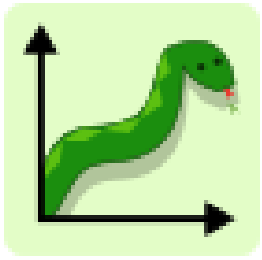
PyMeasure Documentation

Release 0.11.2.dev189+g20fe653.d20230103

PyMeasure Developers

Jan 03, 2023

LEARNING PYMEASURE



PyMeasure

PyMeasure makes scientific measurements easy to set up and run. The package contains a repository of instrument classes and a system for running experiment procedures, which provides graphical interfaces for graphing live data and managing queues of experiments. Both parts of the package are independent, and when combined provide all the necessary requirements for advanced measurements with only limited coding.

Installing Python and PyMeasure are demonstrated in the *Quick Start guide*. From there, checkout the existing *instruments that are available for use*.

PyMeasure is currently under active development, so please report any issues you experience on our Issues page.

The main documentation for the site is organized into a couple sections:

- *Learning PyMeasure*
- *API Reference*
- *About PyMeasure*

Information about development is also available:

- *Getting involved*

CHAPTER

ONE

INTRODUCTION

PyMeasure uses an object-oriented approach for communicating with scientific instruments, which provides an intuitive interface where the low-level SCPI and GPIB commands are hidden from normal use. Users can focus on solving the measurement problems at hand, instead of re-inventing how to communicate with instruments.

Instruments with VISA (GPIB, Serial, etc) are supported through the PyVISA package under the hood. Prologix GPIB adapters are also supported. Communication protocols can be swapped, so that instrument classes can be used with all supported protocols interchangeably.

In order to keep the corresponding numbers and physical units (e.g. 5 meters) together, pint quantities can be used. That way it is easy to handle different orders of magnitude (meters and centimeters) or different units (meters and feet).

Before using PyMeasure, you may find it helpful to be acquainted with basic Python programming for the sciences and understand the concept of objects.

1.1 Instrument ready

The package includes a number of *instruments already defined*. Their definitions are organized based on the manufacturer name of the instrument. For example the class that defines the *Keithley 2400 SourceMeter* can be imported by calling:

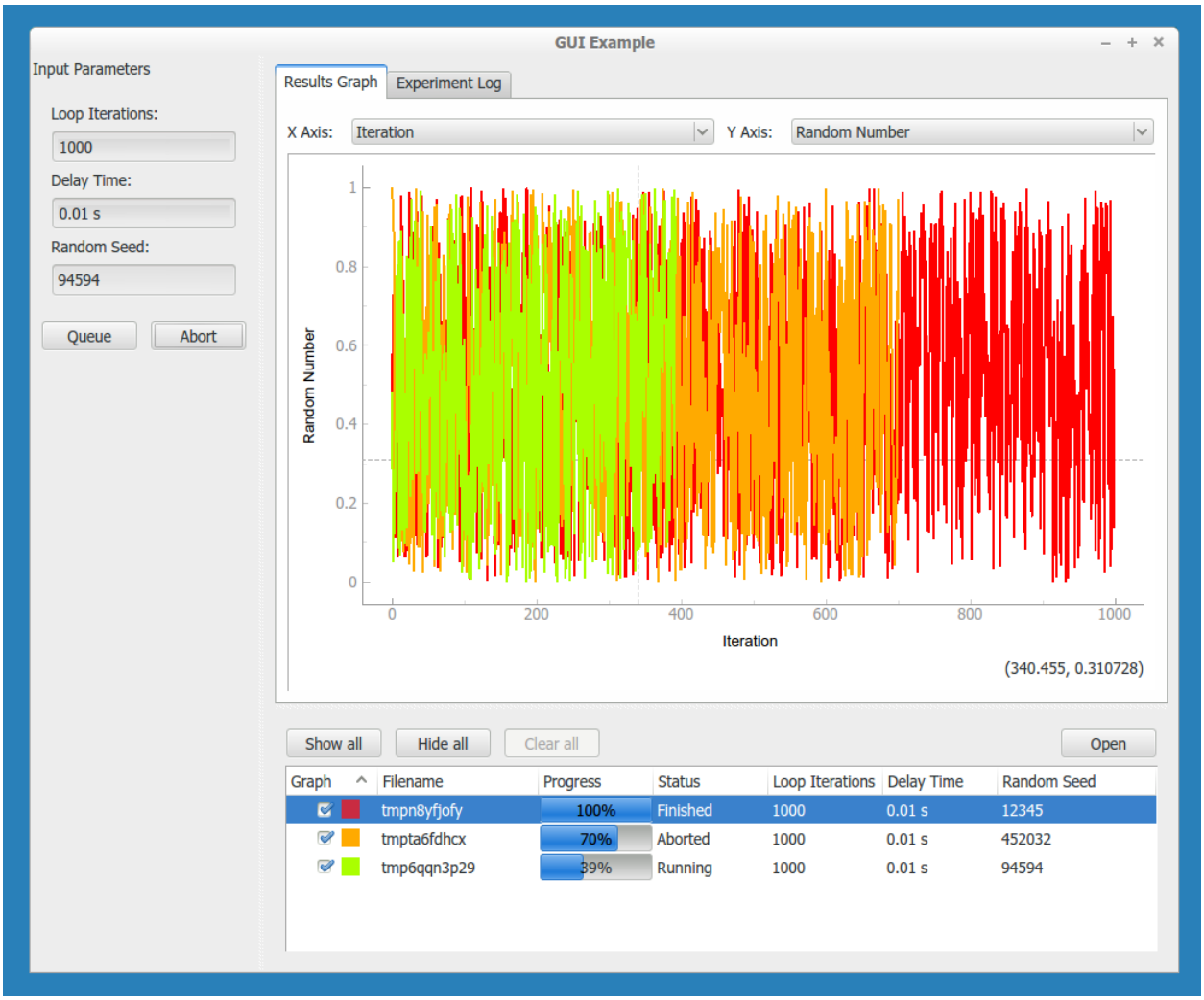
```
from pymeasure.instruments.keithley import Keithley2400
```

The *Tutorials* section will go into more detail on *connecting to an instrument*. If you don't find the instrument you are looking for, but are interested in contributing, see the documentation on *adding an instrument*.

1.2 Graphical displays

Graphical user interfaces (GUIs) can be easily generated to manage execution of measurement procedures with PyMeasure. This includes live plotting for data, and a queue system for managing large numbers of experiments.

These features are explored in the *Using a graphical interface* tutorial.



The GUIs are not restricted to the instruments included in this package. Any python instrument may be used. For example, this script demonstrates how to use an InstrumentKit instrument.

CHAPTER

TWO

QUICK START

This section provides instructions for getting up and running quickly with PyMeasure.

2.1 Setting up Python

The easiest way to install the necessary Python environment for PyMeasure is through the Anaconda distribution, which includes 720 scientific packages. The advantage of using this approach over just relying on the `pip` installer is that it Anaconda correctly installs the required Qt libraries.

Download and install the appropriate Python version of Anaconda for your operating system.

2.2 Installing PyMeasure

2.2.1 Install with conda

If you have the Anaconda distribution you can use the conda package mangager to easily install PyMeasure and all required dependencies.

Open a terminal and type the following commands (on Windows look for the *Anaconda Prompt* in the Start Menu):

```
conda config --add channels conda-forge
conda install pymeasure
```

This will install PyMeasure and all the required dependencies.

2.2.2 Install with `pip`

PyMeasure can also be installed with `pip`.

```
pip install pymeasure
```

Depending on your operating system, using this method may require additional work to install the required dependencies, which include the Qt libraries.

2.2.3 Installing VISA

Typically, communication with your instrument will happen using PyVISA, which is installed automatically. However, this needs a VISA implementation installed to handle device communication. If you do not already know what this means, install the pure-Python `pyvisa-py` package (using the same installation you used above). If you want to know more, consult the PyVISA documentation.

2.2.4 Checking the version

Now that you have Python and PyMeasure installed, open your python environment (e.g. a REPL or Jupyter notebook) to test which version you have installed. Execute the following Python code.

```
import pymeasure
pymeasure.__version__
```

You should see the version of PyMeasure printed out. At this point you have PyMeasure installed, and you are ready to start using it! Are you ready to *connect to an instrument*?

CHAPTER

THREE

TUTORIALS

The following sections provide instructions for getting started with PyMeasure.

3.1 Connecting to an instrument

After following the *Quick Start* section, you now have a working installation of PyMeasure. This section describes connecting to an instrument, using a Keithley 2400 SourceMeter as an example. To follow the tutorial, open a command prompt, IPython terminal, or Jupyter notebook.

First import the instrument of interest.

```
from pymeasure.instruments.keithley import Keithley2400
```

Then construct an object by passing the VISA address. For this example we connect to the instrument over GPIB (using VISA) with an address of 4:

```
sourcemeter = Keithley2400("GPIB::4")
```

Note: Passing an appropriate resource string is the default method when creating pymeasure instruments. See the *adapters* section below for more details.

If you are not sure about the correct resource string identifying your instrument, you can run the *pymeasure.instruments.list_resources()* function to list all available resources:

```
from pymeasure.instruments import list_resources
list_resources()
```

For instruments with standard SCPI commands, an `id` property will return the results of a `*IDN?` SCPI command, identifying the instrument.

```
sourcemeter.id
```

This is equivalent to manually calling the SCPI command.

```
sourcemeter.ask("*IDN?")
```

Here the `ask` method writes the SCPI command, reads the result, and returns that result. This is further equivalent to calling the methods below.

```
sourcemeter.write("*IDN?")
sourcemeter.read()
```

This example illustrates that the top-level methods like `id` are really composed of many lower-level methods. Both can be called depending on the operation that is desired. PyMeasure hides the complexity of these lower-level operations, so you can focus on the bigger picture.

Instruments are also equipped to be used in a `with` statement.

```
with Keithley2400("GPIB::4") as sourcemeter:
    sourcemeter.id
```

When the `with`-block is exited, the `shutdown` method of the instrument will be called, turning the system into a safe state.

```
with Keithley2400("GPIB::4") as sourcemeter:
    sourcemeter.isShutdown == False
sourcemeter.isShutdown == True
```

3.1.1 Using adapters

PyMeasure supports a number of adapters, which are responsible for communicating with the underlying hardware. In the example above, we passed the string "GPIB::4" when constructing the instrument. By default this constructs a VISAAdapter (our most popular, default adapter) to connect to the instrument using VISA. Passing a string (or integer in case of GPIB) is by far the most typical way to create pymeasure instruments.

Sometimes, you might need to go beyond the usual setup, which is also possible. Instead of passing a string, you could equally pass an adapter object.

```
from pymeasure.adapters import VISAAdapter

adapter = VISAAdapter("GPIB::4")
sourcemeter = Keithely2400(adapter)
```

To instead use a Prologix GPIB device connected on `/dev/ttyUSB0` (proper permissions are needed in Linux, see *PrologixAdapter*), the adapter is constructed in a similar way. The Prologix adapter can be shared by many instruments. Therefore, new *PrologixAdapter* instances with different GPIB addresses can be generated from an already existing instance.

```
from pymeasure.adapters import PrologixAdapter

adapter = PrologixAdapter('ASRL/dev/ttyUSB0::INSTR', address=7)
sourcemeter = Keithley2400(adapter)  # at GPIB address 7
multimeter = Keithley2000(adapter.gpib(9))  # at GPIB address 9
```

Some equipment may require the vxi-11 protocol for communication. An example would be a Agilent E5810B ethernet to GPIB bridge. To use this type equipment the python-vxi11 library has to be installed which is part of the extras package requirements.

```
from pymeasure.adapters import VXI11Adapter
from pymeasure.instruments import Instrument

adapter = VXI11Adapter("TCPIP::192.168.0.100::inst0::INSTR")
instr = Instrument(adapter, "my_instrument")
```

3.1.2 Modifying connection settings

Sometimes you want to tweak the connection settings when talking to a device. This might be because you have a non-standard device or connection, or are troubleshooting why a device does not reply.

When using a string or integer to connect to an instrument, a *VISAAdapter* is used internally. Additional settings need to be passed in as keyword arguments. For example, to use a fast baud rate on a quick connection when connecting to the Keithely2400 as above, do

```
sourcemeter = Keithley2400("ASRL2", timeout=500, baud_rate=115200)
```

This overrides any defaults that may be defined for the instrument, either generally valid ones like `timeout` or interface-specific ones like `baud_rate`.

If you use an invalid argument, either misspelled or not valid for the chosen interface, an exception will be raised.

When using a separately-created Adapter instance, you define any custom settings when creating the adapter. Any keyword arguments passed in are discarded.

The above examples illustrate different methods for communicating with instruments, using adapters to keep instrument code independent from the communication protocols. Next we present the methods for setting up measurements.

3.2 Making a measurement

This tutorial will walk you through using PyMeasure to acquire a current-voltage (IV) characteristic using a Keithley 2400. Even if you don't have access to this instrument, this tutorial will explain the method for making measurements with PyMeasure. First we describe using a simple script to make the measurement. From there, we show how *Procedure* objects greatly simplify the workflow, which leads to making the measurement with a graphical interface.

3.2.1 Using scripts

Scripts are a quick way to get up and running with a measurement in PyMeasure. For our IV characteristic measurement, we perform the following steps:

1) Import the necessary packages
2) Set the input parameters to define the measurement
3) Set source_current and measure_voltage parameters
4) Connect to the Keithley 2400
5) Set up the instrument for the IV characteristic
6) Allocate arrays to store the resulting measurements
7) Loop through the current points, measure the voltage, and record
8) Save the final data to a CSV file
9) Shutdown the instrument

These steps are expressed in code as follows.

```
# Import necessary packages
from pymeasure.instruments.keithley import Keithley2400
import numpy as np
import pandas as pd
from time import sleep

# Set the input parameters
data_points = 50
averages = 10
max_current = 0.001
min_current = -max_current

# Set source_current and measure_voltage parameters
current_range = 10e-3  # in Amps
compliance_voltage = 10  # in Volts
measure_nplc = 0.1  # Number of power line cycles
voltage_range = 1  # in VOlts

# Connect and configure the instrument
sourcemeter = Keithley2400("GPIB::24")
sourcemeter.reset()
sourcemeter.use_front_terminals()
sourcemeter.apply_current(current_range, compliance_voltage)
sourcemeter.measure_voltage(measure_nplc, voltage_range)
sleep(0.1)  # wait here to give the instrument time to react
sourcemeter.stop_buffer()
sourcemeter.disable_buffer()

# Allocate arrays to store the measurement results
currents = np.linspace(min_current, max_current, num=data_points)
voltages = np.zeros_like(currents)
voltage_stds = np.zeros_like(currents)

sourcemeter.enable_source()

# Loop through each current point, measure and record the voltage
for i in range(data_points):
    sourcemeter.config_buffer(averages)
    sourcemeter.source_current = currents[i]
    sourcemeter.start_buffer()
    sourcemeter.wait_for_buffer()
    # Record the average and standard deviation
    voltages[i] = sourcemeter.means[0]
    sleep(1.0)
    voltage_stds[i] = sourcemeter.standard_devs[0]

# Save the data columns in a CSV file
data = pd.DataFrame({
    'Current (A)': currents,
    'Voltage (V)': voltages,
    'Voltage Std (V)': voltage_stds,
})
data.to_csv('example.csv')
```

(continues on next page)

(continued from previous page)

```
sourcemeter.shutdown()
```

Running this example script will execute the measurement and save the data to a CSV file. While this may be sufficient for very basic measurements, this example illustrates a number of issues that PyMeasure solves. The issues with the script example include:

- The progress of the measurement is not transparent
- Input parameters are not associated with the data that is saved
- Data is not plotted during the execution (nor at all in this case)
- Data is only saved upon successful completion, which is otherwise lost
- Canceling a running measurement causes the system to end in a undetermined state
- Exceptions also end the system in an undetermined state

The *Procedure* class allows us to solve all of these issues. The next section introduces the *Procedure* class and shows how to modify our script example to take advantage of these features.

