qblox_instruments

Release 0.1

Qblox

Nov 23, 2021

CONTENTS

CHAPTER

ONE

README

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

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

1.1 About

Fig. 1: This software is free to use under the conditions specified in the license.

CHAPTER

TWO

INSTALLATION

Before installing the drivers, you must ensure you have Python 3.8 installed. This can be verified by running in your terminal:

```
$ python --version
```

To install the module drivers, run this command in your terminal:

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

This is the preferred method to install the drivers, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

You can confirm that the drivers are installed and the corresponding version with:

```
$ pip show qblox-instruments
```

After the installation head on to *Connecting to a module*.

CHAPTER

THREE

UPDATING THE MODULES

To update your Pulsar module go to qblox.com and download the Pulsar software from the download section. The Pulsar software includes the Qblox Configuration Manager with which you can configure and update your module. Once you have extracted the Pulsar software, go to the directory in which it was extracted using a command line terminal and execute the commands below using Python 3.8.

3.1 Software update

To update the software:

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

3.2 IP address update

To update the IP address:

```
$ python cfg_man.py -i 192.168.0.{new_ip_digit} 192.168.0.2
```

After executing the command, follow the instructions given by the Qblox Configuration Manager. The module will reboot once, after which the update is complete.

CHAPTER

FOUR

CONNECTING TO A MODULE

In this section you will learn how to connect for the first time to one of our instruments, with minimum background knowledge possible.

Before proceeding, if you did not already, please install the required drivers following the *Installation* section.

4.1 Connecting to a single module

Requirements:

- A Qblox module e.g. a Pulsar QCM or a Pulsar QRM
- **A host PC connected to a network adapter with at least one physical Ethernet port. For example:**
  - A laptop (the host PC) with an Ethernet port should be enough to communicate to single module
  - A USB or Thunderbolt Ethernet adapter
  - An Ethernet Switch
  - A router/modem with LAN ports

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

- A laptop (host PC)
- A USB Ethernet adapter
- A Qblox Pulsar QCM

The following step should allow you to successfully connect to a Qblox Pulsar QCM module, or a similar one.

1. Power up the QCM module. The module is ready when all LEDs become green/blue.
2. Connect the USB Ethernet adapter to your laptop.
3. Configure the network adapter, in this case the USB Ethernet adapter, such that its IP address shares the subnet `192.168.0.X` where `X` is a value between `2` and `255`. This configuration might be slightly different depending on your Operating System. See *Network adapter configuration* for common Operating Systems.
4. Connect the QCM module to the USB Ethernet adapter.

At this point your setup will look similar to:

Tip: After a few seconds, the QCM module should be available on the local network.

You can verify this with:

```
$ ping 192.168.0.2  # replace 192.168.0.2 accordingly if you have changed it
```

If successful the expected output is:

```
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
```

(continues on next page)

(continued from previous page)

```
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
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0.226 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=64 time=0.265 ms
```

press `ctrl + c` to stop.

5. Connect to the QCM module using a Python Interactive Shell or a Jupyter Notebook by running the following snippet (using Python 3.8):

```
from pulsar_qcm.pulsar_qcm import pulsar_qcm
qcm = pulsar_qcm("qcm", "192.168.0.2")
```

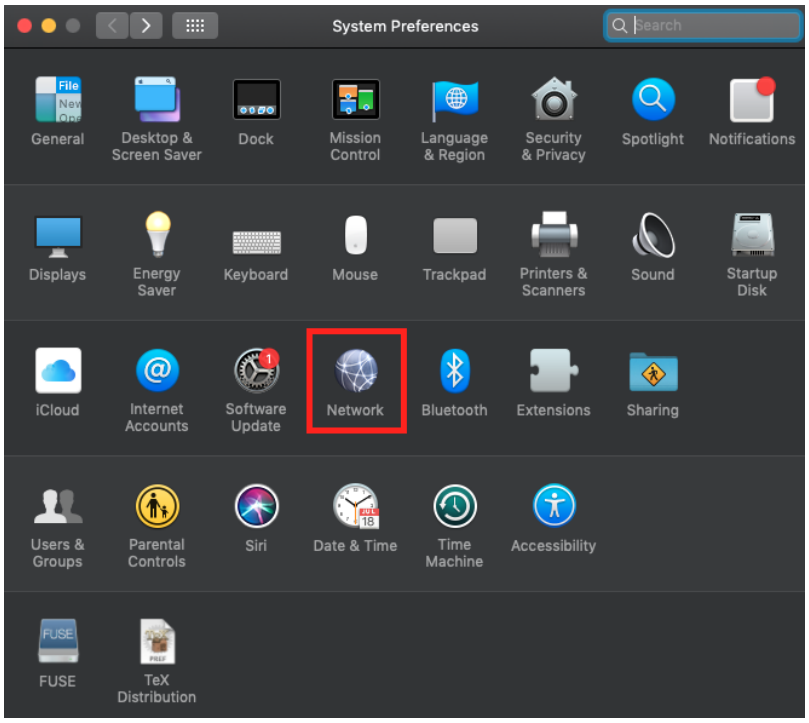
Note: The `192.168.0.2` is the default IP address of the modules, in case this was changed you must change the code accordingly.

Note: For developers and debugging purposes only, the driver allows to bypass important hardware/software compatibility checks with `pulsar_qcm("qcm", "192.168.0.2", debug=True)`. Do not use this option unless you have been instructed to do so!

