]*) – Sequencer index.

Raises

RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

qblox_instruments.native.generic_func.**get_sequencer_state**(*funcs:* FuncRefs, *sequencer: int*, *timeout: int = 0*, *timeout_poll_res: float = 0.02*) → *SequencerState*

Get the sequencer state. If an invalid sequencer index is given, an error is set in system error. If the timeout is set to zero, the function returns the state immediately. If a positive non-zero timeout is set, the function blocks until the sequencer completes. If the sequencer hasn't stopped before the timeout expires, a TimeoutError is thrown.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.

Returns

Tuple containing sequencer status and corresponding flags.

Return type

SequencerState

Raises

TimeoutError – Timeout

qblox_instruments.native.generic_func.**get_acq_acquisition_data**(*instrument: Any*, *funcs:* FuncRefs, *sequencer: int*, *name: str*) → Dict

Get acquisition data of acquisition in acquisition list of indexed sequencer's acquisition path. The acquisition scope and bin data is normalized to a range of -1.0 to 1.0 taking both the bit widths of the processing path and average count into consideration. For the binned integration results, the integration length is not handled during normalization and therefore these values have to be divided by their respective integration lenghts. If an invalid sequencer index is given or if a non-existing acquisition name is given, an error is set in system error.

Parameters

- **sequencer** (*int*) – Sequencer index.

- **name** (*str*) – Acquisition name.

Returns
Dictionary with data of single acquisition.

Return type
dict

Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

qblox_instruments.native.generic_func.**add_waveforms**(*funcs:* FuncRefs, *sequencer:* *int*, *waveforms:* *Dict*) → None

Add all waveforms in JSON compatible dictionary to the AWG waveform list of indexed sequencer. The dictionary must be structured as follows:

- name: waveform name.
  - data: waveform samples in a range of 1.0 to -1.0.
  - **index: optional waveform index used by the sequencer Q1ASM program** to refer to the waveform.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **waveforms** (*dict*) – JSON compatible dictionary with one or more waveforms and weigths.

Raises
KeyError – Missing waveform data of waveform in dictionary.

qblox_instruments.native.generic_func.**delete_waveform**(*funcs:* FuncRefs, *sequencer:* *int*, *name:* *str* = '', *all:* *bool* = *False*) → None

Delete a waveform specified by name in the AWG waveform list of indexed sequencer or delete all waveforms if *all* is True.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Waveform name
- **all** (*bool*) – All waveforms

qblox_instruments.native.generic_func.**get_waveforms**(*funcs:* FuncRefs, *sequencer:* *int*) → Dict

Get all waveforms and weigths in the AWG waveform list of indexed sequencer. The returned dictionary is structured as follows:

- name: waveform name.
  - data: waveform samples in a range of 1.0 to -1.0.
  - **index: waveform index used by the sequencer Q1ASM program to refer** to the waveform.

Parameters
sequencer (*int*) – Sequencer index.

Returns
Dictionary with waveforms.

Return type
dict

qblox_instruments.native.generic_func.**add_weights**(*funcs:* FuncRefs, *sequencer:* *int*, *weights:* *Dict*) → None

Add all weights in JSON compatible dictionary to the aquisition weight list of indexed sequencer. The dictionary must be structured as follows:

- name : weight name.

- data: weight samples in a range of 1.0 to -1.0.
- **index: optional waveweightform index used by the sequencer Q1ASM** program to refer to the weight.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **weights** (*dict*) – JSON compatible dictionary with one or more weigths.

Raises
- **KeyError** – Missing weight data of weight in dictionary.
- **NotImplementedError** – Functionality not available on this module.

qblox_instruments.native.generic_func.**delete_weight**(*funcs:* FuncRefs, *sequencer: int*, *name: str = ''*, *all: bool = False*) → None

Delete a weight specified by name in the acquisition weight list of indexed sequencer or delete all weights if *all* is True.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Weight name
- **all** (*bool*) – All weights

Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**get_weights**(*funcs:* FuncRefs, *sequencer: int*) → Dict

Get all weigths in the acquisition weight lists of indexed sequencer. The returned dictionary is structured as follows:

-name : weight name.
- data: weight samples in a range of 1.0 to -1.0.
- **index: weight index used by the sequencer Q1ASM program to refer** to the weight.

Parameters
sequencer (*int*) – Sequencer index.

Returns
Dictionary with weights.

Return type
dict

Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**get_acquisition_state**(*funcs:* FuncRefs, *sequencer: int*, *timeout: int = 0*, *timeout_poll_res: float = 0.02*, *check_seq_state: bool = True*) → bool

Return acquisition binning completion state of the indexed sequencer. If an invalid sequencer is given, an error is set in system error. If the timeout is set to zero, the function returns the state immediately. If a positive non-zero timeout is set, the function blocks until the acquisition binning completes. If the acquisition hasn't completed before the timeout expires, a TimeoutError is thrown. Note that when sequencer state checking is enabled, the sequencer state is checked using get_sequencer_state with the selected timeout period first and then the acquisition state is checked with the same timeout period. This means that the total timeout period is two times the set timeout period.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.
- **check_seq_state** (*bool*) – Check if sequencer is done before checking acquisition state.

Returns
Indicates the acquisition binning completion state (False = uncompleted, True = completed).

Return type
bool

Raises
- **TimeoutError** – Timeout
- **NotImplementedError** – Functionality not available on this module.

qblox_instruments.native.generic_func.**add_acquisitions**(*funcs:* FuncRefs, *sequencer:* *int*, *acquisitions:* *Dict*) → None

Add all waveforms and weights in JSON compatible dictionary to AWG waveform and aquisition weight lists of indexed sequencer. The dictionary must be structured as follows:
- **name: acquisition name.**
  - num_bins: number of bins in acquisition.
  - **index: optional acquisition index used by the sequencer Q1ASM** program to refer to the acquition.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **acquisitions** (*dict*) – JSON compatible dictionary with one or more acquisitions.

Raises
- **KeyError** – Missing dictionary key in acquisitions.
- **NotImplementedError** – Functionality not available on this module.

qblox_instruments.native.generic_func.**delete_acquisition**(*funcs:* FuncRefs, *sequencer:* *int*, *name:* *str* = '', *all:* *bool* = *False*) → None

Delete an acquisition specified by name in the acquisition list of indexed sequencer or delete all acquisitions if *all* is True.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Weight name
- **all** (*bool*) – All weights

Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**delete_acquisition_data**(*funcs:* FuncRefs, *sequencer:* *int*, *name:* *str* = '', *all:* *bool* = *False*) → None

Delete data from an acquisition specified by name in the acquisition list of indexed sequencer or delete data in all acquisitions if *all* is True.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Weight name

Raises
NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**store_scope_acquisition**(*funcs:* FuncRefs, *sequencer: int*, *name: str*) → None

After an acquisition has completed, store the scope acquisition results in the acquisition specified by name of the indexed sequencers. If an invalid sequencer index is given an error is set in system error. To get access to the acquisition results, the sequencer will be stopped when calling this function.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Acquisition name.

Raises

NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**get_acquisitions**(*funcs:* FuncRefs, *sequencer: int*) → Dict

Get all acquisitions in acquisition lists of indexed sequencer. The acquisition scope and bin data is normalized to a range of -1.0 to 1.0 taking both the bit widths of the processing path and average count into considaration. For the binned integration results, the integration length is not handled during normalization and therefore these values have to be divided by their respective integration lenghts. The returned dictionary is structured as follows:

- name: acquisition name
  - **index: acquisition index used by the sequencer Q1ASM program to refer to the acquisition.**
  - acquisition: acquisition dictionary
    - scope: Scope data
      - path0: input path 0
      - data: acquisition samples in a range of 1.0 to -1.0.
      - **out-of-range: out-of-range indication for the entire acquisition (False = in-range, True = out-of-range).**
      - avg_cnt: number of averages.
      - path1: input path 1
      - data: acquisition samples in a range of 1.0 to -1.0.
      - **out-of-range: out-of-range indication for the entire acquisition (False = in-range, True = out-of-range).**
      - avg_cnt: number of averages.
    - bins: bin data
      - integration: integration data
      - path_0: input path 0 integration result bin list
      - path_1: input path 1 integration result bin list
      - threshold: threshold result bin list
      - valid: list of valid indications per bin
      - avg_cnt: list of number of averages per bin

Parameters

sequencer (*int*) – Sequencer index.

Returns

Dictionary with acquisitions.

Return type

dict

Raises

NotImplementedError – Functionality not available on this module.

qblox_instruments.native.generic_func.**set_sequence**(*funcs:* FuncRefs, *sequencer: int*, *sequence: Union[str, Dict[str, Any]]*, *validation_enable: bool = True*) → None

Set sequencer program, AWG waveforms, acquisition weights and acquisitions from a JSON file or from a dictionary directly. The JSON file or dictionary need to apply the schema specified by *QCM_SEQUENCE_JSON_SCHEMA*, *QRM_SEQUENCE_JSON_SCHEMA*, *WAVE_JSON_SCHEMA* and *ACQ_JSON_SCHEMA*.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **sequence** (*Union[str, Dict[str, Any]]*) – Path to sequence file or dictionary.
- **validation_enable** (*bool*) – Enable JSON schema validation on sequence.

Raises

JsonSchemaValueException – Invalid JSON object.

1.27 Cluster

The Cluster driver is separated into three layers:

- *QCoDeS driver*: Instrument driver based on QCoDeS and the instrument's native interface.
- *Native interface*: Instrument API that provides control over the instrument and is an extension of the the SCPI interface.
- *SCPI interface*: Instrument API based on the SCPI standard which in turn is based on IEEE488.2.

Warning: This is a preliminary driver for the Cluster that is currently under development.

1.27.1 QCoDeS driver

class qblox_instruments.**Cluster**(**args: Any*, ***kwargs: Any*)

Bases: *Cluster*, Instrument

This class connects QCoDeS to the Cluster native interface.

__init__(*name: str*, *identifier: Optional[str] = None*, *port: Optional[int] = None*, *debug: Optional[int] = None*, *dummy_cfg: Optional[Dict] = None*)

Creates Cluster QCoDeS class and adds all relevant instrument parameters. These instrument parameters call the associated methods provided by the native interface.

Parameters

- **name** (*str*) – Instrument name.
- **identifier** (*Optional[str]*) – Instrument identifier. See *resolve()*. If None, the instrument is identified by name.
- **port** (*Optional[int]*) – Override for the TCP port through which we should connect.
- **debug** (*Optional[int]*) – Debug level (0 | None = normal, 1 = no version check, >1 = no version or error checking).
- **dummy_cfg** (*Optional[Dict]*) – Configure as dummy using this configuration. For each slot that needs to be occupied by a module

add the slot index as key and specify the type of module in the slot using the type *ClusterType*.

property modules: List

Get list of modules.

Returns

List of modules.

Return type

list

reset() → None

Resets device, invalidates QCoDeS parameter cache and clears all status and event registers (see SCPI).

QCoDeS instrument parameters

QCoDeS parameters generated by *Cluster*.

Cluster.IDN

Please see QCoDeS for a description.

Properties

- **value**: Anything

Cluster.reference_source

Sets/gets reference source ('internal' = internal 10 MHz, 'external' = external 10 MHz).

Properties

- **value**: Enum: {'internal', 'external'}

QCoDeS module parameters

Besides instrument parameters, the Cluster also has parameters for each module in the Cluster which are described in *QCM-QRM*.

1.27.2 Native interface

class qblox_instruments.native.**Cluster**(*identifier: str*, *port: Optional[int] = None*, *debug: Optional[int] = None*, *dummy_cfg: Optional[Dict] = None*)

Bases: *Cluster*

Class that provides the native API for the Cluster. It provides methods to control all functions and features provided by the Cluster.

Note that the bulk of the functionality of this class is contained in the *generic_func* module so that this functionality can be shared between instruments.

__init__(*identifier: str*, *port: Optional[int] = None*, *debug: Optional[int] = None*, *dummy_cfg: Optional[Dict] = None*)

Creates Cluster native interface object.

Parameters

- **identifier** (*str*) – Instrument identifier. See *resolve()* for more information.
- **port** (*Optional[int]*) – Instrument port. If None, this will be determined automatically.
- **debug** (*Optional[int]*) – Debug level (0 | None = normal, 1 = no version check, >1 = no version or error checking).
- **dummy_cfg** (*Optional[Dict]*) – Configure as dummy using this configuration. For each slot that needs to be occupied by a module add the slot index as key and specify the type of module in the slot using the type *ClusterType*.

Raises

- **RuntimeError** – Instrument cannot be reached due to invalid IP configuration.
- **ConnectionError** – Instrument type is not supported.

property instrument_class: *InstrumentClass*

Get instrument class (e.g. Pulsar, Cluster).

Returns

Instrument class

Return type

InstrumentClass

property instrument_type: *InstrumentType*

Get instrument type (e.g. MM, QRM, QCM).

Returns

Instrument type

Return type

InstrumentType

get_idn() → Dict

Get device identity and build information and convert them to a dictionary.

Returns

Dictionary containing manufacturer, model, serial number and build information. The build information is subdivided into FPGA firmware, kernel module software, application software and driver software build information. Each of those consist of the version, build date, build Git hash and Git build dirty indication.

Return type

dict

get_system_state() → *SystemState*

Get general system state and convert it to a `SystemState`.

Returns

Tuple containing general system status and corresponding flags.

Return type

SystemStatus

arm_sequencer(*slot: Optional[int] = None*, *sequencer: Optional[int] = None*) → None

Prepare the indexed sequencer to start by putting it in the armed state. If no sequencer index is given, all sequencers are armed. Any sequencer that was already running is stopped and rearmed. If an invalid sequencer index is given, an error is set in system error.

Parameters
sequencer (*Optional[int]*) – Sequencer index.
Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

start_sequencer(*slot: Optional[int] = None*, *sequencer: Optional[int] = None*) → None

Start the indexed sequencer, thereby putting it in the running state. If an invalid sequencer index is given or the indexed sequencer was not yet armed, an error is set in system error. If no sequencer index is given, all armed sequencers are started and any sequencer not in the armed state is ignored. However, if no sequencer index is given and no sequencers are armed, and error is set in system error.