3.2.2 Using Procedures

The Procedure object bundles the sequence of steps in an experiment with the parameters required for its successful execution. This simple structure comes with huge benefits, since a number of convenient tools for making the measurement use this common interface.

Let's start with a simple example of a procedure which loops over a certain number of iterations. We make the SimpleProcedure object as a sub-class of Procedure, since SimpleProcedure *is a* Procedure.

```
from time import sleep
from pymeasure.experiment import Procedure
from pymeasure.experiment import IntegerParameter

class SimpleProcedure(Procedure):

    # a Parameter that defines the number of loop iterations
    iterations = IntegerParameter('Loop Iterations')

    # a list defining the order and appearance of columns in our data file
    DATA_COLUMNS = ['Iteration']

    def execute(self):
        """Execute the procedure.

        Loops over each iteration and emits the current iteration,
        before waiting for 0.01 sec, and then checking if the procedure
        should stop.
        """
        for i in range(self.iterations):
            self.emit('results', {'Iteration': i})
            sleep(0.01)
            if self.should_stop():
                break
```

At the top of the SimpleProcedure class we define the required Parameters. In this case, `iterations` is a IntegerParameter that defines the number of loops to perform. Inside our Procedure class we reference the value in the iterations Parameter by the class variable where the Parameter is stored (`self.iterations`). PyMeasure swaps out the Parameters with their values behind the scene, which makes accessing the values of parameters very convenient.

We define the data columns that will be recorded in a list stored in `DATA_COLUMNS`. This sets the order by which columns are stored in the file. In this example, we will store the Iteration number for each loop iteration.

The `execute` methods defines the main body of the procedure. Our example method consists of a loop over the number of iterations, in which we emit the data to be recorded (the Iteration number). The data is broadcast to any number of listeners by using the `emit` method, which takes a topic as the first argument. Data with the `'results'` topic and the proper data columns will be recorded to a file. The sleep function in our example provides two very useful features. The first is to delay the execution of the next lines of code by the time argument in units of seconds. The seconds is that during this delay time, the CPU is free to perform other code. Successful measurements often require the intelligent use of sleep to deal with instrument delays and ensure that the CPU is not hogged by a single script. After our delay, we check to see if the Procedure should stop by calling `self.should_stop()`. By checking this flag, the Procedure will react to a user canceling the procedure execution.

This covers the basic requirements of a Procedure object. Now let's construct our SimpleProcedure object with 100 iterations.

```
procedure = SimpleProcedure()
procedure.iterations = 100
```

Next we will show how to run the procedure.

Running Procedures

A Procedure is run by a Worker object. The Worker executes the Procedure in a separate Python thread, which allows other code to execute in parallel to the procedure (e.g. a graphical user interface). In addition to performing the measurement, the Worker spawns a Recorder object, which listens for the `'results'` topic in data emitted by the Procedure, and writes those lines to a data file. The Results object provides a convenient abstraction to keep track of where the data should be stored, the data in an accessible form, and the Procedure that pertains to those results.

We first construct a Results object for our Procedure.

```
from pymeasure.experiment import Results

data_filename = 'example.csv'
results = Results(procedure, data_filename)
```

Constructing the Results object for our Procedure creates the file using the `data_filename`, and stores the Parameters for the Procedure. This allows the Procedure and Results objects to be reconstructed later simply by loading the file using `Results.load(data_filename)`. The Parameters in the file are easily readable.

We now construct a Worker with the Results object, since it contains our Procedure.

```
from pymeasure.experiment import Worker

worker = Worker(results)
```

The Worker publishes data and other run-time information through specific queues, but can also publish this information over the local network on a specific TCP port (using the optional `port` argument. Using TCP communication allows great flexibility for sharing information with Listener objects. Queues are used as the standard communication method because they preserve the data order, which is of critical importance to storing data accurately and reacting to the measurement status in order.

Now we are ready to start the worker.

```
worker.start()
```

This method starts the worker in a separate Python thread, which allows us to perform other tasks while it is running. When writing a script that should block (wait for the Worker to finish), we need to join the Worker back into the main thread.

```
worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
```

Let's put all the pieces together. Our SimpleProcedure can be run in a script by the following.

```
from time import sleep
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter

class SimpleProcedure(Procedure):

    # a Parameter that defines the number of loop iterations
    iterations = IntegerParameter('Loop Iterations')

    # a list defining the order and appearance of columns in our data file
    DATA_COLUMNS = ['Iteration']

    def execute(self):
        """Execute the procedure.

        Loops over each iteration and emits the current iteration,
        before waiting for 0.01 sec, and then checking if the procedure
        should stop.
        """
        for i in range(self.iterations):
            self.emit('results', {'Iteration': i})
            sleep(0.01)
            if self.should_stop():
                break

if __name__ == "__main__":
    procedure = SimpleProcedure()
    procedure.iterations = 100

    data_filename = 'example.csv'
    results = Results(procedure, data_filename)

    worker = Worker(results)
    worker.start()

    worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
```

Here we have included an if statement to only run the script if the __name__ is __main__. This precaution allows us to import the SimpleProcedure object without running the execution.

Using Logs

Logs keep track of important details in the execution of a procedure. We describe the use of the Python logging module with PyMeasure, which makes it easy to document the execution of a procedure and provides useful insight when diagnosing issues or bugs.

Let's extend our SimpleProcedure with logging.

```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

from time import sleep
from pymeasure.log import console_log
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter

class SimpleProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations')

    DATA_COLUMNS = ['Iteration']

    def execute(self):
        log.info("Starting the loop of %d iterations" % self.iterations)
        for i in range(self.iterations):
            data = {'Iteration': i}
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            sleep(0.01)
            if self.should_stop():
                log.warning("Caught the stop flag in the procedure")
                break

if __name__ == "__main__":
    console_log(log)

    log.info("Constructing a SimpleProcedure")
    procedure = SimpleProcedure()
    procedure.iterations = 100

    data_filename = 'example.csv'
    log.info("Constructing the Results with a data file: %s" % data_filename)
    results = Results(procedure, data_filename)

    log.info("Constructing the Worker")
    worker = Worker(results)
    worker.start()
    log.info("Started the Worker")

    log.info("Joining with the worker in at most 1 hr")
    worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
    log.info("Finished the measurement")
```

First, we have imported the Python logging module and grabbed the logger using the `__name__` argument. This gives

us logging information specific to the current file. Conversely, we could use the `''` argument to get all logs, including those of pymeasure. We use the `console_log` function to conveniently output the log to the console. Further details on how to use the logger are addressed in the Python logging documentation.

Storing metadata

Metadata (*pymeasure.experiment.parameters.Metadata*) allows storing information (e.g. the actual starting time, instrument parameters) about the measurement in the header of the datafile. These Metadata objects are evaluated and stored in the datafile only after the `startup` method has ran; this way it is possible to e.g. retrieve settings from an instrument and store them in the file. Using a Metadata is nearly as straightforward as using a Parameter; extending the example of above to include metadata, looks as follows:

```
from time import sleep, time
from pymeasure.experiment import Procedure
from pymeasure.experiment import IntegerParameter, Metadata

class SimpleProcedure(Procedure):

    # a Parameter that defines the number of loop iterations
    iterations = IntegerParameter('Loop Iterations')

    # the Metadata objects store information after the startup has ran
    starttime = Metadata('Start time', fget=time)
    custom_metadata = Metadata('Custom', default=1)

    # a list defining the order and appearance of columns in our data file
    DATA_COLUMNS = ['Iteration']

    def startup(self):
        self.custom_metadata = 20

    def execute(self):
        """ Loops over each iteration and emits the current iteration,
        before waiting for 0.01 sec, and then checking if the procedure
        should stop
        """
        for i in range(self.iterations):
            self.emit('results', {'Iteration': i})
            sleep(0.01)
            if self.should_stop():
                break
```

As with a Parameter, PyMeasure swaps out the Metadata with their values behind the scene, which makes accessing the values of Metadata very convenient.

The value of a Metadata can be set either using an `fget` method or manually in the startup method. The `fget` method, if provided, is ran after startup method. It can also be provided as a string; in that case it is assumed that the string contains the name of an attribute (either a callable or not) of the Procedure class which returns the value that is to be stored. This also allows to retrieve nested attributes (e.g. in order to store a property or method of an instrument) by separating the attributes with a period: e.g. *instrument_name.attribute_name* (or even *instrument_name.subclass_name.attribute_name*); note that here only the final element (i.e. *attribute_name* in the example) is allowed to refer to a callable. If neither an `fget` method is provided or a value manually set, the Metadata will return to its default value, if set. The formatting of the value of the Metadata-object can be controlled using the *fmt* argument.

Modifying our script

Now that you have a background on how to use the different features of the Procedure class, and how they are run, we will revisit our IV characteristic measurement using Procedures. Below we present the modified version of our example script, now as a IVProcedure class.

```
# Import necessary packages
from pymeasure.instruments.keithley import Keithley2400
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter, FloatParameter
from time import sleep
import numpy as np

from pymeasure.log import log, console_log

class IVProcedure(Procedure):

    data_points = IntegerParameter('Data points', default=20)
    averages = IntegerParameter('Averages', default=8)
    max_current = FloatParameter('Maximum Current', units='A', default=0.001)
    min_current = FloatParameter('Minimum Current', units='A', default=-0.001)

    DATA_COLUMNS = ['Current (A)', 'Voltage (V)', 'Voltage Std (V)']

    def startup(self):
        log.info("Connecting and configuring the instrument")
        self.sourcemeter = Keithley2400("GPIB::24")
        self.sourcemeter.reset()
        self.sourcemeter.use_front_terminals()
        self.sourcemeter.apply_current(100e-3, 10.0)  # current_range = 100e-3,␣
↪compliance_voltage = 10.0
        self.sourcemeter.measure_voltage(0.01, 1.0)  # nplc = 0.01, voltage_range = 1.0
        sleep(0.1)  # wait here to give the instrument time to react
        self.sourcemeter.stop_buffer()
        self.sourcemeter.disable_buffer()

    def execute(self):
        currents = np.linspace(
            self.min_current,
            self.max_current,
            num=self.data_points
        )
        self.sourcemeter.enable_source()
        # Loop through each current point, measure and record the voltage
        for current in currents:
            self.sourcemeter.config_buffer(IVProcedure.averages.value)
            log.info("Setting the current to %g A" % current)
            self.sourcemeter.source_current = current
            self.sourcemeter.start_buffer()
            log.info("Waiting for the buffer to fill with measurements")
            self.sourcemeter.wait_for_buffer()
            data = {
                'Current (A)': current,
```

(continues on next page)

(continued from previous page)

```
                'Voltage (V)': self.sourcemeter.means[0],
                'Voltage Std (V)': self.sourcemeter.standard_devs[0]
            }
            self.emit('results', data)
            sleep(0.01)
            if self.should_stop():
                log.info("User aborted the procedure")
                break

    def shutdown(self):
        self.sourcemeter.shutdown()
        log.info("Finished measuring")


if __name__ == "__main__":
    console_log(log)

    log.info("Constructing an IVProcedure")
    procedure = IVProcedure()
    procedure.data_points = 20
    procedure.averages = 8
    procedure.max_current = -0.001
    procedure.min_current = 0.001

    data_filename = 'example.csv'
    log.info("Constructing the Results with a data file: %s" % data_filename)
    results = Results(procedure, data_filename)

    log.info("Constructing the Worker")
    worker = Worker(results)
    worker.start()
    log.info("Started the Worker")

    log.info("Joining with the worker in at most 1 hr")
    worker.join(timeout=3600)  # wait at most 1 hr (3600 sec)
    log.info("Finished the measurement")
```

The parentheses in the `COLUMN` entries indicate the physical unit of the data in the corresponding column, e.g. `'Voltage Std (V)'` indicates Volts. If you want to indicate a dimensionless value, e.g. Mach number, you can use *(1)* instead. Combined units like *(m/s)* or the long form *(meter/second)* are also possible. The class `Results` ensures, that the data is stored in the correct unit, here Volts. For example a `pint.Quantity` of 500 mV will be stored as 0.5 V. A string will be converted first to a *Quantity* and a mere number (e.g. float, int, ...) is assumed to be already in the right unit (e.g 5 will be stored as 5 V). If the data entry is not compatible, either because it has the wrong unit, e.g. meters which is not a unit of voltage, or because it is no number at all, a warning is logged and *'nan'* will be stored in the file. If you do not specify a unit (i.e. no parentheses), no unit check is performed for this column, unless the data entry is a *Quantity* for that column. In this case, this column's unit is set to the base unit (e.g. meter if unit of the data entry is kilometers) of the data entry. From this point on, unit checks are enabled for this column. Also use columns without unit checks (i.e. without parentheses) for strings or booleans.

At this point, you are familiar with how to construct a Procedure sub-class. The next section shows how to put these procedures to work in a graphical environment, where will have live-plotting of the data and the ability to easily queue up a number of experiments in sequence. All of these features come from using the Procedure object.

3.3 Using a graphical interface

In the previous tutorial we measured the IV characteristic of a sample to show how we can set up a simple experiment in PyMeasure. The real power of PyMeasure comes when we also use the graphical tools that are included to turn our simple example into a full-flegged user interface.

