openDAQ Documentation

Release 0.3.3

Ingen10

Jan 12, 2018

Contents

This is the documentation of the **openDAQ** Python package

Contents:

CHAPTER 1

openDAQ usage in Python

1.1 Device connection and port handling

To establish a connection with the openDAQ through the command line type the following:

```
python

from opendaq import DAQ

daq = DAQ("/dev/ttyUSB0")
```

When creating an object of type DAQ, you have to specify the actual port at wich the openDAQ is connected. This can be done, in UNIX operating systems, typing in the terminal:

```
$ dmesg
```

You should see something like this:

```
...
...
...
for cp210x
[17755.465949] cp210x 1-4.4:1.0: cp210x converter detected
[17755.536101] usb 1-4.4: reset full-speed USB device number 5 using ehci-pci
[17755.629330] usb 1-4.4: cp210x converter now attached to ttyUSB0
```

In this example, openDAQ is attached to the USB port named ttyUSB0.

If you are working in Windows, the name of the port will be something like *COMxx* instead of */dev/ttyUSBxx*. You can check the port in *Control Panel->System->Device Manager*.

Now, with the object *daq* created, we can start working with it. If you want to close the port, simply type the following:

```
daq.close()
```

1.2 ADC reading (Command-Response mode)

First of all, we must configure the ADC,specifying the positive analog input, and the negative analog input if we want to do differential measures.

This can be done using the *conf_adc* function:

```
a.conf_adc(pinput,ninput,gain,nsamples)
```

The values of these parameters are listed in the following table:

Parameter	Description	Value	Notes
pinput	Positive input	1:8	AN1-AN8
ninput	Negative input	M:0,5,6,7,8,25 S: 0, 1:8 N: 0, 1:8	0: ref ground 25: ref 2,5 V rest: input pins
gain	Analog gain	M: 0:4 S: 0:7 N: 0:7	x1/3,x1,x2,x10,x100 x1,x2,x4,x5,x8,x10,x16, x20 x1,x2,x4,x5,x8,x10,x16, x32
nsamples	Number of samples per data point	[0-254]	

There are three options to read the ADC.

If we want the raw data from the ADC, we can use

```
data = daq.read_adc()

print data
```

Much better, if we want the data directly in Volts, just use:

```
data_Volts = daq.read_analog()
```

Finally, we can also read all the analog inputs simultaneously using the function *read_all*:

```
data_Volts = daq.read_all()
```

This function return a list with the readings (in Volts) of all analog inputs.

1.3 DAC setting (CR mode)

As in the case of reading the ADC, there are two functions to set the output of the DAC: *set_analog('V')* and *set_dac('raw')*. The first set DAC output voltage in V betwen the voltage hardware limits :

```
daq.set_analog(1.5)
```

The function *set_dac* set the DAC with the raw binary data value:

```
daq.set_dac(3200)
```

Model	Output Voltage Range
openDAQ[M]	[-4.096V 4.096V]
openDAQ[S]	[0V 4.096V]
openDAQ[N]	[-4.096V 4.096V]

1.4 Stream Experiments Creation (Stream Mode)

OpenDAQ has two main modes of operation: Command-Response Mode and Stream (hardware-timed) Mode.

In command-response mode all communications are initiated by a command from the host PC, wich is followed by a response from openDAQ.

On the other hand, the Stream mode is a continous hardware-timed input mode where a list of channels that are scanned at a specified rate.

Stream Mode can be used in three kind of experiment modes, wich differ in the maximum scan rate allowed and the source of the timing clock (internal or external). We define an experiment as a certain data source with specific configuration, sampling rate and start and stop conditions:

- Stream experiments
- External experiments
- Burst experiments

Once the experiment is configured we can start it:

```
daq.start()
```

or stop it:

```
daq.stop()
```

We can read the data using the method *read*:

```
stream_exp.read()
```

1.4.1 Stream experiments

For Stream Experiments, a specific data source is sampled in regular intervals, using internal timer to keep time count (Timer2). Fastest scan rate in this mode is 1kSPS (1ms of period).

User can configure up to 4 Stream experiments to be running simultaneously. They will have each an internal buffer of about 400 samples, which will be normally enough not to lose any point in the communications.