Parameters
sequencer (*Optional[int]*) – Sequencer index.
Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

stop_sequencer(*slot: Optional[int] = None*, *sequencer: Optional[int] = None*) → None

Stop the indexed sequencer, thereby putting it in the stopped state. If an invalid sequencer index is given, an error is set in system error. If no sequencer index is given, all sequencers are stopped.

Parameters
sequencer (*Optional[int]*) – Sequencer index.
Raises
RuntimeError – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_sequencer_state(*slot: int*, *sequencer: int*, *timeout: int = 0*, *timeout_poll_res: float = 0.02*) → *SequencerState*

Get the sequencer state. If an invalid sequencer index is given, an error is set in system error. If the timeout is set to zero, the function returns the state immediately. If a positive non-zero timeout is set, the function blocks until the sequencer completes. If the sequencer hasn't stopped before the timeout expires, a TimeoutError is thrown.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.

Returns
Tuple containing sequencer status and corresponding flags.
Return type
SequencerState
Raises
TimeoutError – Timeout

get_waveforms(*slot: int*, *sequencer: int*) → Dict

Get all waveforms and weigths in the AWG waveform list of indexed sequencer. The returned dictionary is structured as follows:

- name: waveform name.
  - data: waveform samples in a range of 1.0 to -1.0.
  - **index: waveform index used by the sequencer Q1ASM program to refer** to the waveform.

Parameters
sequencer (*int*) – Sequencer index.

Returns
Dictionary with waveforms.
Return type
dict

get_weights(*slot: int*, *sequencer: int*) → Dict

Get all weigths in the acquisition weight lists of indexed sequencer. The returned dictionary is structured as follows:

-name : weight name.
- data: weight samples in a range of 1.0 to -1.0.
- **index: weight index used by the sequencer Q1ASM program to refer** to the weight.

Parameters
sequencer (*int*) – Sequencer index.
Returns
Dictionary with weights.
Return type
dict
Raises
NotImplementedError – Functionality not available on this module.

get_acquisition_state(*slot: int*, *sequencer: int*, *timeout: int = 0*, *timeout_poll_res: float = 0.02*) → bool

Return acquisition binning completion state of the indexed sequencer. If an invalid sequencer is given, an error is set in system error. If the timeout is set to zero, the function returns the state immediately. If a positive non-zero timeout is set, the function blocks until the acquisition binning completes. If the acquisition hasn't completed before the timeout expires, a TimeoutError is thrown. Note that when sequencer state checking is enabled, the sequencer state is checked using get_sequencer_state with the selected timeout period first and then the acquisition state is checked with the same timeout period. This means that the total timeout period is two times the set timeout period.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.
- **check_seq_state** (*bool*) – Check if sequencer is done before checking acquisition state.

Returns
Indicates the acquisition binning completion state (False = uncompleted, True = completed).
Return type
bool
Raises
- **TimeoutError** – Timeout
- **NotImplementedError** – Functionality not available on this module.

delete_acquisition_data(*slot: int*, *sequencer: int*, *name: str = ''*, *all: bool = False*) → None

Delete data from an acquisition specified by name in the acquisition list of indexed sequencer or delete data in all acquisitions if *all* is True.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Weight name

Raises
NotImplementedError – Functionality not available on this module.

store_scope_acquisition(*slot: int*, *sequencer: int*, *name: str*) → None

After an acquisition has completed, store the scope acquisition results in the acquisition specified by name of the indexed sequencers. If an invalid sequencer index is given an error is set in system error. To get access to the acquisition results, the sequencer will be stopped when calling this function.

Parameters
- **sequencer** (*int*) – Sequencer index.
- **name** (*str*) – Acquisition name.

Raises
NotImplementedError – Functionality not available on this module.

get_acquisitions(*slot: int*, *sequencer: int*) → Dict

Get all acquisitions in acquisition lists of indexed sequencer. The acquisition scope and bin data is normalized to a range of -1.0 to 1.0 taking both the bit widths of the processing path and average count into considaration. For the binned integration results, the integration length is not handled during normalization and therefore these values have to be divided by their respective integration lenghts. The returned dictionary is structured as follows:

- name: acquisition name
  - **index: acquisition index used by the sequencer Q1ASM program to** refer to the acquisition.
  - acquisition: acquisition dictionary
    - scope: Scope data
      - path0: input path 0
      - data: acquisition samples in a range of 1.0 to -1.0.
      - **out-of-range: out-of-range indication for the entire** acquisition (False = in-range, True = out-of-range).
      - avg_cnt: number of averages.
      - path1: input path 1
      - data: acquisition samples in a range of 1.0 to -1.0.
      - **out-of-range: out-of-range indication for the entire** acquisition (False = in-range, True = out-of-range).
      - avg_cnt: number of averages.
    - bins: bin data
      - integration: integration data
      - path_0: input path 0 integration result bin list
      - path_1: input path 1 integration result bin list
      - threshold: threshold result bin list
      - valid: list of valid indications per bin
      - avg_cnt: list of number of averages per bin

Parameters
sequencer (*int*) – Sequencer index.

Returns
Dictionary with acquisitions.

Return type
dict

Raises
NotImplementedError – Functionality not available on this module.

set_dummy_binned_acquisition_data(*slot_idx: int*, *sequencer: int*, *acq_index_name: str*, *data: Iterable[Optional[DummyBinnedAcquisitionData]]*)

Set dummy binned acquisition data for the dummy.

Parameters

- **slot_idx** (*int*) – Slot of the hardware you want to set the data to on a cluster.
- **sequencer** (*int*) – Sequencer.
- **acq_index_name** (*str*) – Acquisition index name.
- **data** (*Iterable[Union[DummyBinnedAcquisitionData, None]]*) – Dummy data for the binned acquisition. An iterable of all the bin values.

set_dummy_scope_acquisition_data(*slot_idx: int*, *sequencer: Optional[int]*, *data:* DummyScopeAcquisitionData)

Set dummy scope acquisition data for the dummy.

Parameters

- **slot_idx** (*int*) – Slot of the hardware you want to set the data to on a cluster.
- **sequencer** (*Union[int, None]*) – Sequencer.
- **data** (DummyScopeAcquisitionData) – Dummy data for the scope acquisition.

1.27.3 SCPI interface

class qblox_instruments.scpi.**Cluster**(*transport:* Transport, *debug: int = 0*)

Bases: *Ieee488_2*

This interface provides an API for the mandatory and required SCPI calls and adds Pulsar related functionality (see SCPI).

__init__(*transport:* Transport, *debug: int = 0*)

Creates SCPI interface object.

Parameters

- **transport** (Transport) – Transport class responsible for the lowest level of communication (e.g. Ethernet).
- **debug** (*int*) – Debug level (0 = normal, 1 = no version check, >1 = no version or error checking).

Raises

ConnectionError – Debug level is 0 and there is a device or version mismatch.

get_system_error() → str

Get system error from queue (see SCPI).

Parameters

None –

Returns

System error description string.

Return type

str

get_num_system_error() → int

Get number of system errors (see SCPI).

Parameters

None –

Returns

Current number of system errors.

Return type
int

`get_system_version()` → str

Get SCPI system version (see SCPI).

Parameters
None –

Returns
SCPI system version.

Return type
str

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`set_name(name: str)` → None

Sets the customer-specified name of the instrument. The name must not contain any newlines, backslashes, or double quotes.

Parameters
name (*str*) – The new name for the device.

Return type
None

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_name()` → str

Returns the customer-specified name of the instrument.

Parameters
None –

Returns
The name of the device.

Return type
str

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`set_ip_config(config: str)` → None

Reconfigures the IP address of this device. The configuration will not go into effect until reboot() is called or the device is power-cycled.

Parameters
config (*str*) –
IP configuration. May be one of the following things:
- an IPv4 address including prefix length, for example 192.168.0.2/24, - the string *dhcp* to enable IPv4 DHCP, - an IPv6 address including prefix length, for example 1:2::3:4/64, or - a semicolon-separated combination of an IPv4 configuration (IP address or *dhcp*) and an IPv6 address.

Return type
None

Raises
- **Exception** – Invalid input parameter type.

- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_ip_config() → str

Returns the IP address configuration that will go into effect when the device reboots.

Parameters

None –

Returns

IP configuration. Can be one of the following things:

- an IPv4 address including prefix length, for example 192.168.0.2/24, - the string *dhcp* to enable IPv4 DHCP, - an IPv6 address including prefix length, for example 1:2::3:4/64, or - a semicolon-separated combination of an IPv4 configuration (IP address or *dhcp*) and an IPv6 address.

Return type

str

Raises

Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

reboot() → None

Reboots the instrument.

Parameters

None –

Return type

None

Raises

Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

clear() → None

Clear all status and event registers (see SCPI).

Parameters

None –

Return type

None

Raises

- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_status_byte() → int

Get status byte register. Register is only cleared when feeding registers are cleared (see SCPI).

Parameters

None –

Returns

Status byte register.

Return type

int

Raises

Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_service_request_enable(*reg: int*) → None

Set service request enable register (see SCPI).

Parameters
reg (*int*) – Service request enable register.
Return type
None
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_service_request_enable() → int

Get service request enable register. The register is cleared after reading it (see SCPI).
Parameters
None –
Returns
Service request enable register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_standard_event_status_enable(*reg: int*) → None

Set standard event status enable register (see SCPI).
Parameters
reg (*int*) – Standard event status enable register.
Return type
None
Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_standard_event_status_enable() → int

Get standard event status enable register. The register is cleared after reading it (see SCPI).
Parameters
None –
Returns
Standard event status enable register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_standard_event_status() → int

Get standard event status register. The register is cleared after reading it (see SCPI).
Parameters
None –
Returns
Standard event status register.
Return type
int
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`set_operation_complete()` → None

Set device in operation complete query active state (see SCPI).

Parameters
`None` –

Return type
None

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_operation_complete()` → bool

Get operation complete state (see SCPI).

Parameters
`None` –

Returns
Operation complete state (False = running, True = completed).

Return type
bool

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`test()` → bool

Run self-test. Currently not implemented (see SCPI).

Parameters
`None` –

Returns
Test result (False = failed, True = success).

Return type
bool

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`wait()` → None

Wait until operations completed before continuing (see SCPI).

Parameters
`None` –

Return type
None

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`preset_system_status()` → None

Preset system status registers. Connects general system status flags for PLL unlock and temperature out-of-range indications to event status enable, status questionable temperature and status questionable frequency registers respectively (see SCPI).

Parameters
`None` –

Return type
None

Raises
`Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_questionable_condition()` → int

Get status questionable condition register (see SCPI).

Parameters
None –

Returns
Status questionable condition register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_questionable_event()` → int

Get status questionable event register (see SCPI).

Parameters
None –

Returns
Status questionable event register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`set_questionable_enable(reg: int)` → None

Set status questionable enable register (see SCPI).

Parameters
reg (*int*) – Status questionable enable register.

Return type
None

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_questionable_enable()` → int

Get status questionable enable register (see SCPI).

Parameters
None –

Returns
Status questionable enable register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_operation_condition()` → int

Get status operation condition register (see SCPI).

Parameters
None –

Returns
Status operation condition register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_events() → int

Get status operation event register (see SCPI).

Parameters
None –

Returns
Status operation event register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

set_operation_enable(*reg: int*) → None

Set status operation enable register (see SCPI).

Parameters
reg (*int*) – Status operation enable register.

Return type
None

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_operation_enable() → int

Get status operation enable register (see SCPI).

Parameters
None –

Returns
Status operation enable register.

Return type
int

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

identify() → None

Toggle frontpanel LEDs to visually identify the instrument.

Parameters
None –

Return type
None

Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_fpga_temperature(*slot: int*) → float

Get current FPGA junction temperature (inside device).

Parameters
slot (*int*) – slot index.

Returns
Current FPGA junction temperature.

Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_maximum_fpga_temperature(*slot: int*) → float
Get maximum FPGA junction temperature since boot or clear (inside device).
Parameters
slot (*int*) – slot index.
Returns
Maximum FPGA junction temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_carrier_temperature(*slot: int*) → float
Get current carrier board temperature (inside device).
Parameters
slot (*int*) – slot index.
Returns
Current carrier board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_maximum_carrier_temperature(*slot: int*) → float
Get maximum carrier board temperature since boot or clear (inside device).
Parameters
slot (*int*) – slot index.
Returns
Maximum carrier board temperature.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_bp_temperature_0() → float
Get current backplane board temperature from sensor 0 (inside device).
Parameters
None –
Returns
Current backplane board temperature from sensor 0.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_maximum_bp_temperature_0() → float
Get maximum backplane board temperature from sensor 0 since boot or clear (inside device).

Parameters
None –
Returns
Maximum backplane board temperature from sensor 0.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_bp_temperature_1() → float

Get current backplane board temperature from sensor 1 (inside device).
Parameters
None –
Returns
Current backplane board temperature from sensor 1.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_maximum_bp_temperature_1() → float

Get maximum backplane board temperature from sensor 1 since boot or clear (inside device).
Parameters
None –
Returns
Maximum backplane board temperature from sensor 1.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_bp_temperature_2() → float

Get current backplane board temperature from sensor 2 (inside device).
Parameters
None –
Returns
Current backplane board temperature from sensor 2.
Return type
float
Raises
Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_maximum_bp_temperature_2() → float

Get maximum backplane board temperature from sensor 2 since boot or clear (inside device).
Parameters
None –
Returns
Maximum backplane board temperature from sensor 2.
Return type
float
Raises

Exception – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_assembler_status(*slot: int*) → bool

Get assembler status. Refer to the assembler log to get more information regarding the assembler result.

Parameters
slot (*int*) – slot index.

Returns
Assembler status (False = failed, True = success).

Return type
bool

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_assembler_log(*slot: int*) → str

Get assembler log.

Parameters
slot (*int*) – slot index.

Returns
Assembler log.

Return type
str

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_current_afe_temperature(*slot: int*) → float

Get current AFE temperature.

Parameters
slot (*int*) – slot index.

Returns
current

Return type
float

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

get_maximum_afe_temperature(*slot: int*) → float

Get maximum AFE temperature.

Parameters
slot (*int*) – slot index.

Returns
maximum

Return type
float

Raises
- **Exception** – Invalid input parameter type.
- **Exception** – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_current_lo_temperature`(*slot: int*) → float

Get current LO temperature.