3.3.1 Using the Plotter

While it lacks the nice features of the ManagedWindow, the Plotter object is the simplest way of getting live-plotting. The Plotter takes a Results object and plots the data at a regular interval, grabbing the latest data each time from the file.

> **Warning:** The example in this section is known to raise issues when executed: a *QApplication was not created in the main thread / nextEventMatchingMask should only be called from the Main Thread* warning is raised. While the example works without issues on some operating systems and python configurations, users are advised not to rely on the plotter while this issue is unresolved. Users can hence skip this example and continue with the *Using the ManagedWindow* section.

Let's extend our SimpleProcedure with a RandomProcedure, which generates random numbers during our loop. This example does not include instruments to provide a simpler example.

```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

import random
from time import sleep
from pymeasure.log import console_log
from pymeasure.display import Plotter
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

class RandomProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations')
    delay = FloatParameter('Delay Time', units='s', default=0.2)
    seed = Parameter('Random Seed', default='12345')

    DATA_COLUMNS = ['Iteration', 'Random Number']

    def startup(self):
        log.info("Setting the seed of the random number generator")
        random.seed(self.seed)

    def execute(self):
        log.info("Starting the loop of %d iterations" % self.iterations)
        for i in range(self.iterations):
            data = {
                'Iteration': i,
                'Random Number': random.random()
```

(continues on next page)

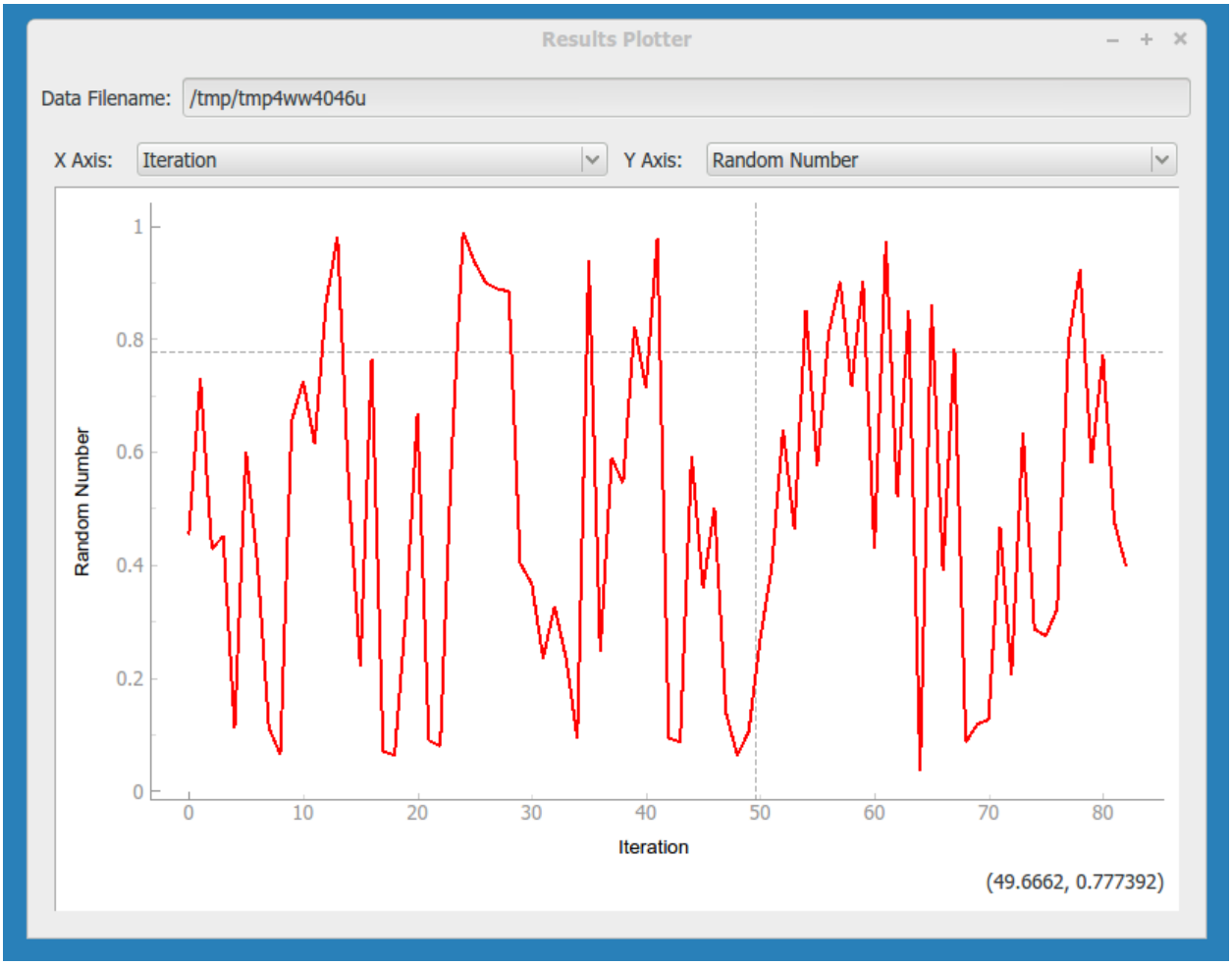
(continued from previous page)

```
            }
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            self.emit('progress', 100 * i / self.iterations)
            sleep(self.delay)
            if self.should_stop():
                log.warning("Caught the stop flag in the procedure")
                break


if __name__ == "__main__":
    console_log(log)

    log.info("Constructing a RandomProcedure")
    procedure = RandomProcedure()
    procedure.iterations = 100

    data_filename = 'random.csv'
    log.info("Constructing the Results with a data file: %s" % data_filename)
    results = Results(procedure, data_filename)

    log.info("Constructing the Plotter")
    plotter = Plotter(results)
    plotter.start()
    log.info("Started the Plotter")

    log.info("Constructing the Worker")
    worker = Worker(results)
    worker.start()
    log.info("Started the Worker")

    log.info("Joining with the worker in at most 1 hr")
    worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
    log.info("Finished the measurement")
```

The important addition is the construction of the Plotter from the Results object.

```
plotter = Plotter(results)
plotter.start()
```

The Plotter is started in a different process so that it can be run on a separate CPU for higher performance. The Plotter launches a Qt graphical interface using pyqtgraph which allows the Results data to be viewed based on the columns in the data.



3.3.2 Using the ManagedWindow

The ManagedWindow is the most convenient tool for running measurements with your Procedure. This has the major advantage of accepting the input parameters graphically. From the parameters, a graphical form is automatically generated that allows the inputs to be typed in. With this feature, measurements can be started dynamically, instead of defined in a script.

Another major feature of the ManagedWindow is its support for running measurements in a sequential queue. This allows you to set up a number of measurements with different input parameters, and watch them unfold on the live-plot. This is especially useful for long running measurements. The ManagedWindow achieves this through the Manager object, which coordinates which Procedure the Worker should run and keeps track of its status as the Worker progresses.

Below we adapt our previous example to use a ManagedWindow.

```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

import sys
import tempfile
import random
from time import sleep
```

(continues on next page)

(continued from previous page)

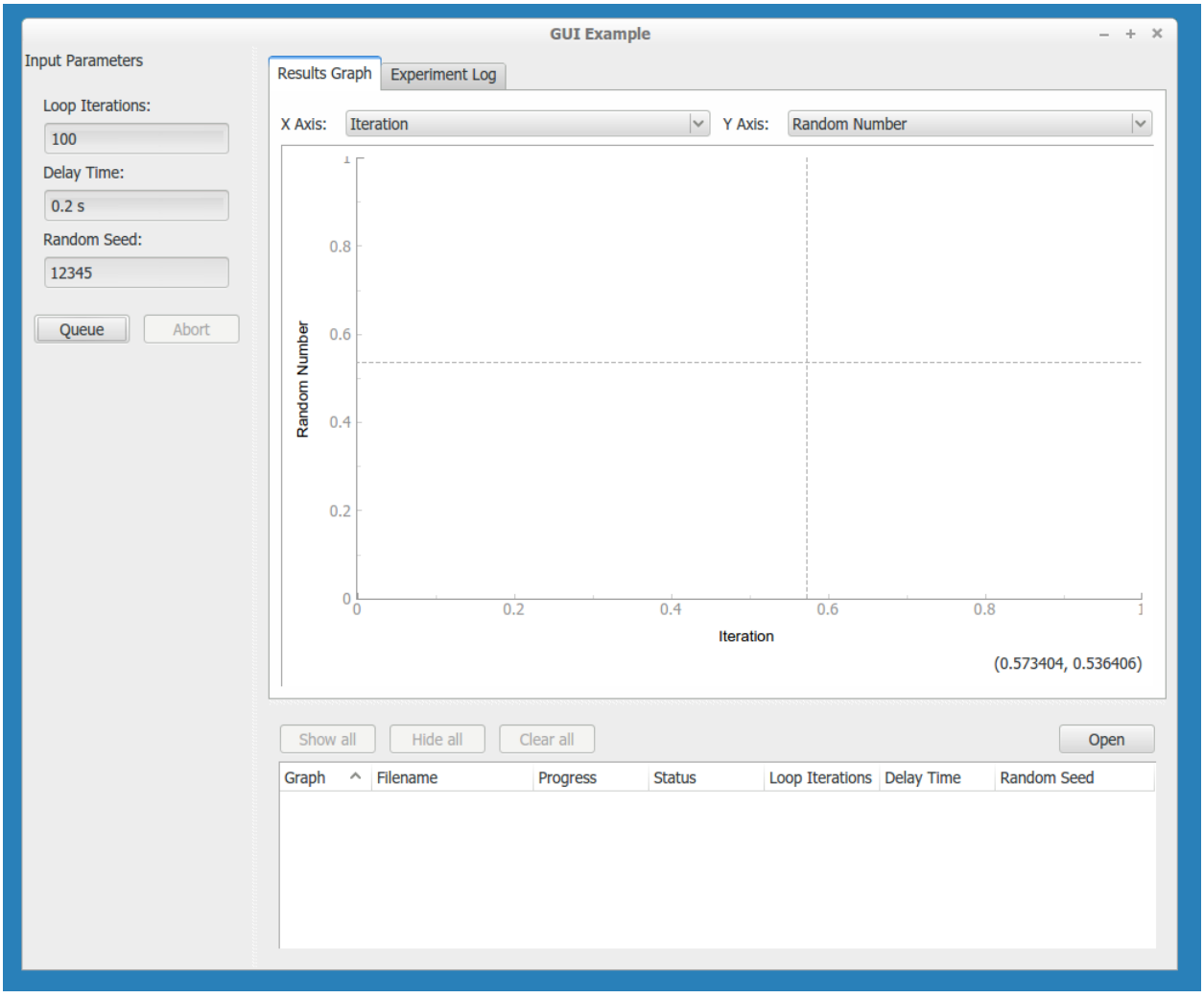
```
from pymeasure.log import console_log
from pymeasure.display.Qt import QtWidgets
from pymeasure.display.windows import ManagedWindow
from pymeasure.experiment import Procedure, Results
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

class RandomProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations')
    delay = FloatParameter('Delay Time', units='s', default=0.2)
    seed = Parameter('Random Seed', default='12345')

    DATA_COLUMNS = ['Iteration', 'Random Number']

    def startup(self):
        log.info("Setting the seed of the random number generator")
        random.seed(self.seed)

    def execute(self):
        log.info("Starting the loop of %d iterations" % self.iterations)
        for i in range(self.iterations):
            data = {
                'Iteration': i,
                'Random Number': random.random()
            }
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            self.emit('progress', 100 * i / self.iterations)
            sleep(self.delay)
            if self.should_stop():
                log.warning("Caught the stop flag in the procedure")
                break


class MainWindow(ManagedWindow):

    def __init__(self):
        super().__init__(
            procedure_class=RandomProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            x_axis='Iteration',
            y_axis='Random Number'
        )
        self.setWindowTitle('GUI Example')

    def queue(self):
        filename = tempfile.mktemp()

        procedure = self.make_procedure()
        results = Results(procedure, filename)
        experiment = self.new_experiment(results)
```

(continues on next page)