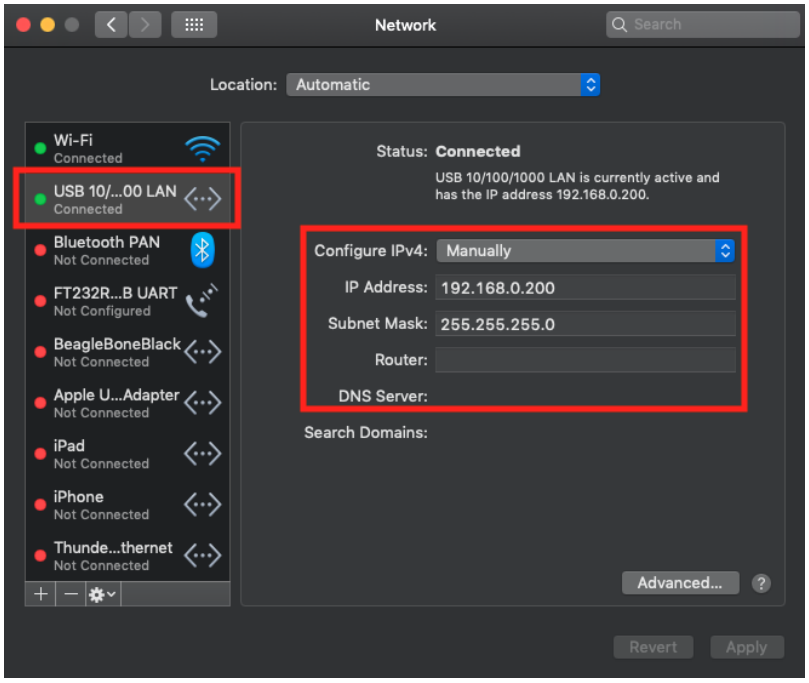
4.1.1 Network adapter configuration

macOS

1. Open *System Preferences* > *Network* and select the adapter that shares the same network as the Qblox module, e.g. (a USB Ethernet adapter)



2. Make sure the IP address has the form `192.168.0.X`, for example `192.168.0.200`. Depending on your setup, you will likely have to set *Configure IPv4* to `Manually` and specify a fixed *IP Address* as shown bellow.



Windows

Go to *Control Panel* > *Network and Internet* > *Network Connections* and follow a similar configuration as shown above for *macOS*.

Ubunto

Go to *System Settings* > *Connections* and follow a similar configuration as shown above for *macOS*.

4.2 Closing the device

As a good practice before closing the Jupyter Notebook (or ipython shell) we close the instantiated (QCoDeS) device:

```
qcm.close()
```

4.3 Connecting to multiple modules

In order to be able to control multiple modules, e.g. one Pulsar QCM and one Pulsar 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. Note that you still need to configure the network adapter of the host PC as described above.

- The IP address of the modules **must** be changed to avoid IP clashes. See *Updating the modules* section for instructions.

An exemplary setup is shown bellow:

The python code that allows to connect to these modules will look similar to:

```
# Import drivers
from pulsar_qcm.pulsar_qcm import pulsar_qcm
from pulsar_qrm.pulsar_qrm import pulsar_qrm

# Instantiate and 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")  # The default IP address was changed
```

Note: When employing multiple modules, depending on your experiment, you might need to synchronize them. This is achieved using the SYNC port and distributing the clock reference signal to all modules.

CHAPTER

FIVE

NEXT STEPS: INSTALLING QUANTIFY

To get the most out of your Pulsar module, we advise you to install Quantify: our measurement control package with built-in support for the Pulsar modules. You can do this by executing the following commands in a command line terminal:

```
$ python -m pip install quantify-core
$ python -m pip install quantify-scheduler
```

Please consult Quantify documentation for more information. We highly recommend following the Quantify tutorials.

CHAPTER

SIX

DEMOS

The hardware demos below are intended to showcase some of the functionalities of our devices with minimal software dependencies. However, these are not intended to be "getting started tutorials" since low-level interfaces to the firmware are being used. When building applications we highly recommend using Quantify framework.

6.1 QCM demo

In this demo we showcase the upload and continuous play of waveform primitives using the Qblox Pulsar QCM. In addition we observe the output on a oscilloscope.

NB: This hardware demo interacts with firmware low-level interfaces. When building applications we highly recommend using Quantify framework.

```
# Set up environment
import os
import scipy.signal
import math
import json
import time
```

6.1.1 Connect to the QCM

```
# Add Pulsar QCM interface
from pulsar_qcm.pulsar_qcm import pulsar_qcm
```

```
# Connect to device over Ethernet
pulsar = pulsar_qcm("qcm", "192.168.0.2")
```