Parameters
`slot` (*int*) – slot index.

Returns
current

Return type
float

Raises
- `Exception` – Invalid input parameter type.
- `Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`get_maximum_lo_temperature`(*slot: int*) → float

Get maximum LO temperature.

Parameters
`slot` (*int*) – slot index.

Returns
maximum

Return type
float

Raises
- `Exception` – Invalid input parameter type.
- `Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

`start_adc_calib`(*slot: int*) → None

Calibrates ADC delay and offset values. This method sets the correct delay values for every input data index (IO data lane) in order to avoid timing violations which occur while sampling ADC data. It also calibrates offsets internal to the ADC.

Parameters
`slot` (*int*) – slot index.

Return type
None

Raises
- `Exception` – Invalid input parameter type.
- `Exception` – An error is reported in system error and debug <= 1. All errors are read from system error and listed in the exception.

1.27.4 Supporting classes and functions

Please have a look at:

- *QCM-QRM*
- *Sequencer*
- *Generic native interface functions*
- *IEEE488.2*
- *Instrument types*

1.28 SPI Rack

The SPI rack driver is separated into two layers:

- *QCoDeS driver*: Instrument driver based on QCoDeS and the instrument's native interface.
- *Native interface*: Instrument API that provides control over the instrument and is an extension of the spirack package.

1.28.1 QCoDeS driver

class qblox_instruments.**SpiRack**(**args: Any*, ***kwargs: Any*)

Bases: *SpiRack*, Instrument

SPI rack driver class based on QCoDeS.

Example usage:

```
from qblox_instruments import SpiRack
from qblox_instruments.qcodes_drivers.spi_rack_modules import S4gModule

spi = SpiRack("my_spi_rack", "COM4")    # connects to an SPI rack on COM␣
↪port 4
spi.add_spi_module(3, "D5a", "alice")  # adds an D5a module with address 3␣
↪named "alice"
spi.add_spi_module(2, "S4g", "bob")    # adds an S4g module with address 2␣
↪named "bob"
spi.add_spi_module(6, S4gModule)       # adds an S4g module with address 6␣
↪with the default name module6

spi.bob.dac0.current(10e-3)            # sets the current of output 1 of␣
↪the S4g module named "bob" to 10 mA
spi.alice.dac6.voltage(-2)             # sets the voltage of output 7 of␣
↪the D5a module named "alice" to -2 V
```

__init__(*name: str*, *address: str*, *baud_rate: int = 9600*, *timeout: float = 1*, *is_dummy: bool = False*)

Instantiates the driver object.

Parameters

- **name** (*str*) – Instrument name.
- **address** (*str*) – COM port used by SPI rack controller unit (e.g. "COM4")
- **baud_rate** (*int*) – Baud rate
- **timeout** (*float*) – Data receive timeout in seconds
- **is_dummy** (*bool*) – If true, the SPI rack driver is operating in "dummy" mode for testing purposes.

add_spi_module(*address: int*, *module_type: Union[*SpiModuleBase, *str]*, *name: Optional[str] = None*, ***kwargs*) → None

Add a module to the driver.

Parameters

- **address** (*int*) – Address that the module is set to (set internally on the module itself with a jumper).

- **module_type** (Union[str, *SpiModuleBase*]) – Either a str that is defined in _MODULES_MAP, or a reference to a class derived from *SpiModuleBase*.
- **name** (*Optional[str]*) – Optional name of the module. If no name is given or is None, a default name of "module{address}" is used.

Raises

ValueError – module_type is not a string or a subclass of *SpiModuleBase*

close() → None

Closes connection to hardware and closes the Instrument.

connect_message(*idn_param: str = 'IDN'*, *begin_time: Optional[float] = None*) → None

Print a standard message on initial connection to an instrument. Overridden from superclass to accommodate IEEE488.2 for IDN.

Parameters

- **idn_param** (*str*) – Name of parameter that returns ID dict. Default `IDN`.
- **begin_time** (*Optional[float]*) – `time.time()` when init started. Default is `self._t0`, set at start of `Instrument.__init__`.

class qblox_instruments.qcodes_drivers.spi_rack_modules.**SpiModuleBase**(*parent*, *name: str*, *address: int*, ***kwargs*)

Bases: *SpiModuleBase*, `InstrumentChannel`

Defines an abstract base class for SPI modules. All module drivers should inherit from this class.

This class defines no actual functionality but rather serves to provide a common interface shared among all modules.

Parameters

- **parent** – Reference to the *SpiRack* parent object. This is handled by the *add_spi_module()* function.
- **name** (*str*) – Name given to the InstrumentChannel.
- **address** (*int*) – Module number set on the hardware (set internally on the module itself with a jumper).

__init__(*parent*, *name: str*, *address: int*, ***kwargs*)

Defines an abstract base class for SPI modules. All module drivers should inherit from this class.

This class defines no actual functionality but rather serves to provide a common interface shared among all modules.

Parameters

- **parent** – Reference to the *SpiRack* parent object. This is handled by the *add_spi_module()* function.
- **name** (*str*) – Name given to the module.
- **address** (*int*) – Module number set on the hardware (set internally on the module itself with a jumper).

QCoDeS instrument parameters

QCoDeS parameters generated by *SpiRack*.

SPIRack.IDN

Please see QCoDeS for a description.

Properties

- **value**: Anything

SPIRack.temperature

Returns the temperature in the C1b module. Reads the temperature from the internal C1b temperature sensor. Accuracy is +- 0.5 degrees in 0-70 degree range.

Properties

- **unit**: C
- **value**: on

SPIRack.battery_voltages

Calculates the battery voltages from the ADC channel values. Returns: [VbatPlus, VbatMin]

Properties

- **unit**: V
- **value**: on

1.28.2 Native interface

class qblox_instruments.native.**SpiRack**(*address: str*, *baud_rate: int = 9600*, *timeout: float = 1*, *is_dummy: bool = False*)

Bases: object

SPI rack native interface class. This class relies on the spirack API.

__init__(*address: str*, *baud_rate: int = 9600*, *timeout: float = 1*, *is_dummy: bool = False*)

Instantiates the driver object.

Parameters

- **address** (*str*) – COM port used by SPI rack controller unit (e.g. "COM4")
- **baud_rate** (*int*) – Baud rate
- **timeout** (*float*) – Data receive timeout in seconds
- **is_dummy** (*bool*) – If true, the SPI rack driver is operating in "dummy" mode for testing purposes.

add_spi_module(*address: int*, *module_type: Union[SpiModuleBase, str]*, *name: Optional[str] = None*, ***kwargs*) → Tuple[str, *SpiModuleBase*]

Add a module to the driver and return module object.

Parameters

- **address** (*int*) – Address that the module is set to (set internally on the module itself with a jumper).
- **module_type** (Union[str, *SpiModuleBase*]) – Either a str that is defined in _MODULES_MAP, or a reference to a class derived from *SpiModuleBase*.
- **name** (*Optional[str]*) – Optional name of the module. If no name is given or is None, a default name of "module{address}" is used.

Returns

- *str* – Name
- *SpiModuleBase* – SPI module object

Raises

ValueError – module_type is not a string or a subclass of *SpiModuleBase*

get_idn() → Dict

Generates the IDN dict.

Returns

The QCoDeS style IDN dictionary. Currently only the firmware version is actually read from hardware.

Return type

dict

close() → None

Closes connection to hardware and closes the Instrument.

set_dacs_zero() → None

Calls the *set_dacs_zero()* function on all the modules, which in turn should cause all output values to be set to 0.

class qblox_instruments.native.spi_rack_modules.**SpiModuleBase**(*parent*, *name: str*, *address: int*, ***kwargs*)

Bases: object

Abstract SPI module base class.

__init__(*parent*, *name: str*, *address: int*, ***kwargs*)

Defines an abstract base class for SPI modules. All module drivers should inherit from this class.

This class defines no actual functionality but rather serves to provide a common interface shared among all modules.

Parameters

- **parent** – Reference to the *SpiRack* parent object. This is handled by the *add_spi_module()* function.
- **name** (*str*) – Name given to the module.
- **address** (*int*) – Module number set on the hardware (set internally on the module itself with a jumper).

abstract set_dacs_zero() → None

" Base method for set_dacs_zero. Should be overridden by subclass.

1.28.3 S4g QCoDeS driver

QCoDeS parameters generated by *S4gModule*.

class qblox_instruments.qcodes_drivers.spi_rack_modules.**S4gModule**(*parent*, *name: str*, *address: int*, *reset_currents: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Bases: *S4gModule*, InstrumentChannel

QCoDeS style instrument channel driver for the S4g SPI module.

NUMBER_OF_DACS = 4

__init__(*parent*, *name: str*, *address: int*, *reset_currents: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Instantiates the driver object. This is the object that should be instantiated by the *add_spi_module()* function.

Parameters

- **parent** – Reference to the *SpiRack* parent object. This is handled by the *add_spi_module()* function.
- **name** (*str*) – Name given to the InstrumentChannel.
- **address** (*int*) – Module number set on the hardware.
- **reset_currents** (*bool*) – If True, then reset all currents to zero and change the span to *range_max_bi*.
- **dac_names** (*Optional[List[str]]*) – List of all the names to use for the dac channels. If no list is given or is None, the default name "dac{i}" is used for the i-th dac channel.
- **is_dummy** (*bool*) – If true, do not connect to physical hardware, but use a dummy module.

Raises

ValueError – Length of the dac names list does not match the number of dacs.

set_dacs_zero() → None

Sets all currents of all outputs to 0.

S4g QCoDeS parameters

S4g.current

Sets the output current of the dac channel. Depending on the value of ramping_enabled, the output value is either achieved through slowly ramping, or instantaneously set.

Properties

- **unit**: A
- **value**: Numbers -0.04<=v<=0.04

S4g.span

Sets the max range of the DACs. Possible values: 'range_max_uni': 0 - 40 mA, 'range_max_bi': -40 - 40 mA, 'range_min_bi': -20 - 20 mA.

Properties

- **value**: Enum: {'range_max_bi', 'range_max_uni', 'range_min_bi'}

S4g.ramp_rate

Limits the rate at which currents can be changed. The size of of steps is still limited by *ramp_max_step*.

Properties

- **unit**: A/s
- **value**: on

S4g.ramp_max_step

Sets the maximum step size for current ramping. The rate at which it ramps is set by *ramp_rate*.

Properties

- **unit**: A
- **value**: on

S4g.ramping_enabled

Turns ramping on or off. Toggling *ramping_enabled* changed the behavior of the setter for the *current* parameter. If enabled, ramping is done at a rate set by *ramp_rate* and in steps specified by *ramp_max_step*.

Properties

- **value**: Boolean

S4g.is_ramping

Returns whether the dac is currently in the process of ramping.

Properties

- **value**: on

`S4g.stepsize`

Returns the smallest current step allowed by the dac for the current settings.

Properties

- **unit**: A
- **value**: on

`S4g.dac_channel`

Returns the dac number of this channel.

Properties

- **value**: on

1.28.4 S4g native interface

class qblox_instruments.native.spi_rack_modules.**S4gModule**(*parent*, *name: str*, *address: int*, *reset_currents: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Bases: *SpiModuleBase*

Native driver for the S4g SPI module.

NUMBER_OF_DACS = 4

__init__(*parent*, *name: str*, *address: int*, *reset_currents: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Instantiates the driver object. This is the object that should be instantiated by the *add_spi_module()* function.

Parameters

- **parent** – Reference to the *SpiRack* parent object. This is handled by the *add_spi_module()* function.
- **name** (*str*) – Name given to the InstrumentChannel.
- **address** (*int*) – Module number set on the hardware.
- **reset_currents** (*bool*) – If True, then reset all currents to zero and change the span to *range_max_bi*.
- **dac_names** (*Optional[List[str]]*) – List of all the names to use for the dac channels. If no list is given or is None, the default name "dac{i}" is used for the i-th dac channel.
- **is_dummy** (*bool*) – If true, do not connect to physical hardware, but use dummy module.

Raises

ValueError – Length of the dac names list does not match the number of dacs.

set_dacs_zero() → None

Sets all currents of all outputs to 0.

1.28.5 D5a QCoDeS driver

QCoDeS parameters generated by *D5aModule*.

class qblox_instruments.qcodes_drivers.spi_rack_modules.**D5aModule**(*parent*, *name: str*, *address: int*, *reset_voltages: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Bases: *D5aModule*, InstrumentChannel

QCoDeS style instrument channel driver for the D5a SPI module.

NUMBER_OF_DACS = 16

__init__(*parent*, *name: str*, *address: int*, *reset_voltages: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Instantiates the driver object. This is the object that should be instantiated by the *add_spi_module()* function.

Parameters

- **parent** – Reference to the *SpiRack* parent object. This is handled by the *add_spi_module()* function.
- **name** (*str*) – Name given to the InstrumentChannel.
- **address** (*int*) – Module number set on the hardware.
- **reset_voltages** (*bool*) – If True, then reset all voltages to zero and change the span to *range_max_bi*.
- **dac_names** (*Optional[List[str]]*) – List of all the names to use for the dac channels. If no list is given or is None, the default name "dac{i}" is used for the i-th dac channel.
- **is_dummy** (*bool*) – If true, do not connect to physical hardware, but use a dummy module.

Raises

ValueError – Length of the dac names list does not match the number of dacs.

D5a QCoDeS parameters

D5a.voltage

Sets the output voltage of the dac channel. Depending on the value of ramping_enabled, the output value is either achieved through slowly ramping, or instantaneously set.

Properties

- **unit**: V
- **value**: Numbers -8.0<=v<=8.0

D5a.span

Sets the max range of the DACs. Possible values: 'range_4V_uni': 0 - 4 V, 'range_8V_uni': 0 - 8 V (only if non-standard 12 V power supply is present), 'range_4V_bi': -4 - 4 V, 'range_8V_bi': -8 - 8 V (only if non-standard 12 V power supply is present), 'range_2V_bi': -2 - 2 V.

Properties

- **value**: Enum: {'range_8V_uni', 'range_4V_bi', 'range_8V_bi', 'range_2V_bi', 'range_4V_uni'}

D5a.ramp_rate

Limits the rate at which currents can be changed. The size of of steps is still limited by *ramp_max_step*.