First of all we have to import the library and the constant definitions:

```
from opendaq import *
from opendaq.daq import *
```

To create an Stream Experiment use the following function:

```
daq.stream_exp = daq.create_stream(ExpMode.ANALOG_IN, 100, 30,␣
↪continuous=False)
```

with parameters:

Parameter	Description	Value	Notes
ExpMode	Define data source or destination	0:5	0:ANALOG_IN 1:ANALOG_OUT 2:DIGITAL_IN 3:DIGITAL_OUT 4:COUNTER_IN 5:CAPTURE_IN
period	Period of the stream experiment	1:65536	
npoints	Total number of points for the experiment	0:65536	0 indicates continous adquisition (By default 10)
continuous	Indicates if experiment is continuous	True or False	False:run once (By default False)

Once created the experiment we can configure the input to read. For example, if we want to read the analog input 6 (AN6), without gain, we should use:

```
stream_exp = daq.create_stream(ExpMode.ANALOG_IN, 200, continuous=True)
```

Now, we have to configure the channel. To do this we use the method *analog_setup* of the class *DAQStream*:

```
stream_exp.analog_setup(pinput=8, ninput=0, gain=Gains.M.x1)
```

with parameters:

For the example above:

```
stream_exp.analog_setup(pinput=7,gain=GAIN_S_X2)
```

1.4.2 External experiments

External experiments use an external digital trigger source to perform readings. Fastest scan rates are in similar ranges as for the Stream experiments. The rest of properties and parameters are similar to Stream experiments.

User can define up to 4 external experiments at the same time, each of one connected to digital inputs D1 to D4 (the number of the internal DataChannel is connected to the digital input number) to act as trigger inputs.

Maximum number of experiments will be 4 in total, including all External and Stream experiments.

To create an External Experiment use the following function:

```
daq.create_external(mode,clock_input,edge,npoints,continuous,buffersize)
```

The new parameters here are *clock_input* and *edge*, which are explained in the following table:

Parameter	Description	Value	Notes
clock_input	Assign a DataChannel number and a digital input for this experiment	1:4	
edge	New data on rising (1) or falling (0) edges	0:1	

For example, we are going to create an external experiment with an analog readin in AN8 (SE):

```
extern_exp = daq.create_external(ExpMode.ANALOG_IN, 1, edge=1, npoints=10,␣
↪continuous=False)
```

As with the stream experiment, now we have to setup the analog input:

```
stream_exp.analog_setup(pinput=8, ninput=0, gain=Gains.M.x1)

daq.start()
```

We can use a while loop in this way:

```
while daq.is_measuring:
    print "data", extern_exp.read()
```

1.4.3 Burst experiments

Burst experiments are also internally timed, like Stream experiments, but they are intended to use a faster sampling rate, up to 10kSPS. The high acquisition rate limits the amount of things that the processor is capable of doing at the same time. Thus, when a Burst experiment is carried out, no more experiments can run at the same time.

Burst experiment use a bigger internal buffer of about 1600 points to temporary store results. However, if the experiment goes on for a long time, the buffer will eventually get full and the firmware will enter "Auto-recovery" mode. This means that it will get no more points until buffer gets empty again, having an time where no sample will be taken.

To create a burst experiment use the following function:

```
burst_exp = daq.create_burst(mode, period, npoints, continuous)
```

Here is an example of a how a burst experiment is configured to do a analog output streaming:

```
preload_buffer = [0.3, 1, 3.3, 2]
burst_source = daq.create_burst(ExpMode.ANALOG_IN, period=200,␣
↪npoints=len(preload_buffer), continous=False)
burst_source.analog_setup()
burst_source.load_signal(preload_buffer)

daq.start()
```

1.4.4 Analog output streaming

With Stream and Burst experiments we can load a generic waveform (of any type) and the device will reproduce it through the DAC. This can be achieved by this way:

- First create the waveform:

```
preload_buffer = [0.3, 1, 3.3, 2] # The waveform
```

- Next, create the experiment (Stream or Burst, see next subsections)
- Finally load the signal to the experiment:

```
exp_name.load_signal(preload_buffer)
```

IMPORTANT NOTE: Analog output streams always use internal DataChannel #4, thus digital input D4 will not be available for an External experiment.