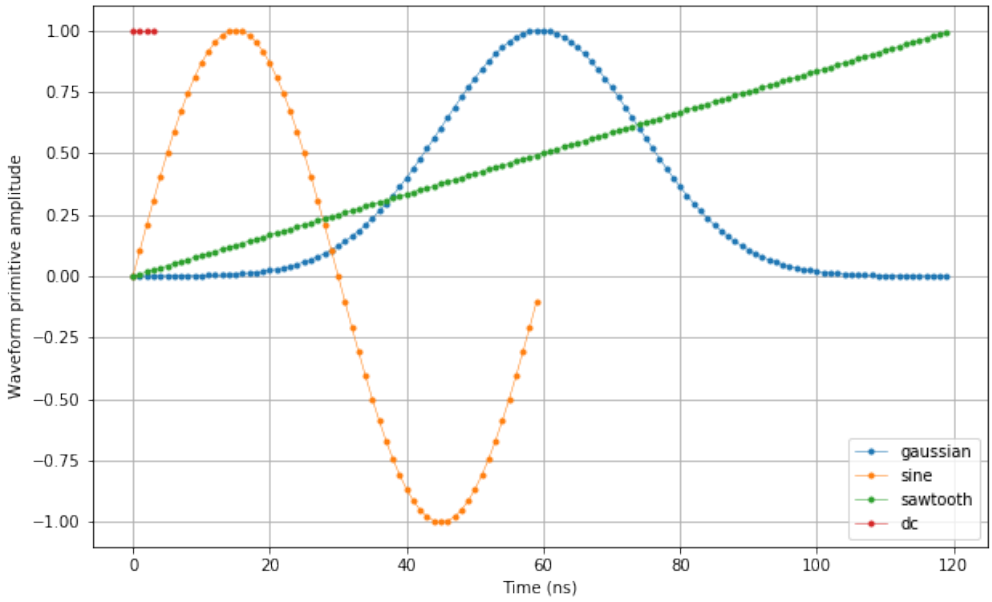
Tip: If you need to re-instantiate the instrument without re-starting the Jupyter Notebook (or the ipython shell) run the following command first `pulsar.close()`. This closes the QCoDeS instrument instance.

```
# Print device information
print(pulsar.get_idn())
# Get system status
print(pulsar.get_system_status())
```

6.1.2 Generate waveforms for QCM

```
# Generate waveforms

waveform_len = 120  # ns

# "index" is used to select the waveforms at the hardware level
waveforms    = {
                  "gaussian": {"data": [], "index": 0},
                  "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_len, std=0.12 *␣
↪waveform_len)

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

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

#Create block waveform
if "dc" in waveforms:
    # NB for DC we only need to generate 4 sample points and play them on repeat
    waveforms["dc"]["data"] = [1.0 for i in range(0, 4)]
```

```
import matplotlib.pyplot as plt
import numpy as np

time = np.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())), 1)

fig, ax = plt.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()
```



In order to play the waveform the firmware needs to receive the waveforms as well as an assembly-like program. In this case we will be running the device in continuous mode, i.e. the same waveforms will be played over and over again on the same outputs. This makes easy to get started and observe the waveforms on an oscilloscope and the required program is trivial.

Currently, the output pair O^1 & O^2 is controlled by its own CPU-like sequence processor (sequencer); and similarly for O^3 & O^4. This will become much more flexible with new firmware and software updates.

Note: Currently, playing waveforms in continuous mode requires the number of samples of the waveforms to be multiples of 4.

6.1.3 Generate program and waveforms file for QCM

We are going to play waveforms in continuous mode. Since this requites bypassing the sequence processor's control over the waveform memory to directly play the waveforms in a continuous repeated mode the sequencer processors needs to stop immediately when started, therefore the only instruction necessary in the program is a `stop` instruction.

Note: For more elaborated experiments there will be an assembly-like program file for each sequencer generated by a compiler from a higher level interface.

```
# Sequencer programs
seq_prog = ["stop", "stop"]
```

```
# Write waveforms and programs to a JSON files
for name in waveforms:
    if str(type(waveforms[name]["data"]).__name__) == "ndarray":
        assert (len(waveforms[name]["data"]) % 4) == 0,\
                "In continuous waveform mode the lenght of a waveform must be a␣
↪mupltiple of 4!"
```

(continues on next page)

(continued from previous page)

```
        waveforms[name]["data"] = waveforms[name]["data"].tolist()  # JSON only supports␣
↪lists

for seq, prog in enumerate(seq_prog):
    wave_and_prog_dict = {"waveforms": {"awg": waveforms},
                          "program": prog}
    with open("demo_seq{}.json".format(seq), 'w', encoding='utf-8') as file:
        json.dump(wave_and_prog_dict, file, indent=4)
        file.close()
```

6.1.4 Upload program and waveforms into QCM

```
# Upload waveforms and programs
for seq in [0, 1]:
    pulsar.set(
        "sequencer{}_waveforms_and_program".format(seq),
        os.path.join(os.getcwd(), "demo_seq{}.json".format(seq))
    )
    print(pulsar.get_assembler_log())
```