Properties

- **unit**: V/s
- **value**: on

D5a.ramp_max_step

Sets the maximum step size for voltage ramping. The rate at which it ramps is set by *ramp_rate*.

Properties

- **unit**: V
- **value**: on

D5a.ramping_enabled

Turns ramping on or off. Toggling *ramping_enabled* changed the behavior of the setter for the *current* parameter. If enabled, ramping is done at a rate set by *ramp_rate* and in steps specified by *ramp_max_step*.

Properties

- **value**: Boolean

D5a.is_ramping

Returns whether the dac is currently in the process of ramping.

Properties

- **value**: on

`D5a.stepsize`

Returns the smallest current step allowed by the dac for the current settings.

Properties

- **unit**: V
- **value**: on

`D5a.dac_channel`

Returns the dac number of this channel.

Properties

- **value**: on

1.28.6 D5a native interface

class qblox_instruments.native.spi_rack_modules.**D5aModule**(*parent*, *name: str*, *address: int*, *reset_voltages: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Bases: *SpiModuleBase*

Native driver for the D5a SPI module.

NUMBER_OF_DACS = 16

__init__(*parent*, *name: str*, *address: int*, *reset_voltages: bool = False*, *dac_names: Optional[List[str]] = None*, *is_dummy: bool = False*)

Instantiates the driver object. This is the object that should be instantiated by the *add_spi_module()* function.

Parameters

- **parent** – Reference to the *SpiRack* parent object. This is handled by the *add_spi_module()* function.
- **name** (*str*) – Name given to the InstrumentChannel.
- **address** (*int*) – Module number set on the hardware.
- **reset_voltages** (*bool*) – If True, then reset all voltages to zero and change the span to *range_max_bi*.
- **dac_names** (*Optional[List[str]]*) – List of all the names to use for the dac channels. If no list is given or is None, the default name "dac{i}" is used for the i-th dac channel.
- **is_dummy** (*bool*) – If true, do not connect to physical hardware, but use dummy module.

Raises

ValueError – Length of the dac names list does not match the number of dacs.

set_dacs_zero() → None

Sets all voltages of all outputs to 0.

1.29 QCM-QRM

The QCM/QRM module is responsible for adding the parameters to a single Pulsar or a single module of a Cluster using *add_qcodes_params()* and *QcmQrm* respectively. The resulting parameters are module type dependend and are described at the bottom of this page. Each module also adds the parameters for 6 sequencers, the parameters of which are described in *Sequencer*.

class qblox_instruments.qcodes_drivers.qcm_qrm.**QcmQrm**(*parent: Instrument*, *name: str*, *slot_idx: int*)

Bases: InstrumentChannel

This class represents a QCM/QRM module. It combines all module specific parameters and functions into a single QCoDes InstrumentChannel.

__init__(*parent: Instrument*, *name: str*, *slot_idx: int*)

Creates a QCM/QRM module class and adds all relevant parameters for the module.

Parameters

- **parent** (*Instrument*) – The QCoDeS class to which this module belongs.
- **name** (*str*) – Name of this module channel
- **slot_idx** (*int*) – The index of this module in the parent instrument, representing which module is controlled by this class.

property slot_idx: int

Get slot index.

Returns
Slot index

Return type
int

property module_type: *InstrumentType*

Get module type (e.g. QRM, QCM).

Returns
Module type

Return type
InstrumentType

Raises
KeyError – Module is not available.

property is_qcm_type: bool

Return if module is of type QCM.

Returns
True if module is of type QCM.

Return type
bool

Raises
KeyError – Module is not available.

property is_qrm_type: bool

Return if module is of type QRM.

Returns
True if module is of type QRM.

Return type
bool

Raises
KeyError – Module is not available.

property is_rf_type: bool

Return if module is of type QCM-RF or QRM-RF.

Returns
True if module is of type QCM-RF or QRM-RF.

Return type
bool

Raises
KeyError – Module is not available.

property sequencers: List

Get list of sequencers submodules.

Returns
List of sequencer submodules.

Return type
list

qblox_instruments.qcodes_drivers.qcm_qrm.**add_qcodes_params**(*parent: Union[Instrument, QcmQrm]*, *num_seq: int*) → None

Add all QCoDeS parameters for a single QCM/QRM module.

Parameters
- **parent** (*Union[Instrument, QcmQrm]*) – Parent object to which the parameters need to be added.
- **num_seq** (*int*) – Number of sequencers to add as submodules.

qblox_instruments.qcodes_drivers.qcm_qrm.**invalidate_qcodes_parameter_cache**(*parent: Union[Instrument, QcmQrm]*) → None

Marks the cache of all QCoDeS parameters in the module as invalid, including in any sequencer submodules the module might have.

Parameters
parent (*Union[Instrument, QcmQrm]*) – Parent module object for which to invalidate the QCoDeS parameters.

qblox_instruments.qcodes_drivers.qcm_qrm.**get_item**(*parent: Union[Instrument, QcmQrm]*, *key: str*) → Union[InstrumentChannel, Parameter, Callable[[Any], Any]]

Get submodule or parameter using string based lookup.

Parameters
- **parent** (*Union[Instrument, QcmQrm]*) – Parent module object to search.
- **key** (*str*) – submodule, parameter or function to retrieve.

Returns
Submodule, parameter or function.

Return type
Union[InstrumentChannel, Parameter, Callable[[Any], Any]]

Raises
KeyError – Submodule, parameter or function does not exist.

1.29.1 Pulsar QCoDeS parameters

QCoDeS parameters generated by *add_qcodes_params()*.

Pulsar QCM parameters

(Pulsar QCM) Pulsar.IDN

Please see QCoDeS for a description.

Properties

- **value**: Anything

(Pulsar QCM) Pulsar.reference_source

Sets/gets reference source (‘internal’ = internal 10 MHz, ‘external’ = external 10 MHz).

Properties

- **value**: Enum: {‘internal’, ‘external’}

(Pulsar QCM) Pulsar.out0_offset

Sets/gets output 0 offset

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

(Pulsar QCM) Pulsar.out1_offset

Sets/gets output 1 offset.

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

(Pulsar QCM) Pulsar.out2_offset

Sets/gets output 2 offset.

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

(Pulsar QCM) Pulsar.out3_offset

Sets/gets output 3 offset.

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

Pulsar QRM parameters

(Pulsar QRM) Pulsar.IDN

Please see QCoDeS for a description.

Properties

- **value**: Anything

(Pulsar QRM) Pulsar.reference_source

Sets/gets reference source ('internal' = internal 10 MHz, 'external' = external 10 MHz).

Properties

- **value**: Enum: {'internal', 'external'}

(Pulsar QRM) Pulsar.in0_gain

Sets/gets input 0 gain in a range of -6dB to 26dB with a resolution of 1dB per step.

Properties

- **unit**: dB
- **value**: Numbers -6<=v<=26

(Pulsar QRM) Pulsar.in1_gain

Sets/gets input 1 gain in a range of -6dB to 26dB with a resolution of 1dB per step.

Properties

- **unit**: dB
- **value**: Numbers -6<=v<=26

(Pulsar QRM) Pulsar.out0_offset

Sets/gets output 0 offset

Properties

- **unit**: V
- **value**: Numbers -0.5<=v<=0.5

(Pulsar QRM) Pulsar.out1_offset

Sets/gets output 1 offset.

Properties

- **unit**: V
- **value**: Numbers -0.5<=v<=0.5

(Pulsar QRM) Pulsar.scope_acq_trigger_mode_path0

Sets/gets scope acquisition trigger mode for input path 0 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

(Pulsar QRM) Pulsar.scope_acq_trigger_mode_path1

Sets/gets scope acquisition trigger mode for input path 1 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

(Pulsar QRM) Pulsar.scope_acq_trigger_level_path0

Sets/gets scope acquisition trigger level when using input level trigger mode for input path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QRM) Pulsar.scope_acq_trigger_level_path1

Sets/gets scope acquisition trigger level when using input level trigger mode for input path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QRM) Pulsar.scope_acq_sequencer_select

Sets/gets sequencer select that specifies which sequencer triggers the scope acquisition when using sequencer trigger mode.

Properties

- **value**: Numbers 0<=v<=6

(Pulsar QRM) Pulsar.scope_acq_avg_mode_en_path0

Sets/gets scope acquisition averaging mode enable for input path 0.

Properties

- **value**: Boolean

(Pulsar QRM) Pulsar.scope_acq_avg_mode_en_path1

Sets/gets scope acquisition averaging mode enable for input path 0.

Properties

- **value**: Boolean

1.29.2 Cluster QCoDeS parameters

QCoDeS parameters generated by *QcmQrm*.

Cluster QCM parameters

(Cluster QCM) module.present

Sets/gets module present status for slot {} in the Cluster.

Properties

- **value**: Boolean

(Cluster QCM) module.out0_offset

Sets/gets output 0 offset

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

(Cluster QCM) module.out1_offset

Sets/gets output 1 offset.

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

(Cluster QCM) module.out2_offset

Sets/gets output 2 offset.

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

(Cluster QCM) module.out3_offset

Sets/gets output 3 offset.

Properties

- **unit**: V
- **value**: Numbers -2.5<=v<=2.5

Cluster QCM-RF parameters

(Cluster QCM-RF) module.present

Sets/gets module present status for slot {} in the Cluster.

Properties

- **value**: Boolean

(Cluster QCM-RF) module.out0_lo_freq

Sets/gets the local oscillator frequency for output 0.

Properties

- **unit**: Hz
- **value**: Numbers 2000000000.0<=v<=18000000000.0

(Cluster QCM-RF) module.out1_lo_freq

Sets/gets the local oscillator frequency for output 1.

Properties

- **unit**: Hz
- **value**: Numbers 2000000000.0<=v<=18000000000.0

(Cluster QCM-RF) module.out0_lo_en

Sets/gets the local oscillator enable for output 0.

Properties

- **value**: Boolean

(Cluster QCM-RF) module.out1_lo_en

Sets/gets the local oscillator enable for output 1.

Properties

- **value**: Boolean

(Cluster QCM-RF) module.out0_att

Sets/gets output attenuation in a range of 0dB to 60dB with a resolution of 2dB per step.

Properties

- **unit**: dB
- **value**: Ints 0<=v<=60, Multiples of 2

(Cluster QCM-RF) module.out1_att

Sets/gets output attenuation in a range of 0dB to 60dB with a resolution of 2dB per step.

Properties

- **unit**: dB
- **value**: Ints 0<=v<=60, Multiples of 2

(Cluster QCM-RF) module.out0_offset_path0

Sets/gets output 0 offset for path 0.

Properties

- **unit**: mV
- **value**: Numbers -84.0<=v<=73.0

(Cluster QCM-RF) module.out0_offset_path1

Sets/gets output 0 offset for path 1.

Properties

- **unit**: mV
- **value**: Numbers -84.0<=v<=73.0

(Cluster QCM-RF) module.out1_offset_path0

Sets/gets output 1 offset for path 0.

Properties

- **unit**: mV
- **value**: Numbers -84.0<=v<=73.0

(Cluster QCM-RF) module.out1_offset_path1

Sets/gets output 1 offset for path 1.

Properties

- **unit**: mV
- **value**: Numbers -84.0<=v<=73.0

Cluster QRM parameters

(Cluster QRM) module.present

Sets/gets module present status for slot {} in the Cluster.

Properties

- **value**: Boolean

(Cluster QRM) module.in0_gain

Sets/gets input 0 gain in a range of -6dB to 26dB with a resolution of 1dB per step.

Properties

- **unit**: dB
- **value**: Numbers -6<=v<=26

(Cluster QRM) module.in1_gain

Sets/gets input 1 gain in a range of -6dB to 26dB with a resolution of 1dB per step.

Properties

- **unit**: dB
- **value**: Numbers -6<=v<=26

(Cluster QRM) module.out0_offset

Sets/gets output 0 offset

Properties

- **unit**: V
- **value**: Numbers -0.5<=v<=0.5

(Cluster QRM) module.out1_offset

Sets/gets output 1 offset.

Properties

- **unit**: V
- **value**: Numbers -0.5<=v<=0.5

(Cluster QRM) module.scope_acq_trigger_mode_path0

Sets/gets scope acquisition trigger mode for input path 0 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

(Cluster QRM) module.scope_acq_trigger_mode_path1

Sets/gets scope acquisition trigger mode for input path 1 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

(Cluster QRM) module.scope_acq_trigger_level_path0

Sets/gets scope acquisition trigger level when using input level trigger mode for input path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM) module.scope_acq_trigger_level_path1

Sets/gets scope acquisition trigger level when using input level trigger mode for input path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM) module.scope_acq_sequencer_select

Sets/gets sequencer select that specifies which sequencer triggers the scope acquisition when using sequencer trigger mode.

Properties

- **value**: Numbers 0<=v<=6

(Cluster QRM) module.scope_acq_avg_mode_en_path0

Sets/gets scope acquisition averaging mode enable for input path 0.

Properties

- **value**: Boolean

(Cluster QRM) module.scope_acq_avg_mode_en_path1

Sets/gets scope acquisition averaging mode enable for input path 0.

Properties

- **value**: Boolean

Cluster QRM-RF parameters

(Cluster QRM-RF) module.present

Sets/gets module present status for slot {} in the Cluster.

Properties

- **value**: Boolean

(Cluster QRM-RF) module.out0_in0_lo_freq

Sets/gets the local oscillator frequency for output 0 and input 0.

Properties

- **unit**: Hz
- **value**: Numbers 2000000000.0<=v<=18000000000.0

(Cluster QRM-RF) module.out0_in0_lo_en

Sets/gets the local oscillator enable for output 0 and input 0.

Properties

- **value**: Boolean

(Cluster QRM-RF) module.in0_att

Sets/gets input attenuation in a range of 0dB to 30dB with a resolution of 2dB per step.

Properties

- **unit**: dB
- **value**: Ints 0<=v<=30, Multiples of 2

(Cluster QRM-RF) module.out0_att

Sets/gets output attenuation in a range of 0dB to 60dB with a resolution of 2dB per step.