1.4.5 Triggering experiments

From version 0.2.1 of the library, openDAQ allows setting trigger modes to start executing experiments. Trigger sources may be software triggered (default), digital input trigger (rising or falling edge) or analog value (input value above or below a specific limit).

```
stream1.trigger_setup(type,value)
```

where

type	Value	Notes
SW_TRG	•	software trigger (default)
DIN1_TRG	0/1	digital trigger
DIN2_TRG	0/1	digital trigger
DIN3_TRG	0/1	digital trigger
DIN4_TRG	0/1	digital trigger
DIN5_TRG	0/1	digital trigger
DIN6_TRG	0/1	digital trigger
ABIG_TRG	any	analog trigger
ASML_TRG	any	analog trigger

1.5 Capture Input

The capture input permits measuring the time length of incoming digital signals. It makes use of device internal timer to calculate the time elapsed between changes in state (high to low or low to high) of an external signal. OpenDAQ has a main clock running at 16MHz, which limits the minimum periods that the device is able to measure to several microseconds.

The input in this mode is D5 (DIO 5 pin)

There are three methods associated with this mode: *init_capture*, *stop_capture* and *get_capture*. To start measuring use

```
daq.init_capture(period)
```

where period is the estimated period of the wave (in microseconds), and its range is *32 bits*. Now , we can get the Capture reading:

```
daq.get_capture(mode)
```

where

Parameter	Value	Notes
mode	0:3	0: Low cycle 1: High cycle 2: Full period

Finally, stop the capture when the experiment has finished:

```
daq.stop_capture(mode)
```

1.6 Counter Input

The counter input is also based on Timer 1, and its functionality consists on counting number of edges coming through the port (D6). This can be useful to measure the frequency of very fast signal or to read some kind of sensors.

User can select which kind of digital edges will the peripheral detect (high or low), and he can also read and reset the counter back to 0 whenever it is necessary.

The edges are counted in a *32-bit counter*.

To start counting type the following:

```
daq.init_counter(edge)
```

This method configure which edge increments the count: Low-to-High (1) or High-to-Low (0). To get the counter value:

```
daq.get_counter(reset)
```

If *reset>0* , the counter is reset after perform the reading.

1.7 Encoder Input

The encoder input is based on external interrupts on pin D6. Its functionality consists on counting number of edges coming through the digital input D6 while keeping track of the direction of the movement, by reading D5 on each interrupt.

User can select the maximum resolution of the encoder.

To work in this mode there are three methods. The first start the encoder function:

```
daq.init_encoder(resolution)
```

Resolution is the maximum number of ticks per round (32-bit counter).This command configures external interrupts on D6 and resets the pulse counter to 0. Next, to get the current encoder relative position use:

```
daq.get_encoder()
```

This method returns the actual encoder value. Finally, stop the encoder:

```
daq.stop_encoder()
```

1.8 PWM Output

Pulse Width Modulator generates a continuous digital signal at a given frequency. Duty refers to the portion of time that the signal spends in High state.

PWM output is connected to port D6 of openDAQ.

To start the PWM Output mode use the following method:

```
daq.init_pwm(duty,period)
```

Duty is the high time of the signal ([0:1023]). If 0, the signal is always low. Period is the period of the signal in microseconds. To stop the PWM:

```
daq.stop_pwm()
```

1.9 PIO Configuration and control (CR mode)

The openDAQ has 6 DIO (digital Inputs/Outputs). We have 4 DIO lines on the right side screw terminal block (D1-D4), and the two others on the left terminal block (D5-D6).

D5 is a multipurpose terminal that is also connected with internal microprocessor's Timer/Counter 2. Apart from being used as a DIO, this terminal can be configured as PWM output, Counter input or Capture input.

All the digital I/O lines include an internal series resistor and a protective diode that provides overvoltage/short-circuit protection. The series resistors (about 100Ω) also limit the ability of these lines to sink or source current.

The DIOs have 3 possible states: input, output-high, or output-low. Each line of I/O can be configured individually. When configured as an input, the line has a 50kΩ pull-up resistor to 5.0 volts. When configured as output-high, the line is connected to the internal 5.0 volt supply (through a series resistor).