(continued from previous page)

```
        self.manager.queue(experiment)


if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

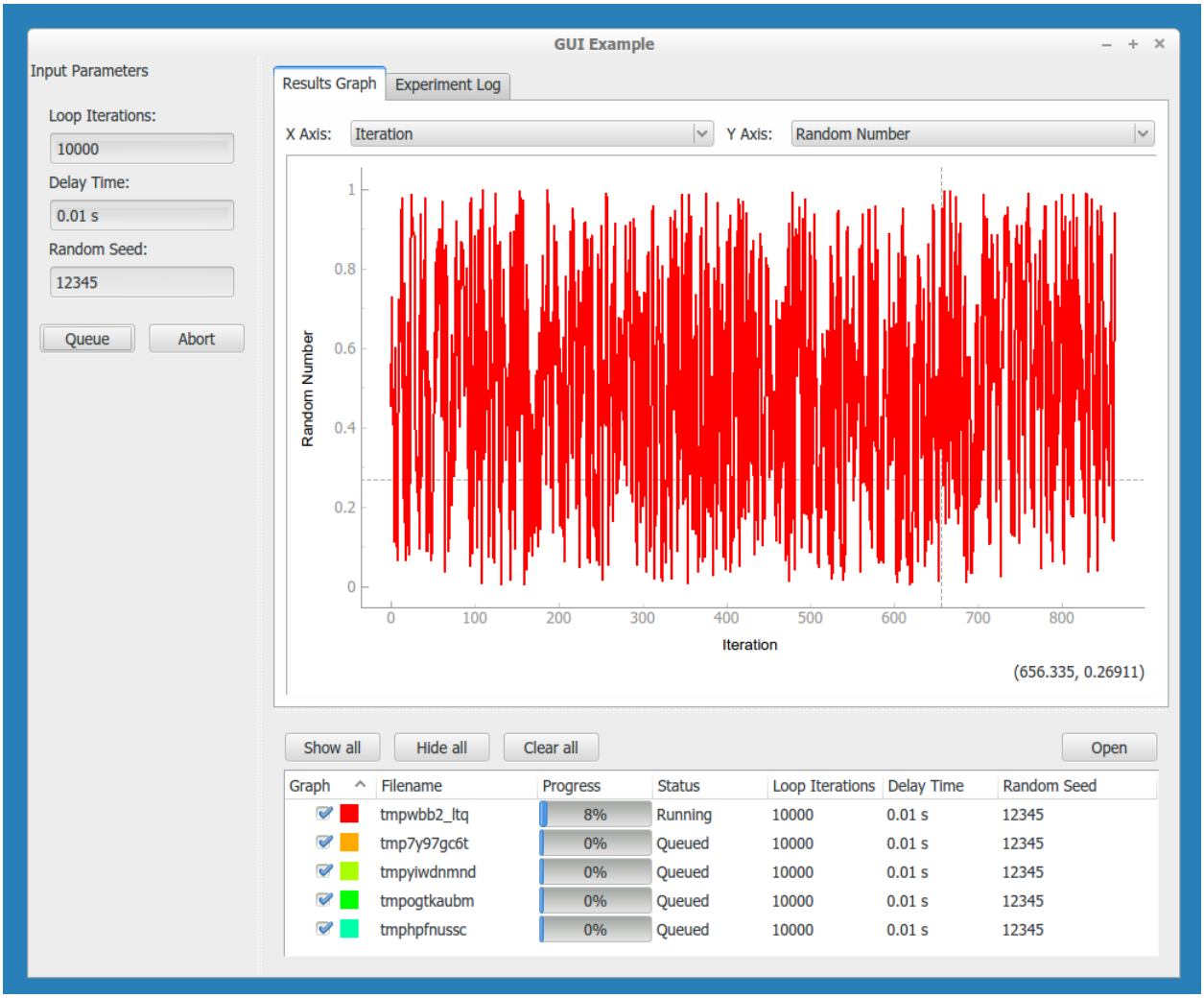
This results in the following graphical display.



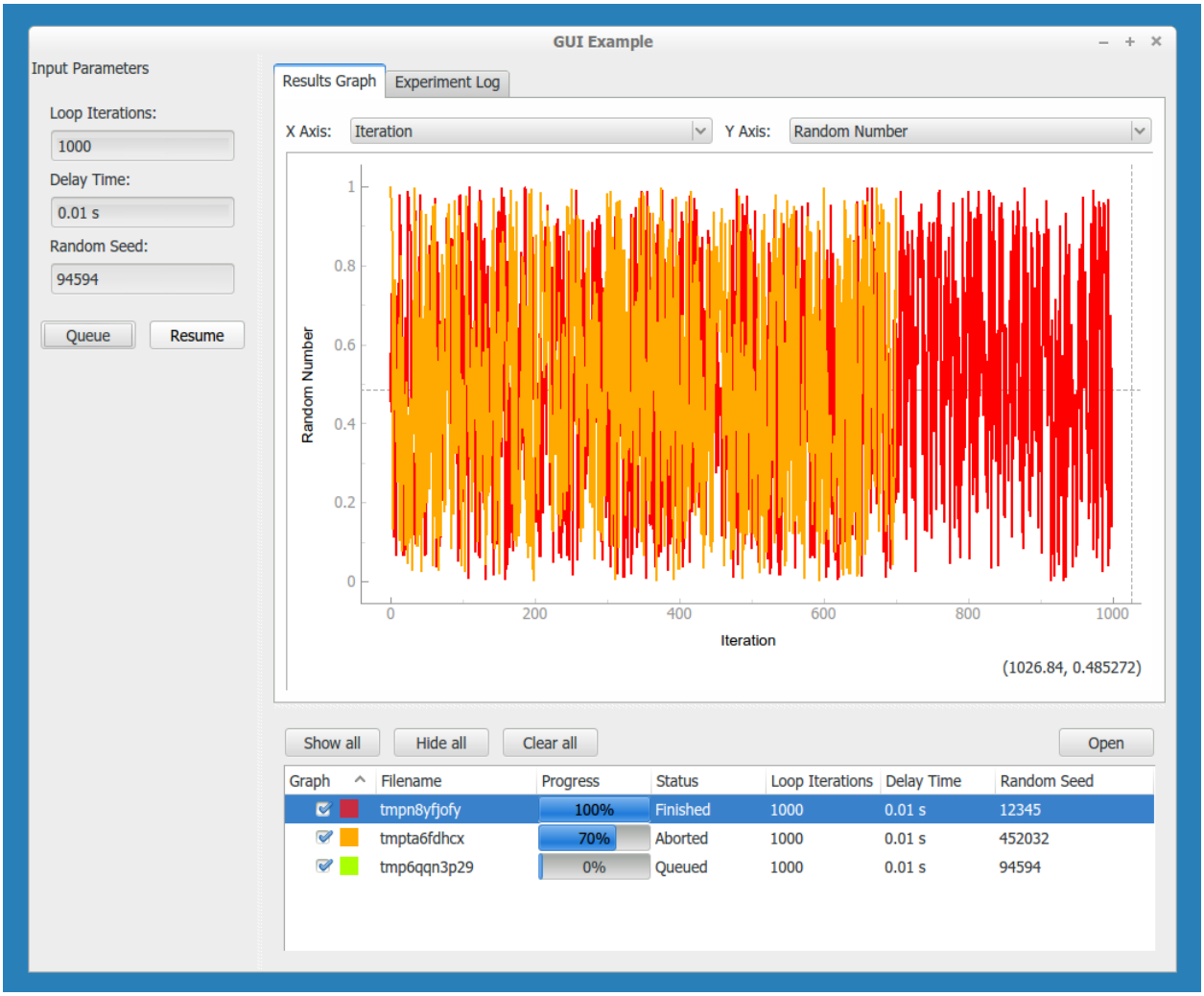
In the code, the MainWindow class is a sub-class of the ManagedWindow class. We override the constructor to provide information about the procedure class and its options. The `inputs` are a list of Parameters class-variable names, which the display will generate graphical fields for. When the list of inputs is long, a boolean key-word argument `inputs_in_scrollarea` is provided that adds a scrollbar to the input area. The `displays` is a list similar to the `inputs` list, which instead defines the parameters to display in the browser window. This browser keeps track of the experiments being run in the sequential queue.

The `queue` method establishes how the Procedure object is constructed. We use the `self.make_procedure` method to create a Procedure based on the graphical input fields. Here we are free to modify the procedure before putting it

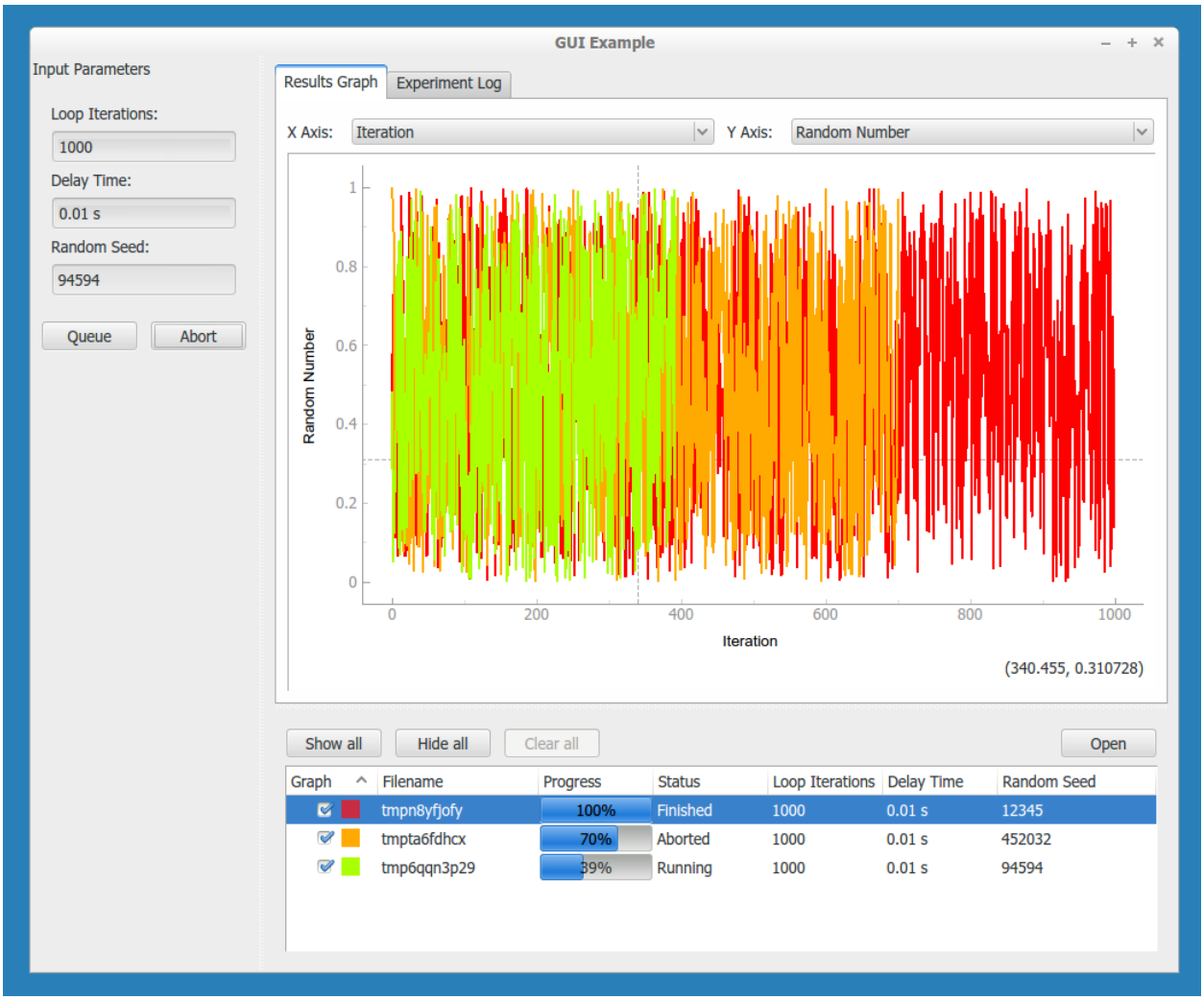
on the queue. In this context, the Manager uses an Experiment object to keep track of the Procedure, Results, and its associated graphical representations in the browser and live-graph. This is then given to the Manager to queue the experiment.



By default the Manager starts a measurement when its procedure is queued. The abort button can be pressed to stop an experiment. In the Procedure, the `self.should_stop` call will catch the abort event and halt the measurement. It is important to check this value, or the Procedure will not be responsive to the abort event.



If you abort a measurement, the resume button must be pressed to continue the next measurement. This allows you to adjust anything, which is presumably why the abort was needed.



Now that you have learned about the ManagedWindow, you have all of the basics to get up and running quickly with a measurement and produce an easy to use graphical interface with PyMeasure.

Note: For performance reasons, the default linewidth of all the graphs has been set to 1. If performance is not an issue, the linewidth can be changed to 2 (or any other value) for better visibility by using the *linewidth* keyword-argument in the *Plotter* or the *ManagedWindow*. Whenever a linewidth of 2 is prefered and a better performance is required, it is possible to enable using OpenGL in the import section of the file:

```
import pyqtgraph as pg
pg.setConfigOption("useOpenGL", True)
```

3.3.3 Customising the plot options

For both the PlotterWindow and ManagedWindow, plotting is provided by the pyqtgraph library. This library allows you to change various plot options, as you might expect: axis ranges (by default auto-ranging), logarithmic and semilogarithmic axes, downsampling, grid display, FFT display, etc. There are two main ways you can do this:

1. You can right click on the plot to manually change any available options. This is also a good way of getting an overview of what options are available in pyqtgraph. Option changes will, of course, not persist across a restart of your program.
2. You can programmatically set these options using pyqtgraph's PlotItem API, so that the window will open with these display options already set, as further explained below.

For *Plotter*, you can make a sub-class that overrides the *setup_plot()* method. This method will be called when the Plotter constructs the window. As an example

```
class LogPlotter(Plotter):
    def setup_plot(self, plot):
        # use logarithmic x-axis (e.g. for frequency sweeps)
        plot.setLogMode(x=True)
```

For *ManagedWindow*, the mechanism to customize plots is much more flexible by using specialization via inheritance. Indeed *ManagedWindowBase* is the base class for *ManagedWindow* and *ManagedImageWindow* which are subclasses ready to use for GUI.

3.3.4 Defining your own ManagedWindow's widgets

The parameter `widget_list` in *ManagedWindowBase* constructor allow to introduce user's defined widget in the GUI results display area. The user's widget should inherit from *TabWidget* and could reimplement any of the methods that needs customization. In order to get familiar with the mechanism, users can check the following widgets already provided:

- *LogWidget*
- *PlotWidget*
- *ImageWidget*
- `DockWidget`

3.3.5 Using the sequencer

As an extension to the way of graphically inputting parameters and executing multiple measurements using the *ManagedWindow*, *SequencerWidget* is provided which allows users to queue a series of measurements with varying one, or more, of the parameters. This sequencer thereby provides a convenient way to scan through the parameter space of the measurement procedure.

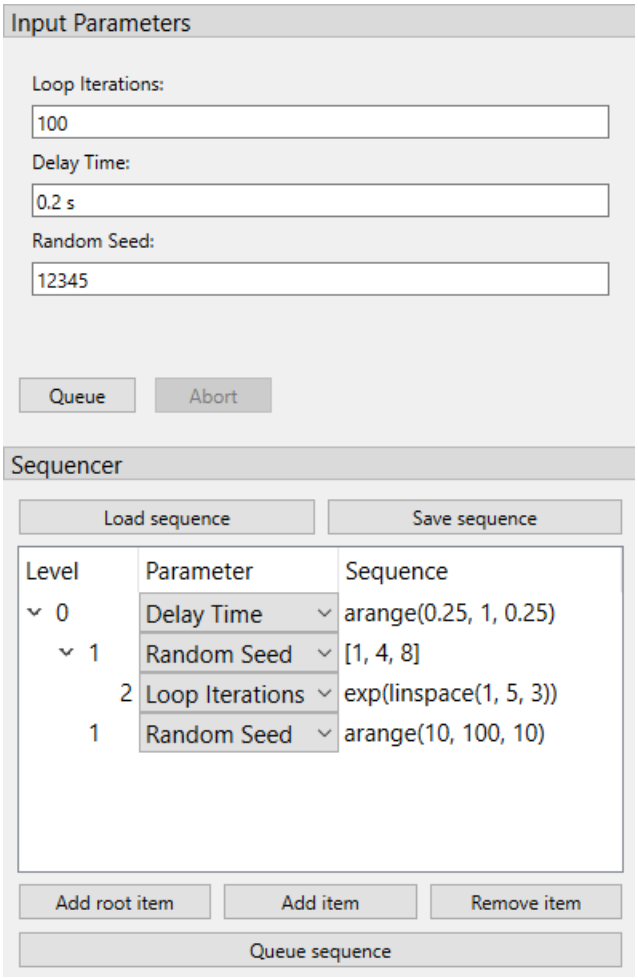
To activate the sequencer, two additional keyword arguments are added to *ManagedWindow*, namely `sequencer` and `sequencer_inputs`. `sequencer` accepts a boolean stating whether or not the sequencer has to be included into the window and `sequencer_inputs` accepts either `None` or a list of the parameter names are to be scanned over. If no list of parameters is given, the parameters displayed in the manager queue are used.

In order to be able to use the sequencer, the *ManagedWindow* class is required to have a `queue` method which takes a keyword (or better keyword-only for safety reasons) argument `procedure`, where a procedure instance can be passed. The sequencer will use this method to queue the parameter scan.

In order to implement the sequencer into the previous example, only the *ManagedWindow* has to be modified slightly (where modified lines are marked):