Properties

- **unit**: dB
- **value**: Ints 0<=v<=60, Multiples of 2

(Cluster QRM-RF) module.out0_offset_path0

Sets/gets output 0 offset for path 0.

Properties

- **unit**: mV
- **value**: Numbers -84.0<=v<=73.0

(Cluster QRM-RF) module.out0_offset_path1

Sets/gets output 0 offset for path 1.

Properties

- **unit**: mV
- **value**: Numbers -84.0<=v<=73.0

(Cluster QRM-RF) module.scope_acq_trigger_mode_path0

Sets/gets scope acquisition trigger mode for input path 0 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

(Cluster QRM-RF) module.scope_acq_trigger_mode_path1

Sets/gets scope acquisition trigger mode for input path 1 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

(Cluster QRM-RF) module.scope_acq_trigger_level_path0

Sets/gets scope acquisition trigger level when using input level trigger mode for input path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM-RF) module.scope_acq_trigger_level_path1

Sets/gets scope acquisition trigger level when using input level trigger mode for input path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM-RF) module.scope_acq_sequencer_select

Sets/gets sequencer select that specifies which sequencer triggers the scope acquisition when using sequencer trigger mode.

Properties

- **value**: Numbers 0<=v<=6

(Cluster QRM-RF) module.scope_acq_avg_mode_en_path0

Sets/gets scope acquisition averaging mode enable for input path 0.

Properties

- **value**: Boolean

(Cluster QRM-RF) module.scope_acq_avg_mode_en_path1

Sets/gets scope acquisition averaging mode enable for input path 0.

Properties

- **value**: Boolean

1.30 Sequencer

The sequencer module is responsible for adding the parameters of a single sequencer within a module.

class qblox_instruments.qcodes_drivers.sequencer.**Sequencer**(*parent: Union[Instrument, InstrumentChannel]*, *name: str*, *seq_idx: int*)

Bases: InstrumentChannel

This class represents a single sequencer. It combines all sequencer specific parameters and functions into a single QCoDes InstrumentChannel.

__init__(*parent: Union[Instrument, InstrumentChannel]*, *name: str*, *seq_idx: int*)

Creates a sequencer class and adds all relevant parameters for the sequencer.

Parameters

- **parent** (*Union[Instrument, InstrumentChannel]*) – The QCoDeS class to which this sequencer belongs.
- **name** (*str*) – Name of this sequencer channel
- **seq_idx** (*int*) – The index of this sequencer in the parent instrument, representing which sequencer is controlled by this class.

property seq_idx: int

Get sequencer index.

Returns

Sequencer index

Return type

int

1.30.1 Pulsar QCoDeS parameters

QCoDeS parameters generated by *Sequencer*.

Pulsar QCM parameters

(Pulsar QCM) sequencer.channel_map_path0_out0_en

Sets/gets sequencer channel map enable of path 0 to output 0.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.channel_map_path1_out1_en

Sets/gets sequencer channel map enable of path 1 to output 1.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.channel_map_path0_out2_en

Sets/gets sequencer channel map enable of path 0 to output 2.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.channel_map_path1_out3_en

Sets/gets sequencer channel map enable of path 1 to output 3.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.sync_en

Sets/gets sequencer synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.nco_freq

Sets/gets sequencer NCO frequency in Hz with a resolution of 0.25 Hz. Be aware that the outputs have low-pass filters with a cut-off frequency of 350 MHz

Properties

- **unit**: Hz
- **value**: Numbers -500000000.0<=v<=500000000.0

(Pulsar QCM) sequencer.nco_phase_offs

Sets/gets sequencer NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Pulsar QCM) sequencer.nco_prop_delay_comp

Sets/gets a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delay is applied on top of a default delay of 146 ns.

Properties

- **unit**: ns
- **value**: Numbers -50<=v<=50

(Pulsar QCM) sequencer.nco_prop_delay_comp_en

Sets/gets the enable for a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delays the frequency update as well.

Properties

- **unit**: ns
- **value**: Boolean

(Pulsar QCM) sequencer.marker_ovr_en

Sets/gets sequencer marker override enable.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.marker_ovr_value

Sets/gets sequencer marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

(Pulsar QCM) sequencer.sequence

Sets sequencer's AWG waveforms, acquistion weights, acquisitions and Q1ASM program. Valid input is a string representing the JSON filename or a JSON compatible dictionary.

Properties

- **value**: MultiType: Strings, Dict

(Pulsar QCM) sequencer.cont_mode_en_awg_path0

Sets/gets sequencer continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.cont_mode_en_awg_path1

Sets/gets sequencer continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

(Pulsar QCM) sequencer.cont_mode_waveform_idx_awg_path0

Sets/gets sequencer continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

(Pulsar QCM) sequencer.cont_mode_waveform_idx_awg_path1

Sets/gets sequencer continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

(Pulsar QCM) sequencer.upsample_rate_awg_path0

Sets/gets sequencer upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

(Pulsar QCM) sequencer.upsample_rate_awg_path1

Sets/gets sequencer upsample rate for AWG path 1.

Properties

- **value**: Numbers 0<=v<=65535

(Pulsar QCM) sequencer.gain_awg_path0

Sets/gets sequencer gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QCM) sequencer.gain_awg_path1

Sets/gets sequencer gain for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QCM) sequencer.offset_awg_path0

Sets/gets sequencer offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QCM) sequencer.offset_awg_path1

Sets/gets sequencer offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QCM) sequencer.mixer_corr_phase_offset_degree

Sets/gets sequencer mixer phase imbalance correction for AWG; applied to AWG path 1 relative to AWG path 0 and measured in degrees

Properties

- **value**: Numbers -45.0<=v<=45.0

(Pulsar QCM) sequencer.mixer_corr_gain_ratio

Sets/gets sequencer mixer gain imbalance correction for AWG; equal to AWG path 1 amplitude divided by AWG path 0 amplitude.

Properties

- **value**: Numbers 0.5<=v<=2.0

(Pulsar QCM) sequencer.mod_en_awg

Sets/gets sequencer modulation enable for AWG.

Properties

- **value**: Boolean

Pulsar QRM parameters

(Pulsar QRM) sequencer.channel_map_path0_out0_en

Sets/gets sequencer channel map enable of path 0 to output 0.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.channel_map_path1_out1_en

Sets/gets sequencer channel map enable of path 1 to output 1.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.sync_en

Sets/gets sequencer synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.nco_freq

Sets/gets sequencer NCO frequency in Hz with a resolution of 0.25 Hz. Be aware that the outputs have low-pass filters with a cut-off frequency of 350 MHz

Properties

- **unit**: Hz
- **value**: Numbers -500000000.0<=v<=500000000.0

(Pulsar QRM) sequencer.nco_phase_offs

Sets/gets sequencer NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Pulsar QRM) sequencer.nco_prop_delay_comp

Sets/gets a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delay is applied on top of a default delay of 146 ns.

Properties

- **unit**: ns
- **value**: Numbers -50<=v<=50

(Pulsar QRM) sequencer.nco_prop_delay_comp_en

Sets/gets the enable for a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delays the frequency update as well.

Properties

- **unit**: ns
- **value**: Boolean

(Pulsar QRM) sequencer.marker_ovr_en

Sets/gets sequencer marker override enable.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.marker_ovr_value

Sets/gets sequencer marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

(Pulsar QRM) sequencer.sequence

Sets sequencer's AWG waveforms, acquistion weights, acquisitions and Q1ASM program. Valid input is a string representing the JSON filename or a JSON compatible dictionary.

Properties

- **value**: MultiType: Strings, Dict

(Pulsar QRM) sequencer.cont_mode_en_awg_path0

Sets/gets sequencer continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.cont_mode_en_awg_path1

Sets/gets sequencer continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.cont_mode_waveform_idx_awg_path0

Sets/gets sequencer continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

(Pulsar QRM) sequencer.cont_mode_waveform_idx_awg_path1

Sets/gets sequencer continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

(Pulsar QRM) sequencer.upsample_rate_awg_path0

Sets/gets sequencer upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

(Pulsar QRM) sequencer.upsample_rate_awg_path1

Sets/gets sequencer upsample rate for AWG path 1.

Properties

- **value**: Numbers 0<=v<=65535

(Pulsar QRM) sequencer.gain_awg_path0

Sets/gets sequencer gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QRM) sequencer.gain_awg_path1

Sets/gets sequencer gain for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QRM) sequencer.offset_awg_path0

Sets/gets sequencer offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QRM) sequencer.offset_awg_path1

Sets/gets sequencer offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QRM) sequencer.mixer_corr_phase_offset_degree

Sets/gets sequencer mixer phase imbalance correction for AWG; applied to AWG path 1 relative to AWG path 0 and measured in degrees

Properties

- **value**: Numbers -45.0<=v<=45.0

(Pulsar QRM) sequencer.mixer_corr_gain_ratio

Sets/gets sequencer mixer gain imbalance correction for AWG; equal to AWG path 1 amplitude divided by AWG path 0 amplitude.

Properties

- **value**: Numbers 0.5<=v<=2.0

(Pulsar QRM) sequencer.mod_en_awg

Sets/gets sequencer modulation enable for AWG.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.demod_en_acq

Sets/gets sequencer demodulation enable for acquisition.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.integration_length_acq

Sets/gets sequencer integration length in number of samples for non-weighed acquisitions on paths 0 and 1. Must be a multiple of 4

Properties

- **value**: Ints 4<=v<=16777212, Multiples of 4

(Pulsar QRM) sequencer.phase_rotation_acq

Sets/gets sequencer integration result phase rotation in degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Pulsar QRM) sequencer.discretization_threshold_acq

Sets/gets sequencer discretization threshold for discretizing the phase rotation result. Discretization is done by comparing the threshold to the rotated integration result of path 0. This comparison is applied before normalization (i.e. division) of the rotated value with the integration length and therefore the threshold needs to be compensated (i.e. multiplied) with this length for the discretization to function properly.

Properties

- **value**: Numbers -16777212.0<=v<=16777212.0

(Pulsar QRM) sequencer.ttl_acq_auto_bin_incr_en

Sets/gets whether the bin index is automatically incremented when acquiring multiple triggers. Disabling the TTL trigger acquisition path resets the bin index.

Properties

- **value**: Boolean

(Pulsar QRM) sequencer.ttl_acq_threshold

Sets/gets the threshold value with which to compare the input ADC values of the selected input path.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Pulsar QRM) sequencer.ttl_acq_input_select

Sets/gets the input used to compare against the threshold value in the TTL trigger acquisition path.

Properties

- **value**: Numbers 0<=v<=1

1.30.2 Cluster QCoDeS parameters

QCoDeS parameters generated by *Sequencer*.

Cluster QCM parameters

(Cluster QCM) sequencer.channel_map_path0_out0_en

Sets/gets sequencer channel map enable of path 0 to output 0.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.channel_map_path1_out1_en

Sets/gets sequencer channel map enable of path 1 to output 1.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.channel_map_path0_out2_en

Sets/gets sequencer channel map enable of path 0 to output 2.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.channel_map_path1_out3_en

Sets/gets sequencer channel map enable of path 1 to output 3.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.sync_en

Sets/gets sequencer synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.nco_freq

Sets/gets sequencer NCO frequency in Hz with a resolution of 0.25 Hz. Be aware that the outputs have low-pass filters with a cut-off frequency of 350 MHz

Properties

- **unit**: Hz
- **value**: Numbers -500000000.0<=v<=500000000.0

(Cluster QCM) sequencer.nco_phase_offs

Sets/gets sequencer NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Cluster QCM) sequencer.nco_prop_delay_comp

Sets/gets a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delay is applied on top of a default delay of 146 ns.

Properties

- **unit**: ns
- **value**: Numbers -50<=v<=50

(Cluster QCM) sequencer.nco_prop_delay_comp_en

Sets/gets the enable for a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delays the frequency update as well.

Properties

- **unit**: ns
- **value**: Boolean

(Cluster QCM) sequencer.marker_ovr_en

Sets/gets sequencer marker override enable.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.marker_ovr_value

Sets/gets sequencer marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

(Cluster QCM) sequencer.sequence

Sets sequencer's AWG waveforms, acquistion weights, acquisitions and Q1ASM program. Valid input is a string representing the JSON filename or a JSON compatible dictionary.

Properties

- **value**: MultiType: Strings, Dict

(Cluster QCM) sequencer.cont_mode_en_awg_path0

Sets/gets sequencer continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.cont_mode_en_awg_path1

Sets/gets sequencer continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

(Cluster QCM) sequencer.cont_mode_waveform_idx_awg_path0

Sets/gets sequencer continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QCM) sequencer.cont_mode_waveform_idx_awg_path1

Sets/gets sequencer continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QCM) sequencer.upsample_rate_awg_path0

Sets/gets sequencer upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QCM) sequencer.upsample_rate_awg_path1

Sets/gets sequencer upsample rate for AWG path 1.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QCM) sequencer.gain_awg_path0

Sets/gets sequencer gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM) sequencer.gain_awg_path1

Sets/gets sequencer gain for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM) sequencer.offset_awg_path0

Sets/gets sequencer offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM) sequencer.offset_awg_path1

Sets/gets sequencer offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM) sequencer.mixer_corr_phase_offset_degree

Sets/gets sequencer mixer phase imbalance correction for AWG; applied to AWG path 1 relative to AWG path 0 and measured in degrees

Properties

- **value**: Numbers -45.0<=v<=45.0

(Cluster QCM) sequencer.mixer_corr_gain_ratio

Sets/gets sequencer mixer gain imbalance correction for AWG; equal to AWG path 1 amplitude divided by AWG path 0 amplitude.

Properties

- **value**: Numbers 0.5<=v<=2.0

(Cluster QCM) sequencer.mod_en_awg

Sets/gets sequencer modulation enable for AWG.