When configured as output-low, a bit is connected to GND (through a series resistor). All digital I/O are configured to be inputs at power up.

We have two couples of commands to control the digital I/O lines. The first two ones control each line individually, one to set or read the line direction (input or output), and the other to read or set the line value (high or low). The other two commands control the six lines at a time, one function to read or set the lines direction, and the other command to read or set the lines values.

Method	Arguments	Notes
set_pio_dir	number: 1:6 output: 0:1	PIO number 0: input; 1: output
set_pio	number: 1:6 value: 0:1	PIO number Digital value: 0 Low, 1 High
read_pio	number: 1:6	PIO number
set_port_dir	output: 0:1	0: input; 1: output
set_port	value: 0:1	Digital value: 0 Low, 1 High
read_port		

1.10 Bit-bang SPI Output

The Serial Peripheral Interface (SPI) is a very popular communications bus, used widely in electronics to control slave devices. This utility allows openDAQ to communicate with other low level devices, like external port expanders, PGAs, switches or other peripherals.

SPI is a synchronous serial data link that operates in full duplex mode, using a master/slave scheme, where the master device always initiates the data frame. Multiple slave devices are allowed with separated select lines.

The SPI bus specifies four logic signals:

- SCLK: serial clock (output from master)
- MOSI: master output, slave input (output from master)
- MISO: master input, slave output (output from slave)
- SS: slave select (active low, output from master)

To begin a communication, the bus master first configures the clock, and then transmits the logic 0 for the desired chip over the chip select line (SS). During each SPI clock cycle, a full duplex data transmission occurs:

- The master sends a bit on the MOSI line, and the slave reads it from that same line
- The slave sends a bit on the MISO line, and the master reads it from that same line

Transmissions may involve any number of clock cycles.

A relevant issue concerning SPI transmissions, is how the SCLK behaves, and when the MISO and MOSI lines should be read. By convention, these options are named CPOL (clock polarity) and CPHA (clock phase). At CPOL=0 the base value of the clock, when inactive, is zero. CPHA=0 means sample on the leading (first) clock edge, while CPHA=1 means sample on the trailing (second) clock edge, regardless of whether that clock edge is rising or falling. Taking this into consideration, we can define up to four SPI modes, by combining the two possible values of each option.

OpenDAQ uses a so called bit-bang SPI mode, as the bus signals are generated entirely by software (no specific hardware is used).

Specific commands are available to configure the functions of the pins (which DIO number will be used for each SPI line) and the SPI mode (CPOL and CPHA). The SS lines must be controlled separately, using any of the DIO terminals not configured as SPI line (PIO command must be used).

To configure Bit-bang SPI use this method:

```
daq.spi_config(cpol,cpha)
```

Here, *cpol* is the clock polarity (clock pin state when inactive) and *chpa* is the clock phase (leading 0, or trailing 1 edges read).

To select the PIO numbers to use, we have the following method:

```
daq.spi_setup(nbytes,sck,mosi,miso)
```

where

Parameter	Value	Notes
nbytes		Number of bytes
sck	1 by default	Clock pin
mosi	2 by default	MOSI pin
miso	3 by default	MISO pin

Finally, to transfer (send and receive) a byte or a word use:

```
daq.spi_write(value,word)
```

If *word* = *True* , then we are sending a 2-byte word instead of a byte.

1.11 Other functions

There are other methods that can be used with the openDAQ. They are listed below:

Method	Arguments	Notes
enable_crc	on	Enable/Disable the cyclic redundancy check
set_led	color	0:off ; 1: green ; 2: red ; 3: orange
set_id	id: [000:999]	Identify openDAQ device
device_info	None	Read device configuration: Hardware version Firmware version Device ID number

1.12 Calibration

IMPORTANT NOTE: The functions used for openDAQ calibration have been redesigned completely from firmware version 1.4.0 and python library version 0.3

Use the tool **opendaq-utils**, which is installed with the rest of the scripts, for device calibrating and updating.