6.1.5 Configure the QCM

```
# Configure the sequencers
for seq in [0, 1]:
    pulsar.set("sequencer{}_sync_en".format(seq),                True)  # Enable the␣
↪sequecer
    pulsar.set("sequencer{}_cont_mode_en_awg_path0".format(seq), True)  # Set␣
↪continuous waveform repetition on O1/O3 outputs
    pulsar.set("sequencer{}_cont_mode_en_awg_path1".format(seq), True)  # Set␣
↪continuous waveform repetition on O2/O4 outputs
    pulsar.set("sequencer{}_gain_awg_path0".format(seq),         1.0)   # Set unitary␣
↪gain on O1/O3 outputs
    pulsar.set("sequencer{}_gain_awg_path1".format(seq),         1.0)   # Set unitary␣
↪gain on O2/O4 outputs
    pulsar.set("sequencer{}_offset_awg_path0".format(seq),       0)     # No offsets on␣
↪O1/O3 outputs
    pulsar.set("sequencer{}_offset_awg_path1".format(seq),       0)     # No offsets on␣
↪O2/O4 outputs
    pulsar.set("sequencer{}_mod_en_awg".format(seq),             False) # Disable␣
↪modulation

# Which waveform on which output?
pulsar.set("sequencer0_cont_mode_waveform_idx_awg_path0".format(seq), 0) # Gaussian on O1
pulsar.set("sequencer0_cont_mode_waveform_idx_awg_path1".format(seq), 1) # Sine on O2
pulsar.set("sequencer1_cont_mode_waveform_idx_awg_path0".format(seq), 2) # Sawtooth on O3
pulsar.set("sequencer1_cont_mode_waveform_idx_awg_path1".format(seq), 3) # DC on O4
```

6.1.6 Play waveforms on the QCM

```
# Arm the sequencers
pulsar.arm_sequencer()
```

```
# Start the sequencers
pulsar.start_sequencer()

# Print status
for seq in range(0, 2):
    print(pulsar.get_sequencer_state(seq))
```

6.1.7 Visualize the signals on an oscilloscope

We connect all output channels of the QCM to the four channels of an oscilloscope. On the scope we are able to see that all waveforms are being generated correctly:

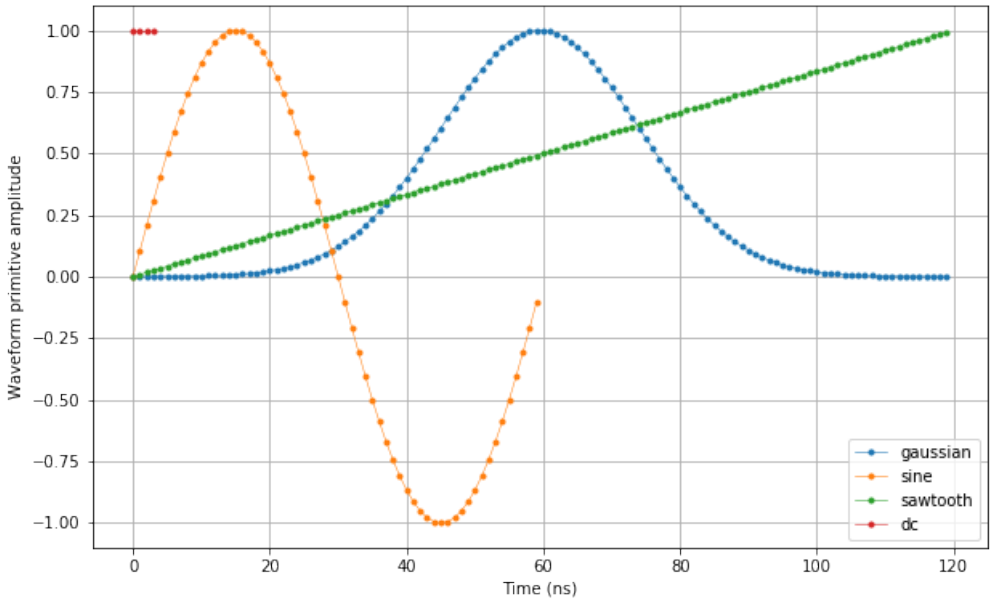

```
# Stop the sequencers
pulsar.stop_sequencer()
for seq in range(0, 2):
    print(pulsar.get_sequencer_state(seq))
```

```
# It is also possible to retrieve back the waveforms in the memory
wvs = pulsar.get_waveforms(0)["awg"]

import matplotlib.pyplot as plt

time = np.arange(0, max(map(lambda d: len(d["data"]), wvs.values())), 1)

fig, ax = plt.subplots(1,1, figsize=(10, 10/1.61))

ax.set_ylabel("Waveform primitive amplitude")
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Time (ns)")

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

ax.legend(loc=4)
ax.yaxis.grid()
ax.xaxis.grid()
```



Since we are using a QCoDeS interface for our instrument we can retrieve an overview of its parameters:

```
# Print the instrument snapshot
# See QCoDeS documentation for details
pulsar.print_readable_snapshot(update=True)
```

6.2 QRM demo

In this demo we showcase the upload, play and digitization of a waveform using the Qblox Pulsar QRM.

NB: This hardware demo interacts with firmware low-level interfaces. When building applications we highly recommend using Quantify framework.

6.2.1 Physical setup

In this demo we use a single QRM and we connect its outputs to its own inputs as follows using two SMA cables

- $O^1 \rightarrow I^1$
- $O^2 \rightarrow I^2$

```
# Import python utilities
import os
import scipy.signal
import math
import json
import matplotlib.pyplot as plt
```

6.2.2 Connect to the QRM

```
# Add Pulsar QRM interface
from pulsar_qrm.pulsar_qrm import pulsar_qrm
```

```
# Connect to device over Ethernet
pulsar = pulsar_qrm("qrm", "192.168.0.3")
```

Tip: If you need to re-instantiate the instrument without re-starting the Jupyter Notebook (or the ipython shell) run the following command first `pulsar.close()`. This closes the QCoDeS instrument instance.

```
# Print device information
print(pulsar.get_idn())
# Get system status
print(pulsar.get_system_status())
```

