Qblox Instruments

Release 0.1

Qblox

Nov 23, 2021

GETTING STARTED

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

Caution: The instrument drivers are still in a **beta** state; major changes are expected. Use for testing and development purposes only.

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 ($O^{[1-4]}$) for a Pulsar QCM; or 2 outputs and 2 input channels ($I^{[1-2]}$) for a Pulsar QRM.
- **4 x SMP female connectors**: Marker output channels.
- **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.
  - **REF^{out}**: 10MHz reference clock output.
  - **TRIG^{in}**: External trigger input.

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

In this section we will explain how to connect a Qblox instrument (e.g. a 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.

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:


QBLOX
PULSAR QCM
QUBIT CONTROL MODULE

```
# Import driver
from pulsar_qcm.pulsar_qcm import pulsar_qcm

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

Tip: Close the connection to the module using `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 a device (e.g. a Pulsar Series QCM or QRM). We will show how to scan a range of IP addresses on the `192.168.0.X` subnet.

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

(a) Follow steps 1-3 described in *Connecting to multiple modules*.

(b) 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.

(c) Scan the IP address within the subnet:

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).

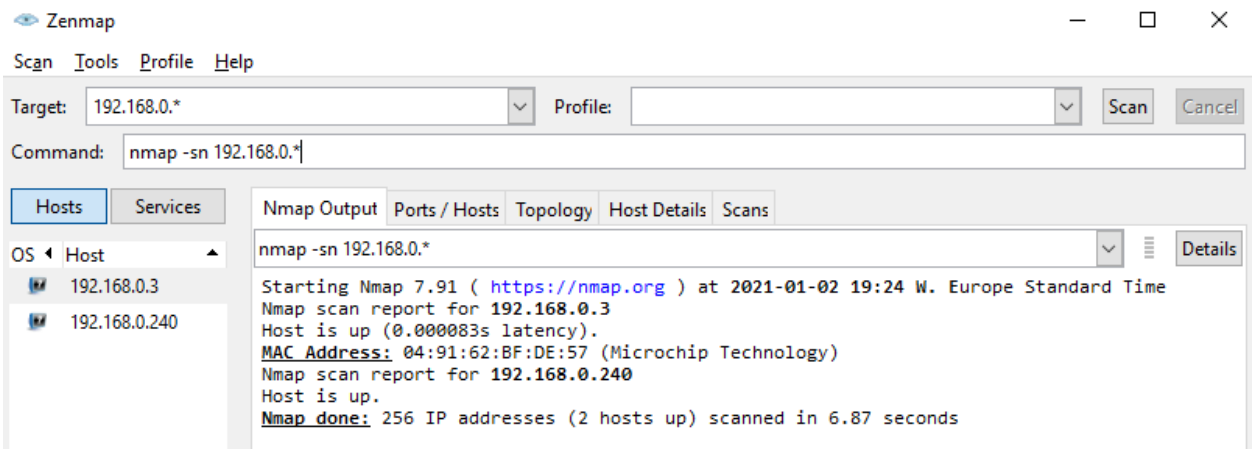
(a) Visit download section at nmap.org and download the *Latest stable release self-installer* under the *Microsoft Windows binaries* section.

(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 pulsar_qcm.pulsar_qcm import pulsar_qcm
from pulsar_qrm.pulsar_qrm import pulsar_qrm

# Connect to modules
qcm = pulsar_qcm("qcm", "192.168.0.2")  # This module uses the default IP␣
↪address.
qrm = pulsar_qrm("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 Updating

In this section we will explain how to update the firmware and IP address of your module. Go to Qblox.com and download the firmware from the download section. The firmware includes the Qblox Configuration Manager with which you can configure and update your module.

Once you have extracted the firmware, go to the directory in which it was extracted using the terminal of your choice and execute the following commands using Python 3.8.

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.

To update the firmware:

```
$ python cfg_man.py -u 192.168.0.2
```

To update the IP address:

```
$ python cfg_man.py -i 192.168.0.<new_ip_digit(s)> 192.168.0.2
```

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

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 LEDs will turn purple/red, which indicates the reboot is in progress. When the LEDs turn green/blue again, the reboot is finished. If the module does not reboot within a few minutes (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 module.

Please make sure your installed Qblox instruments package is compatible with the installed firmware. (See Qblox instruments PyPI and section *Installation*)

Tip: Executing `$ python cfg_man.py -h` will show you a list of additional functions, like querying the firmware version and reverting the last firmware update.

1.5 What's next

1.5.1 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.

1.5.2 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.

1.6 General

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

1.6.1 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.

1.6.2 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 signal) [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 signal). 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*).

1.6.3 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_status()`. This returns the following status and accompanying flags that elaborate on the status:

- **Status**:
  - **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 board PLL unlocked**: No reference clock found.
  - **FPGA PLL unlocked**: No reference clock found.

- **FPGA temperature out-of-range**: FPGA temperature has surpassed 80°C.
- **Carrier board temperature out-of-range**: Carrier board temperature has surpassed 100°C.
- **Analog frontend temperature out-of-range**: Analog frontend board temperature has surpassed 100°C.

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	**Green**	Okay.
	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.

1.7 Pulsar

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

1.7.1 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.

1.7.2 QRM Overview

The Qubit Readout Module is an instrument that targets qubit readout. To accomplish this, it shares the same architecture as the QCM, but in addition, each sequencer also 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.

Note: The acquisition path is still in development and only supports raw input captures (i.e. *scope mode*) at the moment. More modes will be added in the near future.

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.

1.7.3 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 number of these sequencers to target multiple qubits with one instrument. See section *Sequencer* for more information on how to program and control them.

Note: Currently the number of sequencers per instrument is limited to the number of output channel pairs. This will be expanded in the near future.

---

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. This will become more flexible in the future with a calibration matrix.

---

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.

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_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#_nco_mod_en 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.

Note: Currently there is only one sequencer for every output pair. When this is expanded in the near future, the multiplexer will be added.

---

7. Mixer correction

Every in- and output pair has access to a mixer correction module ideal for up or down conversion using an external mixer. The mixer correction module can compensate for mixer imperfections like IQ imbalance, phase offset and DC offset.

Note: The mixer correction will be implemented in the near future.

---

8. High-speed data converters

The Pulsar instruments use state-of-art 1Gbps 16-bit DACs and 1Gbps 12-bit ADCs.

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()`.

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.

1.8 Sequencer

This section will explain how the sequencers 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.

1.8.1 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.

1.8.2 Sequence

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

- **program**: Single string containing the entire sequence processor Q1ASM program.
- **awg**: 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.
- **acq**: Indicates that the following waveforms are intended for the integration unit of the acquisition path (only used by the Pulsar QRM).
  - **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.

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 to be placed in between these consecutively 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	Description
Control				

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	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				
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*.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Description
jlt	Register	Immediate	Immediate, Register, Label	If *argument 0* is less than *argument 1*, jump to the instruction indicated by *argument 2*.
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	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 1* and *argument 0* and move the result to *argument 2*.
or	Register	Immediate, Register	Register	Bit-wise OR *argument 1* and *argument 0* and move the result to *argument 2*.
xor	Register	Immediate, Register	Register	Bit-wise XOR *argument 1* and *argument 0* and move the result to *argument 2*.
asl	Register	Immediate, Register	Register	Bit-wise left-shift *argument 1* by *argument 0* number of bits and move the result to *argument 2*.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Description
asr	Register	Immediate, Register	Register	Bit-wise right-shift *argument 1* by *argument 0* 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				
set_mrk	Immediate, Register	–	–	Set marker output channels to *argument 0* (bits 0-3), where the bit index corresponds to the channel index. The set value is OR´ed by that of other sequencers. The parameters are cached and only updated when the *upd_param* or *play* instructions are executed.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Description
set_ph	Immediate, Register	Immediate, Register	Immediate, Register	Set the AWG and acquisition phase of the NCO. The phase is divided into a coarse (*argument 0*), fine (*argument 1*) and ultra-fine (*argument 2*) segment. The coarse segment is divided into 400 steps of 0.9°. The fine segment is divided into 400 steps of 2.25e-3°. And the ultra-fine segment is divided into 6250 steps of 3.6e-7°. The parameters are cached and only updated when the *upd_param* or *play* 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	Description
set_ph_delta	Immediate, Register	Immediate, Register	Immediate, Register	Set the AWG and acquisition phase offset of the NCO. Offset that is applied on top of the phase set using *set_phase*. See *set_phase* for more details regarding the arguments. The parameters are cached and only updated when the *upd_param* or *play* instructions are executed.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	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` or *play* 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	Description
set_acq_gain	Immediate, Register	Immediate, Register	–	Set acquisition 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* or *play* 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	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* or *play* 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	Description
set_acq_offs	Immediate, Register	Immediate, Register	–	Set acquisition 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* or *play* instructions are executed. The arguments are either all set through immediates or registers.
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	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	Description
acquire	Immediate, Register	Immediate, Register	Immediate	Update the marker, phase, phase offset, gain and offset parameters set using their respective instruction, start acquisition using integration waveforms stored at indexes *argument 0* for path 0 and *argument 1* for path 1 and finally wait for *argument 2* number of nanoseconds. The arguments are either all set through immediates or registers.
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.

continues on next page

Table 1 – continued from previous page

Instructions	Argument 0	Argument 1	Argument 2	Description
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*, *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 (e.g. `R0`)
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.
```

(continues on next page)

(continued from previous page)

```
    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 and acquisition paths use the same format for these waveforms for different purposes. The AWG path uses these waveforms to parametrically generate pulses on its outputs and the acquisition path uses these waveforms as integration functions.

Waveform playback is started by the *play* and *acquire* instructions. Each waveform is paired with an index, which is used by these instructions 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()`.

Note: The integration feature in the acquisition path is not implemented yet and the waveforms in that path are therefore not used.

Acquisitions

Acquisitions are started by the *acquire* instruction 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 be done by calling *store_acquisition()*, which moves the samples into 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 lists containing floating point values in a range of 1.0 to -1.0 with a resolution of one nanosecond per sample. The list can be cleared from the RAM of the instrument by calling *delete_acquisitions()*.

The acquisition path also has an averaging function set through the `sequencer#_avg_mode_en_acq_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_acquisition()* to store the accumulated result in the acquisition list. Once retrieved from the instrument you will need to divide the accumulated samples 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.

Note: The acquisition path is still in development and more acquisitions modes will be added in the near future.

1.8.3 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#_waveforms_and_program()`, `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.

1.9 Synchronization

In this section we explain how to synchronize multiple instruments in your setup including Qblox 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.

1.9.1 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 (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.

1.9.2 SYNQ

To synchronize the start event of the instruments, Qblox SYNQ technology can be used to greatly simply 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 ever 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.

1.9.3 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 as shown in the figure below (see section *Overview*) and have any sequencer in the Qblox instrument participating in the experiment execute the *wait_trigger* instruction (see section *Instructions*). Please take into account that the delay between triggering and waveform playback on any output is 95.2 ns.

1.10 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. 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.	If the LED is red, it indicates a serious error that needs to be resolved. Query `get_system_status()` 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_status()` see what the problem is.
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.
I cannot connect to the instrument.	Make sure that the Ethernet cables are firmly inserted into the instrument and host PC (see section *Connecting*).
	Make sure that the instrument and host PC are within the same subnet of the same network (see section *Connecting*).
	Make sure that you are using the IP address of the instrument. If you you do not know the IP

See also:

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

`cont_wave_mode.ipynb`

1.11 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.

This tutorial is designed with the Pulsar QCM in mind, but can easily be used with a Pulsar QRM as well. Change any Pulsar QCM reference to Pulsar QRM and only use one sequencer.

1.11.1 Setup

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

[1]:
```
#Set up the environment.
import pprint
import os
import scipy.signal
import math
import json
import matplotlib.pyplot
import numpy

from pulsar_qcm.pulsar_qcm import pulsar_qcm

#Connect to the Pulsar QCM at default IP address.
pulsar = pulsar_qcm("qcm", "192.168.0.2")

#Reset the instrument for good measure.
pulsar.reset()
print("Status:")
print(pulsar.get_system_status())
```