1.12.1 Theory of operation

AIN and DAC commands are transmitted between the host PC and the device in raw binary using the full 16-bit range of the binary transmissions. For example, raw code -32768 correspond in the ADC readings of the openDAQ [M] to -4.096V, while it is equivalent to -12.0V for the openDAQ [S]. Maximum ADC raw values range up to 32767, which is equivalent to 4.095V in openDAQ [M] and to 12.0V in openDAQ[S].

The same happens for the DAC values: in all openDAQ models maximum raw value (32767) is equivalent to a +4.096V output, and in case of openDAQ [M] minimum value is -32768 or -4.095V. Minimum DAC value for openDAQ [S] is 0V which is equivalent to 0 raw code.

In the case of the ADC inputs the situation is more complex, as there are different gain settings that do affect the conversion between raw codes and real voltage values.

The devices always use the raw values for the internal calculations and data transmission, and it is the *daq.py* library who has the duty to translate those binary codes into actual voltage values.

The relationships between the voltage values and raw codes are always linear, and a good approximation to transform the raw codes into voltages would be just to use the theorical formulas that could be deduced from previous paragraphs. Anyhow, the voltage values calculated from the theorical formulas would have some error, because the components inside the circuits of the openDAQ devices do not have a perfect ideal behaviour. Thus, a specific calibration is used for each openDAQ device, so that the values read by the ADCs and set in the DAC are far more similar to the ideal values.

These values are stored in the permanent EEPROM memory of the openDAQs and used by the *opendaq-python* library to calculate the formulas between the raw codes and voltage values. Those calculations are carried in a slighly different manner depending on the openDAQ model. The code of the conversions is in the *model.py* file.

1.12.2 DAC calibration

The functions that manage the DAC calibration are:

```
daq.set_dac_calib(*list of CalibReg registers*)
daq.get_dac_calib()
```

These methods set and read the device DAC calibration, where *CalibReg* are pairs of slope and offset coefficients (*[dac_corr, dac_offset]*). The values are the coefficients of the line that corrects the deviation between the ideal values and the actual values that the device outputs when it applies no calibration.

In the case of the of the DAC output the mathematical function between the theorical value and the raw binary code is exactly the same:

$$raw_dac_code = volts/dac_base_gain$$

And applying the calibration:

$$raw_dac_code = (volts - dac_offset)/(dac_base_gain * dac_corr)$$

1.12.3 ADC calibration

The functions that manage the DAC calibration are:

```
daq.get_adc_calib(*list of CalibReg registers*)
daq.get_adc_calib()
```

Where as in the case of the DAC calibration, *CalibReg* are pairs of slope and offset coefficients (*[adc_corr, adc_offset]*).

- *adc_corr* is the slope of the calibration lines, the read value divided by the real voltage value at the input.
- *adc_offset* is the zero crossing of the line, in this case the raw ADC value for a 0V input (in this case, it is not a voltage but a raw binary code).

In the case of the ADC, several facts have to be taken into consideration:

- Each analog input will have a different calibration line
- In the case of openDAQ [M] each gain setting must be calibrated separately, as the gains are set by resistor values with a relatively high tolerance. This is not the case of the

openDAQ [S] and [N], which use a PGA with factory calibration for all ranges. - The inputs of the openDAQ [S] have a different calibration if they are used as single ended (SE) or differential (DE). In the case of openDAQ [M] the calibration can be the same for both modes, because the inputs are just multiplexed.

All of this translates into the following:

- openDAQ [M] has a total of 13 ADC calibration slots, 8 for each analog input, and 5 for each gain setting.
- openDAQ [S] has 16 ADC calibration slots, 8 for each analog input in SE mode, and 8 for each input in DE mode.
- openDAQ [N] has 16 ADC calibration slots, 8 for each analog input in SE mode, and 8 for each input in DE mode.

The mathematical function between the raw code given by the device and the real analog value is given by an equation depending on the device model (check file *model.py*):

$$volts = raw/(adc_base_gain * gain_ampli)$$

Where *adc_base_gain* is the relationship between binary codes and volts at *gain 1x*, and *gain_ampli* the actual gain amplification being used.