6.2.3 Generate waveforms for QRM

```
# Generate waveforms
waveform_len = 1000 # ns
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_len, std=0.
↪133*waveform_len)

#Create gaussian waveform
if "sine" in waveforms:
```

(continues on next page)

(continued from previous page)

```
    waveforms["sine"]["data"] = [math.sin((2*math.pi/waveform_len)*i) for i in range(0,␣
↪waveform_len)]

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

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

```
import matplotlib.pyplot as plt
import numpy as np

time = np.arange(0, max(map(lambda d: len(d["data"]), waveforms.values())), 1)

fig, ax = plt.subplots(1,1)

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"])
```

6.2.4 Upload program and waveforms into QRM

In order to play the waveforms and digitize the input the firmware needs to receive the waveforms and an assembly-like program.

The program for this demo is somewhat elaborated and we will not examine it in detail. It is intended to be generated by a higher level compiler.

```
# Sequencer programs

seq_prog = (""
"                   wait_sync    4                   #Wait for synchronization\n"
"start:             move         4,R0                #Init number of waveforms\n"
"                   move         0,R1                #Init waveform index\n"
""
"mult_wave_loop:    move         166,R2              #Init number of singe wave loops␣
↪(increasing wait)\n"
"                   move         166,R3              #Init number of singe wave loops␣
↪(decreasing wait)\n"
"                   move         24,R4               #Init number of dynamic wait time␣
↪(total 4us)\n"
"                   move         3976,R5             #Init number of dynamic wait time\n"
"                   move         32768,R6            #Init gain\n"
""
"sngl_wave_loop_0: move          800,R7              #Init number of long delay cycles\n"
"                   set_mrk      15                  #Set marker to 0xF\n"
```

(continues on next page)

(continued from previous page)

```
"                   upd_param    4                    #Update all parameters and wait 4ns\
↪n"
"                   set_mrk      0                    #Set marker to 0\n"
"                   upd_param    96                   #Update all parameters and wait␣
↪96ns\n"
""
"                   wait         R4                   #Dynamic wait\n"
"                   add          R4,24,R4             #Increase wait\n"
""
"                   set_mrk      1                    #Set marker to 1\n"
"                   play         R1,R1,4              #Play waveform and wait 992ns\n"
"                   acquire      R1,R1,992            #Acquire while playing\n"
"                   set_mrk      0                    #Set marker to 0\n"
"                   upd_param    4                    #Update all parameters and wait for␣
↪4ns\n"
""
"                   wait         R5                   #Compensate previous dynamic wait\n"
"                   sub          R5,24,R5             #Decrease wait\n"
""
"                   sub          R6,98,R6             #Decrease gain\n"
"                   nop\n"
"                   set_awg_gain R6,R6                #Set gain\n"
""
"long_wait_loop_0:  wait         50000                #Wait 50 us\n"
"                   loop         R7,@long_wait_loop_0 #Wait number of long delay cycles\n"
"                   loop         R2,@sngl_wave_loop_0 #Repeat single wave\n"
""
"sngl_wave_loop_1:  move         800,R7               #Init number of long delay cycles\n"
"                   set_mrk      15                   #Set marker to 0xF\n"
"                   upd_param    8                    #Update all parameters and wait 8ns\
↪n"
"                   set_mrk      0                    #Set marker to 0\n"
"                   upd_param    92                   #Update all parameters and wait␣
↪92ns\n"
""
"                   wait         R4                   #Dynamic wait\n"
"                   sub          R4,24,R4             #Decrease wait\n"
""
"                   set_mrk      1                    #Set marker to 1\n"
"                   play         R1,R1,4              #Play waveform and wait 992ns\n"
"                   acquire      R1,R1,992            #Acquire while playing\n"
"                   set_mrk      0                    #Set marker to 0\n"
"                   upd_param    4                    #Update all parameters and wait 4ns\
↪n"
""
"                   wait         R5                   #Compensate previous dynamic wait\n"
"                   add          R5,24,R5             #Increase wait\n"
""
"                   sub          R6,98,R6             #Decrease gain\n"
"                   nop\n"
"                   set_awg_gain R6,R6                #Set gain\n"
""
```

(continues on next page)

(continued from previous page)

```
"long_wait_loop_1: wait         50000                #Wait for 50 us\n"
"                  loop         R7,@long_wait_loop_1 #Wait number of long delay cycles\n"
"                  loop         R3,@sngl_wave_loop_1 #Repeat single wave\n"
""
"                  add          R1,1,R1              #Adjust waveform index\n"
"                  loop         R0,@mult_wave_loop   #Move to next waveform\n"
"                  jmp          @start               #Jump back to start\n")
```

```
# Write waveforms and programs to JSON files
for name in waveforms:
    if str(type(waveforms[name]["data"]).__name__) == "ndarray":
        waveforms[name]["data"] = waveforms[name]["data"].tolist()

wave_and_prog_dict = {"waveforms": {"awg": waveforms,
                                    "acq": waveforms},
                      "program": seq_prog}

seq = 0
with open("demo_seq{}.json".format(seq), 'w', encoding='utf-8') as file:
    json.dump(wave_and_prog_dict, file, indent=4)
    file.close()
```