```
Status:
{'status': 'OKAY', 'flags': []}
```

1.11.2 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.

[2]:
```
#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": [], "index": 0},
```

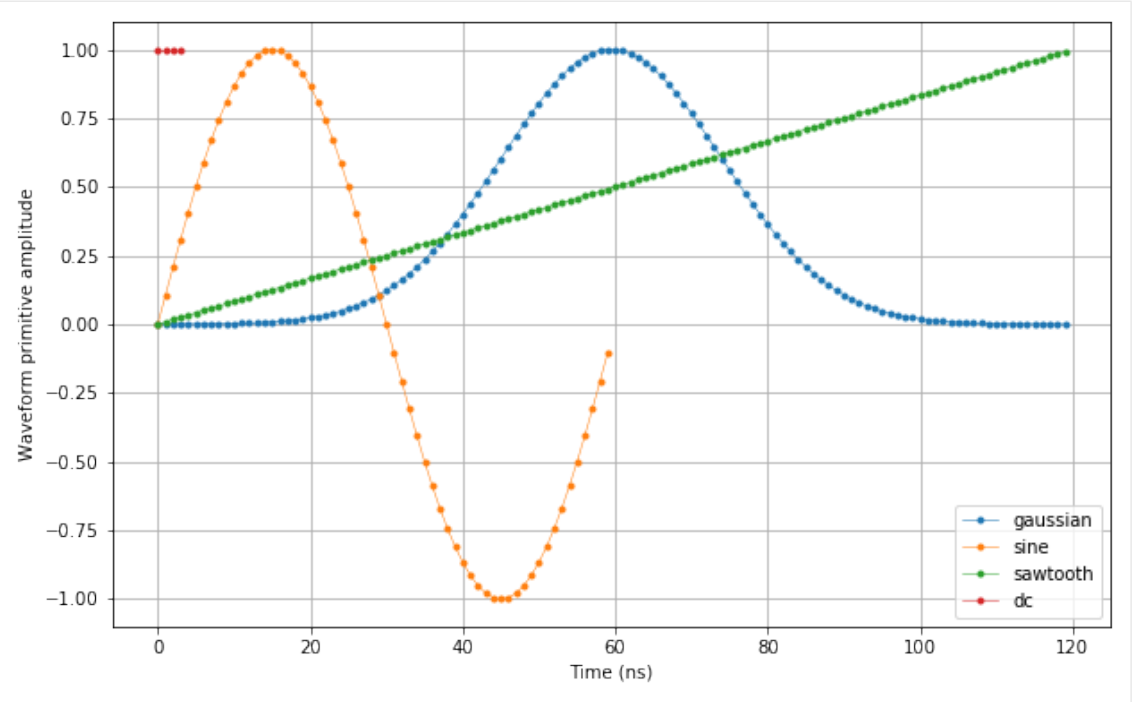
(continues on next page)

(continued from previous page)

```
                "sine":     {"data": [], "index": 1},
                "sawtooth": {"data": [], "index": 2},
                "dc":       {"data": [], "index": 3}
            }

#Create gaussian waveform
if "gaussian" in waveforms:
    waveforms["gaussian"]["data"] = scipy.signal.gaussian(waveform_length,␣
↪std=0.12 * waveform_length)

#Create sine waveform
if "sine" in waveforms:
    waveforms["sine"]["data"] = [math.sin((2 * math.pi / (0.5 * waveform_
↪length)) * i ) for i in range(0, waveform_length//2)]

#Create sawtooth waveform
if "sawtooth" in waveforms:
    waveforms["sawtooth"]["data"] = [(1.0 / (waveform_length)) * i for i in␣
↪range(0, waveform_length)]

#Create DC waveform
if "dc" in waveforms:
     waveforms["dc"]["data"] = [1.0 for i in range(0, 4)] #DC only needs to 4␣
↪samples and can be repeated
```

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

[3]:
```
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))

ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

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()

matplotlib.pyplot.draw()
matplotlib.pyplot.show()   # add this at EOF to prevent execution stall
```



1.11.3 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`).

[4]:
```
#Sequence program.
seq_prog = "stop"

#Reformat waveforms to lists if necessary.
for name in waveforms:
    if str(type(waveforms[name]["data"]).__name__) == "ndarray":
        waveforms[name]["data"] = waveforms[name]["data"].tolist()  # JSON␣
↪only supports lists
    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.
wave_and_prog_dict = {"waveforms": {"awg": waveforms}, "program": seq_prog}
with open("cont_wave_mode.json", 'w', encoding='utf-8') as file:
    json.dump(wave_and_prog_dict, 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.

[5]:
```
#Upload waveforms and programs.
pulsar.sequencer0_waveforms_and_program(os.path.join(os.getcwd(), "cont_wave_
↪mode.json"))
pulsar.sequencer1_waveforms_and_program(os.path.join(os.getcwd(), "cont_wave_
↪mode.json"))
```

1.11.4 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.

[6]:
```
#Configure the sequencers to run in continuous waveform mode.
for sequencer in range(0, 2):
    pulsar.set("sequencer{}_cont_mode_en_awg_path0".format(sequencer), True)
↪#On outputs O1 and O3.
    pulsar.set("sequencer{}_cont_mode_en_awg_path1".format(sequencer), True)
↪#On outputs O2 and O4.

#Set specific waveform to specific output.
pulsar.sequencer0_cont_mode_waveform_idx_awg_path0(0) #Gaussian on O1
pulsar.sequencer0_cont_mode_waveform_idx_awg_path1(1) #Sine on O2
pulsar.sequencer1_cont_mode_waveform_idx_awg_path0(2) #Sawtooth on O3
pulsar.sequencer1_cont_mode_waveform_idx_awg_path1(3) #DC on O4
```

Now let's start playback.

[7]:
```
#Arm and start both sequencers.
pulsar.arm_sequencer()
pulsar.start_sequencer()

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

```
Status:
{'status': 'Q1 STOPPED', 'flags': []}
{'status': 'Q1 STOPPED', 'flags': []}
```

1.11.5 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 Pulsar QCM to 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

1.11.6 Stop

Finally, let's stop the playback and close the instrument connection.

[8]:
```
#Stop both sequencers.
pulsar.stop_sequencer()

#Print status of both sequencers (should now say it is stopped).
print("Status:")
print(pulsar.get_sequencer_state(0))
print(pulsar.get_sequencer_state(1))
print()

#Print an overview of the instrument parameters.
print("Snapshot:")
pulsar.print_readable_snapshot(update=True)

#Close the instrument connection.
pulsar.close()
```

```
Status:
{'status': 'STOPPED', 'flags': ['FORCED STOP']}
{'status': 'STOPPED', 'flags': ['FORCED STOP']}

Snapshot:
qcm:
        parameter                                       value
--------------------------------------------------------------------------------
↪-
```

(continues on next page)

(continued from previous page)

```
IDN                                          :   {'manufacturer': 'Qblox',
↪'devi...
reference_source                             :  internal
sequencer0_cont_mode_en_awg_path0            :      True
sequencer0_cont_mode_en_awg_path1            :      True
sequencer0_cont_mode_waveform_idx_awg_path0  :     0
sequencer0_cont_mode_waveform_idx_awg_path1  :     1
sequencer0_gain_awg_path0                    :        1
sequencer0_gain_awg_path1                    :        1
sequencer0_marker_ovr_en                     :        False
sequencer0_marker_ovr_value                  :        0
sequencer0_mod_en_awg                        :        False
sequencer0_nco_freq                          : 0 (Hz)
sequencer0_nco_phase_offs                    :        0 (Degrees)
sequencer0_offset_awg_path0                  :        0
sequencer0_offset_awg_path1                  :        0
sequencer0_sync_en                           : False
sequencer0_upsample_rate_awg_path0           :       0
sequencer0_upsample_rate_awg_path1           :       0
sequencer0_waveforms_and_program             :        C:\Users\jordy\Projects\
↪pulsar_...
sequencer1_cont_mode_en_awg_path0            :      True
sequencer1_cont_mode_en_awg_path1            :      True
sequencer1_cont_mode_waveform_idx_awg_path0  :     2
sequencer1_cont_mode_waveform_idx_awg_path1  :     3
sequencer1_gain_awg_path0                    :        1
sequencer1_gain_awg_path1                    :        1
sequencer1_marker_ovr_en                     :        False
sequencer1_marker_ovr_value                  :        0
sequencer1_mod_en_awg                        :        False
sequencer1_nco_freq                          : 0 (Hz)
sequencer1_nco_phase_offs                    :        0 (Degrees)
sequencer1_offset_awg_path0                  :        0
sequencer1_offset_awg_path1                  :        0
sequencer1_sync_en                           : False
sequencer1_upsample_rate_awg_path0           :       0
sequencer1_upsample_rate_awg_path1           :       0
sequencer1_waveforms_and_program             :        C:\Users\jordy\Projects\
↪pulsar_...
```

See also:

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

`basic_sequencing.ipynb`

1.12 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.

This tutorial is designed with the Pulsar QCM in mind, but can easily be used with a Pulsar QRM as well. Change any Pulsar QCM reference to Pulsar QRM and only use one sequencer.

1.12.1 Setup

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

[1]:
```
#Set up the environment.
import pprint
import os
import scipy.signal
import math
import json
import matplotlib.pyplot
import numpy

from pulsar_qcm.pulsar_qcm import pulsar_qcm

#Connect to the Pulsar QCM at default IP address.
pulsar = pulsar_qcm("qcm", "192.168.0.2")

#Reset the instrument for good measure.
pulsar.reset()
print("Status:")
print(pulsar.get_system_status())
```

```
Status:
{'status': 'OKAY', 'flags': []}
```

1.12.2 Generate waveforms

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

[2]:
```
#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": [], "index": 0},
                "block":    {"data": [], "index": 1}
```

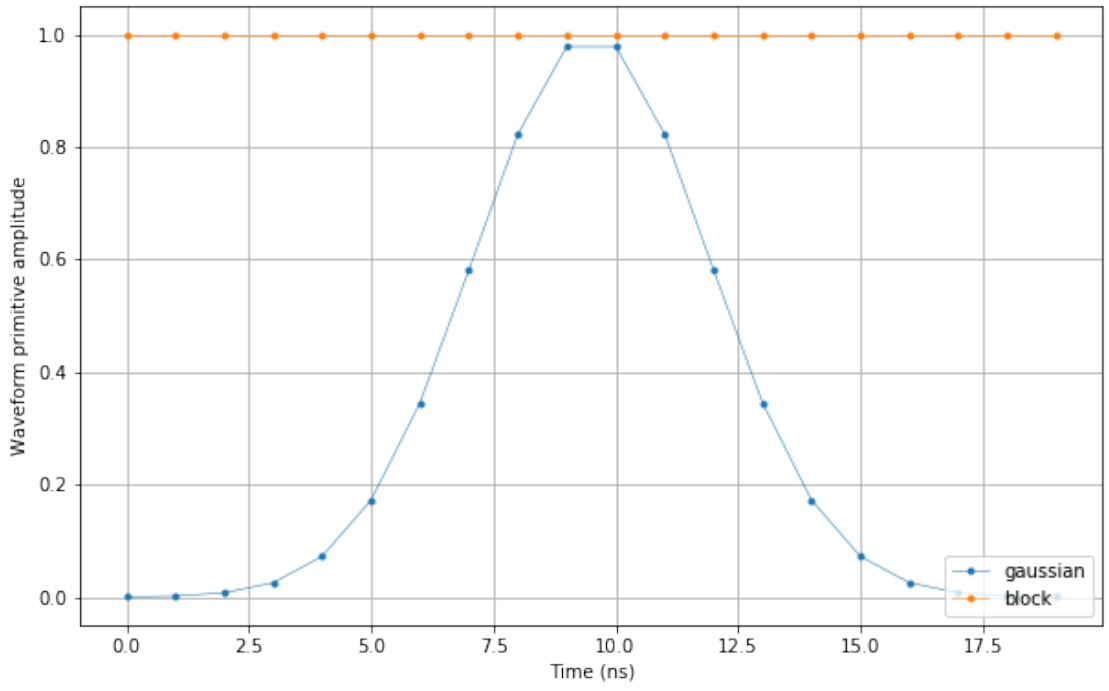
(continues on next page)

(continued from previous page)

```
            }

#Create gaussian waveform
if "gaussian" in waveforms:
    waveforms["gaussian"]["data"] = scipy.signal.gaussian(waveform_length,␣
↪std=0.12 * waveform_length)

#Create block waveform
if "block" in waveforms:
    waveforms["block"]["data"] = [1.0 for i in range(0, waveform_length)]
```

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

[3]:
```
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))

ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

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()

matplotlib.pyplot.draw()
matplotlib.pyplot.show()   # add this at EOF to prevent execution stall
```



1.12.3 Create Q1ASM program

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

[4]:
```
#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.
       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.
"""
```

1.12.4 Upload sequence

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

[5]:
```
#Reformat waveforms to lists if necessary.
for name in waveforms:
    if str(type(waveforms[name]["data"]).__name__) == "ndarray":
        waveforms[name]["data"] = waveforms[name]["data"].tolist()  # JSON␣
↪only supports lists

#Add sequence program and waveforms to single dictionary and write to JSON␣
↪file.
wave_and_prog_dict = {"waveforms": {"awg": waveforms}, "program": seq_prog}
with open("sequence.json", 'w', encoding='utf-8') as file:
    json.dump(wave_and_prog_dict, 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.