```
class MainWindow(ManagedWindow):

    def __init__(self):
        super().__init__(
            procedure_class=TestProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            x_axis='Iteration',
            y_axis='Random Number',
            sequencer=True,                                      # Added line
            sequencer_inputs=['iterations', 'delay', 'seed'],    # Added line
            sequence_file="gui_sequencer_example_sequence.txt",  # Added line, optional
        )
        self.setWindowTitle('GUI Example')

    def queue(self, procedure=None):                             # Modified line
        filename = tempfile.mktemp()

        if procedure is None:                                    # Added line
            procedure = self.make_procedure()                    # Indented

        results = Results(procedure, filename)
        experiment = self.new_experiment(results)

        self.manager.queue(experiment)
```

This adds the sequencer underneath the the input panel.



The widget contains a tree-view where you can build the sequence. It has three columns: `level` (indicated how deep an item is nested), `parameter` (a drop-down menu to select which parameter is being sequenced by that item), and `sequence` (the text-box where you can define the sequence). While the two former columns are rather straightforward, filling in the later requires some explanation.

In order to maintain flexibility, the sequence is defined in a text-box, allowing the user to enter any list-generating single-line piece of code. To assist in this, a number of functions is supported, either from the main python library (namely `range`, `sorted`, and `list`) or the numpy library. The supported numpy functions (prepending `numpy.` or any abbreviation is not required) are: `arange`, `linspace`, `arccos`, `arcsin`, `arctan`, `arctan2`, `ceil`, `cos`, `cosh`, `degrees`, `e`, `exp`, `fabs`, `floor`, `fmod`, `frexp`, `hypot`, `ldexp`, `log`, `log10`, `modf`, `pi`, `power`, `radians`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.

As an example, `arange(0, 10, 1)` generates a list increasing with steps of 1, while using `exp(arange(0, 10, 1))` generates an exponentially increasing list. This way complex sequences can be entered easily.

The sequences can be extended and shortened using the buttons `Add root item`, `Add item`, and `Remove item`. The later two either add a item as a child of the currently selected item or remove the selected item, respectively. To queue the entered sequence the button `Queue` sequence can be used. If an error occurs in evaluating the sequence text-boxes, this is mentioned in the logger, and nothing is queued.

Finally, it is possible to create a sequence file such that the user does not need to write the sequence again each time. The sequence file can be created by saving current sequence built within the GUI using the `Save sequence` button or directly writing a simple text file. Once created, the sequence can be loaded with the `Load sequence` button.

In the sequence file each line adds one item to the sequence tree, starting with a number of dashes (`-`) to indicate the

level of the item (starting with 1 dash for top level), followed by the name of the parameter and the sequence string, both as a python string between parentheses.

An example of such a sequence file is given below, resulting in the sequence shown in the figure above.

```
- "Delay Time", "arange(0.25, 1, 0.25)"
-- "Random Seed", "[1, 4, 8]"
--- "Loop Iterations", "exp(linspace(1, 5, 3))"
-- "Random Seed", "arange(10, 100, 10)"
```

This file can also be automatically loaded at the start of the program by adding the key-word argument `sequence_file="filename.txt"` to the `super().__init__` call, as was done in the example.

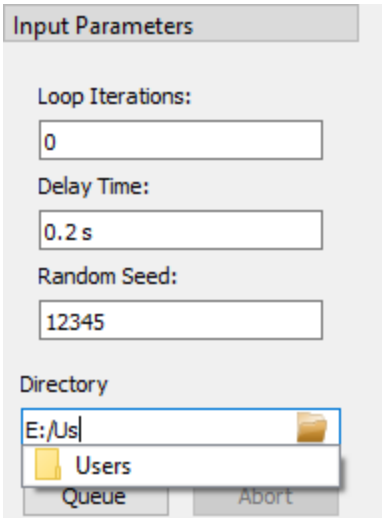
3.3.6 Using the directory input

It is possible to add a directory input in order to choose where the experiment's result will be saved. This option is activated by passing a boolean key-word argument `directory_input` during the *ManagedWindow* init. The value of the directory can be retrieved and set using the property `directory`. A default directory can be defined by setting the `directory` property in the MainWindow init.

Only the MainWindow needs to be modified in order to use this option (modified lines are marked).

```
class MainWindow(ManagedWindow):

    def __init__(self):
        super().__init__(
            procedure_class=TestProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            x_axis='Iteration',
            y_axis='Random Number',
            directory_input=True,                                   # Added line, enables␣
↪directory widget
        )
        self.setWindowTitle('GUI Example')
        self.directory = r'C:/Path/to/default/directory'            # Added line, sets␣
↪default directory for GUI load

    def queue(self):
        directory = self.directory                                  # Added line
        filename = unique_filename(directory)                       # Modified line

        results = Results(procedure, filename)
        experiment = self.new_experiment(results)

        self.manager.queue(experiment)
```

This adds the input line above the Queue and Abort buttons.



A completer is implemented allowing to quickly select an existing folder, and a button on the right side of the input widget opens a browse dialog.

3.3.7 Using the estimator widget

In order to provide estimates of the measurement procedure, an *EstimatorWidget* is provided that allows the user to define and calculate estimates. The widget is automatically activated when the `get_estimates` method is added in the `Procedure`.

The quickest and most simple implementation of the `get_estimates` function simply returns the estimated duration of the measurement in seconds (as an `int` or a `float`). As an example, in the example provided in the *Using the ManagedWindow* section, the `Procedure` is changed to:

```
class RandomProcedure(Procedure):

    # ...

    def get_estimates(self, sequence_length=None, sequence=None):

        return self.iterations * self.delay
```

This will add the estimator widget at the dock on the left. The duration and finishing-time of a single measurement is always displayed in this case. Depending on whether the SequencerWidget is also used, the length, duration and finishing-time of the full sequence is also shown.

For maximum flexibility (e.g. for showing multiple and other types of estimates, such as the duration, filesize, finishing-time, etc.) it is also possible that the `get_estimates` returns a list of tuples. Each of these tuple consists of two strings: the first is the name (label) of the estimate, the second is the estimate itself.

As an example, in the example provided in the *Using the ManagedWindow* section, the `Procedure` is changed to:

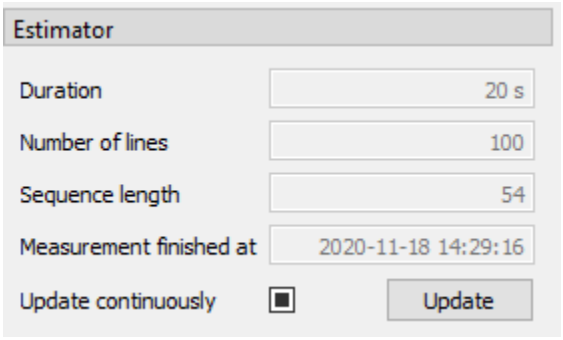
```
class RandomProcedure(Procedure):

    # ...

    def get_estimates(self, sequence_length=None, sequence=None):

```

(continues on next page)

(continued from previous page)

```
        duration = self.iterations * self.delay

        estimates = [
            ("Duration", "%d s" % int(duration)),
            ("Number of lines", "%d" % int(self.iterations)),
            ("Sequence length", str(sequence_length)),
            ('Measurement finished at', str(datetime.now() +␣
↪timedelta(seconds=duration))),
        ]

        return estimates
```

This will add the estimator widget at the dock on the left.



Note that after the initialisation of the widget both the label of the estimate as of course the estimate itself can be modified, but the amount of estimates is fixed.

The keyword arguments are not required in the implementation of the function, but are passed if asked for (i.e. `def get_estimates(self)` does also works). Keyword arguments that are accepted are `sequence`, which contains the full sequence of the sequencer (if present), and `sequence_length`, which gives the length of the sequence as integer (if present). If the sequencer is not present or the sequence cannot be parsed, both `sequence` and `sequence_length` will contain `None`.

The estimates are automatically updated every 2 seconds. Changing this update interval is possible using the "Update continuously"-checkbox, which can be toggled between three states: off (i.e. no updating), auto-update every two seconds (default) or auto-update every 100 milliseconds. Manually updating the estimates (useful whenever continuous updating is turned off) is also possible using the "update"-button.

3.3.8 Flexible hiding of inputs

There can be situations when it may be relevant to turn on or off a number of inputs (e.g. when a part of the measurement script is skipped upon turning of a single `BooleanParameter`). For these cases, it is possible to assign a `Parameter` to a controlling `Parameter`, which will hide or show the `Input` of the `Parameter` depending on the value of the `Parameter`. This is done with the `group_by` key-word argument.

```
toggle = BooleanParameter("toggle", default=True)
param = FloatParameter('some parameter', group_by='toggle')
```

When both the `toggle` and `param` are visible in the `InputsWidget` (via `inputs=['iterations', 'delay', 'seed']` as demonstrated above) one can control whether the input-field of `param` is visible by checking and unchecking the checkbox of `toggle`. By default, the group will be visible if the value of the `group_by` `Parameter` is `True`

(which is only relevant for a `BooleanParameter`), but it is possible to specify other value as conditions using the `group_condition` keyword argument.

```
iterations = IntegerParameter('Loop Iterations', default=100)
param = FloatParameter('some parameter', group_by='iterations', group_condition=99)
```

Here the input of `param` is only visible if `iterations` has a value of 99. This works with any type of `Parameter` as `group_by` parameter.

To allow for even more flexibility, it is also possible to pass a (lambda)function as a condition:

```
iterations = IntegerParameter('Loop Iterations', default=100)
param = FloatParameter('some parameter', group_by='iterations', group_condition=lambda␣
↪v: 50 < v < 100)
```

Now the input of `param` is only shown if the value of `iterations` is between 51 and 99.

Using the `hide_groups` keyword-argument of the `ManagedWindow` you can choose between hiding the groups (`hide_groups = True`) and disabling / graying-out the groups (`hide_groups = False`).

Finally, it is also possible to provide multiple parameters to the `group_by` argument, in which case the input will only be visible if all of the conditions are true. Multiple parameters for grouping can either be passed as a dict of string: condition pairs, or as a list of strings, in which case the `group_condition` can be either a single condition or a list of conditions:

```
iterations = IntegerParameter('Loop Iterations', default=100)
toggle = BooleanParameter('A checkbox')
param_A = FloatParameter('some parameter', group_by=['iterations', 'toggle'], group_
↪condition=[lambda v: 50 < v < 100, True])
param_B = FloatParameter('some parameter', group_by={'iterations': lambda v: 50 < v <␣
↪100, 'toggle': True})
```

Note that in this example, `param_A` and `param_B` are identically grouped: they're only visible if `iterations` is between 51 and 99 and if the *toggle* checkbox is checked (i.e. True).

3.3.9 Using the ManagedDockWindow

Building off the *Using the ManagedWindow* section where we used a `ManagedWindow`, we can also use *ManagedDockWindow* to build a graphical interface with multiple graphs that can be docked in the main GUI window or popped out into their own window.

To start with, let's make the following highlighted edits to the code example from *Using the ManagedWindow*:

1. On line 10 we now import *ManagedDockWindow*
2. On line 44 we make `MainWindow` a subclass of `ManagedDockWindow`
3. On line 51 we will pass in a list of strings from `DATA_COLUMNS` to the `x_axis` argument
4. On line 52 we will pass in a list of strings from `DATA_COLUMNS` to the `y_axis` argument

```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

import sys
import tempfile
```

(continues on next page)

(continued from previous page)

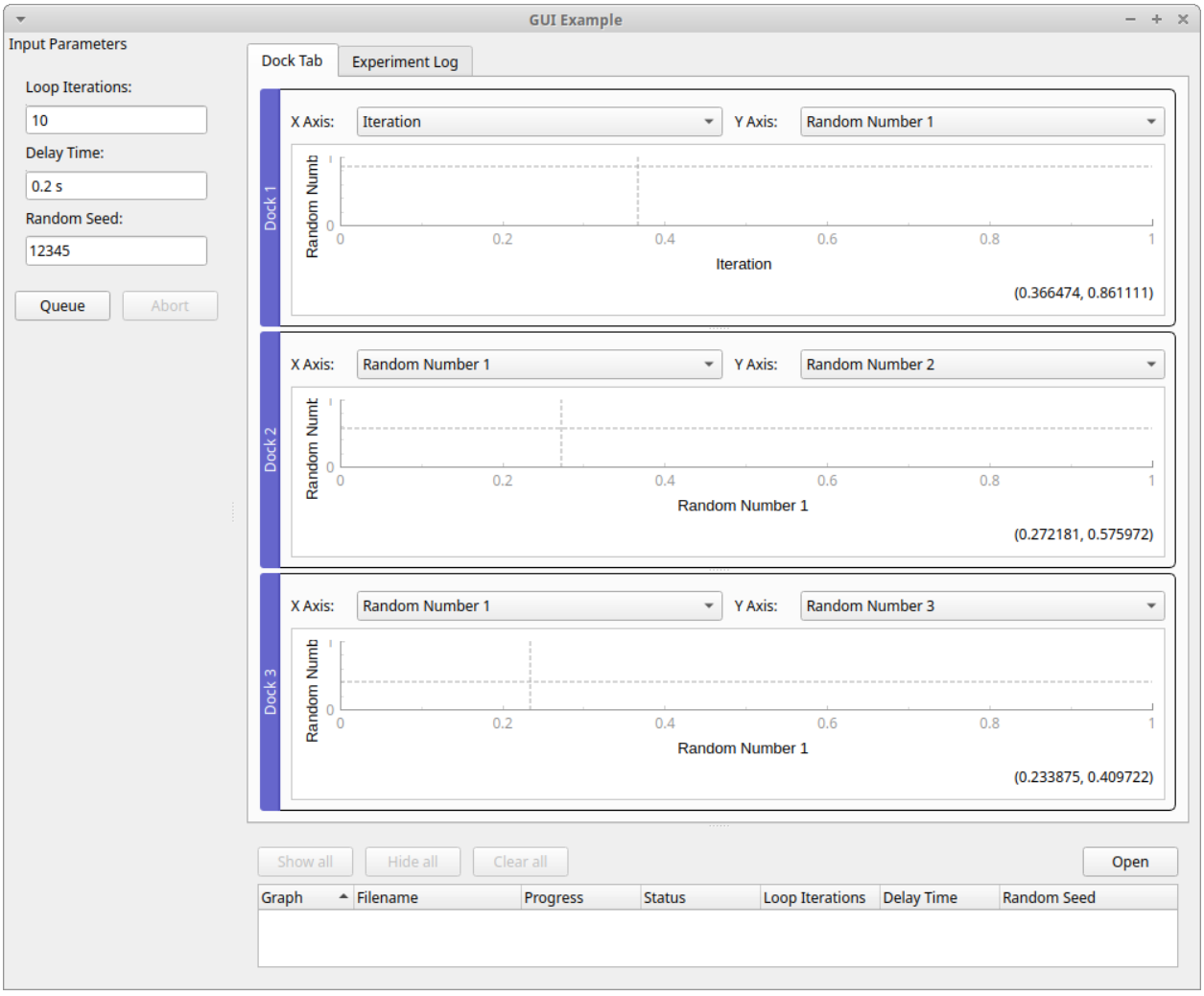
```
import random
from time import sleep
from pymeasure.display.Qt import QtWidgets
from pymeasure.display.windows.managed_dock_window import ManagedDockWindow
from pymeasure.experiment import Procedure, Results
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

class RandomProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations', default=10)
    delay = FloatParameter('Delay Time', units='s', default=0.2)
    seed = Parameter('Random Seed', default='12345')