```
# Upload waveforms and programs
pulsar.set("sequencer{}_waveforms_and_program".format(seq),
        os.path.join(os.getcwd(), "demo_seq{}.json".format(seq)))
print(pulsar.get_assembler_log())
```

6.2.5 Configure the QRM

```
# Configure the sequencers

pulsar.set("sequencer{}_sync_en".format(seq),                    True)  # Enable␣
↪the sequecer
pulsar.set("sequencer{}_cont_mode_en_awg_path0".format(seq),     False) # Disable␣
↪continuous waveform repetition on O1
pulsar.set("sequencer{}_cont_mode_en_awg_path1".format(seq),     False) # Disable␣
↪continuous waveform repetition on O2
pulsar.set("sequencer{}_gain_awg_path0".format(seq),             1.0)   # Set␣
↪unitary gain on O1
pulsar.set("sequencer{}_gain_awg_path1".format(seq),             1.0)   # Set␣
↪unitary gain on O2
pulsar.set("sequencer{}_offset_awg_path0".format(seq),           0)     # No␣
↪offset on O1
pulsar.set("sequencer{}_offset_awg_path1".format(seq),           0)     # No␣
↪offset on O1
pulsar.set("sequencer{}_mod_en_awg".format(seq),                 True)  # Enable␣
↪modulation
pulsar.set("sequencer{}_nco_freq".format(seq),                   10e6)  # Set␣
↪modulation frequency
pulsar.set("sequencer{}_nco_phase".format(seq),                  0)     # Set␣
↪modulation phase
```

(continues on next page)

(continued from previous page)

```
pulsar.set("sequencer{}_trigger_mode_acq_path0".format(seq),          False) #␣
↪Acquisition will be triggered "manually" in this notebook
pulsar.set("sequencer{}_trigger_mode_acq_path1".format(seq),          False) #␣
↪Acquisition will be triggered "manually" in this notebook
```

6.2.6 Play the modulated waveforms and digitize the inputs

```
# Arm the sequencer
pulsar.arm_sequencer()
```

```
# Start the sequencers
pulsar.start_sequencer()
print(pulsar.get_sequencer_state(seq))
```

```
# Stop the sequencers
pulsar.stop_sequencer()
print(pulsar.get_sequencer_state(seq))
```

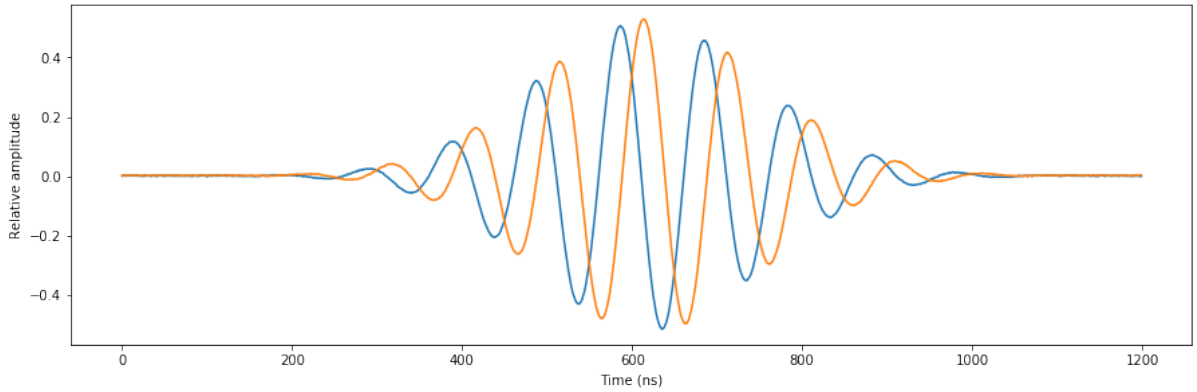
6.2.7 Retrieve acquired data

The data acquired above was stored in a temporary memory (FPGA memory). This memory is fast but at the cost of being relatively small in size. Its contents are overwritten (automatically) on each new acquisition!

Because of this there is an intermediate memory (CPU RAM of the Pulsar) to which the acquisitions are stored and later retrieved into the host PC for offline analysis. Note that it is possible to copy many acquisitions into the intermediate memory before retrieving them all.

```
# Get acquisition

# Deletes any previous data stored in the intermediate memory (CPU RAM of the Pulsar)
pulsar.delete_acquisitions(seq)

# Copies last aquired data from the FPGA memory into the intermediate memory
# NB: The FPGA memory is only accessible when the sequencer is stopped
pulsar.store_acquisition(seq, "meas_0", 1200) # Copy only the first 1200 samples, if not␣
↪specified will dump full FPGA memory

# Retriev aquired data from the intermediate memory of the Pulsar into the host PC
acq = pulsar.get_acquisitions(seq)
```