[6]:
```
#Upload waveforms and programs.
pulsar.sequencer0_waveforms_and_program(os.path.join(os.getcwd(), "sequence.
↪json"))
pulsar.sequencer1_waveforms_and_program(os.path.join(os.getcwd(), "sequence.
↪json"))
```

1.12.5 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.

[7]:
```
#Configure the sequencers to synchronize.
pulsar.sequencer0_sync_en(True)
pulsar.sequencer1_sync_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.

[8]:
```
#Arm and start both sequencers.
pulsar.arm_sequencer()
pulsar.start_sequencer()

#Print status of both sequencers.
print("Status:")
print(pulsar.get_sequencer_state(0))
print(pulsar.get_sequencer_state(1))
```

```
Status:
{'status': 'STOPPED', 'flags': []}
{'status': 'STOPPED', 'flags': []}
```

1.12.6 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection.

[9]:
```
#Stop both sequencers.
pulsar.stop_sequencer()

#Print status of both sequencers (should now say it is stopped).
print("Status:")
print(pulsar.get_sequencer_state(0))
print(pulsar.get_sequencer_state(1))
print()

#Print an overview of the instrument parameters.
print("Snapshot:")
pulsar.print_readable_snapshot(update=True)

#Close the instrument connection.
pulsar.close()
```

```
Status:
{'status': 'STOPPED', 'flags': ['FORCED STOP']}
{'status': 'STOPPED', 'flags': ['FORCED STOP']}

Snapshot:
qcm:
        parameter                                        value
--------------------------------------------------------------------------------
↪-
IDN                                          :   {'manufacturer': 'Qblox',
↪'devi...
reference_source                             :  internal
sequencer0_cont_mode_en_awg_path0            :      False
sequencer0_cont_mode_en_awg_path1            :      False
sequencer0_cont_mode_waveform_idx_awg_path0 :     0
sequencer0_cont_mode_waveform_idx_awg_path1 :     0
sequencer0_gain_awg_path0                    :        1
sequencer0_gain_awg_path1                    :        1
sequencer0_marker_ovr_en                     :        False
sequencer0_marker_ovr_value                  :        0
sequencer0_mod_en_awg                        :        False
sequencer0_nco_freq                          : 0 (Hz)
sequencer0_nco_phase_offs                    :        0 (Degrees)
sequencer0_offset_awg_path0                  :        0
sequencer0_offset_awg_path1                  :        0
sequencer0_sync_en                           : True
sequencer0_upsample_rate_awg_path0           :       0
sequencer0_upsample_rate_awg_path1           :       0
sequencer0_waveforms_and_program             :        C:\Users\jordy\Projects\
↪pulsar_...
sequencer1_cont_mode_en_awg_path0            :      False
sequencer1_cont_mode_en_awg_path1            :      False
sequencer1_cont_mode_waveform_idx_awg_path0 :     0
sequencer1_cont_mode_waveform_idx_awg_path1 :     0
sequencer1_gain_awg_path0                    :        1
sequencer1_gain_awg_path1                    :        1
sequencer1_marker_ovr_en                     :        False
sequencer1_marker_ovr_value                  :        0
sequencer1_mod_en_awg                        :        False
sequencer1_nco_freq                          : 0 (Hz)
sequencer1_nco_phase_offs                    :        0 (Degrees)
sequencer1_offset_awg_path0                  :        0
sequencer1_offset_awg_path1                  :        0
sequencer1_sync_en                           : True
sequencer1_upsample_rate_awg_path0           :       0
sequencer1_upsample_rate_awg_path1           :       0
sequencer1_waveforms_and_program             :        C:\Users\jordy\Projects\
↪pulsar_...
```

See also:

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

`advanced_sequencing.ipynb`

1.13 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 markour 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.

This tutorial is designed with the Pulsar QCM in mind, but can easily be used with a Pulsar QRM as well. Change any Pulsar QCM reference to Pulsar QRM and only use one sequencer.

1.13.1 Setup

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

[1]:
```
#Set up the environment.
import pprint
import os
import scipy.signal
import math
import json
import matplotlib.pyplot
import numpy

from pulsar_qcm.pulsar_qcm import pulsar_qcm

#Connect to the Pulsar QCM at default IP address.
pulsar = pulsar_qcm("qcm", "192.168.0.2")

#Reset the instrument for good measure.
pulsar.reset()
print("Status:")
print(pulsar.get_system_status())
```

```
Status:
{'status': 'OKAY', 'flags': []}
```

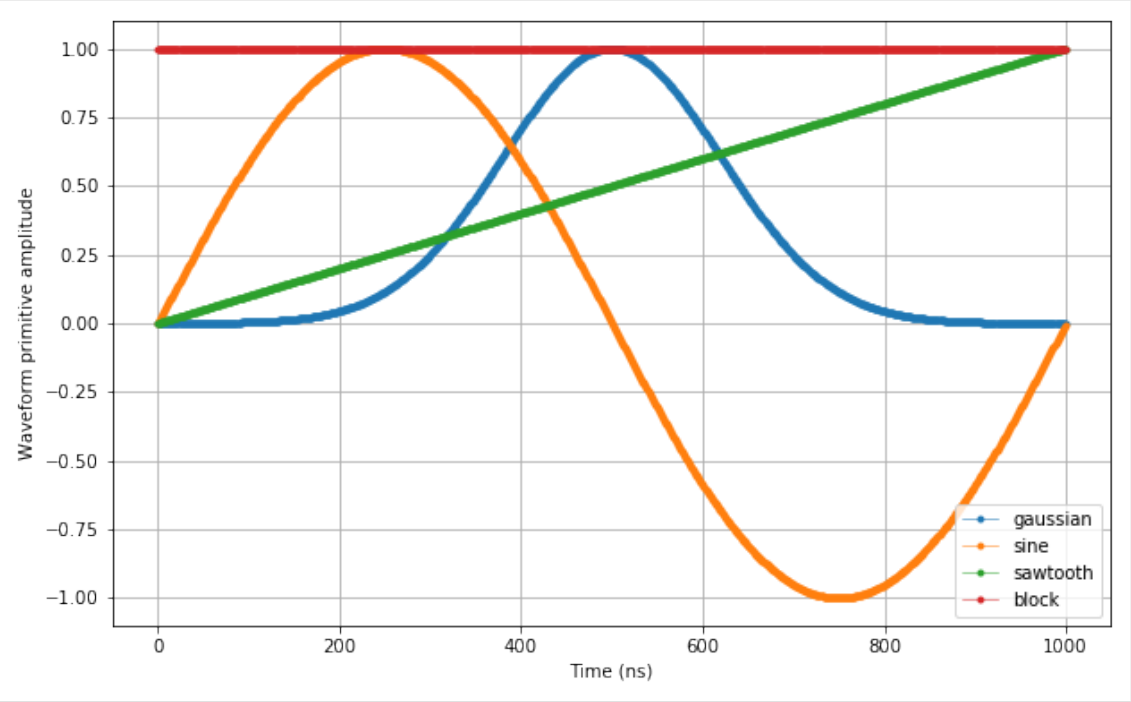
1.13.2 Generate waveforms

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

[2]:
```
#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": [], "index": 0},
                "sine":     {"data": [], "index": 1},
                "sawtooth": {"data": [], "index": 2},
                "block":    {"data": [], "index": 3}
            }

#Create gaussian waveform
if "gaussian" in waveforms:
    waveforms["gaussian"]["data"] = scipy.signal.gaussian(waveform_length,␣
↪std=0.12 * waveform_length)

#Create sine waveform
if "sine" in waveforms:
    waveforms["sine"]["data"] = [math.sin((2 * math.pi / waveform_length) * i␣
↪) for i in range(0, waveform_length)]

#Create sawtooth waveform
if "sawtooth" in waveforms:
    waveforms["sawtooth"]["data"] = [(1.0 / (waveform_length)) * i for i in␣
↪range(0, waveform_length)]

#Create block waveform
if "block" in waveforms:
    waveforms["block"]["data"] = [1.0 for i in range(0, waveform_length)]
```

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

[3]:
```
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))

ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

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()

matplotlib.pyplot.draw()
matplotlib.pyplot.show()   # add this at EOF to prevent execution stall
```



1.13.3 Create Q1ASM program

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

```
[4]: #Sequence program.
seq_prog = """
                  wait_sync    4                  #Wait for synchronization
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
                  set_mrk      0                  #Set marker to 0
                  upd_param    96                 #Update all parameters and␣
↪wait 96ns
```

(continues on next page)

(continued from previous page)

```
                  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

long_wait_loop_1: wait         50000                 #Wait for 50 us
                  loop         R7,@long_wait_loop_1 #Wait total of 40ms
```

(continues on next page)

(continued from previous page)

```
                    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
"""
```

1.13.4 Upload sequence

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

```
[5]: #Reformat waveforms to lists if necessary.
     for name in waveforms:
         if str(type(waveforms[name]["data"]).__name__) == "ndarray":
             waveforms[name]["data"] = waveforms[name]["data"].tolist()  # JSON␣
     ↪only supports lists

     #Add sequence program and waveforms to single dictionary and write to JSON␣
     ↪file.
     wave_and_prog_dict = {"waveforms": {"awg": waveforms}, "program": seq_prog}
     with open("sequence.json", 'w', encoding='utf-8') as file:
         json.dump(wave_and_prog_dict, 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.

```
[6]: #Upload waveforms and programs.
     pulsar.sequencer0_waveforms_and_program(os.path.join(os.getcwd(), "sequence.
     ↪json"))
     pulsar.sequencer1_waveforms_and_program(os.path.join(os.getcwd(), "sequence.
     ↪json"))
```

1.13.5 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.

```
[7]: #Configure the sequencers to synchronize and enable modulation at 10MHz.
     pulsar.sequencer0_sync_en(True)
     pulsar.sequencer0_mod_en_awg(True)
     pulsar.sequencer0_nco_freq(10e6)
     pulsar.sequencer1_sync_en(True)
     pulsar.sequencer1_mod_en_awg(True)
     pulsar.sequencer1_nco_freq(10e6)
```

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.

[8]:
```
#Arm and start both sequencers.
pulsar.arm_sequencer()
pulsar.start_sequencer()

#Print status of both sequencers.
print("Status:")
print(pulsar.get_sequencer_state(0))
print(pulsar.get_sequencer_state(1))
```

```
Status:
{'status': 'RUNNING', 'flags': []}
{'status': 'RUNNING', 'flags': []}
```

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

1.13.6 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection.

[9]:
```
#Stop both sequencers.
pulsar.stop_sequencer()

#Print status of both sequencers (should now say it is stopped).
print("Status:")
print(pulsar.get_sequencer_state(0))
print(pulsar.get_sequencer_state(1))
print()

#Print an overview of the instrument parameters.
print("Snapshot:")
pulsar.print_readable_snapshot(update=True)

#Close the instrument connection.
pulsar.close()
```

```
Status:
{'status': 'STOPPED', 'flags': ['FORCED STOP']}
{'status': 'STOPPED', 'flags': ['FORCED STOP']}

Snapshot:
qcm:
        parameter                                        value
--------------------------------------------------------------------------------
↪-
IDN                                          :   {'manufacturer': 'Qblox',
↪'devi...
reference_source                             :  internal
sequencer0_cont_mode_en_awg_path0            :      False
sequencer0_cont_mode_en_awg_path1            :      False
```

(continues on next page)

(continued from previous page)

```
sequencer0_cont_mode_waveform_idx_awg_path0 :     0
sequencer0_cont_mode_waveform_idx_awg_path1 :     0
sequencer0_gain_awg_path0                   :        1
sequencer0_gain_awg_path1                   :        1
sequencer0_marker_ovr_en                    :        False
sequencer0_marker_ovr_value                 :        0
sequencer0_mod_en_awg                       :        True
sequencer0_nco_freq                         : 1e+07 (Hz)
sequencer0_nco_phase_offs                   :        0 (Degrees)
sequencer0_offset_awg_path0                 :        0
sequencer0_offset_awg_path1                 :        0
sequencer0_sync_en                          : True
sequencer0_upsample_rate_awg_path0          :       0
sequencer0_upsample_rate_awg_path1          :       0
sequencer0_waveforms_and_program            :        C:\Users\jordy\Projects\
↪pulsar_...
sequencer1_cont_mode_en_awg_path0           :      False
sequencer1_cont_mode_en_awg_path1           :      False
sequencer1_cont_mode_waveform_idx_awg_path0 :     0
sequencer1_cont_mode_waveform_idx_awg_path1 :     0
sequencer1_gain_awg_path0                   :        1
sequencer1_gain_awg_path1                   :        1
sequencer1_marker_ovr_en                    :        False
sequencer1_marker_ovr_value                 :        0
sequencer1_mod_en_awg                       :        True
sequencer1_nco_freq                         : 1e+07 (Hz)
sequencer1_nco_phase_offs                   :        0 (Degrees)
sequencer1_offset_awg_path0                 :        0
sequencer1_offset_awg_path1                 :        0
sequencer1_sync_en                          : True
sequencer1_upsample_rate_awg_path0          :       0
sequencer1_upsample_rate_awg_path1          :       0
sequencer1_waveforms_and_program            :        C:\Users\jordy\Projects\
↪pulsar_...
```

See also:

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

`acquisition.ipynb`

1.14 Acquisition

In this tutorial we will demonstrate the sequencer based acquisition procedure as well as how to average multiple acquisitions in hardware (see section Acquisition). We will do this by using a Pulsar QRM and directly connect outputs $O^{[1-2]}$ to inputs $I^{[1-2]}$ respectively. We will then use the Pulsar QRM's sequencers to sequence waveforms on the outputs and simultaneously acquire the resulting waveforms on the inputs.