Properties

- **value**: Boolean

Cluster QCM-RF parameters

(Cluster QCM-RF) sequencer.channel_map_path0_out0_en

Sets/gets sequencer channel map enable of path 0 to output 0.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.channel_map_path1_out1_en

Sets/gets sequencer channel map enable of path 1 to output 1.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.channel_map_path0_out2_en

Sets/gets sequencer channel map enable of path 0 to output 2.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.channel_map_path1_out3_en

Sets/gets sequencer channel map enable of path 1 to output 3.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.sync_en

Sets/gets sequencer synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.nco_freq

Sets/gets sequencer NCO frequency in Hz with a resolution of 0.25 Hz. Be aware that the outputs have low-pass filters with a cut-off frequency of 300 MHz

Properties

- **unit**: Hz
- **value**: Numbers -500000000.0<=v<=500000000.0

(Cluster QCM-RF) sequencer.nco_phase_offs

Sets/gets sequencer NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Cluster QCM-RF) sequencer.nco_prop_delay_comp

Sets/gets a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delay is applied on top of a default delay of 146 ns.

Properties

- **unit**: ns
- **value**: Numbers -50<=v<=50

(Cluster QCM-RF) sequencer.nco_prop_delay_comp_en

Sets/gets the enable for a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delays the frequency update as well.

Properties

- **unit**: ns
- **value**: Boolean

(Cluster QCM-RF) sequencer.marker_ovr_en

Sets/gets sequencer marker override enable.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.marker_ovr_value

Sets/gets sequencer marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

(Cluster QCM-RF) sequencer.sequence

Sets sequencer's AWG waveforms, acquistion weights, acquisitions and Q1ASM program. Valid input is a string representing the JSON filename or a JSON compatible dictionary.

Properties

- **value**: MultiType: Strings, Dict

(Cluster QCM-RF) sequencer.cont_mode_en_awg_path0

Sets/gets sequencer continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.cont_mode_en_awg_path1

Sets/gets sequencer continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

(Cluster QCM-RF) sequencer.cont_mode_waveform_idx_awg_path0

Sets/gets sequencer continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QCM-RF) sequencer.cont_mode_waveform_idx_awg_path1

Sets/gets sequencer continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QCM-RF) sequencer.upsample_rate_awg_path0

Sets/gets sequencer upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QCM-RF) sequencer.upsample_rate_awg_path1

Sets/gets sequencer upsample rate for AWG path 1.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QCM-RF) sequencer.gain_awg_path0

Sets/gets sequencer gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM-RF) sequencer.gain_awg_path1

Sets/gets sequencer gain for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM-RF) sequencer.offset_awg_path0

Sets/gets sequencer offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM-RF) sequencer.offset_awg_path1

Sets/gets sequencer offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QCM-RF) sequencer.mixer_corr_phase_offset_degree

Sets/gets sequencer mixer phase imbalance correction for AWG; applied to AWG path 1 relative to AWG path 0 and measured in degrees

Properties

- **value**: Numbers -45.0<=v<=45.0

(Cluster QCM-RF) sequencer.mixer_corr_gain_ratio

Sets/gets sequencer mixer gain imbalance correction for AWG; equal to AWG path 1 amplitude divided by AWG path 0 amplitude.

Properties

- **value**: Numbers 0.5<=v<=2.0

(Cluster QCM-RF) sequencer.mod_en_awg

Sets/gets sequencer modulation enable for AWG.

Properties

- **value**: Boolean

Cluster QRM parameters

(Cluster QRM) sequencer.channel_map_path0_out0_en

Sets/gets sequencer channel map enable of path 0 to output 0.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.channel_map_path1_out1_en

Sets/gets sequencer channel map enable of path 1 to output 1.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.sync_en

Sets/gets sequencer synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.nco_freq

Sets/gets sequencer NCO frequency in Hz with a resolution of 0.25 Hz. Be aware that the outputs have low-pass filters with a cut-off frequency of 350 MHz

Properties

- **unit**: Hz
- **value**: Numbers -500000000.0<=v<=500000000.0

(Cluster QRM) sequencer.nco_phase_offs

Sets/gets sequencer NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Cluster QRM) sequencer.nco_prop_delay_comp

Sets/gets a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delay is applied on top of a default delay of 146 ns.

Properties

- **unit**: ns
- **value**: Numbers -50<=v<=50

(Cluster QRM) sequencer.nco_prop_delay_comp_en

Sets/gets the enable for a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delays the frequency update as well.

Properties

- **unit**: ns
- **value**: Boolean

(Cluster QRM) sequencer.marker_ovr_en

Sets/gets sequencer marker override enable.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.marker_ovr_value

Sets/gets sequencer marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

(Cluster QRM) sequencer.sequence

Sets sequencer's AWG waveforms, acquistion weights, acquisitions and Q1ASM program. Valid input is a string representing the JSON filename or a JSON compatible dictionary.

Properties

- **value**: MultiType: Strings, Dict

(Cluster QRM) sequencer.cont_mode_en_awg_path0

Sets/gets sequencer continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.cont_mode_en_awg_path1

Sets/gets sequencer continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.cont_mode_waveform_idx_awg_path0

Sets/gets sequencer continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QRM) sequencer.cont_mode_waveform_idx_awg_path1

Sets/gets sequencer continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QRM) sequencer.upsample_rate_awg_path0

Sets/gets sequencer upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QRM) sequencer.upsample_rate_awg_path1

Sets/gets sequencer upsample rate for AWG path 1.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QRM) sequencer.gain_awg_path0

Sets/gets sequencer gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM) sequencer.gain_awg_path1

Sets/gets sequencer gain for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM) sequencer.offset_awg_path0

Sets/gets sequencer offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM) sequencer.offset_awg_path1

Sets/gets sequencer offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM) sequencer.mixer_corr_phase_offset_degree

Sets/gets sequencer mixer phase imbalance correction for AWG; applied to AWG path 1 relative to AWG path 0 and measured in degrees

Properties

- **value**: Numbers -45.0<=v<=45.0

(Cluster QRM) sequencer.mixer_corr_gain_ratio

Sets/gets sequencer mixer gain imbalance correction for AWG; equal to AWG path 1 amplitude divided by AWG path 0 amplitude.

Properties

- **value**: Numbers 0.5<=v<=2.0

(Cluster QRM) sequencer.mod_en_awg

Sets/gets sequencer modulation enable for AWG.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.demod_en_acq

Sets/gets sequencer demodulation enable for acquisition.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.integration_length_acq

Sets/gets sequencer integration length in number of samples for non-weighed acquisitions on paths 0 and 1. Must be a multiple of 4

Properties

- **value**: Ints 4<=v<=16777212, Multiples of 4

(Cluster QRM) sequencer.phase_rotation_acq

Sets/gets sequencer integration result phase rotation in degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Cluster QRM) sequencer.discretization_threshold_acq

Sets/gets sequencer discretization threshold for discretizing the phase rotation result. Discretization is done by comparing the threshold to the rotated integration result of path 0. This comparison is applied before normalization (i.e. division) of the rotated value with the integration length and therefore the threshold needs to be compensated (i.e. multiplied) with this length for the discretization to function properly.

Properties

- **value**: Numbers -16777212.0<=v<=16777212.0

(Cluster QRM) sequencer.ttl_acq_auto_bin_incr_en

Sets/gets whether the bin index is automatically incremented when acquiring multiple triggers. Disabling the TTL trigger acquisition path resets the bin index.

Properties

- **value**: Boolean

(Cluster QRM) sequencer.ttl_acq_threshold

Sets/gets the threshold value with which to compare the input ADC values of the selected input path.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM) sequencer.ttl_acq_input_select

Sets/gets the input used to compare against the threshold value in the TTL trigger acquisition path.

Properties

- **value**: Numbers 0<=v<=1

Cluster QRM-RF parameters

(Cluster QRM-RF) sequencer.channel_map_path0_out0_en

Sets/gets sequencer channel map enable of path 0 to output 0.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.channel_map_path1_out1_en

Sets/gets sequencer channel map enable of path 1 to output 1.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.sync_en

Sets/gets sequencer synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.nco_freq

Sets/gets sequencer NCO frequency in Hz with a resolution of 0.25 Hz. Be aware that the outputs have low-pass filters with a cut-off frequency of 300 MHz

Properties

- **unit**: Hz
- **value**: Numbers -500000000.0<=v<=500000000.0

(Cluster QRM-RF) sequencer.nco_phase_offs

Sets/gets sequencer NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Cluster QRM-RF) sequencer.nco_prop_delay_comp

Sets/gets a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delay is applied on top of a default delay of 146 ns.

Properties

- **unit**: ns
- **value**: Numbers -50<=v<=50

(Cluster QRM-RF) sequencer.nco_prop_delay_comp_en

Sets/gets the enable for a delay that compensates the NCO phase to the input path with respect to the instrument's combined output and input propagation delay. This delays the frequency update as well.

Properties

- **unit**: ns
- **value**: Boolean

(Cluster QRM-RF) sequencer.marker_ovr_en

Sets/gets sequencer marker override enable.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.marker_ovr_value

Sets/gets sequencer marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

(Cluster QRM-RF) sequencer.sequence

Sets sequencer's AWG waveforms, acquistion weights, acquisitions and Q1ASM program. Valid input is a string representing the JSON filename or a JSON compatible dictionary.

Properties

- **value**: MultiType: Strings, Dict

(Cluster QRM-RF) sequencer.cont_mode_en_awg_path0

Sets/gets sequencer continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.cont_mode_en_awg_path1

Sets/gets sequencer continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.cont_mode_waveform_idx_awg_path0

Sets/gets sequencer continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QRM-RF) sequencer.cont_mode_waveform_idx_awg_path1

Sets/gets sequencer continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

(Cluster QRM-RF) sequencer.upsample_rate_awg_path0

Sets/gets sequencer upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QRM-RF) sequencer.upsample_rate_awg_path1

Sets/gets sequencer upsample rate for AWG path 1.

Properties

- **value**: Numbers 0<=v<=65535

(Cluster QRM-RF) sequencer.gain_awg_path0

Sets/gets sequencer gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM-RF) sequencer.gain_awg_path1

Sets/gets sequencer gain for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM-RF) sequencer.offset_awg_path0

Sets/gets sequencer offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM-RF) sequencer.offset_awg_path1

Sets/gets sequencer offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

(Cluster QRM-RF) sequencer.mixer_corr_phase_offset_degree

Sets/gets sequencer mixer phase imbalance correction for AWG; applied to AWG path 1 relative to AWG path 0 and measured in degrees

Properties

- **value**: Numbers -45.0<=v<=45.0

(Cluster QRM-RF) sequencer.mixer_corr_gain_ratio

Sets/gets sequencer mixer gain imbalance correction for AWG; equal to AWG path 1 amplitude divided by AWG path 0 amplitude.

Properties

- **value**: Numbers 0.5<=v<=2.0

(Cluster QRM-RF) sequencer.mod_en_awg

Sets/gets sequencer modulation enable for AWG.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.demod_en_acq

Sets/gets sequencer demodulation enable for acquisition.

Properties

- **value**: Boolean

(Cluster QRM-RF) sequencer.integration_length_acq

Sets/gets sequencer integration length in number of samples for non-weighed acquisitions on paths 0 and 1. Must be a multiple of 4

Properties

- **value**: Ints 4<=v<=16777212, Multiples of 4

(Cluster QRM-RF) sequencer.phase_rotation_acq

Sets/gets sequencer integration result phase rotation in degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

(Cluster QRM-RF) sequencer.discretization_threshold_acq

Sets/gets sequencer discretization threshold for discretizing the phase rotation result. Discretization is done by comparing the threshold to the rotated integration result of path 0. This comparison is applied before normalization (i.e. division) of the rotated value with the integration length and therefore the threshold needs to be compensated (i.e. multiplied) with this length for the discretization to function properly.

Properties

- **value**: Numbers -16777212.0<=v<=16777212.0

1.31 Configuration management

The following class allows you to manage Pulsar and Cluster instruments from within Python.

Note: There is also a command-line tool for this, that comes with the Qblox instruments, named `qblox-cfg`. Run `qblox-cfg help` for its documentation.

---

class qblox_instruments.**ConfigurationManager**(*identifier: Union[str, AddressInfo, ConnectionInfo]*)

Bases: `object`

Class that provides configuration management functionality.

__init__(*identifier: Union[str, AddressInfo, ConnectionInfo]*)

Creates a configuration management interface object for the given device. Use close() when you're done with the object, or a `with` clause:

```
with cfg_man(...) as cm:
    # do stuff with cm here
    pass
```

Note: Depending on the software running on the device and the connectivity to the device, not all features may be available. See *get_protocol()*.

Table 2: Feature availability

Feature	scpi	legacy	pnp
set_name	Yes	No	Yes
download_log	Yes	tgz only	No
set_ip_config	Yes	192.168.x.x/24 only	Yes
update	Yes	Yes	No
rollback	Yes	Yes	No
reboot	Yes	Yes	Yes

scpi will be used if available, but:

- devices running an old software version may not support it, in which case legacy will be used; and
- if the embedded software version is new enough but the device IP configuration is incompatible with your network settings, pnp will be used.

Parameters
identifier (*Union[str, AddressInfo, ConnectionInfo]*) – Instrument identifier. See *resolve()*.
Raises
Exception – If we can't connect.

close()

Closes the underlying connection. The object must not be used anymore after this call.
Parameters
None –
Return type
None

get_connection_info() → *ConnectionInfo*

Returns the connection information object.
Parameters
None –
Returns
The connection information object.
Return type
ConnectionInfo

get_protocol() → str

Returns the protocol used for this connection.
Parameters
None –
Returns
The protocol, either "scpi", "legacy", or "pnp".
Return type
str

has_capability(*cmd: str*) → bool

Returns whether our connection type supports the given command.
Parameters
cmd (*str*) – The command name.
Returns
Whether the command is supported. Note that some commands are only

partially supported; in this case, this still reports True.

Return type
bool

download_log(*source: str = 'app', fmt: ~typing.Union[str, int] = 'tail', file: ~typing.Union[str, ~typing.BinaryIO, ~typing.TextIO] = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>, tail: int = 100*) → None

Downloads log data from the device.