    DATA_COLUMNS = ['Iteration', 'Random Number 1', 'Random Number 2', 'Random Number 3']

    def startup(self):
        log.info("Setting the seed of the random number generator")
        random.seed(self.seed)

    def execute(self):
        log.info("Starting the loop of %d iterations" % self.iterations)
        for i in range(self.iterations):
            data = {
                'Iteration': i,
                'Random Number 1': random.random(),
                'Random Number 2': random.random(),
                'Random Number 3': random.random()
            }
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            self.emit('progress', 100 * i / self.iterations)
            sleep(self.delay)
            if self.should_stop():
                log.warning("Caught the stop flag in the procedure")
                break


class MainWindow(ManagedDockWindow):

    def __init__(self):
        super().__init__(
            procedure_class=RandomProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            x_axis=['Iteration', 'Random Number 1'],
            y_axis=['Random Number 1','Random Number 2', 'Random Number 3']
        )
        self.setWindowTitle('GUI Example')

    def queue(self):
        filename = tempfile.mktemp()
```

(continues on next page)

(continued from previous page)

```
        procedure = self.make_procedure()
        results = Results(procedure, filename)
        experiment = self.new_experiment(results)

        self.manager.queue(experiment)


if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

Now we can see our `ManagedDockWindow`:



As you can see from the above screenshot, our example code created three docks with following "X Axis" and "Y Axis" labels:

1. **X Axis:** "Iteration" **Y Axis:** "Random Number 1"
2. **X Axis:** "Random Number 1" **Y Axis:** "Random Number 2"

3. **X Axis:** "Random Number 1" **Y Axis:** "Random Number 3"

The list of strings for `x_axis` and `y_axis` set the default labels for each dockable plot and the longest list determines how many dockable plots are created. To highlight this point, in our example we define `x_axis` and `y_axis` with the following lists:

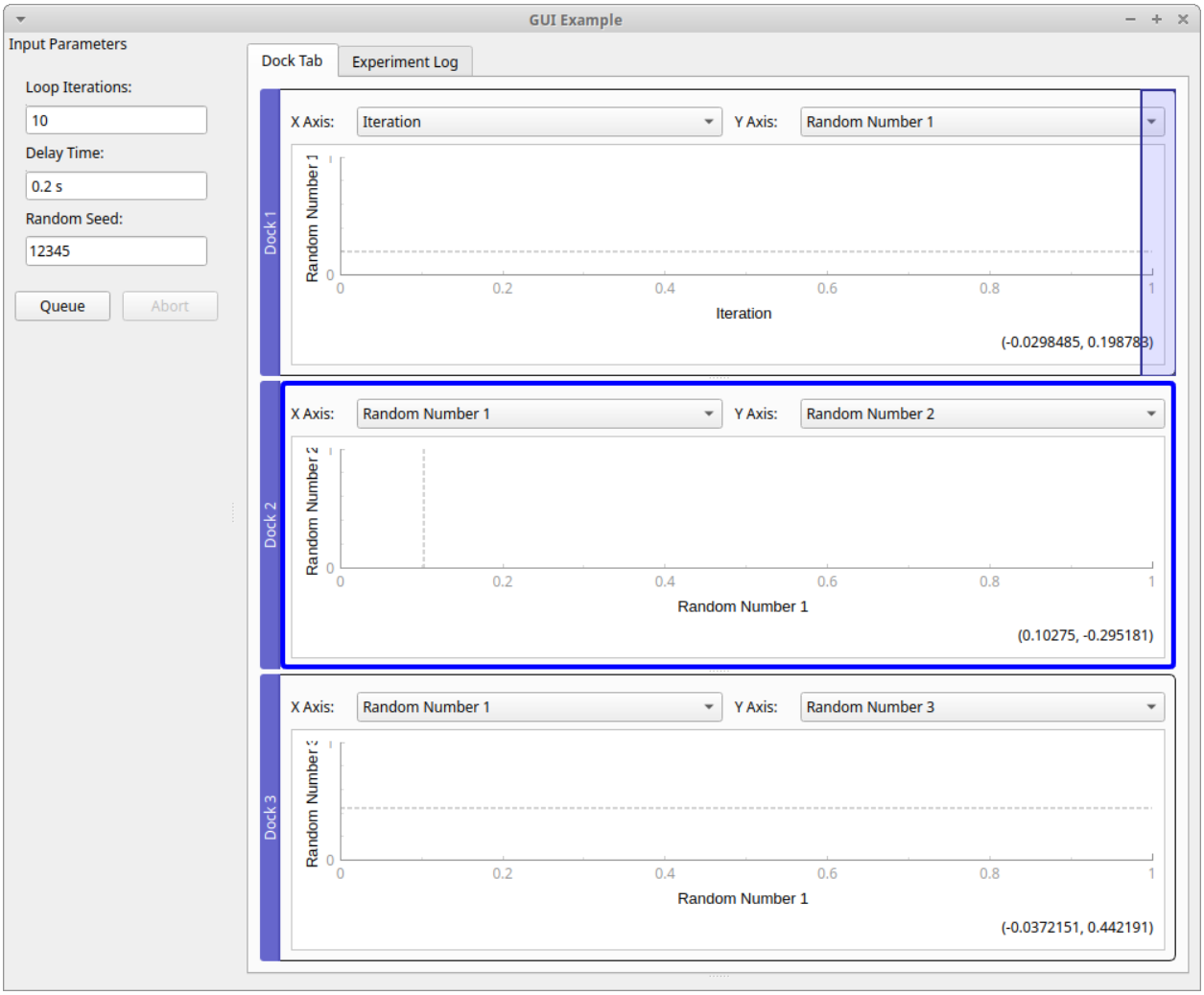
```
x_axis=['Iteration', 'Random Number 1'],
y_axis=['Random Number 1','Random Number 2', 'Random Number 3']
```

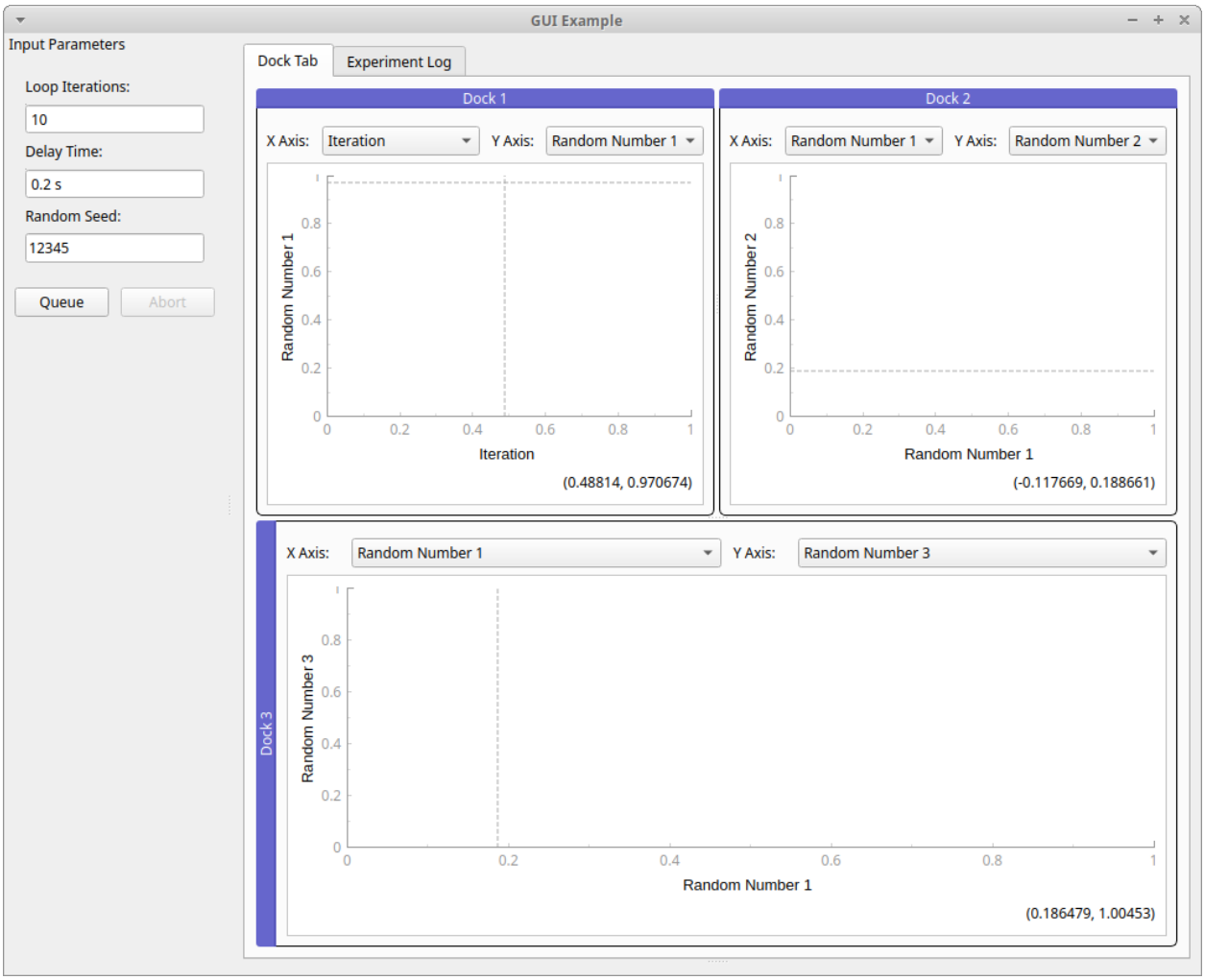
If one list is longer than the last element if the other list is used as the default label for the rest of the dockable plots. In our example that is why we have two **X Axis** labels with "Random Number 1". The longest list between `x_axis` and `y_axis` determines the number of plots. In our example `y_axis` has the longest list with a length of three so three plots are created.

You can pop out a dockable plot from the main dock window to its own window by double clicking the blue "Dock #" title bar, which is to the left of each plot by default:

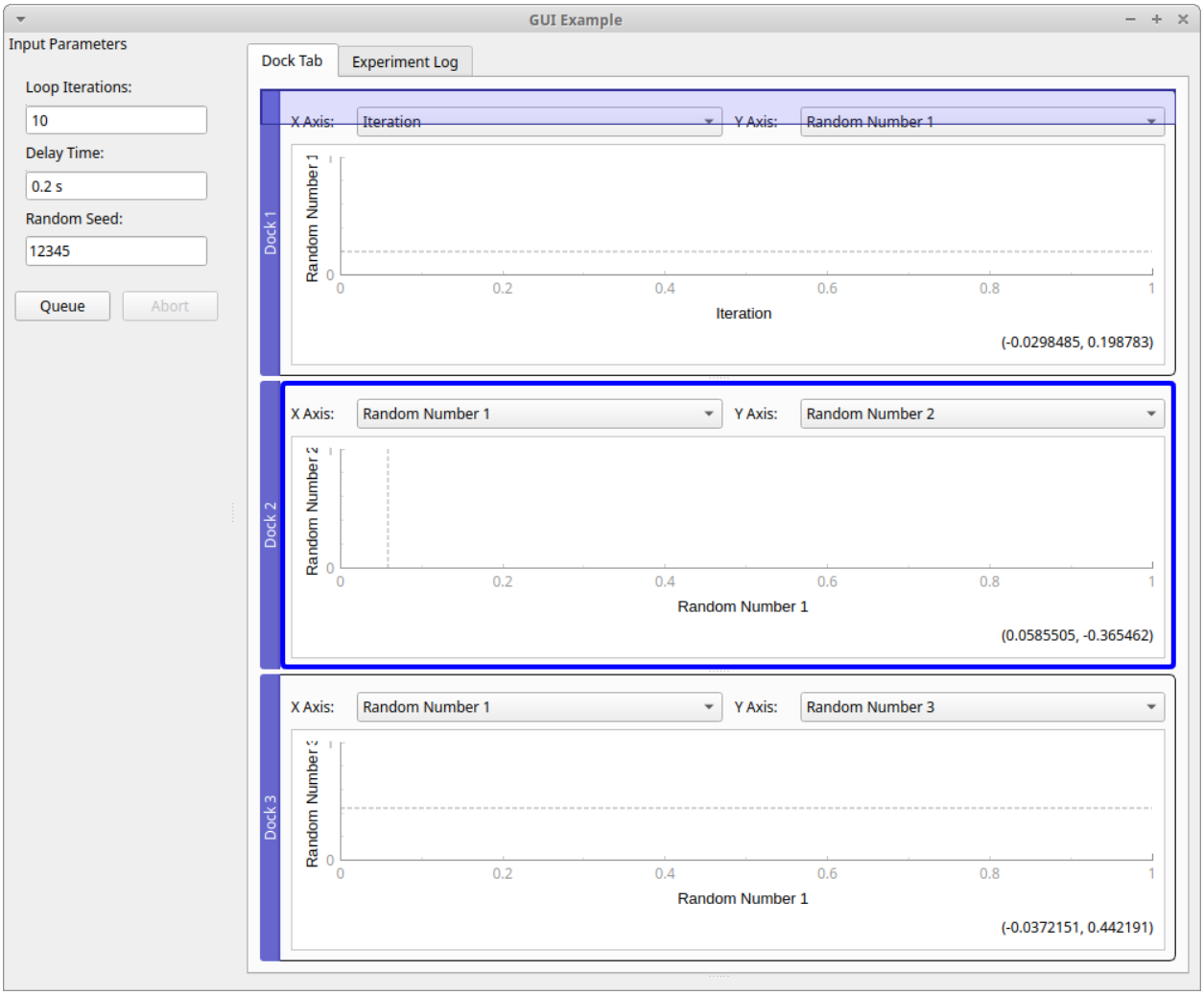
You can return the popped out window to the main window by clicking the close icon X in the top right.