```
# Plot aquired signal on both inputs

fig, ax = plt.subplots(1, 1, figsize=(15, 15/2/1.61))
ax.plot(acq["meas_0"]["path_0"]["data"])
ax.set_xlabel('Time (ns)')
ax.set_ylabel('Relative amplitude')
ax.plot(acq["meas_0"]["path_1"]["data"])
plt.show()
```



The digitized signal values are relative to the ADC reference, see datasheet for details.

Above we showcase the capabilities of outputting and digitizing I and Q signals with a Gaussian envelope.

Retrieving multiple acquisitions at once

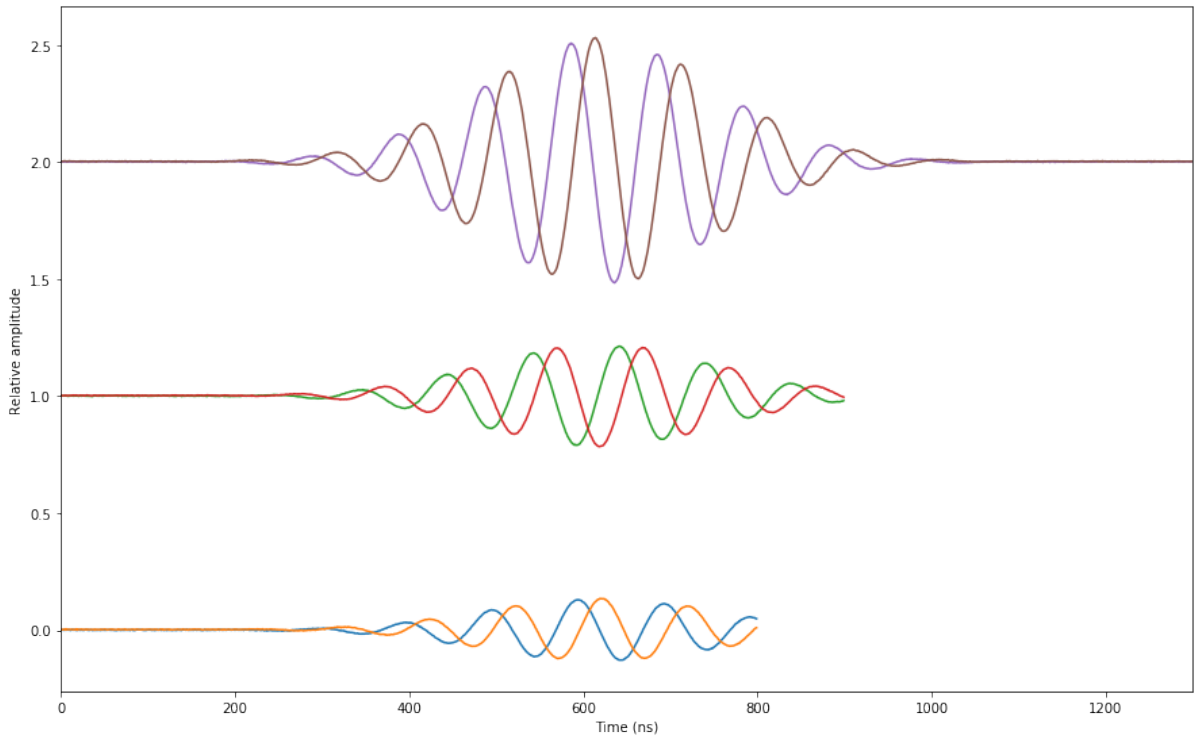
Below we showcase the retrieving of multiple measurements results at once. We also exemplify how distinct parts of the FPGA memory can be stored.

```
pulsar.delete_acquisitions(seq) # Clear intermediate memory

# Run and store first measurement
pulsar.set("sequencer{}_gain_awg_path0".format(seq), 0.25)
pulsar.set("sequencer{}_gain_awg_path1".format(seq), 0.25)
pulsar.arm_sequencer()
pulsar.start_sequencer()
print(pulsar.get_sequencer_state(seq))
pulsar.stop_sequencer()
pulsar.store_acquisition(seq, "meas_0", 800)

# Run and store second measurement
pulsar.set("sequencer{}_gain_awg_path0".format(seq), 0.5)
pulsar.set("sequencer{}_gain_awg_path1".format(seq), 0.3)
pulsar.arm_sequencer()
pulsar.start_sequencer()
print(pulsar.get_sequencer_state(seq))
pulsar.stop_sequencer()
pulsar.store_acquisition(seq, "meas_1", 900)

# Run and store third measurement
pulsar.set("sequencer{}_gain_awg_path0".format(seq), 1.0)
pulsar.set("sequencer{}_gain_awg_path1".format(seq), 1.0)
pulsar.arm_sequencer()
pulsar.start_sequencer()
print(pulsar.get_sequencer_state(seq))
pulsar.stop_sequencer()
pulsar.store_acquisition(seq, "meas_2") # Store full FPGA memory

# Retriev aquired data from the intermediate memory of the Pulsar into the host PC
acq = pulsar.get_acquisitions(seq)
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(1, 1, figsize=(15, 15 / 1.61))

for msmt, plt_offset in zip(["meas_0", "meas_1", "meas_2"], [0, 1, 2]):
    ax.plot(np.array(acq[msmt]["path_0"]["data"]) + plt_offset,
            label="{} I".format(msmt))  # Plot I quadrature
    ax.plot(np.array(acq[msmt]["path_1"]["data"]) + plt_offset,
            label="{} q".format(msmt))  # Plot Q quadrature

    ax.set_xlabel('Time (ns)')
    ax.set_ylabel('Relative amplitude')

# Plot only relevant region
ax.set_xlim(0, 1300)
```



Since we are using a QCoDeS interface for our instrument we can retrieve an overview of its parameters:

```
# Print the instrument snapshot
# See QCoDeS documentation for details
pulsar.print_readable_snapshot(update=True)
```

CHAPTER

SEVEN

API REFERENCE

Contents:

7.1 ieee488_2

class ieee488_2.transport.**file_transport**(*out_file_name: str*, *in_file_name: str = ''*)
Input/output from/to file to support driver testing

class ieee488_2.transport.**ip_transport**(*host: str*, *port: int = 5025*, *timeout=10.0*, *snd_buf_size: int = 524288*)
IP socket transport

class ieee488_2.transport.**pulsar_dummy_transport**(*cfg_format*)
Dummy transport

class ieee488_2.transport.**transport**
Abstract base class for data transport to instruments

7.2 pulsar_qcm

7.2.1 QCoDeS driver

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

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

class pulsar_qcm.pulsar_qcm.**pulsar_qcm_qcodes**(*name*, *transport_inst*, *debug=0*)
The Pulsar QCM QCoDeS interface.

7.2.2 Native interface

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

arm_sequencer(*sequencer=None*)
Arm sequencer.

get_idn()
Get device identity and build information.

get_sequencer_state(*sequencer*)
Get sequencer state.

get_system_status()
Get general system status.

get_waveforms(*sequencer*)
Return all waveforms in a dictionary

start_sequencer(*sequencer=None*)
Start sequencer.

stop_sequencer(*sequencer=None*)
Stop sequencer.

7.2.3 SCPI interface

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

clear()
Clear all status registers.

get_assembler_log()
Get assembler log.

get_assembler_status()
Get assembler status.

get_current_afe_temperature()
Get analog frontend temperature.

get_current_carrier_temperature()
Get current carrier temperature.

get_current_fpga_temperature()
Get current FPGA temperature.

get_maximum_afe_temperature()
Get analog frontend temperature.

get_maximum_carrier_temperature()
Get maximum carrier temperature.

get_maximum_fpga_temperature()
Get maximum FPGA temperature.

get_num_system_error()
Get number of system errors.

get_operation_complete()
Get operation complete status.

get_operation_condition()
Get operation condition.

get_operation_enable()
Get operation enable.

get_operation_events()
Get operation events.

get_questionable_condition()
Get questionable condition.

get_questionable_enable()
Get questionable enable.

get_questionable_event()
Get questionable events.

get_service_request_enable()
Get service request enable.

get_standard_event_status()
Get standard event status.

get_standard_event_status_enable()
Get standard event status enable.

get_status_byte()
Get status byte.

get_system_error()
Get system error from queue.

get_system_version()
Get system version.

preset_system_status()
Preset system status registers.

reset()
Reset device.

set_operation_complete()
Set operation complete command.

set_operation_enable(*reg*)
Get operation enable.

set_questionable_enable(*reg*)
Set questionable enable.

set_service_request_enable(*reg*)
Set service request enable.

set_standard_event_status_enable(*reg*)
Set standard event status enable.

test()
Run self-test.

wait()
Wait-to-continue.

7.3 pulsar_qrm

7.3.1 QCoDeS driver

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

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

class pulsar_qrm.pulsar_qrm.**pulsar_qrm_qcodes**(*name*, *transport_inst*, *debug=0*)
The Pulsar QRM QCoDeS interface.

7.3.2 Native interface

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

arm_sequencer(*sequencer=None*)
Arm sequencer.

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

get_acquisitions(*sequencer*)
Return all acquisitions in a dictionary

get_idn()
Get device identity and build information.

get_sequencer_state(*sequencer*)
Get sequencer state.

get_system_status()
Get general system status.

get_waveforms(*sequencer*)
Return all waveforms in a dictionary

start_sequencer(*sequencer=None*)
Start sequencer.

stop_sequencer(*sequencer=None*)
Stop sequencer.

store_acquisition(*sequencer*, *name*, *size=2147483648*)
Add acquisition to sequencer acquisition list.

7.3.3 SCPI interface

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

clear()
Clear all status registers.

get_assembler_log()
Get assembler log.

get_assembler_status()
Get assembler status.

get_current_afe_temperature()
Get analog frontend temperature.

get_current_carrier_temperature()
Get current carrier temperature.

get_current_fpga_temperature()
Get current FPGA temperature.

get_maximum_afe_temperature()
Get analog frontend temperature.

get_maximum_carrier_temperature()
Get maximum carrier temperature.

get_maximum_fpga_temperature()
Get maximum FPGA temperature.

get_num_system_error()
Get number of system errors.

get_operation_complete()
Get operation complete status.

get_operation_condition()
Get operation condition.

get_operation_enable()
Get operation enable.

get_operation_events()
Get operation events.

get_questionable_condition()
Get questionable condition.

get_questionable_enable()
Get questionable enable.

get_questionable_event()
Get questionable events.

get_service_request_enable()
Get service request enable.

get_standard_event_status()
Get standard event status.

get_standard_event_status_enable()
Get standard event status enable.

get_status_byte()
Get status byte.

get_system_error()
Get system error from queue.

get_system_version()
Get system version.

preset_system_status()
Preset system status registers.

reset()
Reset device.

set_operation_complete()
Set operation complete command.

set_operation_enable(*reg*)
Get operation enable.

set_questionable_enable(*reg*)
Set questionable enable.

set_service_request_enable(*reg*)
Set service request enable.

set_standard_event_status_enable(*reg*)
Set standard event status enable.

test()
Run self-test.

wait()
Wait-to-continue.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

p

INDEX

I

M

P

R

S

T

W