Parameters
- **source** (*str*) – The log source. Currently this must be "app" (default), "system", or "cfg_man", or the device will return an error.
- **fmt** (*Union[str, int]*) – File format:
  - If "tail" (default): return the latest <tail> messages in plaintext.
  - If a positive integer: return the latest <fmt> messages in plaintext.
  - If "txt" or zero: return the log file currently in rotation in plaintext.
  - If "tgz" or a negative integer: return all log files in rotation in a tar.gz archive.
- **file** (*Union[str, BinaryIO, TextIO]*) – The file object to write to. For textual formats, the file may be opened in either binary or unicode mode (in the latter case, the contents will be buffered in memory first); for tgz, it must be open in binary mode. If a string, this is a pattern for the filename to write to; the file will be opened internally. The following substitutions are made for convenience:
  - %s -> device serial number;
  - %n -> device name (be careful, device names are not necessarily valid filenames);
  - %i -> address of the device we're connecting to (usually IP+port, but will be the serial number when this is a plug & play connection).
- **tail** (*int*) – If fmt is "tail", this specifies the number of lines returned. Unused otherwise.

Return type
None

Raises
- **NotImplementedError** – If the underlying protocol we're connecting with does not support this command.
- **Exception** – If the command failed.

set_name(*name: str*) → None

Renames the device. The name change will be processed immediately.

Parameters
name (*str*) – The new name. Names may not contain newlines, backslashes, or double-quotes.

Return type
None

Raises
- **NotImplementedError** – If the underlying protocol we're connecting with does not support this command.
- **Exception** – If the command failed.

set_ip_config(*config: str*) → None

Reconfigures the IP configuration of the device. Changes will only go into effect after the

device is rebooted.

Note: If this is a plug & play connection, this will also reboot the device immediately.

Parameters

config (*str*) – The IP configuration. Must be one of:

- a static IPv4 address including prefix length;
- a static IPv6 address including prefix length;
- "dhcp" to get an IPv4 address via DHCP;
- a combination of an IPv4 and IPv6 configuration separated by a semicolon.

Return type

None

Raises

- **NotImplementedError** – If the underlying protocol we're connecting with does not support this command.
- **Exception** – If the command failed.

update(*package: Union[str, UpdateFile]*) → None

Updates the device with the given update file. The changes will only go into effect once the device is rebooted.

Parameters

package (*Union[str, UpdateFile]*) – The update package.

Return type

None

Raises

- **NotImplementedError** – If the underlying protocol we're connecting with does not support this command.
- **Exception** – If the command failed.

rollback() → None

Instructs the device to attempt a rollback to the previous version. The changes will only go into effect once the device is rebooted.

Parameters

None –

Return type

None

Raises

- **NotImplementedError** – If the underlying protocol we're connecting with does not support this command.
- **Exception** – If the command failed.

reboot() → None

Instructs the device to reboot.

Parameters

None –

Return type

None

Raises

- **NotImplementedError** – If the underlying protocol we're connecting with does not support this command.
- **Exception** – If the command failed.

static cmd_line(**args: Iterable[str]*) → None

Runs the configuration manager with the given command-line arguments.

Parameters

***args** (*Iterable[str]*) – The command-line arguments.

Return type

None

Raises

RuntimeError – If the command-line tool returns a nonzero exit status.

1.31.1 Supporting classes and functions

class qblox_instruments.cfg_man.probe.**PortInfo**(*protocol*, *version*, *device*)

Bases: tuple

Protocol information for a particular IP/TCP port pair, supporting the legacy configuration manager protocol and SCPI.

device

Device information structure.

Type

DeviceInfo

protocol

The type of host we're connected to, which will be one of the following values.

- "legacy_cfg_man": a legacy configuration manager. The update() function from this file can be used to update the device, but other features will not work (use an older configuration manager if you need them).
- "legacy_app": a legacy application without configuration management commands.
- "cfg_man": the configuration manager application via SCPI. This application can manage the device at the given host, but only that device.
- "app": the instrument application via SCPI. This means the connection is fully-featured, including the ability to configure modules (if this is a CMM).

Type

str

version

Configuration manager server version.

Type

str

qblox_instruments.cfg_man.probe.**probe_port**(*host: str*, *port: int*, *version: Tuple[int, int, int]*, *timeout: float = 10.0*) → *PortInfo*

Automatically detects what type of application is listening on the given host and port.

Parameters

- **host** (*str*) – IP address or hostname of the server to connect to.
- **port** (*int*) – Port to connect to.
- **version** (*Tuple[int, int, int]*) – Our client version.
- **timeout** (*float*) – Socket timeout in seconds.

Returns

Information about the protocol.

Return type

PortInfo

Raises

ValueError – If the configuration manager returned something we didn't expect.

class qblox_instruments.cfg_man.probe.**ConnectionInfo**(*identifier*, *protocol*, *address*, *slot_index*, *ip_config*, *server_version*, *client_version*, *device*, *name*, *all_models*)

Bases: tuple

Configuration manager connection information structure.

address

Two-tuple of the IP address and port we need to use to connect for legacy and SCPI connections, or the device serial number for plug & play.

Type

Union[str, tuple[str, int]]

all_models

Set of lowercase model names that will need to be present in the update package. Must include `{device}`, but the cluster management module may for instance request more model names.

Type

set[str]

client_version

Configuration manager client version.

Type

tuple[int, int, int]

device

Device information structure.

Type

DeviceInfo

identifier

Device identifier or address, as passed to *probe_device()*.

Type

Union[str, *AddressInfo*]

ip_config

The IP configuration of the device that will be applied when the device is rebooted, if known. May or may not match the address field, as the configuration may have changed since the instrument was last rebooted, and the local IP address of the instrument may differ from what we're connecting to if NAT is involved.

Type

str

name

Customer-specified name of the instrument, if known.

Type

Optional[str]

protocol

The protocol that must be used to connect. Can be:

- `"legacy"` for the legacy configuration manager protocol;
- `"scpi"` for the SCPI-based configuration manager protocol; or
- `"pnp"` when the device is not accessible due to IP address misconfiguration.

Type

str

server_version

Configuration manager server version, if known. Will be None for plug & play.

Type

Optional[tuple[int, int, int]]

slot_index

None for entire device, slot index if only a single module in the device will be affected.

Type

Optional[int]

qblox_instruments.cfg_man.probe.**represent_address**(*ci:* ConnectionInfo) → str

Returns a human-readable string representation of the address.

Parameters

ci (ConnectionInfo) – The connection information object to represent the address of.

Returns

String representation of the address.

Return type

str

qblox_instruments.cfg_man.probe.**represent_connection**(*ci:* ConnectionInfo) → str

Returns a human-readable string representation of the connection.

Parameters

ci (ConnectionInfo) – The connection information object to represent the connection of.

Returns

String representation of the connection.

Return type

str

qblox_instruments.cfg_man.probe.**represent_device**(*ci:* ConnectionInfo) → str

Returns a human-readable string representation of the device we're connecting to.

Parameters

ci (ConnectionInfo) – The connection information object to represent the device of.

Returns

String representation of the device.

Return type

str

qblox_instruments.cfg_man.probe.**pprint_connection_info**(*ci: ~qblox_instruments.cfg_man.probe.ConnectionInfo, output: ~typing.Callable[[str], None] = <function info>*) → None

Pretty-prints information about a connection information object.

Parameters

- **ci** (ConnectionInfo) – The connection information object to pretty-print.
- **output** (*Callable[[str], None]*) – The function used for printing. Each call represents a line.

qblox_instruments.cfg_man.probe.**probe_device**(*identifier: Union[str,* AddressInfo, ConnectionInfo*], quiet: bool = False*) → *ConnectionInfo*

Automatically detects how to manage the given device.

Parameters
- **identifier** (*str*) – Instrument identifier. See *resolve()* for more information.
- **quiet** (*bool*) – When set, don't log anything.

Returns
The detected connection and device information.

Return type
ConnectionInfo

Raises
RuntimeError – if we failed to connect.

qblox_instruments.cfg_man.probe.**get_device_info**(*identifier: Union[str, AddressInfo, ConnectionInfo]*) → *DeviceInfo*

Fetches a complete *DeviceInfo* structure for the given device.

Parameters
identifier (*Union[str, AddressInfo, ConnectionInfo]*) – Instrument identifier. See *resolve()* for more information.

Returns
The device information.

Return type
DeviceInfo

Raises
RuntimeError – if we failed to connect.

class qblox_instruments.cfg_man.update_file.**UpdateFile**(*fname: str*, *check_version: bool = True*)

Bases: object

Representation of a device update file.

__init__(*fname: str*, *check_version: bool = True*)

Loads an update file.

Parameters
- **fname** (*str*) – The file to load.
- **check_version** (*bool*) – Whether to throw a NotImplementedError if the minimum configuration management client version reported by the update file is newer than our client version.

close()

Cleans up any operating resources that we may have claimed.

needs_confirmation() → Optional[str]

Returns whether the update file requests the user to confirm something before application, and if so, what message should be printed.

Returns
None if there is nothing exceptional about this file, otherwise this is the confirmation message.

Return type
Optional[str]

summarize() → str

Returns a summary of the update file format.

Returns
Update file summary.

Return type
str

pprint(*output: ~typing.Callable[[str], None] = <function info>*) → None

Pretty-prints the update file metadata.

Parameters

output (*Callable[[str], None]*) – The function used for printing. Each call represents a line.

load(*ci:* ConnectionInfo) → BinaryIO

Loads an update file, checking whether the given update file is compatible within the given connection context. Returns a file-like object opened in binary read mode if compatible, or throws a ValueError if there is a problem.

Parameters

ci (ConnectionInfo) – Connection information object retrieved from autoconf(), to verify that the update file is compatible, or to make it compatible, if possible.

Returns

Binary file-like object for the update file. Will at least be opened for reading, and rewound to the start of the file. This may effectively be `open(fname, "rb")`, but could also be a `tempfile.TemporaryFile` to an update file specifically converted to be compatible with the given environment. It is the responsibility of the caller to close the file.

Return type

BinaryIO

Raises

ValueError – If there is a problem with the given update file.

1.32 Plug & Play

The following class allows you to send Qblox Plug & Play commands from within Python.

Note: There is also a command-line tool for this, that comes with the Qblox instruments, named `qblox-pnp`. Run `qblox-pnp help` for its documentation.

---

class qblox_instruments.**PlugAndPlay**

Bases: object

Class that provides device discovery and IP address (re)configuration functionality, for instance to convert customer-controlled device names or serial numbers to IP addresses we can connect to via the usual interfaces.

__init__()

Creates a plug & play interface object. Use close() when you're done with the object, or a `with` clause:

```
with PlugAndPlay() as p:
    # do stuff with p here
    pass
```

Raises

OSError – If creating the network socket fails.

close()

Closes the underlying socket. The object must not be used anymore after this call.

list_devices(*timeout: float = 1.0*) → Dict[str, dict]

Lists all observable devices on the network.

Parameters

timeout (*float*) – Timeout in seconds to wait for responses.

Returns

Mapping from serial number to device description record as returned by the device. If a device returned an invalid structure, its dict will be {}.

Return type

Dict[str, dict]

Raises

OSError – If transmission or reception fails.

print_devices(*timeout: float = 1.0*) → None

Like list_devices(), but prints a user-friendly device list instead of returning a data structure.

Parameters

timeout (*float*) – Timeout in seconds to wait for responses.

Raises

OSError – If transmission or reception fails.

identify(*serial_or_name: str*, *retries: int = 3*, *timeout: float = 1.0*) → None

Visually identifies the device with the given serial number or customer-given name by having it blink its LEDs for a while.

Parameters

- **serial_or_name** (*str*) – Serial number of the device that is to be identified.
- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Raises

- **TypeError** – If serial_or_name is invalid.
- **ValueError** – If serial_or_name is invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.

identify_all(*count: int = 3*) → None

Instructs all devices visible on the network to blink their LEDs.

Parameters

count (*int*) – Number of times to repeat the command packet, to reduce the odds of packet loss being a problem.

Raises

OSError – If transmission or reception fails.

describe(*serial_or_name: str*, *retries: int = 3*, *timeout: float = 1.0*) → dict

Returns the device description structure corresponding to the device with the given serial number or customer-given name.

Parameters

- **serial_or_name** (*str*) – Serial number or customer-given name of the device that is to be queried.
- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Returns
The device description structure.
Return type
dict
Raises
- **TypeError** – If serial_or_name is invalid.
- **ValueError** – If serial_or_name is invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.

get_serial(*name: str*, *retries: int = 3*, *timeout: float = 1.0*) → str

Returns the serial number of the device with the given customer-given name.
Parameters
- **name** (*str*) – Customer-given name of the device that is to be queried.
- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Returns
The serial number of the device.
Return type
str
Raises
- **TypeError** – If name is invalid.
- **ValueError** – If name is invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.
- **KeyError** – If the device response did not contain the requested information.

get_name(*serial: str*, *retries: int = 3*, *timeout: float = 1.0*) → str

Returns the customer-given name of the device with the given serial number.
Parameters
- **serial** (*str*) – Serial number of the device that is to be queried.
- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Returns
The customer-given name of the device.
Return type
str
Raises
- **TypeError** – If serial is invalid.
- **ValueError** – If serial is invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.
- **KeyError** – If the device response did not contain the requested in-

formation.

set_name(*serial_or_name: str*, *new_name: str*, *retries: int = 3*, *timeout: float = 1.0*) → None

Renames the device with the given serial number or name.

Parameters

- **serial_or_name** (*str*) – Serial number or customer-given name of the device that is to be reconfigured.
- **new_name** (*str*) – The new customer-given name for the device. May not contain newlines, double quotes, or backslashes.
- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Raises

- **TypeError** – If serial_or_name or new_name are invalid.
- **ValueError** – If serial_or_name or new_name are invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.

get_ip(*serial_or_name: str*, *retries: int = 3*, *timeout: float = 1.0*) → str

Returns the IP address of the device with the given serial number or customer-given name.

Parameters

- **serial_or_name** (*str*) – Serial number or customer-given name of the device that is to be queried.
- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Returns

The IP address of the device.

Return type

str

Raises

- **TypeError** – If serial_or_name is invalid.
- **ValueError** – If serial_or_name is invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.
- **KeyError** – If the device response did not contain the requested information.

set_ip(*serial_or_name: str*, *ip_address: str*, *retries: int = 3*, *timeout: float = 1.0*) → None

Adjusts the IP address configuration of the device with the given serial number or customer-given name. The device will reboot as a result of this.