You can drag a dockable plot to reposition it in reference to other plots in the main dock window in several ways. You can drag the blue "Dock #" title bar to the left or right side of another plot to reposition a plot to be side by side with another plot:

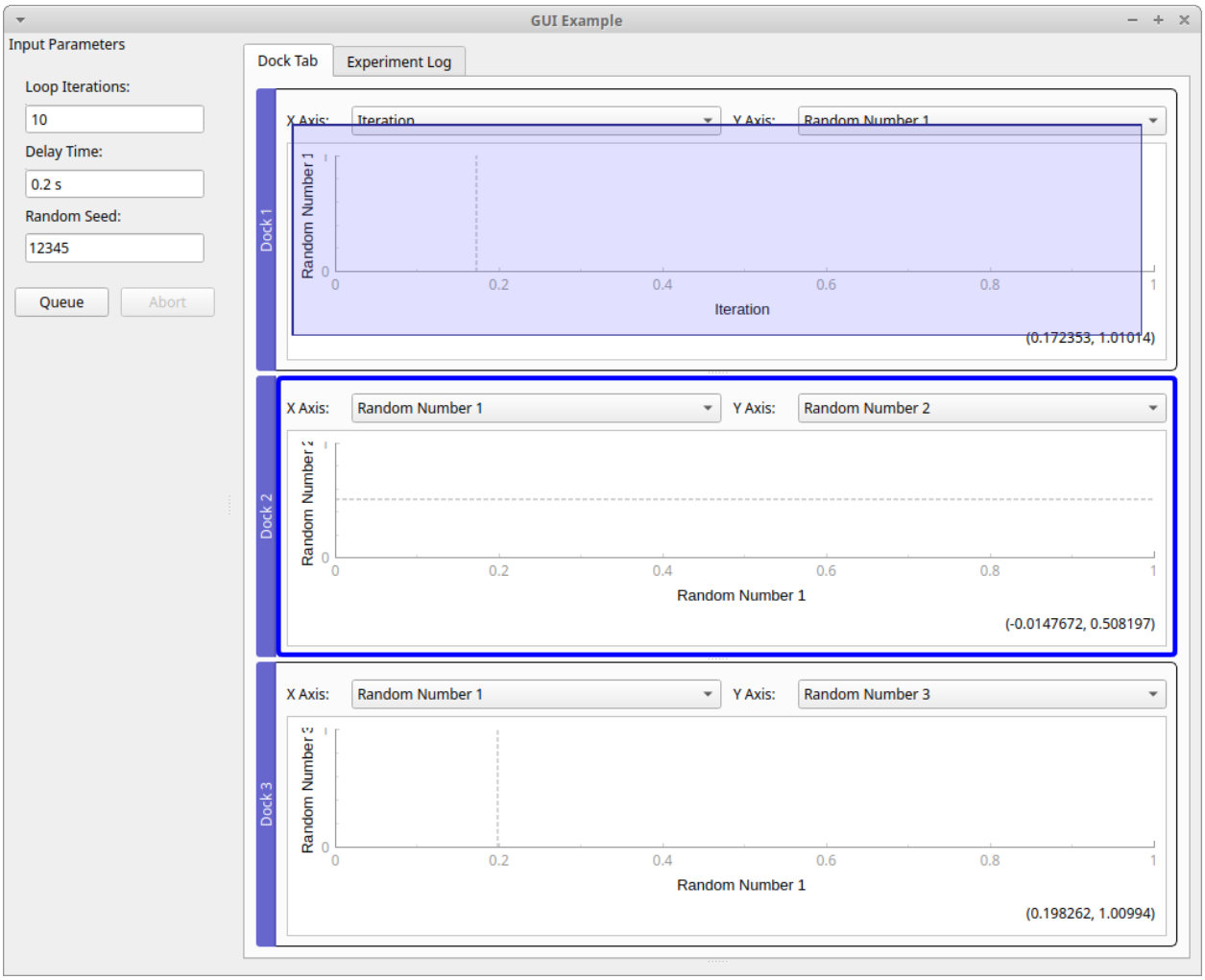


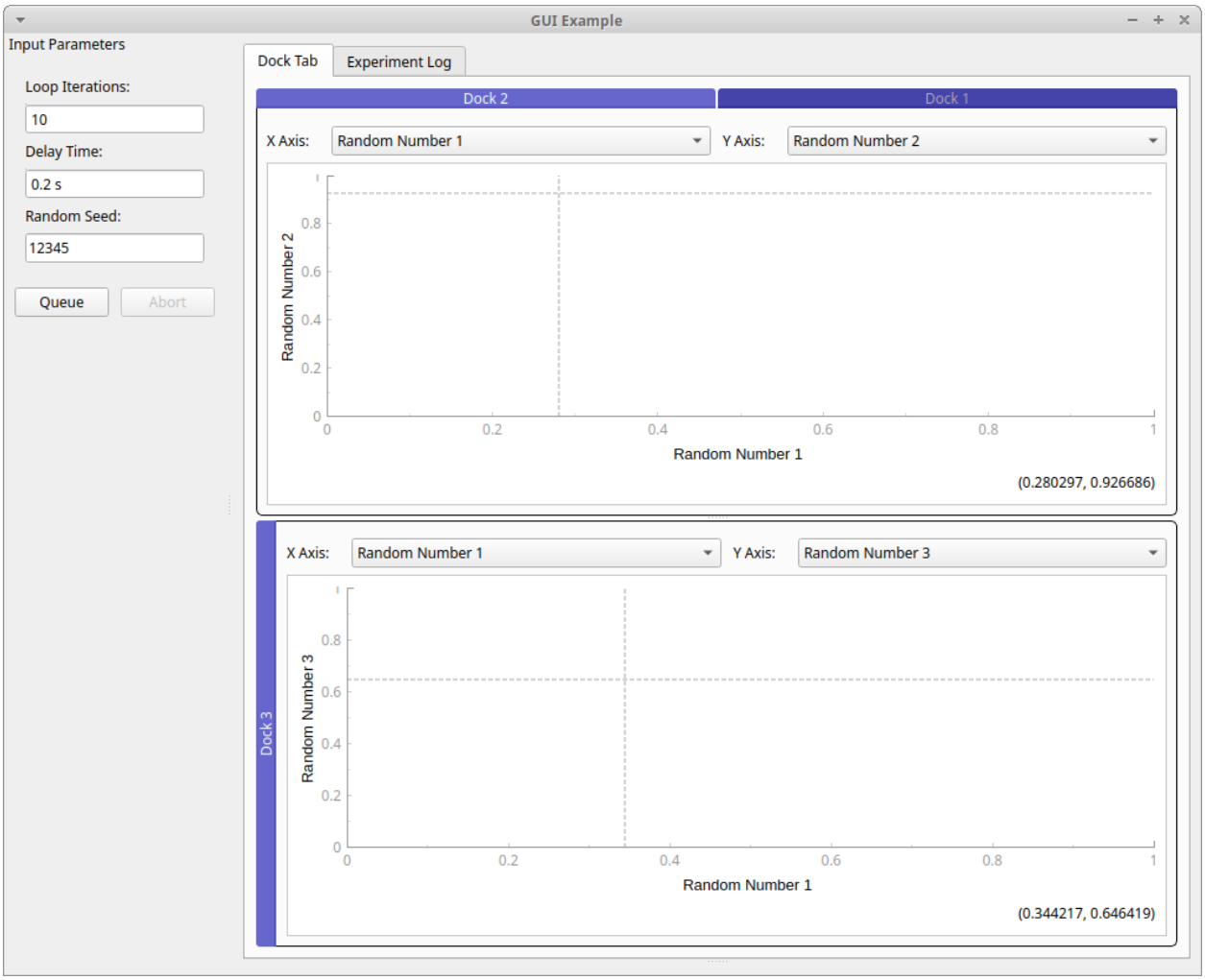


You can also drag the blue "Dock #" title bar to the top or bottom side of another plot to reposition a plot to rearrange the vertical order of the plots:



Finally, you can drag the blue "Dock #" title bar to the middle of another plot to reposition a plot to create a tabbed view of the two plots:


GUI Example
Input Parameters
Loop Iterations:
10
Delay Time:
0.2 s
Random Seed:
12345
Queue
Abort
Dock Tab
Experiment Log
Dock 1
X Axis:
Iteration
Y Axis:
Random Number 1
Random Number 1
0
0
0.2
0.4
0.6
0.8
1
Iteration
(0.172353, 1.01014)
Dock 2
X Axis:
Random Number 1
Y Axis:
Random Number 2
Random Number 2
0
0
0.2
0.4
0.6
0.8
1
Random Number 1
(-0.0147672, 0.508197)
Dock 3
X Axis:
Random Number 1
Y Axis:
Random Number 3
Random Number 3
0
0
0.2
0.4
0.6
0.8
1
Random Number 1
(0.198262, 1.00994)


GUI Example
Input Parameters
Loop Iterations:
10
Delay Time:
0.2 s
Random Seed:
12345
Queue
Abort
Dock Tab
Experiment Log
Dock 2
Dock 1
X Axis:
Random Number 1
Y Axis:
Random Number 2
Random Number 2
Random Number 1
(0.280297, 0.926686)
Dock 3
X Axis:
Random Number 1
Y Axis:
Random Number 3
Random Number 3
Random Number 1
(0.344217, 0.646419)

CHAPTER

FOUR

PYMEASURE.ADAPTERS

The adapter classes allow the instruments to be independent of the communication method used. The instrument implementation takes care of any potential quirks in its commmunication protocol (see *Advanced communication protocols*), and the adapter takes care of the details of the over-the-wire communication with the hardware device. In the vast majority of cases, it will be sufficient to pass a connection string or integer to the instrument (see *Connecting to an instrument*), which uses the `pymeasure.adapters.VISAAdapter` in the background.

4.1 Adapter base class

class pymeasure.adapters.**Adapter**(*preprocess_reply=None*, *log=None*, ***kwargs*)

Base class for Adapter child classes, which adapt between the Instrument object and the connection, to allow flexible use of different connection techniques.

This class should only be inherited from.

Parameters

- **preprocess_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.

  Deprecated since version 0.11: Implement it in the instrument's *read* method instead.
- **log** – Parent logger of the 'Adapter' logger.
- **kwargs** – Keyword arguments just to be cooperative.

ask(*command*)

Write the command to the instrument and returns the resulting ASCII response.

Deprecated since version 0.11: Call *Instrument.ask* instead.

Parameters **command** – SCPI command string to be sent to the instrument

Returns String ASCII response of the instrument

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary_values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

close()
Close the connection.

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

Do not override in a subclass!

Parameters **kwargs** – Keyword arguments for the connection itself.

Returns str ASCII response of the instrument (excluding read_termination).

read_binary_values(*header_bytes=0*, *termination_bytes=None*, *dtype=<class 'numpy.float32'>*, ***kwargs*)
Returns a numpy array from a query for binary data

Parameters

- **header_bytes** (*int*) – Number of bytes to ignore in header.
- **termination_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.
- **kwargs** – Further arguments for the NumPy fromstring method.

Returns NumPy array of values

read_bytes(*count=- 1*, ***kwargs*)
Read a certain number of bytes from the instrument.

Do not override in a subclass!

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the connection itself.

Returns bytes Bytes response of the instrument (including termination).

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*)
Write a command to the instrument and returns a list of formatted values from the result.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string. If not specified, the Adapter default is used if available, otherwise no preprocessing is done.

Returns A list of the desired type, or strings where the casting fails

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Do not override in a subclass!

Parameters

- **command** (*str*) – Command string to be sent to the instrument (without termination).
- **kwargs** – Keyword arguments for the connection itself.

write_binary_values(*command*, *values*, *termination=''*, ***kwargs*)
Write binary data to the instrument, e.g. waveform for signal generators

Parameters

- **command** – command string to be sent to the instrument
- **values** – iterable representing the binary values
- **termination** – String added afterwards to terminate the message.
- **kwargs** – Key-word arguments to pass onto `Adapter._format_binary_values()`

Returns number of bytes written

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

Do not override in a subclass!

Parameters

- **content** (*bytes*) – The bytes to write to the instrument.
- **kwargs** – Keyword arguments for the connection itself.

4.2 VISA adapter

class pymeasure.adapters.**VISAAdapter**(*resource_name*, *visa_library=''*, *preprocess_reply=None*, *query_delay=0*, *log=None*, ***kwargs*)
Bases: *pymeasure.adapters.adapter.Adapter*

Adapter class for the VISA library, using PyVISA to communicate with instruments.

The workhorse of our library, used by most instruments.

Parameters

- **resource_name** – A VISA resource string or GPIB address integer that identifies the target of the connection
- **visa_library** – PyVISA VisaLibrary Instance, path of the VISA library or VisaLibrary spec string (`@py` or `@ivi`). If not given, the default for the platform will be used.
- **preprocess_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.

  Deprecated since version 0.11: Implement it in the instrument's *read* method instead.
- **query_delay** (*float*) – Time in s to wait after writing and before reading.

  Deprecated since version 0.11: Implement it in the instrument's *wait_until_read* method instead.
- **log** – Parent logger of the 'Adapter' logger.
- ****kwargs** – Keyword arguments for configuring the PyVISA connection.

Kwargs Keyword arguments are used to configure the connection created by PyVISA. This is complicated by the fact that *which* arguments are valid depends on the interface (e.g. serial, GPIB, TCPI/IP, USB) determined by the current `resource_name`.

A flexible process is used to easily define reasonable *default values* for different instrument interfaces, but also enable the instrument user to *override any setting* if their situation demands it.

A kwarg that names a pyVISA interface type (most commonly `asrl`, `gpib`, `tcpip`, or `usb`) is a dictionary with keyword arguments defining defaults specific to that interface. Example: `asrl={'baud_rate': 4200}`.

All other kwargs are either generally valid (e.g. `timeout=500`) or override any default settings from the interface-specific entries above. For example, passing `baud_rate=115200` when connecting via a resource name `ASRL1` would override a default of 4200 defined as above.