Applying calibration to the equation above:

$$volts = (raw - adc_offset1 - (adc_offset2 * gain_ampli))/(adc_corr1 * adc_corr2 * adc_base_gain * gain_ampli)$$

CHAPTER 2

API documentation

2.1 opendaq.daq module

Main functions used to communicate with the device. See `usage.rst` for additional info.

class opendaq.daq.**CMD**
Bases: enum.IntEnum

AIN = 1

AIN_ALL = 4

AIN_CFG = 2

BURST_CREATE = 21

CAPTURE_INIT = 14

CAPTURE_STOP = 15

CHANNEL_CFG = 22

CHANNEL_DESTROY = 57

CHANNEL_FLUSH = 45

CHANNEL_SETUP = 32

COUNTER_INIT = 41

EEPROM_READ = 31

EEPROM_WRITE = 30

ENABLE_CRC = 55

ENCODER_INIT = 50

ENCODER_STOP = 51

EXTERNAL_CREATE = 20

GET_CALIB = 36

GET_CAPTURE = 16

GET_COUNTER = 42

GET_ENCODER = 52

GET_STATE_CHANNEL = 35

GET_TRIGGER_MODE = 34

ID_CONFIG = 39

LED_W = 18

PIO = 3

PIO_DIR = 5

PORT = 7

PORT_DIR = 9

PWM_DUTY = 12

PWM_INIT = 10

PWM_STOP = 11

RESET = 27

RESET_CALIB = 38

SET_ANALOG = 24

SET_CALIB = 37

SET_DAC = 13

SIGNAL_LOAD = 23

SPISW_CONFIG = 26

SPISW_SETUP = 28

SPISW_TRANSFER = 29

STREAM_CREATE = 19

STREAM_DATA = 25

STREAM_START = 64

STREAM_STOP = 80

TRIGGER_SETUP = 33

WAIT_MS = 17

class opendaq.daq.**DAQ**(*port*, *debug=False*)

Bases: object

This class represents an OpenDAQ device.

clear_experiments()

Delete the whole experiment list.

close()

Close the serial port.

conf_adc(*pinput=8*, *ninput=0*, *gain=0*, *nsamples=20*)
Configure the analog-to-digital converter.

Get the parameters for configure the analog-to-digital converter.

Parameters

- **pinput** – Positive input [1:8].
- **ninput** – Negative input.
- **gain** – Analog gain.
- **nsamples** – Number of samples per data point [0-255).

Raises ValueError

create_burst(**args*, ***kwargs*)
Create Burst experiment.

See the *DAQBurst* class constructor for more info.

create_external(*mode*, *clock_input*, **args*, ***kwargs*)
Create External experiment.

See the *DAQExternal* class constructor for more info.

create_stream(*mode*, **args*, ***kwargs*)
Create Stream experiment.

See the *DAQStream* class constructor for more info.

enable_crc(*on*)
Enable/Disable the cyclic redundancy check.

Parameters **on** – Enable/disable CRC checking (bool).

flush()
Flush internal buffers.

flush_channel(*number*)
Flush the channel.

Parameters **number** – Number of DataChannel to flush.

Returns ValueError

get_adc_calib()
Get the ADC calibration.

Returns List of ADC calibration registers

get_capture(*mode*)
Get Capture reading for the period length.

Parameters **mode** – Period length (0: Low cycle, 1: High cycle, 2: Full period)

Returns

- mode
- period: The period length in microseconds

Raises ValueError

get_counter(*reset*)
Get the counter value.

Parameters **reset** – reset the counter after perform reading (boolean).

get_dac_calib()
Get the DAC calibration.

Returns List of DAC calibration registers

get_encoder()
Get current encoder relative position.

Returns Position: The actual encoder value.

get_info()
Read device information.

Returns [hardware_version, firmware_version, device_id]

get_state_ch(*number*)
Get state of the DataChannel.

Parameters **number** – Number of the DataChannel.

Raises ValueError

init_capture(*period*)
Start Capture Mode around a given period.

Parameters **period** – Estimated period of the wave (in microseconds).

Raises ValueError

init_counter(*edge*)
Initialize the edge counter and configure which edge increments the count.

Parameters **edge** – high-to-low (False) or low-to-high (True).

init_encoder(*resolution*)
Start Encoder function.

Parameters **resolution** – Maximum number of ticks per round [0:65535].