1.14.1 Setup

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

[1]:
```
#Set up the environment.
import pprint
import os
import scipy.signal
import math
import json
import matplotlib.pyplot
import numpy

from pulsar_qrm.pulsar_qrm import pulsar_qrm

#Connect to the Pulsar QRM at default IP address.
pulsar = pulsar_qrm("qrm", "192.168.0.2", debug=1)

#Reset the instrument for good measure.
pulsar.reset()
print("Status:")
print(pulsar.get_system_status())
```

```
Status:
{'status': 'OKAY', 'flags': []}
```

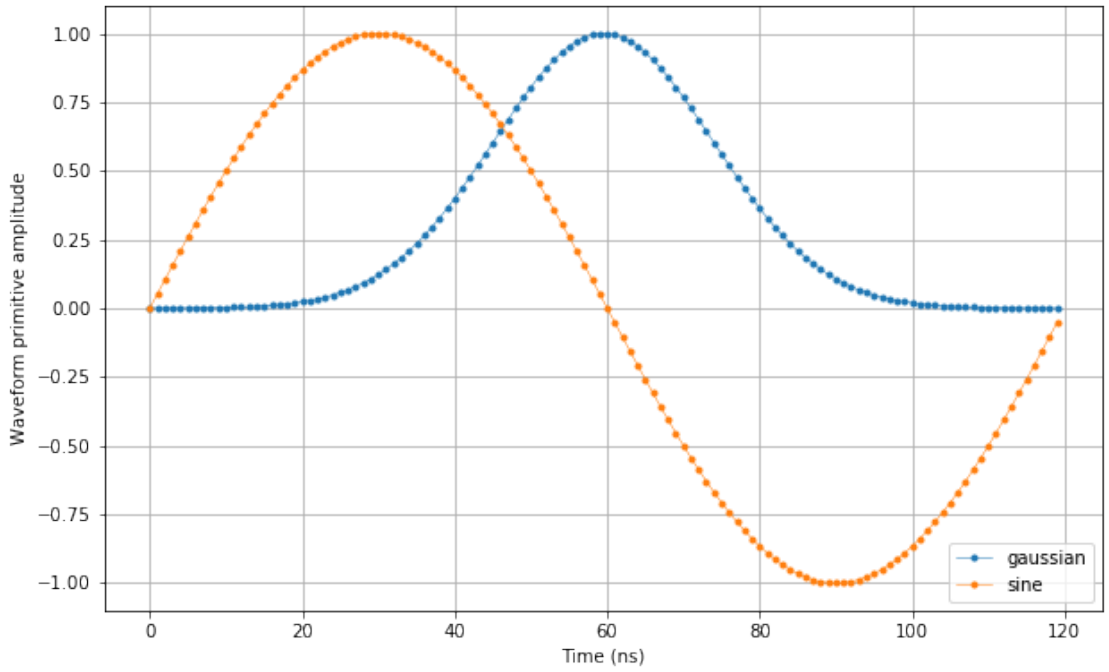
1.14.2 Generate waveforms

Next, we need to create the waveforms for the sequence.

[2]:
```
#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": [], "index": 0},
                "sine":     {"data": [], "index": 1}
            }

#Create gaussian waveform
if "gaussian" in waveforms:
    waveforms["gaussian"]["data"] = scipy.signal.gaussian(waveform_length,␣
↪std=0.12 * waveform_length)

#Create sine waveform
if "sine" in waveforms:
    waveforms["sine"]["data"] = [math.sin((2*math.pi/waveform_length)*i) for i␣
↪in range(0, waveform_length)]
```

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

[3]:
```
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))

ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

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()

matplotlib.pyplot.draw()
matplotlib.pyplot.show()   # add this at EOF to prevent execution stall
```



1.14.3 Create Q1ASM program

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

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

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.

[5]:
```
#Reformat waveforms to lists if necessary.
for name in waveforms:
    if str(type(waveforms[name]["data"]).__name__) == "ndarray":
        waveforms[name]["data"] = waveforms[name]["data"].tolist()  # JSON␣
↪only supports lists

#Add sequence program and waveforms to single dictionary and write to JSON␣
↪file.
wave_and_prog_dict = {"waveforms": {"awg": waveforms, "acq": waveforms},
↪"program": seq_prog}
with open("sequence.json", 'w', encoding='utf-8') as file:
    json.dump(wave_and_prog_dict, 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 $I^{[1-2]}$.

[6]:
```
#Upload waveforms and programs.
pulsar.sequencer0_waveforms_and_program(os.path.join(os.getcwd(), "sequence.
↪json"))
```

1.14.5 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.

[7]:
```
#Configure the sequencer to trigger the acquisition.
pulsar.sequencer0_trigger_mode_acq_path0("sequencer")
pulsar.sequencer0_trigger_mode_acq_path1("sequencer")
```

Now let's start the sequence.

[8]:
```
#Arm and start sequencer.
pulsar.arm_sequencer()
pulsar.start_sequencer()

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

```
Status:
{'status': 'STOPPED', 'flags': ['ACQ WAVE CAPTURE DONE PATH 0', 'ACQ WAVE␣
↪CAPTURE DONE PATH 1']}
```

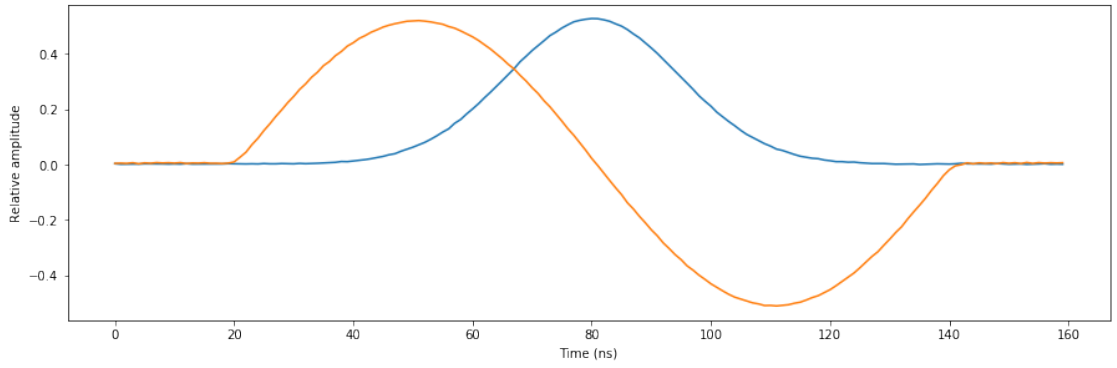
1.14.6 Retrieve acquisition

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.

[9]:
```
#Wait for the sequencer to stop with a timeout period of one second.
pulsar.get_sequencer_state(0, 1)

#Wait for the acquisition to finish with a timeout period of one second.
pulsar.get_acquisition_state(0, 1)

#Move acquisition data from temporary memory to acquisition list.
pulsar.store_acquisition(0, "measurement")

#Get acquisition list from instrument.
acq = pulsar.get_acquisitions(0)
```

Let's plot the result.

[10]:
```
#Plot acquired signal on both inputs.
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15/2/1.61))
ax.plot(acq["measurement"]["path_0"]["data"][100:260])
ax.plot(acq["measurement"]["path_1"]["data"][100:260])
ax.set_xlabel('Time (ns)')
ax.set_ylabel('Relative amplitude')
matplotlib.pyplot.show()
```



1.14.7 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.

[11]:
```
#Clear acquisition list of any previously made acquisitions
pulsar.delete_acquisitions(0)

#First run
pulsar.sequencer0_gain_awg_path0(0.33)
```

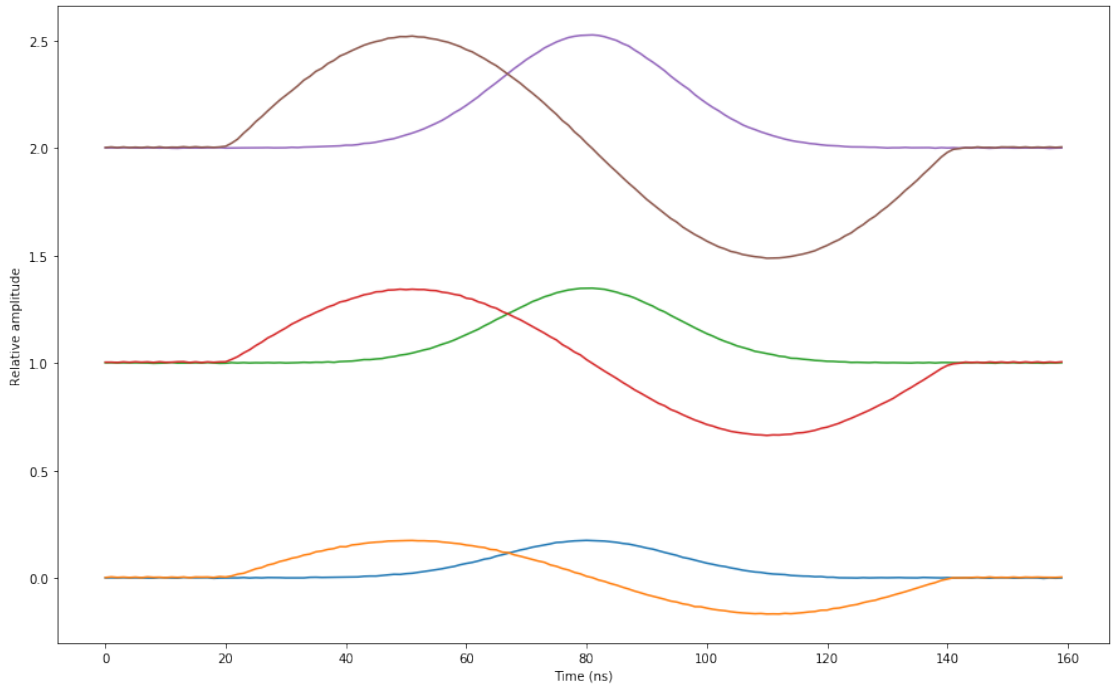
(continues on next page)

(continued from previous page)

```
pulsar.sequencer0_gain_awg_path1(0.33)

pulsar.arm_sequencer()
pulsar.start_sequencer()

pulsar.get_sequencer_state(0, 1)
pulsar.get_acquisition_state(0, 1)

pulsar.store_acquisition(0, "measurement_0")

#Second run
pulsar.sequencer0_gain_awg_path0(0.66)
pulsar.sequencer0_gain_awg_path1(0.66)

pulsar.arm_sequencer()
pulsar.start_sequencer()

pulsar.get_sequencer_state(0, 1)
pulsar.get_acquisition_state(0, 1)

pulsar.store_acquisition(0, "measurement_1")

#Second run
pulsar.sequencer0_gain_awg_path0(1)
pulsar.sequencer0_gain_awg_path1(1)

pulsar.arm_sequencer()
pulsar.start_sequencer()

pulsar.get_sequencer_state(0, 1)
pulsar.get_acquisition_state(0, 1)

pulsar.store_acquisition(0, "measurement_2")

#Get acquisition list from instrument.
acq = pulsar.get_acquisitions(0)
```

Let's plot the result again.

[12]:
```
#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(acq["measurement_{}".format(acq_idx)]["path_0"]["data
↪"][100:260]) + acq_idx)
    ax.plot(numpy.array(acq["measurement_{}".format(acq_idx)]["path_1"]["data
↪"][100:260]) + acq_idx)
    ax.set_xlabel('Time (ns)')
    ax.set_ylabel('Relative amplitude')
matplotlib.pyplot.show()
```



1.14.8 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.