Parameters

- **serial_or_name** (*str*) – Serial number or customer-given name of the device that is to be reconfigured.
- **ip_address** (*str*) – The new IP address configuration for the device. This may be an IPv4 address including prefix length (*x.x.x.x/x*), an IPv6 address including prefix length (e.g. *x:x::x:x/x*), a combination thereof separated via a semicolon, or the string *dhcp* to have the device obtain an IPv4 address via DHCP.

- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Raises

- **TypeError** – If serial_or_name is invalid.
- **ValueError** – If serial_or_name is invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.

set_all_dhcp(*count: int = 3*) → None

Instructs all devices on the network to reboot and obtain an IP address via DHCP.

Parameters

count (*int*) – Number of times to repeat the command packet, to reduce the odds of packet loss being a problem.

Raises

OSError – If transmission or reception fails.

reboot(*serial_or_name: str*, *retries: int = 3*, *timeout: float = 1.0*) → None

Reboots the device with the given serial number or customer-given name.

Parameters

- **serial_or_name** (*str*) – Serial number of the device that is to be rebooted.
- **retries** (*int*) – Number of times to retry sending the command, if no response is received.
- **timeout** (*float*) – Timeout in seconds to wait for a response, per retry.

Raises

- **TypeError** – If serial_or_name is invalid.
- **ValueError** – If serial_or_name is invalid.
- **OSError** – If transmission or reception fails.
- **TimeoutError** – If no respone is received from the device.
- **RuntimeError** – If an unexpected response is received from the device.

reboot_all(*count: int = 3*) → None

Instructs all devices on the network to reboot.

Parameters

count (*int*) – Number of times to repeat the command packet, to reduce the odds of packet loss being a problem.

Raises

OSError – If transmission or reception fails.

recover_device() → None

Attempts to recover a device with a severely broken IP configuration, by instructing ALL devices on the network to revert back to 192.168.0.2/24. ONLY RUN THIS COMMAND WHEN YOU ARE ONLY CONNECTED TO A SINGLE DEVICE, OR YOU WILL GET IP ADDRESS CONFLICTS.

Raises

OSError – If recovery packet transmission fails.

static cmd_line(**args: Iterable[str]*) → Any

Runs the plug & play command-line tool with the given arguments.

Parameters
***args** (*Iterable[str]*) – The command-line arguments.
Returns
If the given command logically returns something, it will be returned as a Python value in addition to being printed as a string. Otherwise, None will be returned.
Return type
Any
Raises
RuntimeError – If the command-line tool returns a nonzero exit status.

1.32.1 Supporting classes and functions

qblox_instruments.**resolve**(*identifier: Union[str,* AddressInfo*]*) → *AddressInfo*

Converts a device identifier to an IP address or (if only reachable via plug and play) a serial number. For IP connections, also returns the ports that the device should be listening on.

Parameters
identifier (*Union[str, AddressInfo]*) – If a string, this can be one of the following things:
- "[ip://]<ip-address>[/#]",
- "[pnp://]<device-name>[/#]",
- "[pnp://]<serial-number>[/#]", or
- "[ip://]<hostname>[/#]",

where:
- <ip-address> is a valid IPv4 or IPv6 address for the device to be resolved;
- <device-name> is the customer-specified name of the instrument (programmed into it with qblox-cfg, qblox-pnp, or the set_name() method);
- <serial-number> is the serial number of the instrument;
- <hostname> is a hostname that will resolve to the IP address of the instrument via DNS (some routers support this if the device connects via DHCP; the hostname requested by the device will be its customer-specified name in lowercase, using dashes (-) for sequences of non-alphanumeric characters);
- either ip:// or pnp:// may be prefixed to disambiguate between the various methods of device identification, if necessary; and
- the optional /# suffix may be used for cluster devices to return information for a specific cluster module, where # is the slot index of the module.

The four identification methods will be attempted in the sequence shown.

If an *AddressInfo* object is passed as input, this function simply returns it as-is.
Returns
Address information structure.
Return type
AddressInfo
Raises
- **ValueError** – If the identifier is invalid.
- **RuntimeError** – If we failed to determine what this identifier is.

class qblox_instruments.**AddressInfo**(*protocol*, *address*, *slot_index*, *scpi_port*, *cfg_port*, *zmq_rr_port*, *zmq_ps_port*)

Bases: tuple

Connection information structure. Can be constructed from an instrument identifier via resolve().

`address`

IP address string for IP connections, or the device serial number for plug & play.

Type

str

`cfg_port`

The port number for configuration manager connections.

Type

int

`protocol`

The protocol that must be used to connect. Can be:

- `"ip"` for a normal IP-based connection; or
- `"pnp"` when the device is not accessible due to IP address misconfiguration.

Type

str

`scpi_port`

The port number for SCPI connections.

Type

int

`slot_index`

For clusters, this specifies which specific module to connect to, if any. If None, the whole cluster will be treated as a single instrument. For Pulsar devices, this should always be None.

Type

Optional[int]

`zmq_ps_port`

The port number for Qblox native ZeroMQ/CBOR PUB-SUB connections.

Type

int

`zmq_rr_port`

The port number for Qblox native ZeroMQ/CBOR REQ-REP connections.

Type

int

1.33 Build information

The following functions and classes allow you to programmatically retrieve version and build information for various components.

1.33.1 Information about Qblox instruments

qblox_instruments.**get_build_info**() → *BuildInfo*

Get build information for Qblox Instruments.

Returns

Build information structure for Qblox Instruments.

Return type

BuildInfo

class qblox_instruments.**BuildInfo**(*version: Union[str, Tuple[int, int, int]]*, *build: Union[str, int, datetime]*, *hash: Union[str, int]*, *dirty: Union[str, bool]*)

Bases: object

Class representing build information for a particular component.

__init__(*version: Union[str, Tuple[int, int, int]]*, *build: Union[str, int, datetime]*, *hash: Union[str, int]*, *dirty: Union[str, bool]*)

Makes a build information object.

Parameters

- **version** (*Union[str, Tuple[int, int, int]]*) – Either a canonical version string or a three-tuple of integers.
- **build** (*Union[str, int, datetime],*) – The build timestamp, either as a string formatted like "17/11/2021-19:04:53" (as used in *IDN?), a Unix timestamp in seconds, or a Python datetime object.
- **hash** (*Union[str, int]*) – The git hash of the repository that the build was run from, either as a hex string with at least 8 characters, or as an integer. If 0x is prefixed, the hash may have less than 8 digits, implying zeros in front.
- **dirty** (*Union[str, bool]*) – Whether the git repository was dirty at the time of the build, either as a 0 or 1 string (as in *IDN?) or as the boolean itself.

property version: Tuple[int, int, int]

The version as a three-tuple.

Type

Tuple[int, int, int]

property version_str: str

The version as a string.

Type

str

property build: datetime

The build timestamp as a datetime object.

Type

datetime

property build_str: str

The build time as a string, as formatted for `*IDN?`.

Type

str

property build_iso: str

The build time as a string, formatted using the ISO date format.

Type

str

property build_unix: int

The build time as a unix timestamp in seconds.

Type

int

property hash: int

The git hash as an integer.

Type

int

property hash_str: str

The git hash as a string.

Type

str

property dirty: bool

Whether the repository was dirty during the build.

Type

bool

property dirty_str: str

The dirty flag as a `0` or `1` string (as used for `*IDN?`).

Type

str

classmethod from_idn(*idn: str*, *prefix: str = ''*) → Optional[*BuildInfo*]

Constructs a build information structure from an `*IDN?` string.

Parameters

- **idn** (*str*) – The `*IDN?` string.
- **prefix** (*str*) – The prefix used for each key (currently `fw`, `kmod`, `sw`, or `cfgMan`).

Returns

The build information structure if data is available for the given key, or None if not.

Return type

Optional[*BuildInfo*]

to_idn(*prefix: str = ''*) → str

Formats this build information object in the same way `*IDN?` is formatted.

Parameters

prefix (*str*) – The prefix used for each key (currently `fw`, `kmod`, `sw`, or `cfgMan`).

Returns

The part of the `*IDN?` string for this build information object.

Return type

str

classmethod from_dict(*build_data: dict*) → *BuildInfo*

Constructs a build information structure from a JSON-capable dict, as used in ZeroMQ/CBOR descriptions, plug&play descriptions, update file metadata, and various other places.

Parameters

build_data (*dict*) – Dictionary with (at least) the following keys:

- "version": iterable of three integers representing the version;
- "build": Unix timestamp in seconds representing the build timestamp;
- "hash": the first 8 hex digits of the git hash as an integer; and
- "dirty": boolean dirty flag.

Returns

The build information structure.

Return type

BuildInfo

to_dict() → dict

Formats this build information object as a JSON-capable dict, as used in ZeroMQ/CBOR descriptions, plug&play descriptions, update file metadata, and various other places.

Parameters

None –

Returns

The generated dictionary, having the following keys:

- "version": iterable of three integers representing the version;
- "build": Unix timestamp in seconds representing the build timestamp;
- "hash": the first 8 hex digits of the git hash as an integer; and
- "dirty": boolean dirty flag.

Return type

dict

to_idn_dict() → dict

Formats this build information object as a human-readable JSON-capable dict, as used in get_idn.

Returns

The generated dictionary, having the following keys:

- "version": string representation of the version;
- "build": string representation of timestamp in seconds representing the build timestamp;
- "hash": string representation of the first 8 hex digits of the git hash; and
- "dirty": boolean dirty flag.

Return type

dict

to_tuple() → tuple

Formats this build information object as a tuple for ordering purposes.

Parameters

None –

Returns

A tuple, containing all the information in this structure in a canonical format.

Return type

tuple

1.33.2 Information about an instrument

qblox_instruments.**get_device_info**(*identifier: Union[str, AddressInfo, ConnectionInfo]*) → *DeviceInfo*

Fetches a complete *DeviceInfo* structure for the given device.

Parameters

identifier (*Union[str, AddressInfo, ConnectionInfo]*) – Instrument identifier. See *resolve()* for more information.

Returns

The device information.

Return type

DeviceInfo

Raises

RuntimeError – if we failed to connect.

class qblox_instruments.**DeviceInfo**(*manufacturer: str, model: str, serial: Optional[str] = None, sw_build: Optional[BuildInfo] = None, fw_build: Optional[BuildInfo] = None, kmod_build: Optional[BuildInfo] = None, cfg_man_build: Optional[BuildInfo] = None*)

Bases: object

Class representing the build and model information of a device. Has the same information content as what *IDN? returns.

__init__(*manufacturer: str, model: str, serial: Optional[str] = None, sw_build: Optional[BuildInfo] = None, fw_build: Optional[BuildInfo] = None, kmod_build: Optional[BuildInfo] = None, cfg_man_build: Optional[BuildInfo] = None*)

property manufacturer: str

The manufacturer name, in lowercase_with_underscores format.

Type

str

property model: str

The model name, in lowercase_with_underscores format.

Type

str

property device: str

The model name, in lowercase_with_underscores format.

Type

str

property serial: Optional[str]

The serial number, if known.

Type

Optional[str]

property sw_build: Optional[*BuildInfo*]

The software/application build information, if known.

Type

Optional[*BuildInfo*]

property fw_build: Optional[*BuildInfo*]

The FPGA firmware build information, if known.

Type
Optional[*BuildInfo*]

property kmod_build: Optional[*BuildInfo*]

The kernel module build information, if known.

Type
Optional[*BuildInfo*]

property cfg_man_build: Optional[*BuildInfo*]

The configuration management build information, if known.

Type
Optional[*BuildInfo*]

get_build_info(*key: str*) → Optional[*BuildInfo*]

Returns build information for the given key.

Parameters
key (*str*) – The key. Must be one of:
- `"sw"`: returns the application build info;
- `"fw"`: returns the FPGA firmware build info;
- `"kmod"`: returns the kernel module build info; or
- `"cfg_man"` or `"cfgMan"`: returns the configuration manager build info.

Returns
The build information structure, if known.

Return type
Optional[*BuildInfo*]

Raises
KeyError – For unknown keys.

classmethod from_idn(*idn: str*) → *DeviceInfo*

Constructs a device information structure from an `*IDN?` string.

Parameters
idn (*str*) – The `*IDN?` string.

Returns
The parsed device information structure.

Return type
DeviceInfo

to_idn() → str

Formats this device information object in the same way `*IDN?` is formatted.

Returns
The `*IDN?` string.

Return type
str

classmethod from_dict(*description: dict*) → *DeviceInfo*

Constructs a device information structure from a JSON-capable dict, as used in ZeroMQ/CBOR descriptions, plug&play descriptions, update file metadata, and various other places.

Parameters
description (*dict*) – Dictionary with the following keys:
- `"manufacturer"`: manufacturer name (string);
- `"model"`: model name (string);
- `"ser"`: serial number (string);
- `"sw"`: application build information (dict);
- `"fw"`: FPGA firmware build information (dict);

- "kmod": kernel module build information (dict); and
- "cfg_man": configuration management build information (dict);

Returns
The build information structure.

Return type
DeviceInfo

to_dict() → dict

Formats this device information object as a JSON-capable dict, as used in ZeroMQ/CBOR descriptions, plug&play descriptions, update file metadata, and various other places.

Returns

The generated dictionary, having the following keys:
- "manufacturer": manufacturer name (string);
- "model": model name (string);
- "ser": serial number (string);
- "sw": application build information (dict);
- "fw": FPGA firmware build information (dict);
- "kmod": kernel module build information (dict); and/or
- "cfg_man": configuration management build information (dict);

Some keys may be omitted if the information is not available.

Return type
dict

to_idn_dict() → dict

Formats this device information object as a human-readable JSON-capable dict, as used get_idn.

Returns

The generated dictionary, having the following keys:
- "manufacturer": manufacturer name (string);
- "model": model name (string);
- "serial_number": serial number (string);
- **"firmware": build info (dict);**
  - "fpga": FPGA firmware build information (dict);
  - "kernel_mod": kernel module build information (dict);
  - "application": application build information (dict); and
  - "driver": driver build information (dict);

Some keys may be omitted if the information is not available.

Return type
dict

to_tuple() → tuple

Formats this device information object as a tuple for ordering purposes.

Returns
A tuple, containing all the information in this structure in a canonical format.

Return type
tuple

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C

D

E

F

G

N

O

P

Q

R

S