Raises ValueError

init_pwm(*duty*, *period*)
Start PWM output with a given period and duty cycle.

Parameters

- **duty** – High time of the signal [0:1023](0 always low, 1023 always high).
- **period** – Period of the signal (microseconds) [0:65535].

Raises ValueError

is_measuring
True if any experiment is going on.

open()
Open the serial port.

read_adc()
Read data from ADC and return the raw value.

Returns Raw ADC value.

read_all(*nsamples=20*, *gain=0*)
Read data from all analog inputs

Parameters

- **nsamples** – Number of samples per data point [0-255] (default=20)
- **gain** – Analog gain (default=1)

Returns Values[0:7]: List of the analog reading on each input

read_analog()
Read data from ADC in volts.

Returns Voltage value.

read_eeprom(*pos*)
Read a byte from the EEPROM.

Parameters

- **val** – value to write.
- **pos** – position in memory.

Raises ValueError

read_pio(*number*)
Read PIO input value (0: low, 1: high).

Parameters **number** – PIO number.

Returns Read value.

Raises ValueError

read_port()
Read all PIO values.

Returns Binary value of the port.

remove_experiment(*experiment*)
Delete a single experiment.

Parameters **experiment** – reference of the experiment to remove.

Raises ValueError

send_command(*command*, *ret_fmt=None*)
Build a command packet, send it to the openDAQ and process the response.

Parameters

- **command** – Command string.
- **ret_fmt** – Payload format of the response using python 'struct' format characters. I ret_fmt is None, no response is expected.

Returns Command ID and arguments of the response.

Raises LengthError: The legth of the response is not the expected.

serial_str

set_adc_calib(*regs*)
Set the ADC calibration.

Parameters **regs** – A list of CalibReg objects.

Raises ValueError, IndexError

set_analog(*volts*, *number=1*)
Set DAC output (volts). Set the output voltage of the DAC. :param volts: DAC output value in volts. :raises: ValueError

set_dac(*raw*, *number=1*)
Set DAC output (raw value). Set the raw value of the DAC.

"param raw: Raw ADC value. :raises: ValueError

set_dac_calib(*regs*)
Set the DAC calibration.

Parameters **regs** – A list of CalibReg objects.

Raises ValueError, IndexError

set_id(*id*)
Identify openDAQ device.

Parameters **id** – id number of the device [000:999]

Raises ValueError

set_led(*color*, *number=1*)
Choose LED status. LED switch on (green, red or orange) or switch off.

Parameters **color** – LED color (use *LedColor*).

Raises ValueError

set_pio(*number*, *value*)
Write PIO output value. Set the value of the PIO terminal (0: low, 1: high).

Parameters

- **number** – PIO number.
- **value** – digital value (0: low, 1: high)

Raises ValueError

set_pio_dir(*number*, *output*)
Configure PIO direction. Set the direction of a specific PIO terminal (D1-D6).

Parameters

- **number** – PIO number.
- **output** – PIO direction (0 input, 1 output).

Raises ValueError

set_port(*value*)
Write all PIO values. Set the value of all Dx terminals.

Parameters **value** – Port output byte (bits: 0:low, 1:high).

Raises ValueError

set_port_dir(*output*)
Configure all PIOs directions. Set the direction of all D1-D6 terminals.

Parameters **output** – Port directions byte (bits: 0:input, 1:output).

Raises ValueError

spi_config(*cpol*, *cpha*)
Bit-Bang SPI configure (clock properties).

Parameters

- **cpol** – Clock polarity (clock pin state when inactive).
- **cpha** – Clock phase (leading 0, or trailing 1 edges read).

Raises ValueError

spi_setup(*nbytes*, *sck=1*, *mosi=2*, *miso=3*)
Bit-Bang SPI setup (PIO numbers to use).

Parameters

- **nbytes** – Number of bytes.
- **sck** – Clock pin.
- **mosi** – MOSI pin (master out / slave in).
- **miso** – MISO pin (master in / slave out).

Raises ValueError

spi_write(*value*, *word=False*)
Bit-bang SPI transfer (send+receive) a byte or a word.

Parameters

- **value** – Data to send (byte/word to transmit).
- **word** – send a 2-byte word, instead of a byte.