[13]:
```
#Sequence program.
seq_prog = """
      move    1000,R0   #Loop iterator.

loop: play    0,1,4     #Play waveforms and wait 4ns.
      acquire 0,0,16380 #Acquire waveforms remaining duration of 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.

[14]:
```
#Clear acquisition list of any previously made acquisitions
pulsar.delete_acquisitions(0)

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

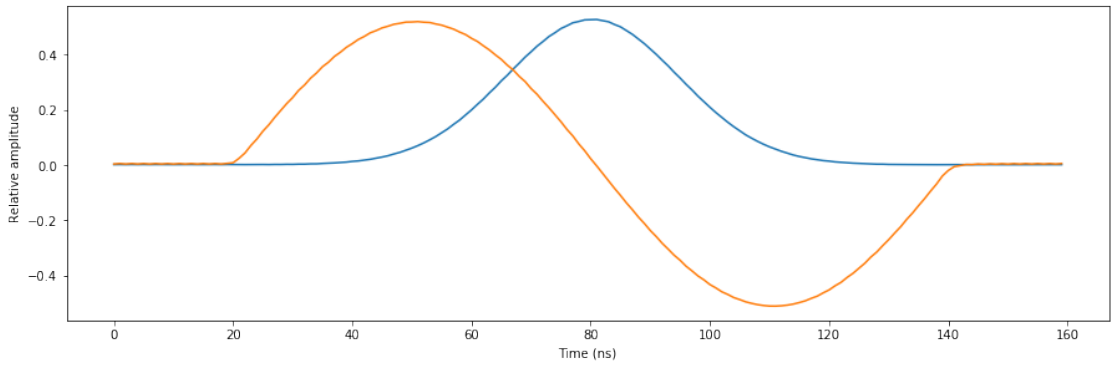
(continues on next page)

(continued from previous page)

```
    file.close()

#Upload waveforms and programs.
pulsar.sequencer0_waveforms_and_program(os.path.join(os.getcwd(), "sequence_
↪avg.json"))

#Configure the sequencer to trigger the acquisition.
pulsar.sequencer0_trigger_mode_acq_path0("sequencer")
pulsar.sequencer0_trigger_mode_acq_path1("sequencer")
pulsar.sequencer0_avg_mode_en_acq_path0(True)
pulsar.sequencer0_avg_mode_en_acq_path1(True)

#Arm and start sequencer.
pulsar.arm_sequencer()
pulsar.start_sequencer()

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

#Move accumulated result from temporary memory to the instrument's acquisition␣
↪list.
pulsar.store_acquisition(0, "measurement_avg")

#Get acquisition list from instrument.
acq = pulsar.get_acquisitions(0)
```

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

[15]:
```
#Divide accumulated result by a 1000 and plot results.
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15/2/1.61))
ax.plot(numpy.array(acq["measurement_avg"]["path_0"]["data"][100:260]) / 1000)
ax.plot(numpy.array(acq["measurement_avg"]["path_1"]["data"][100:260]) / 1000)
ax.set_xlabel('Time (ns)')
ax.set_ylabel('Relative amplitude')
matplotlib.pyplot.show()
```



1.14.9 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connection.

[16]:
```
#Stop sequencer.
pulsar.stop_sequencer()

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

#Print an overview of the instrument parameters.
print("Snapshot:")
pulsar.print_readable_snapshot(update=True)

#Close the instrument connection.
pulsar.close()
```

```
Status:
{'status': 'STOPPED', 'flags': ['FORCED STOP', 'ACQ WAVE CAPTURE DONE PATH 0',
↪'ACQ WAVE CAPTURE OVERWRITTEN PATH 0', 'ACQ WAVE CAPTURE DONE PATH 1', 'ACQ␣
↪WAVE CAPTURE OVERWRITTEN PATH 1']}

Snapshot:
qrm:
        parameter                                        value
--------------------------------------------------------------------------------
↪-
IDN                                          :   {'manufacturer': 'Qblox',
↪'devi...
in0_amp_gain                                 : -6 (dB)
in1_amp_gain                                 : -6 (dB)
reference_source                             :  internal
sequencer0_avg_mode_en_acq_path0             :      True
sequencer0_avg_mode_en_acq_path1             :      True
sequencer0_cont_mode_en_awg_path0            :      False
sequencer0_cont_mode_en_awg_path1            :      False
sequencer0_cont_mode_waveform_idx_awg_path0  :     0
sequencer0_cont_mode_waveform_idx_awg_path1  :     0
sequencer0_gain_awg_path0                    :        1
sequencer0_gain_awg_path1                    :        1
sequencer0_marker_ovr_en                     :        False
sequencer0_marker_ovr_value                  :        0
sequencer0_mod_en_awg                        :        False
sequencer0_nco_freq                          : 0 (Hz)
sequencer0_nco_phase_offs                    :        0 (Degrees)
sequencer0_offset_awg_path0                  :        0
sequencer0_offset_awg_path1                  :        0
sequencer0_sync_en                           : False
sequencer0_trigger_level_acq_path0           :       0
sequencer0_trigger_level_acq_path1           :       0
sequencer0_trigger_mode_acq_path0            :       sequencer
sequencer0_trigger_mode_acq_path1            :       sequencer
```

(continues on next page)

(continued from previous page)

```
sequencer0_upsample_rate_awg_path0          :       0
sequencer0_upsample_rate_awg_path1          :       0
sequencer0_waveforms_and_program            :        C:\Users\jordy\Projects\
↪pulsar_...
```

See also:

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

`synchronization.ipynb`

1.15 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 Pulsar QCM and one Pulsar QRM and we will be acquiring waveforms sequenced by the Pulsar QCM using the Pulsar QRM. By synchronizing the two instruments using the SYNQ technology, timing the acquisition of the waveforms becomes trivial.

For this tutorial to work, we need to connect both instruments to the same network, connect the REF^{out} of the Pulsar QCM to the REF^{in} of the Pulsar QRM using a 50cm coaxial cable, connect their SYNQ ports using the SYNQ cable and finally connect $\text{O}^{[1-2]}$ of the Pulsar QCM to $\text{I}^{[1-2]}$ of the Pulsar QRM respectively.

This tutorial is designed with the Pulsar QCM as output instrument in mind, but the Pulsar QCM can easily be swapped with another Pulsar QRM as well. Just change the Pulsar QCM instantiation to a Pulsar QRM instantiation.

1.15.1 Setup

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

[1]:
```
#Set up the environment.
import pprint
import os
import scipy.signal
import math
import json
import matplotlib.pyplot
import numpy

from pulsar_qcm.pulsar_qcm import pulsar_qcm
from pulsar_qrm.pulsar_qrm import pulsar_qrm

#Connect to the Pulsar QCM at default IP address.
pulsar_qcm = pulsar_qcm("qcm", "192.168.0.2")

#Reset the Pulsar QCM for good measure.
pulsar_qcm.reset()
print("QCM status:")
print(pulsar_qcm.get_system_status())
print()
```

(continues on next page)

(continued from previous page)

```
#Connect to the Pulsar QRM at alternate address.
pulsar_qrm = pulsar_qrm("qrm", "192.168.0.3")

#Reset the Pulsar QRM for good measure.
pulsar_qrm.reset()
print("QRM status:")
print(pulsar_qrm.get_system_status())
```

```
QCM status:
{'status': 'OKAY', 'flags': []}

QRM status:
{'status': 'OKAY', 'flags': []}
```

We also need to configure the reference clock sources of the instruments. The Pulsar QCM is used as the overal reference source and needs to be configured to use its internal reference clock (the default setting). The Pulsar QRM will use the Pulsar QCM's reference clock and needs to be configured to use the external reference clock source.

[2]:
```
#Set reference clock source.
pulsar_qrm.reference_source("external")
```

1.15.2 Generate waveforms

Next, we need to create the waveforms for the sequence.

[3]:
```
#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": [], "index": 0},
                "sine":     {"data": [], "index": 1}
            }

#Create gaussian waveform
if "gaussian" in waveforms:
    waveforms["gaussian"]["data"] = scipy.signal.gaussian(waveform_length,␣
↪std=0.12 * waveform_length)

#Create sine waveform
if "sine" in waveforms:
    waveforms["sine"]["data"] = [math.sin((2*math.pi/waveform_length)*i) for i␣
↪in range(0, waveform_length)]
```

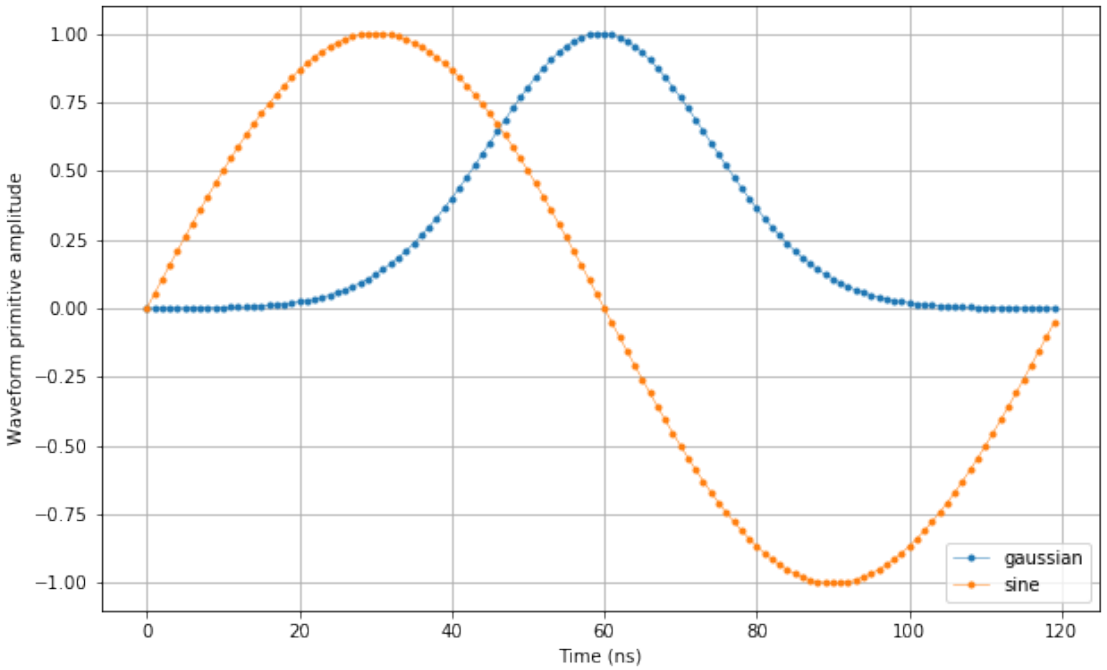
Let's plot the waveforms to see what we have created.

[4]:
```
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))
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("Waveform primitive amplitude")
ax.set_xlabel("Time (ns)")

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()

matplotlib.pyplot.draw()
matplotlib.pyplot.show()   # add this at EOF to prevent execution stall
```



1.15.3 Create Q1ASM programs

Now that we have the waveforms for the sequence, we need a simple Q1ASM program that sequences the waveforms in the Pulsar QCM and acquires the waveforms in the Pulsar QRM.

[5]:
```
#Pulsar QCM sequence program.
qcm_seq_prog = """
wait_sync 4         #Synchronize sequencers over multiple instruments.
play      0,1,16384 #Play waveforms and wait duration of acquisition.
stop                #Stop.
"""

#Pulsar QRM sequence program.
qrm_seq_prog = """
wait_sync 4         #Synchronize sequencers over multiple instruments.
acquire   0,0,16384 #Acquire waveforms and wait duration of acquisition.
```

(continues on next page)

(continued from previous page)

```
stop                #Stop.
"""
```

1.15.4 Upload sequences

Now that we have the waveforms and Q1ASM programs, we can combine them in the sequences stored in JSON files.

[6]:
```
#Reformat waveforms to lists if necessary.
for name in waveforms:
    if str(type(waveforms[name]["data"]).__name__) == "ndarray":
        waveforms[name]["data"] = waveforms[name]["data"].tolist()  # JSON␣
↪only supports lists

#Add QCM sequence program and waveforms to single dictionary and write to JSON␣
↪file.
wave_and_prog_dict = {"waveforms": {"awg": waveforms}, "program": qcm_seq_prog}
with open("qcm_sequence.json", 'w', encoding='utf-8') as file:
    json.dump(wave_and_prog_dict, file, indent=4)
    file.close()

#Add QRM sequence program and waveforms to single dictionary and write to JSON␣
↪file.
wave_and_prog_dict = {"waveforms": {"acq": waveforms}, "program": qrm_seq_prog}
with open("qrm_sequence.json", 'w', encoding='utf-8') as file:
    json.dump(wave_and_prog_dict, file, indent=4)
    file.close()
```

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

[7]:
```
#Upload waveforms and programs to Pulsar QCM.
pulsar_qcm.sequencer0_waveforms_and_program(os.path.join(os.getcwd(), "qcm_
↪sequence.json"))

#Upload waveforms and programs to Pulsar QRM.
pulsar_qrm.sequencer0_waveforms_and_program(os.path.join(os.getcwd(), "qrm_
↪sequence.json"))
```

1.15.5 Play sequences

The sequence has been uploaded to the instruments. Now we need to configure the sequencers of both the Pulsar QCM and Pulsar QRM to use the `wait_sync` instruction to synchronize and we need to configure the sequencer of the Pulsar QRM to trigger the acquisition with the `acquire` instruction. Furthermore we also need to attenuate the Pulsar QCM's outputs to 40% to be able to capture the full range of the waveforms on the Pulsar QRM's inputs.

Attenuation $= Input/Output = 2V/5V = 0.4$

[8]:
```
#Configure the sequencer of the Pulsar QCM.
pulsar_qcm.sequencer0_sync_en(True)
pulsar_qcm.sequencer0_gain_awg_path0(0.35) #Adding a bit of margin to the 0.4
pulsar_qcm.sequencer0_gain_awg_path1(0.35)

#Configure the sequencer of the Pulsar QRM.
pulsar_qrm.sequencer0_sync_en(True)
pulsar_qrm.sequencer0_trigger_mode_acq_path0("sequencer")
pulsar_qrm.sequencer0_trigger_mode_acq_path1("sequencer")
```

Now let's start the sequences.

[9]:
```
#Arm and start sequencer of the Pulsar QCM (only sequencer 0).
pulsar_qcm.arm_sequencer(0)
pulsar_qcm.start_sequencer(0)

#Print status of sequencer of the Pulsar QCM.
print("QCM status:")
print(pulsar_qcm.get_sequencer_state(0))
print()

#Arm and start sequencer of the Pulsar QRM (only sequencer 0).
pulsar_qrm.arm_sequencer(0)
pulsar_qrm.start_sequencer(0)

#Print status of sequencer of the Pulsar QRM.
print("QRM status:")
print(pulsar_qrm.get_sequencer_state(0))
```

```
QCM status:
{'status': 'Q1 STOPPED', 'flags': []}

QRM status:
{'status': 'STOPPED', 'flags': ['ACQ WAVE CAPTURE DONE PATH 0', 'ACQ WAVE␣
↪CAPTURE DONE PATH 1']}
```

1.15.6 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.

[10]:
```
#Wait for the sequencers to stop with a timeout period of one second.
pulsar_qcm.get_sequencer_state(0, 1)
pulsar_qrm.get_sequencer_state(0, 1)

#Wait for the acquisition to finish with a timeout period of one second.
pulsar_qrm.get_acquisition_state(0, 1)

#Move acquisition data from temporary memory to acquisition list.
pulsar_qrm.store_acquisition(0, "measurement")
```

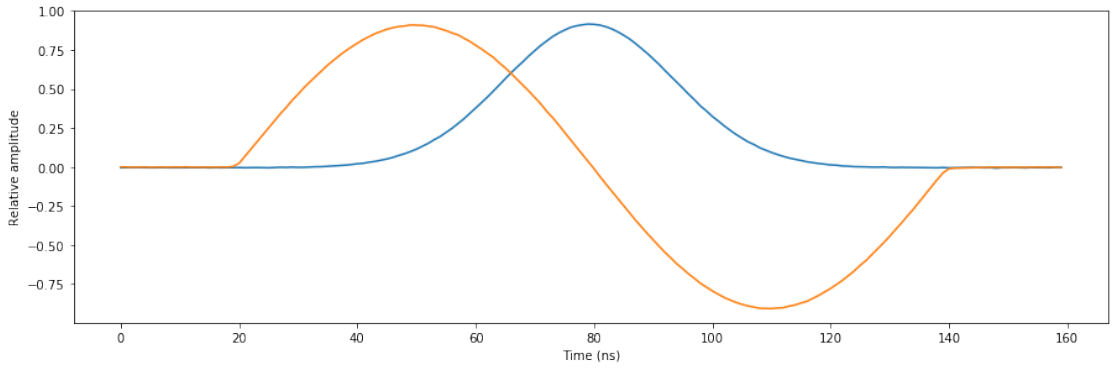
(continues on next page)

(continued from previous page)

```
#Get acquisition list from instrument.
acq = pulsar_qrm.get_acquisitions(0)
```

Let's plot the result.

[11]:
```
#Plot acquired signal on both inputs.
fig, ax = matplotlib.pyplot.subplots(1, 1, figsize=(15, 15/2/1.61))
ax.plot(acq["measurement"]["path_0"]["data"][100:260])
ax.plot(acq["measurement"]["path_1"]["data"][100:260])
ax.set_xlabel('Time (ns)')
ax.set_ylabel('Relative amplitude')
matplotlib.pyplot.show()
```



1.15.7 Stop

Finally, let's stop the sequencers if they haven't already and close the instrument connections.

[12]:
```
#Stop sequencers.
pulsar_qcm.stop_sequencer()
pulsar_qrm.stop_sequencer()

#Print status of sequencers.
print("QCM status:")
print(pulsar_qcm.get_sequencer_state(0))
print()

print("QRM status:")
print(pulsar_qrm.get_sequencer_state(0))
print()

#Print an overview of instrument parameters.
print("QCM snapshot:")
pulsar_qcm.print_readable_snapshot(update=True)
print()

print("QRM snapshot:")
pulsar_qrm.print_readable_snapshot(update=True)

#Close the instrument connections.
```

(continues on next page)

(continued from previous page)

```
pulsar_qcm.close()
pulsar_qrm.close()
```

```
QCM status:
{'status': 'STOPPED', 'flags': ['FORCED STOP']}

QRM status:
{'status': 'STOPPED', 'flags': ['FORCED STOP', 'ACQ WAVE CAPTURE DONE PATH 0',
↪'ACQ WAVE CAPTURE DONE PATH 1']}

QCM snapshot:
qcm:
        parameter                                   value
--------------------------------------------------------------------------------
↪-
IDN                                       :   {'manufacturer': 'Qblox',
↪'devi...
reference_source                          :  internal
sequencer0_cont_mode_en_awg_path0         :      False
sequencer0_cont_mode_en_awg_path1         :      False
sequencer0_cont_mode_waveform_idx_awg_path0 :     0
sequencer0_cont_mode_waveform_idx_awg_path1 :     0
sequencer0_gain_awg_path0                 :        0.34999
sequencer0_gain_awg_path1                 :        0.34999
sequencer0_marker_ovr_en                  :        False
sequencer0_marker_ovr_value               :        0
sequencer0_mod_en_awg                     :        False
sequencer0_nco_freq                       : 0 (Hz)
sequencer0_nco_phase_offs                 :        0 (Degrees)
sequencer0_offset_awg_path0               :        0
sequencer0_offset_awg_path1               :        0
sequencer0_sync_en                        : True
sequencer0_upsample_rate_awg_path0        :       0
sequencer0_upsample_rate_awg_path1        :       0
sequencer0_waveforms_and_program          :        C:\Users\jordy\Projects\
↪pulsar_...
sequencer1_cont_mode_en_awg_path0         :      False
sequencer1_cont_mode_en_awg_path1         :      False
sequencer1_cont_mode_waveform_idx_awg_path0 :     0
sequencer1_cont_mode_waveform_idx_awg_path1 :     0
sequencer1_gain_awg_path0                 :        1
sequencer1_gain_awg_path1                 :        1
sequencer1_marker_ovr_en                  :        False
sequencer1_marker_ovr_value               :        0
sequencer1_mod_en_awg                     :        False
sequencer1_nco_freq                       : 0 (Hz)
sequencer1_nco_phase_offs                 :        0 (Degrees)
sequencer1_offset_awg_path0               :        0
sequencer1_offset_awg_path1               :        0
sequencer1_sync_en                        : False
sequencer1_upsample_rate_awg_path0        :       0
sequencer1_upsample_rate_awg_path1        :       0
```

(continues on next page)

(continued from previous page)

```
sequencer1_waveforms_and_program             :        None

QRM snapshot:
qrm:
        parameter                                      value
--------------------------------------------------------------------------------
↪-
IDN                                          :   {'manufacturer': 'Qblox',
↪'devi...
in0_amp_gain                                 : -6 (dB)
in1_amp_gain                                 : -6 (dB)
reference_source                             :  external
sequencer0_avg_mode_en_acq_path0             :      False
sequencer0_avg_mode_en_acq_path1             :      False
sequencer0_cont_mode_en_awg_path0            :      False
sequencer0_cont_mode_en_awg_path1            :      False
sequencer0_cont_mode_waveform_idx_awg_path0  :     0
sequencer0_cont_mode_waveform_idx_awg_path1  :     0
sequencer0_gain_awg_path0                    :        1
sequencer0_gain_awg_path1                    :        1
sequencer0_marker_ovr_en                     :        False
sequencer0_marker_ovr_value                  :        0
sequencer0_mod_en_awg                        :        False
sequencer0_nco_freq                          : 0 (Hz)
sequencer0_nco_phase_offs                    :        0 (Degrees)
sequencer0_offset_awg_path0                  :        0
sequencer0_offset_awg_path1                  :        0
sequencer0_sync_en                           : True
sequencer0_trigger_level_acq_path0           :       0
sequencer0_trigger_level_acq_path1           :       0
sequencer0_trigger_mode_acq_path0            :       sequencer
sequencer0_trigger_mode_acq_path1            :       sequencer
sequencer0_upsample_rate_awg_path0           :       0
sequencer0_upsample_rate_awg_path1           :       0
sequencer0_waveforms_and_program             :        C:\Users\jordy\Projects\
↪pulsar_...
```

1.16 Pulsar QCM

The Pulsar QCM 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.

1.16.1 QCoDeS driver

class pulsar_qcm.pulsar_qcm.**pulsar_qcm_qcodes**(*name*, *transport_inst*, *debug=0*)

Bases: *pulsar_qcm.pulsar_qcm_ifc.pulsar_qcm_ifc*, qcodes.instrument.base.Instrument

This class connects QCoDeS to the Pulsar QCM native interface. Do not directly instantiate this class, but instead instantiate either the *pulsar_qcm* or *pulsar_qcm_dummy*.

__init__(*name*, *transport_inst*, *debug=0*)

Creates Pulsar QCM 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.
- **transport_inst** (*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 **Exception** – Debug level is 0 and there is a version mismatch.

Note: To get a complete of list of the QCoDeS parameters, run the following code.

```
from pulsar_qcm.pulsar_qcm import pulsar_qcm_dummy

qcm = pulsar_qcm_dummy("qrm")
for call in qcm.snapshot()['parameters']:
    print(getattr(qcm, call).__doc__)
```

class pulsar_qcm.pulsar_qcm.**pulsar_qcm**(*name*, *host*, *port=5025*, *debug=0*)

Bases: *pulsar_qcm.pulsar_qcm.pulsar_qcm_qcodes*

Pulsar QCM driver class based on QCoDeS that uses an IP socket to communicate with the instrument.

__init__(*name*, *host*, *port=5025*, *debug=0*)

Creates Pulsar QCM driver object.

Parameters

- **name** (*str*) – Instrument name.
- **host** (*str*) – Instrument IP address.
- **port** (*int*) – Instrument port.
- **debug** (*int*) – Debug level (0 = normal, 1 = no version check, >1 = no version or error checking).

Raises **Exception** – Debug level is 0 and there is a version mismatch.

class pulsar_qcm.pulsar_qcm.**pulsar_qcm_dummy**(*name*, *debug=1*)

Bases: *pulsar_qcm.pulsar_qcm.pulsar_qcm_qcodes*

Pulsar QCM driver class based on QCoDeS that uses the *pulsar_dummy_transport* layer to substitute an actual Pulsar QCM to allow software stack development without hardware.

__init__(*name*, *debug=1*)

Creates Pulsar QCM driver object. The debug level must be set to >= 1.

Parameters

- **name** (*str*) – Instrument name.
- **debug** (*int*) – Debug level (0 = normal, 1 = no version check, >1 = no version or error checking).

QCoDeS parameters

QCoDeS parameters generated by *pulsar_qcm_qcodes*.

Note: Only sequencer 0's parameters are listed, but all other sequencers have the same parameters.

pulsar_qcm.IDN

Please see QCoDeS for a description.

Properties

- **value**: Anything

pulsar_qcm.reference_source

Sets/gets reference source ('internal' = internal 10 MHz, 'external' = external 10 MHz).

Properties

- **value**: Enum: {'external', 'internal'}

pulsar_qcm.sequencer0_sync_en

Sets/gets sequencer 0 synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

pulsar_qcm.sequencer0_nco_freq

Sets/gets sequencer 0 NCO frequency in Hz with a resolution of 0.25 Hz

Properties

- **unit**: Hz
- **value**: Numbers -300000000.0<=v<=300000000.0

pulsar_qcm.sequencer0_nco_phase_offs

Sets/gets sequencer 0 NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

pulsar_qcm.sequencer0_marker_ovr_en

Sets/gets sequencer 0 marker override enable.

Properties

- **value**: Boolean

pulsar_qcm.sequencer0_marker_ovr_value

Sets/gets sequencer 0 marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

pulsar_qcm.sequencer0_waveforms_and_program

Sets sequencer 0 AWG waveforms and ASM program. Valid input is a string representing the JSON filename.