Raises ValueError

start()
Start all available experiments.

stop(*clear=False*)
Stop all running experiments and exit threads.

Parameters **clear** – If True, the experiment list will be cleared. The

experiments will no longer be available.

stop_capture()
Stop Capture mode.

stop_encoder()
Stop encoder

stop_pwm()
Stop PWM

trigger_mode(*number*)
Get the trigger mode of the DataChannel.

Parameters **number** – Number of the DataChannel.

Raises ValueError

write_eeprom(*pos*, *val*)
Write a byte in the EEPROM.

Parameters **id** – id number of the device [000:999].

Raises ValueError

class opendaq.daq.**LedColor**
Bases: enum.IntEnum

Valid LED colors.

GREEN = 1

OFF = 0

ORANGE = 3

RED = 2

2.1.1 Experiment classes

class opendaq.experiment.**DAQBurst**(*mode*, *period*, *npoints=10*, *continuous=False*, *buffersize=4000*)
Bases: *opendaq.experiment.DAQExperiment*

Burst experiment.

Parameters

- **mode** – Define data source or destination (use *ExpMode*).
- **period** – Period of the stream experiment (microseconds) [1:65536]
- **npoints** – Total number of points for the experiment [0:65536]
- **continuous** – Indicates if the experiment is continuous (False: run once, True: continuous).
- **buffersize** – Buffer size

Raises LengthError (too many experiments at the same time), ValueError (values out of range)

class opendaq.experiment.**DAQExperiment**
Bases: object

add_points(*points*)
Write a single point into the ring buffer.

analog_setup(*pinput=1*, *ninput=0*, *gain=1*, *nsamples=20*)
Configure a channel for a generic stream experiment.

get_mode()
Return mode.

get_params()
Return gain, pinput and ninput.

get_preload_data()
Return preload_data and preload_offset.

load_signal(*data*, *offset=0*)

read()
Return all available points from the ring buffer.

trigger_setup(*mode=<Trigger.SW: 0>*, *value=0*)
Channge the trigger mode of datachannel.

Parameters

- **mode** – Trigger mode (use *Trigger*).

- **value** – Value of the trigger mode.

Raises ValueError

class opendaq.experiment.**DAQExternal**(*mode*, *clock_input*, *edge=1*, *npoints=10*, *continuous=False*, *buffersize=1000*)

Bases: *opendaq.experiment.DAQExperiment*

External experiment.

Parameters

- **mode** – Define data source or destination (use *ExpMode*).
- **clock_input** – Digital input used as external clock
- **edge** – New data on rising (1) or falling (0) edges [0:1]
- **npoints** – Total number of points for the experiment [0:65536]
- **continuous** – Indicates if the experiment is continuous (False: run once, True: continuous).
- **buffersize** – Buffer size

Raises LengthError (too many experiments at the same time, ValueError (values out of range)

class opendaq.experiment.**DAQStream**(*mode*, *number*, *period*, *npoints=10*, *continuous=False*, *buffersize=1000*)

Bases: *opendaq.experiment.DAQExperiment*

Stream experiment.

Parameters

- **mode** – Define data source or destination (use *ExpMode*).
- **period** – Period of the stream experiment (milliseconds) [1:65536]
- **npoints** – Total number of points for the experiment [0:65536] (0 indicates continuous acquisition).
- **continuous** – Indicates if experiment is continuous (True) or one-shot (False).
- **buffersize** – Buffer size.

Raises LengthError (too many experiments at the same time), ValueError (values out of range)

class opendaq.experiment.**ExpMode**

Bases: enum.IntEnum

Valid experiment modes.

ANALOG_IN = 0

ANALOG_OUT = 1

CAPTURE_IN = 5

COUNTER_IN = 4

DIGITAL_IN = 2

DIGITAL_OUT = 3

class opendaq.experiment.**Trigger**

Bases: enum.IntEnum

Valid trigger modes.

```
ABIG = 10
ASML = 20
DIN1 = 1
DIN2 = 2
DIN3 = 3
DIN4 = 4
DIN5 = 5
DIN6 = 6
SW = 0
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

I

L

O

P

R

S

T

W