Properties

- **value**: Strings

pulsar_qcm.sequencer0_cont_mode_en_awg_path0

Sets/gets sequencer 0 continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

pulsar_qcm.sequencer0_cont_mode_en_awg_path1

Sets/gets sequencer 0 continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

pulsar_qcm.sequencer0_cont_mode_waveform_idx_awg_path0

Sets/gets sequencer 0 continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

pulsar_qcm.sequencer0_cont_mode_waveform_idx_awg_path1

Sets/gets sequencer 0 continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

pulsar_qcm.sequencer0_upsample_rate_awg_path0

Sets/gets sequencer 0 upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

pulsar_qcm.sequencer0_upsample_rate_awg_path1

Sets/gets sequencer 0 upsample rate for AWG path 1

Properties

- **value**: Numbers 0<=v<=65535

pulsar_qcm.sequencer0_gain_awg_path0

Sets/gets sequencer 0 gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qcm.sequencer0_gain_awg_path1

Sets/gets sequencer 0 gain for AWG path 1

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qcm.sequencer0_offset_awg_path0

Sets/gets sequencer 0 offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qcm.sequencer0_offset_awg_path1

Sets/gets sequencer 0 offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qcm.sequencer0_mod_en_awg

Sets/gets sequencer 0 modulation enable for AWG.

Properties

- **value**: Boolean

1.16.2 Native interface

class pulsar_qcm.pulsar_qcm_ifc.**pulsar_qcm_ifc**(*transport_inst*, *debug=0*)

Bases: *pulsar_qcm.pulsar_qcm_scpi_ifc.pulsar_qcm_scpi_ifc*

Class that provides the native API for the Pulsar QCM. It provides methods to control all functions and features provided by the Pulsar QCM, like sequencer and waveform handling.

__init__(*transport_inst*, *debug=0*)

Creates Pulsar QCM native interface object.

Parameters

- **transport_inst** (*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).

Returns Pulsar QCM native interface object.

Return type *pulsar_qcm_ifc*

Raises **Exception** – Debug level is 0 and there is a version mismatch.

get_idn()

Get device identity and build information.

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

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_system_status()

Get general system state.

Returns

Dictionary containing general status and corresponding flags:

Status

- 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.

Flags

- CARRIER_PLL_UNLOCK: Carrier board PLL is unlocked.

- FPGA_PLL_UNLOCK: FPGA PLL is unlocked.
- FPGA_TEMP_OR: FPGA temperature is out-of-range.
- CARRIER_TEMP_OR: Carrier board temperature is out-of-range.
- AFE_TEMP_OR: Analog frontend board temperature is out-of-range.

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.

arm_sequencer(*sequencer=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** (*int*) – Sequencer index.

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_sequencer(*sequencer=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** (*int*) – Sequencer index.

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.

stop_sequencer(*sequencer=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** (*int*) – Sequencer index.

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_sequencer_state(*sequencer*, *timeout=0*, *timeout_poll_res=0.1*)

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 timeout exception is thrown.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.

- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.

Returns

Concatinated list of strings separated by the semicolon character. Status is indicated by one status string and an optional number of flags respectively ordered as:

Status

- 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.

Flags

- 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.
- AWG WAVE PLAYBACK INVALID PATH 0: AWG path 0 tried to play an unknown waveform.
- AWG WAVE PLAYBACK INVALID PATH 1: AWG path 1 tried to play an unknown waveform.
- CLOCK INSTABILITY: Clock source instability occurred.

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.
- **TypeError** – Timeout is not an integer.
- **TimeoutError** – Timeout

get_waveforms(*sequencer*)

Get all waveforms in AWG waveform list of indexed sequencer. The returned dictionary is structured as follows:

- awg
  - 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.

Returns Dictionary with waveforms.

Return type dict

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.

1.16.3 SCPI interface

class pulsar_qcm.pulsar_qcm_scpi_ifc.**pulsar_qcm_scpi_ifc**(*transport_inst*, *debug=0*)
Bases: *ieee488_2.ieee488_2.ieee488_2*

This interface provides an API for the mandatory and required SCPI calls and adds Pulsar related functionality (see SCPI).

__init__(*transport_inst*, *debug=0*)
Creates SCPI interface object.

Parameters

- **transport_inst** (*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).

Returns SCPI interface object.

Return type *pulsar_qcm_scpi_ifc*

Raises **Exception** – Debug level is 0 and there is a version mismatch.

reset()
Reset device and clear all status and event registers (see SCPI).

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()
Clear all status and event registers (see SCPI).

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()
Get status byte register. Register is only cleared when feeding registers are cleared (see SCPI).

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*)
Set service request enable register (see SCPI).

Parameters **reg** (*int*) – Service request enable register.

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()
Get service request enable register. The register is cleared after reading it (see SCPI).

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*)
Get standard event status enable register. The register is cleared after reading it (see SCPI).

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_enable()
Set standard event status enable register (see SCPI).

Parameters **reg** (*int*) – Standard event status enable register.

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()
Get standard event status register. The register is cleared after reading it (see SCPI).

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()
Set device in operation complete query active state (see SCPI).

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()
Get operation complete state (see SCPI).

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()
Run self-test. Currently not implemented (see SCPI).

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()
Wait until operations completed before continuing (see SCPI).

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_system_error()
Get system error from queue (see SCPI).

Returns System error description string.

Return type str

get_num_system_error()
Get number of system errors (see SCPI).

Returns Current number of number of system errors.

Return type int

get_system_version()
Get SCPI system version (see SCPI).

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.

preset_system_status()
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).

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()
Get status questionable condition register (see SCPI).

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()
Get status questionable event register (see SCPI).

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*)
Set status questionable enable register (see SCPI).

Parameters **reg** (*int*) – Status questionable enable register.

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()
Get status questionable enable register (see SCPI).

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_current_fpga_temperature()
Get current FPGA junction temperature (inside device).

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()
Get maximum FPGA junction temperature since boot or clear (inside device).

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()
Get current carrier board temperature (inside device).

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()
Get maximum carrier board temperature since boot or clear (inside device).

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_afe_temperature()
Get current analog frontend board temperature (inside device).

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()
Get maximum analog frontend board temperature since boot or clear (inside device).

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_operation_condition()
Get status operation condition register (see SCPI).

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()
Get status operation event register (see SCPI).

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*)
Set status operation enable register (see SCPI).

Parameters **reg** (*int*) – Status operation enable register.

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()
Get status operation enable register (see SCPI).

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.

get_assembler_status()
Get assembler status. Refer to the assembler log to get more information regarding the assembler result.

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()
Get assembler log.

Returns Formatted assembler log string.

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.

1.17 Pulsar QRM

The Pulsar QRM 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.

1.17.1 QCoDeS driver

class pulsar_qrm.pulsar_qrm.**pulsar_qrm_qcodes**(*name*, *transport_inst*, *debug=0*)
Bases: *pulsar_qrm.pulsar_qrm_ifc.pulsar_qrm_ifc*, qcodes.instrument.base.Instrument

This class connects QCoDeS to the Pulsar QRM native interface. Do not directly instantiate this class, but instead instantiate either the *pulsar_qrm* or *pulsar_qrm_dummy*.

__init__(*name*, *transport_inst*, *debug=0*)
Creates Pulsar QRM 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.
- **transport_inst** (*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 **Exception** – Debug level is 0 and there is a version mismatch.

Note: To get a complete of list of the QCoDeS parameters, run the following code.

```
from pulsar_qrm.pulsar_qrm import pulsar_qrm_dummy

qrm = pulsar_qrm_dummy("qrm")
for call in qrm.snapshot()['parameters']:
    print(getattr(qrm, call).__doc__)
```

class pulsar_qrm.pulsar_qrm.**pulsar_qrm**(*name*, *host*, *port=5025*, *debug=0*)
Bases: *pulsar_qrm.pulsar_qrm.pulsar_qrm_qcodes*

Pulsar QRM driver class based on QCoDeS that uses an IP socket to communicate with the instrument.

__init__(*name*, *host*, *port=5025*, *debug=0*)
Creates Pulsar QRM driver object.

Parameters

- **name** (*str*) – Instrument name.
- **host** (*str*) – Instrument IP address.
- **port** (*int*) – Instrument port.
- **debug** (*int*) – Debug level (0 = normal, 1 = no version check, >1 = no version or error checking).

Raises **Exception** – Debug level is 0 and there is a version mismatch.

class pulsar_qrm.pulsar_qrm.**pulsar_qrm_dummy**(*name*, *debug=1*)
Bases: *pulsar_qrm.pulsar_qrm.pulsar_qrm_qcodes*

Pulsar QRM driver class based on QCoDeS that uses the *pulsar_dummy_transport* layer to substitute an actual Pulsar QRM to allow software stack development without hardware.

__init__(*name*, *debug=1*)
Creates Pulsar QRM driver object. The debug level must be set to >= 1.

Parameters

- **name** (*str*) – Instrument name.
- **debug** (*int*) – Debug level (0 = normal, 1 = no version check, >1 = no version or error checking).

QCoDeS parameters

QCoDeS parameters generated by *pulsar_qrm_qcodes*.

Note: Only sequencer 0's parameters are listed, but all other sequencers have the same parameters.

pulsar_qrm.IDN

Please see QCoDeS for a description.

Properties

- **value**: Anything

pulsar_qrm.reference_source

Sets/gets reference source ('internal' = internal 10 MHz, 'external' = external 10 MHz).

Properties

- **value**: Enum: {'external', 'internal'}

pulsar_qrm.in0_amp_gain

Sets/gets input 0 amplifier 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.in1_amp_gain

Sets/gets input 1 amplifier 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.sequencer0_sync_en

Sets/gets sequencer 0 synchronization enable which enables party-line synchronization.

Properties

- **value**: Boolean

pulsar_qrm.sequencer0_nco_freq

Sets/gets sequencer 0 NCO frequency in Hz with a resolution of 0.25 Hz.

Properties

- **unit**: Hz
- **value**: Numbers -300000000.0<=v<=300000000.0

pulsar_qrm.sequencer0_nco_phase_offs

Sets/gets sequencer 0 NCO phase offset in degrees with a resolution of 3.6e-7 degrees.

Properties

- **unit**: Degrees
- **value**: Numbers 0<=v<=360

pulsar_qrm.sequencer0_marker_ovr_en

Sets/gets sequencer 0 marker override enable.

Properties

- **value**: Boolean

pulsar_qrm.sequencer0_marker_ovr_value

Sets/gets sequencer 0 marker override value. Bit index corresponds to marker channel index.

Properties

- **value**: Numbers 0<=v<=15

pulsar_qrm.sequencer0_waveforms_and_program

Sets sequencer 0 AWG and acquistion waveforms and ASM program. Valid input is a string representing the JSON filename.

Properties

- **value**: Strings

pulsar_qrm.sequencer0_cont_mode_en_awg_path0

Sets/gets sequencer 0 continous waveform mode enable for AWG path 0.

Properties

- **value**: Boolean

pulsar_qrm.sequencer0_cont_mode_en_awg_path1

Sets/gets sequencer 0 continous waveform mode enable for AWG path 1.

Properties

- **value**: Boolean

pulsar_qrm.sequencer0_cont_mode_waveform_idx_awg_path0

Sets/gets sequencer 0 continous waveform mode waveform index or AWG path 0.

Properties

- **value**: Numbers 0<=v<=1023

pulsar_qrm.sequencer0_cont_mode_waveform_idx_awg_path1

Sets/gets sequencer 0 continous waveform mode waveform index or AWG path 1.

Properties

- **value**: Numbers 0<=v<=1023

pulsar_qrm.sequencer0_upsample_rate_awg_path0

Sets/gets sequencer 0 upsample rate for AWG path 0.

Properties

- **value**: Numbers 0<=v<=65535

pulsar_qrm.sequencer0_upsample_rate_awg_path1

Sets/gets sequencer 0 upsample rate for AWG path 1.

Properties

- **value**: Numbers 0<=v<=65535

pulsar_qrm.sequencer0_gain_awg_path0

Sets/gets sequencer 0 gain for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qrm.sequencer0_gain_awg_path1

Sets/gets sequencer 0 gain for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qrm.sequencer0_offset_awg_path0

Sets/gets sequencer 0 offset for AWG path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qrm.sequencer0_offset_awg_path1

Sets/gets sequencer 0 offset for AWG path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qrm.sequencer0_mod_en_awg

Sets/gets sequencer 0 modulation enable for AWG.

Properties

- **value**: Boolean

pulsar_qrm.sequencer0_trigger_mode_acq_path0

Sets/gets sequencer 0 trigger mode for acquisition path 0 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

pulsar_qrm.sequencer0_trigger_mode_acq_path1

Sets/gets sequencer 0 trigger mode for acquisition path 1 ('sequencer' = triggered by sequencer, 'level' = triggered by input level).

Properties

- **value**: Enum: {'level', 'sequencer'}

pulsar_qrm.sequencer0_trigger_level_acq_path0

Sets/gets sequencer 0 trigger level when using input level trigger mode for acquisition path 0.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qrm.sequencer0_trigger_level_acq_path1

Sets/gets sequencer 0 trigger level when using input level trigger mode for acquisition path 1.

Properties

- **value**: Numbers -1.0<=v<=1.0

pulsar_qrm.sequencer0_avg_mode_en_acq_path0

Sets/gets sequencer 0 averaging mode enable for acquisition path 0.

Properties

- **value**: Boolean

pulsar_qrm.sequencer0_avg_mode_en_acq_path1

Sets/gets sequencer 0 averaging mode enable for acquisition path 1.

Properties

- **value**: Boolean

1.17.2 Native interface

class pulsar_qrm.pulsar_qrm_ifc.**pulsar_qrm_ifc**(*transport_inst*, *debug=0*)

Bases: *pulsar_qrm.pulsar_qrm_scpi_ifc.pulsar_qrm_scpi_ifc*

Class that provides the native API for the Pulsar QrM. It provides methods to control all functions and features provided by the Pulsar QRM, like sequencer, waveform and acquistion handling.

__init__(*transport_inst*, *debug=0*)

Creates Pulsar QRM native interface object.

Parameters

- **transport_inst** (*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).

Returns Pulsar QRM native interface object.

Return type *pulsar_qrm_ifc*

Raises **Exception** – Debug level is 0 and there is a version mismatch.

get_idn()

Get device identity and build information.

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

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_system_status()`
Get general system state.

Returns

Dictionary containing general status and corresponding flags:

Status

- 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.

Flags

- CARRIER_PLL_UNLOCK: Carrier board PLL is unlocked.
- FPGA_PLL_UNLOCK: FPGA PLL is unlocked.
- FPGA_TEMP_OR: FPGA temperature is out-of-range.
- CARRIER_TEMP_OR: Carrier board temperature is out-of-range.
- AFE_TEMP_OR: Analog frontend board temperature is out-of-range.

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.

`arm_sequencer(sequencer=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` (*int*) – Sequencer index.

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_sequencer(sequencer=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` (*int*) – Sequencer index.

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.

stop_sequencer(*sequencer=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** (*int*) – Sequencer index.

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_sequencer_state(*sequencer*, *timeout=0*, *timeout_poll_res=0.1*)

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 timeout exception is thrown.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.

Returns

Concatinated list of strings separated by the semicolon character. Status is indicated by one status string and an optional number of flags respectively ordered as:

Status

- 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.

Flags

- 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.
- AWG WAVE PLAYBACK INVALID PATH 0: AWG path 0 tried to play an unknown waveform.

- AWG WAVE PLAYBACK INVALID PATH 1: AWG path 1 tried to play an unknown waveform.
- ACQ WAVE PLAYBACK INVALID PATH 0: Acquisition path 0 tried to play an unknown waveform.
- ACQ WAVE PLAYBACK INVALID PATH 1: Acquisition path 1 tried to play an unknown waveform.
- ACQ WAVE CAPTURE OUT-OF-RANGE PATH 0: Acquisition path 0 data was out-of-range.
- ACQ WAVE CAPTURE DONE PATH 0: Acquisition path 0 has finished.
- ACQ WAVE CAPTURE OVERWRITTEN PATH 0: Acquisition path 0 data was overwritten.
- ACQ WAVE CAPTURE OUT-OF-RANGE PATH 1: Acquisition path 1 data was out-of-range.
- ACQ WAVE CAPTURE DONE PATH 1: Acquisition path 1 has finished.
- ACQ WAVE CAPTURE OVERWRITTEN PATH 1: Acquisition path 1 data was overwritten.
- CLOCK INSTABILITY: Clock source instability occurred.

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.
- **TypeError** – Timeout is not an integer.
- **TimeoutError** – Timeout

get_waveforms(*sequencer*)

Get all waveforms in AWG and acquisition waveform lists of indexed sequencer. The returned dictionary is structured as follows:

- awg
  - 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.
- acq

  -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.

Returns Dictionary with waveforms.

Return type dict

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_acquisition_state(*sequencer*, *timeout=0*, *timeout_poll_res=0.1*)

Return acquisition completion state of both acquisition paths 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 completes on both paths. If the acquisition hasn't completed before the timeout expires, a timeout exception is thrown.

Parameters

- **sequencer** (*int*) – Sequencer index.
- **timeout** (*int*) – Timeout in minutes.
- **timeout_poll_res** (*float*) – Timeout polling resolution in seconds.

Returns Tuple of booleans indicating the acquisition completion state of both acquisition paths (False = uncompleted, True = completed).

Return type tuple

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.
- **TypeError** – Timeout is not an integer.
- **TimeoutError** – Timeout

store_acquisition(*sequencer*, *name*, *size=2147483648*)

After an acquisition has completed, store the results of both acquisition paths in the acquisition list of the indexed sequencers. If an invalid sequencer index is given an error is set in system error. The acquisition names 'all' and 'ALL' are reserved and adding waveforms with those names will also result in an error being set in system error. The size argument can be used to specify the number of samples to store of the acquisition results of both acquisition paths. If the size argument is not provided, all samples will be stored. 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.
- **size** (*int*) – Number of samples to store of both acquisition results.

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.

delete_acquisitions(*sequencer*)
Delete all acquisitions from acquisition list of indexed sequencer.

Parameters **sequencer** (*int*) – Sequencer index.

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_acquisitions(*sequencer*)
Get all acquisitions in acquisition lists of indexed sequencer. The returned dictionary is structured as follows:

- name: acquisition name
  - path_0: Input path 0
    - data: acquisition samples in a range of 1.0 to -1.0.
    - out-of-range: out-of-range indication for each acquisition sample (0 = in-range, 1 = out-of-range).
  - path_1: Input path 1
    - data: acquisition samples in a range of 1.0 to -1.0.
    - out-of-range: out-of-range indication for each acquisition sample (0 = in-range, 1 = out-of-range).

Parameters **sequencer** (*int*) – Sequencer index.

Returns Dictionary with acquisitions.

Return type dict

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.

1.17.3 SCPI interface

class pulsar_qrm.pulsar_qrm_scpi_ifc.**pulsar_qrm_scpi_ifc**(*transport_inst*, *debug=0*)
Bases: *ieee488_2.ieee488_2.ieee488_2*

This interface provides an API for the mandatory and required SCPI calls and adds Pulsar related functionality (see SCPI).

__init__(*transport_inst*, *debug=0*)
Creates SCPI interface object.

Parameters

- **transport_inst** (*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).

Returns SCPI interface object.

Return type *pulsar_qrm_scpi_ifc*

Raises **Exception** – Debug level is 0 and there is a version mismatch.

reset()
Reset device and clear all status and event registers (see SCPI).

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()
Clear all status and event registers (see SCPI).

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()
Get status byte register. Register is only cleared when feeding registers are cleared (see SCPI).

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*)
Set service request enable register (see SCPI).

Parameters **reg** (*int*) – Service request enable register.

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()
Get service request enable register. The register is cleared after reading it (see SCPI).

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*)
Get standard event status enable register. The register is cleared after reading it (see SCPI).

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_enable()
Set standard event status enable register (see SCPI).

Parameters **reg** (*int*) – Standard event status enable register.

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()
Get standard event status register. The register is cleared after reading it (see SCPI).

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()
Set device in operation complete query active state (see SCPI).

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()
Get operation complete state (see SCPI).

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()
Run self-test. Currently not implemented (see SCPI).

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()
Wait until operations completed before continuing (see SCPI).

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_system_error()
Get system error from queue (see SCPI).

Returns System error description string.

Return type str

get_num_system_error()
Get number of system errors (see SCPI).

Returns Current number of number of system errors.

Return type int

get_system_version()
Get SCPI system version (see SCPI).

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.

`preset_system_status()`
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).

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()`
Get status questionable condition register (see SCPI).

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()`
Get status questionable event register (see SCPI).

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)`
Set status questionable enable register (see SCPI).

Parameters `reg` (*int*) – Status questionable enable register.

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()`
Get status questionable enable register (see SCPI).

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_current_fpga_temperature()`
Get current FPGA junction temperature (inside device).

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()`
Get maximum FPGA junction temperature since boot or clear (inside device).

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()`
Get current carrier board temperature (inside device).

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()`
Get maximum carrier board temperature since boot or clear (inside device).

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_afe_temperature()`
Get current analog frontend board temperature (inside device).

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()`
Get maximum analog frontend board temperature since boot or clear (inside device).

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_operation_condition()`
Get status operation condition register (see SCPI).

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()`
Get status operation event register (see SCPI).

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*)
Set status operation enable register (see SCPI).

Parameters `reg` (*int*) – Status operation enable register.

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`()
Get status operation enable register (see SCPI).

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.

`get_assembler_status`()
Get assembler status. Refer to the assembler log to get more information regarding the assembler result.

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`()
Get assembler log.

Returns Formatted assembler log string.

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.

1.18 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)."

1.18.1 IEEE488.2 layer

class ieee488_2.ieee488_2.**ieee488_2**(*transport_inst: <module 'ieee488_2.transport' from '/home/docs/checkouts/readthedocs.org/user_builds/qblox-qblox-instruments/checkouts/v0.2.3/ieee488_2/transport.py'>*)

Bases: `object`

Class that implements the IEEE488.2 interface.

__init__(*transport_inst: <module 'ieee488_2.transport' from '/home/docs/checkouts/readthedocs.org/user_builds/qblox-qblox-instruments/checkouts/v0.2.3/ieee488_2/transport.py'>*) → None

Creates IEEE488.2 interface object.

Parameters **transport_inst** (*transport*) – Transport class responsible for the lowest level of communication (e.g. ethernet).

1.18.2 Transport layer

class ieee488_2.transport.**transport**

Bases: `object`

Abstract base class for data transport to instruments.

close() → None

Abstract method to close instrument.

write(*cmd_str: str*) → None

Abstract method to write command to instrument.

Parameters **cmd_str** (*str*) – Command

write_binary(*data: bytes*) → None

Abstract method to write binary data to instrument.

Parameters **data** (*bytes*) – Binary data

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

readline() → str

Abstract method to read data from instrument.

Returns String with data.

Return type str

class ieee488_2.transport.**ip_transport**(*host: str*, *port: int = 5025*, *timeout: float = 60.0*, *snd_buf_size: int = 524288*)

Bases: *ieee488_2.transport.transport*

Class for data transport of IP socket.

__init__(*host: str*, *port: int = 5025*, *timeout: float = 60.0*, *snd_buf_size: int = 524288*) → None
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 ieee488_2.transport.**file_transport**(*out_file_name: str*, *in_file_name: str = ''*)
Bases: *ieee488_2.transport.transport*

Class implementing file I/O to support driver testing.

__init__(*out_file_name: str*, *in_file_name: str = ''*) → None
Create file transport class.

Parameters

- **out_file_name** (*str*) – Output file name/path to write all commands to.
- **in_file_name** (*str*) – Input file name/path to read all command responses from.

close() → None
Close file descriptors.

write(*cmd_str: str*) → None
Write command to file.

Parameters **cmd_str** (*str*) – Command

write_binary(*data: bytes*) → None
Write binary data to file.

Parameters **data** (*bytes*) – Binary data

read_binary(*size: int*) → bytes
Read binary data from file.

Parameters **size** (*int*) – Number of bytes

Returns Binary data array of length "size".

Return type bytes

readline() → str
Read data from file.

Returns String with data.

Return type str

class ieee488_2.transport.**pulsar_dummy_transport**(*cfg_format: str*)
Bases: *ieee488_2.transport.transport*

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.

__init__(*cfg_format: str*) → None
Create Pulsar dummy transport class.

Parameters **cfg_format** (*str*) – Configuration format based on struct.pack format used to calculate configration transaction size.

close() → None
Close and resets Pulsar dummy transport class.

write(*cmd_str: str*) → None
Write command to Pulsar dummy. Stores command in command history (see *ieee488_2.transport.pulsar_dummy_transport.get_cmd_hist()*).

Parameters **cmd_str** (*str*) – Command

write_binary(*data: bytes*) → None
Write binary data to Pulsar dummy. Stores command in command history (see *ieee488_2.transport.pulsar_dummy_transport.get_cmd_hist()*).

Parameters **data** (*bytes*) – Binary data

read_binary(*size: int*) → bytes
Read binary data from Pulsar dummy.

Parameters **size** (*int*) – Number of bytes

Returns Binary data array of length "size".

Return type bytes

readline() → str
Read data from Pulsar dummy.

Returns String with data.

Return type str

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

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

p

INDEX

Symbols

A

C

D

F

G

I

M

P

R