See *Modifying connection settings* for how to tweak settings when *connecting* to an instrument. See *Defining default connection settings* for how to best define default settings when *implementing an instrument*.

ask(*command*)

Writes the command to the instrument and returns the resulting ASCII response

Deprecated since version 0.11: Call *Instrument.ask* instead.

Parameters **command** – SCPI command string to be sent to the instrument

Returns String ASCII response of the instrument

ask_values(*command*, ***kwargs*)

Writes a command to the instrument and returns a list of formatted values from the result. This leverages the *query_ascii_values* method in PyVISA.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **kwargs** – Key-word arguments to pass onto *query_ascii_values*

Returns Formatted response of the instrument.

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary_values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

close()

Close the connection.

flush_read_buffer()

Flush and discard the input buffer

As detailed by pyvisa, discard the read buffer contents and if data was present in the read buffer and no END-indicator was present, read from the device until encountering an END indicator (which causes loss of data).

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

Do not override in a subclass!

Parameters **kwargs** – Keyword arguments for the connection itself.

Returns str ASCII response of the instrument (excluding read_termination).

read_binary_values(*header_bytes=0*, *termination_bytes=None*, *dtype=<class 'numpy.float32'>*, ***kwargs*)
Returns a numpy array from a query for binary data

Parameters

- **header_bytes** (*int*) – Number of bytes to ignore in header.
- **termination_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.
- **kwargs** – Further arguments for the NumPy fromstring method.

Returns NumPy array of values

read_bytes(*count=- 1*, ***kwargs*)
Read a certain number of bytes from the instrument.

Do not override in a subclass!

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the connection itself.

Returns bytes Bytes response of the instrument (including termination).

values(*command*, *separator=', '*, *cast=<class 'float'>*, *preprocess_reply=None*)
Write a command to the instrument and returns a list of formatted values from the result.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string. If not specified, the Adapter default is used if available, otherwise no preprocessing is done.

Returns A list of the desired type, or strings where the casting fails

wait_for_srq(*timeout=25*, *delay=0.1*)
Block until a SRQ, and leave the bit high

Parameters

- **timeout** – Timeout duration in seconds

- **delay** – Time delay between checking SRQ in seconds

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Do not override in a subclass!

Parameters

- **command** (*str*) – Command string to be sent to the instrument (without termination).
- **kwargs** – Keyword arguments for the connection itself.

write_binary_values(*command*, *values*, *termination=''*, ***kwargs*)
Write binary data to the instrument, e.g. waveform for signal generators

Parameters

- **command** – command string to be sent to the instrument
- **values** – iterable representing the binary values
- **termination** – String added afterwards to terminate the message.
- **kwargs** – Key-word arguments to pass onto `Adapter._format_binary_values()`

Returns number of bytes written

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

Do not override in a subclass!

Parameters

- **content** (*bytes*) – The bytes to write to the instrument.
- **kwargs** – Keyword arguments for the connection itself.

4.3 Serial adapter

class pymeasure.adapters.**SerialAdapter**(*port*, *preprocess_reply=None*, *write_termination=''*, *read_termination=''*, ***kwargs*)
Bases: *pymeasure.adapters.adapter.Adapter*

Adapter class for using the Python Serial package to allow serial communication to instrument

Parameters

- **port** – Serial port
- **preprocess_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.

  Deprecated since version 0.11: Implement it in the instrument's *read* method instead.
- **write_termination** – String appended to messages before writing them.
- **read_termination** – String expected at end of read message and removed.
- **kwargs** – Any valid key-word argument for serial.Serial

_format_binary_values(*values*, *datatype='f'*, *is_big_endian=False*, *header_fmt='ieee'*)
Format values in binary format, used internally in *Adapter.write_binary_values()*.

Parameters

- **values** – data to be written to the device.
- **datatype** – the format string for a single element. See struct module.
- **is_big_endian** – boolean indicating endianess.
- **header_fmt** – Format of the header prefixing the data ("ieee", "hp", "empty").

Returns binary string.

Return type bytes

ask(*command*)

Write the command to the instrument and returns the resulting ASCII response.

Deprecated since version 0.11: Call *Instrument.ask* instead.

Parameters **command** – SCPI command string to be sent to the instrument

Returns String ASCII response of the instrument

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary_values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

close()

Close the connection.

read(***kwargs*)

Read up to (excluding) *read_termination* or the whole read buffer.

Do not override in a subclass!

Parameters **kwargs** – Keyword arguments for the connection itself.

Returns str ASCII response of the instrument (excluding read_termination).

read_binary_values(*header_bytes=0*, *termination_bytes=None*, *dtype=<class 'numpy.float32'>*, ***kwargs*)

Returns a numpy array from a query for binary data

Parameters

- **header_bytes** (*int*) – Number of bytes to ignore in header.
- **termination_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.
- **kwargs** – Further arguments for the NumPy fromstring method.

Returns NumPy array of values

read_bytes(*count=- 1*, ***kwargs*)

Read a certain number of bytes from the instrument.

Do not override in a subclass!

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the connection itself.

Returns bytes Bytes response of the instrument (including termination).

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*)
Write a command to the instrument and returns a list of formatted values from the result.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string. If not specified, the Adapter default is used if available, otherwise no preprocessing is done.

Returns A list of the desired type, or strings where the casting fails

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Do not override in a subclass!

Parameters

- **command** (*str*) – Command string to be sent to the instrument (without termination).
- **kwargs** – Keyword arguments for the connection itself.

write_binary_values(*command*, *values*, *termination=''*, ***kwargs*)
Write binary data to the instrument, e.g. waveform for signal generators

Parameters

- **command** – command string to be sent to the instrument
- **values** – iterable representing the binary values
- **termination** – String added afterwards to terminate the message.
- **kwargs** – Key-word arguments to pass onto `Adapter._format_binary_values()`

Returns number of bytes written

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

Do not override in a subclass!

Parameters

- **content** (*bytes*) – The bytes to write to the instrument.
- **kwargs** – Keyword arguments for the connection itself.

4.4 Prologix adapter

class pymeasure.adapters.**PrologixAdapter**(*resource_name*, *address=None*, *rw_delay=0*, *serial_timeout=None*, *preprocess_reply=None*, ***kwargs*)

Bases: *pymeasure.adapters.visa.VISAAdapter*

Encapsulates the additional commands necessary to communicate over a Prologix GPIB-USB Adapter, using the *VISAAdapter*.

Each PrologixAdapter is constructed based on a connection to the Prologix device itself and the GPIB address of the instrument to be communicated to. Connection sharing is achieved by using the *gpib()* method to spawn new PrologixAdapters for different GPIB addresses.

Parameters

- **resource_name** – A VISA resource string that identifies the connection to the Prologix device itself, for example "ASRL5" for the 5th COM port.
- **address** – Integer GPIB address of the desired instrument.
- **rw_delay** – An optional delay to set between a write and read call for slow to respond instruments.

  Deprecated since version 0.11: Implement it in the instrument's *wait_until_read* method instead.
- **preprocess_reply** – optional callable used to preprocess strings received from the instrument. The callable returns the processed string.

  Deprecated since version 0.11: Implement it in the instrument's *read* method instead.
- **kwargs** – Key-word arguments if constructing a new serial object

Variables **address** – Integer GPIB address of the desired instrument.

Usage example:

```
adapter = PrologixAdapter("ASRL5::INSTR", 7)
sourcemeter = Keithley2400(adapter)  # at GPIB address 7
# generate another instance with a different GPIB address:
adapter2 = adapter.gpib(9)
multimeter = Keithley2000(adapter2)  # at GPIB address 9
```

To allow user access to the Prologix adapter in Linux, create the file: `/etc/udev/rules.d/51-prologix.rules`, with contents:

```
SUBSYSTEMS=="usb",ATTRS{idVendor}=="0403",ATTRS{idProduct}=="6001",MODE="0666"
```

Then reload the udev rules with:

```
sudo udevadm control --reload-rules
sudo udevadm trigger
```

_format_binary_values(*values*, *datatype='f'*, *is_big_endian=False*, *header_fmt='ieee'*)

Format values in binary format, used internally in *write_binary_values()*.

Parameters

- **values** – data to be writen to the device.
- **datatype** – the format string for a single element. See struct module.

- **is_big_endian** – boolean indicating endianess.
- **header_fmt** – Format of the header prefixing the data ("ieee", "hp", "empty").

Returns binary string.

Return type bytes

ask(*command*)
Ask the Prologix controller.

Deprecated since version 0.11: Call *Instrument.ask* instead.

Parameters **command** – SCPI command string to be sent to instrument

ask_values(*command*, ***kwargs*)
Writes a command to the instrument and returns a list of formatted values from the result. This leverages the *query_ascii_values* method in PyVISA.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **kwargs** – Key-word arguments to pass onto *query_ascii_values*

Returns Formatted response of the instrument.

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)
Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary_values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

close()
Close the connection.

flush_read_buffer()
Flush and discard the input buffer

As detailed by pyvisa, discard the read buffer contents and if data was present in the read buffer and no END-indicator was present, read from the device until encountering an END indicator (which causes loss of data).

gpib(*address*, ***kwargs*)
Return a PrologixAdapter object that references the GPIB address specified, while sharing the Serial connection with other calls of this function

Parameters

- **address** – Integer GPIB address of the desired instrument
- **kwargs** – Arguments for the initialization

Returns PrologixAdapter for specific GPIB address

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

Do not override in a subclass!

Parameters **kwargs** – Keyword arguments for the connection itself.

Returns str ASCII response of the instrument (excluding read_termination).

read_binary_values(*header_bytes=0*, *termination_bytes=None*, *dtype=<class 'numpy.float32'>*, ***kwargs*)
Returns a numpy array from a query for binary data

Parameters

- **header_bytes** (*int*) – Number of bytes to ignore in header.
- **termination_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.
- **kwargs** – Further arguments for the NumPy fromstring method.

Returns NumPy array of values

read_bytes(*count=- 1*, ***kwargs*)
Read a certain number of bytes from the instrument.

Do not override in a subclass!

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the connection itself.

Returns bytes Bytes response of the instrument (including termination).

set_defaults()
Set up the default behavior of the Prologix-GPIB adapter

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*)
Write a command to the instrument and returns a list of formatted values from the result.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string. If not specified, the Adapter default is used if available, otherwise no preprocessing is done.

Returns A list of the desired type, or strings where the casting fails

wait_for_srq(*timeout=25*, *delay=0.1*)
Blocks until a SRQ, and leaves the bit high

Parameters

- **timeout** – Timeout duration in seconds.

- **delay** – Time delay between checking SRQ in seconds.

Raises **TimeoutError** – "Waiting for SRQ timed out."

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

If the GPIB address in *address* is defined, it is sent first.

Parameters

- **command** (*str*) – Command string to be sent to the instrument (without termination).
- **kwargs** – Keyword arguments for the connection itself.

write_binary_values(*command*, *values*, ***kwargs*)
Write binary data to the instrument, e.g. waveform for signal generators.

values are encoded in a binary format according to IEEE 488.2 Definite Length Arbitrary Block Response Data block.

Parameters

- **command** – SCPI command to be sent to the instrument
- **values** – iterable representing the binary values
- **kwargs** – Key-word arguments to pass onto *_format_binary_values()*

Returns number of bytes written

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

Do not override in a subclass!

Parameters

- **content** (*bytes*) – The bytes to write to the instrument.
- **kwargs** – Keyword arguments for the connection itself.

4.5 VXI-11 adapter

class pymeasure.adapters.**VXI11Adapter**(*host*, *preprocess_reply=None*, ***kwargs*)
Bases: *pymeasure.adapters.adapter.Adapter*

VXI11 Adapter class. Provides a adapter object that wraps around the read, write and ask functionality of the vxi11 library.

Deprecated since version 0.11: Use VISAAdapter instead.

Parameters

- **host** – string containing the visa connection information.
- **preprocess_reply** – (deprecated) optional callable used to preprocess strings received from the instrument. The callable returns the processed string.

ask(*command*)
Wrapper function for the ask command using the vx11 interface.

Deprecated since version 0.11: Call *Instrument.ask* instead.

Parameters **command** – string with the command that will be transmitted to the instrument.

:returns string containing a response from the device.

ask_raw(*command*)
Wrapper function for the ask_raw command using the vx11 interface.

Deprecated since version 0.11: Use *Instrument.write_bytes* and *Instrument.read_bytes* instead.

Parameters **command** – binary string with the command that will be transmitted to the instrument

:returns binary string containing the response from the device.

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)
Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary_values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

close()
Close the connection.

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

Do not override in a subclass!

Parameters **kwargs** – Keyword arguments for the connection itself.

Returns str ASCII response of the instrument (excluding read_termination).

read_binary_values(*header_bytes=0*, *termination_bytes=None*, *dtype=<class 'numpy.float32'>*, ***kwargs*)
Returns a numpy array from a query for binary data

Parameters

- **header_bytes** (*int*) – Number of bytes to ignore in header.
- **termination_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.
- **kwargs** – Further arguments for the NumPy fromstring method.

Returns NumPy array of values

read_bytes(*count=- 1*, ***kwargs*)
Read a certain number of bytes from the instrument.

Do not override in a subclass!

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the connection itself.

Returns bytes Bytes response of the instrument (including termination).

read_raw()
Read bytes from the device.

Deprecated since version 0.11: Use *read_bytes* instead.

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*)
Write a command to the instrument and returns a list of formatted values from the result.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string. If not specified, the Adapter default is used if available, otherwise no preprocessing is done.

Returns A list of the desired type, or strings where the casting fails

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Do not override in a subclass!

Parameters

- **command** (*str*) – Command string to be sent to the instrument (without termination).
- **kwargs** – Keyword arguments for the connection itself.

write_binary_values(*command*, *values*, *termination=''*, ***kwargs*)
Write binary data to the instrument, e.g. waveform for signal generators

Parameters

- **command** – command string to be sent to the instrument
- **values** – iterable representing the binary values
- **termination** – String added afterwards to terminate the message.
- **kwargs** – Key-word arguments to pass onto `Adapter._format_binary_values()`

Returns number of bytes written

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

Do not override in a subclass!

Parameters

- **content** (*bytes*) – The bytes to write to the instrument.
- **kwargs** – Keyword arguments for the connection itself.

write_raw(*command*)
Write bytes to the device.

Deprecated since version 0.11: Use *write_bytes* instead.

4.6 Telnet adapter

class pymeasure.adapters.**TelnetAdapter**(*host*, *port=0*, *query_delay=0*, *preprocess_reply=None*, ***kwargs*)

Bases: *pymeasure.adapters.adapter.Adapter*

Adapter class for using the Python telnetlib package to allow communication to instruments

Parameters

- **host** – host address of the instrument
- **port** – TCPIP port
- **query_delay** – delay in seconds between write and read in the ask method
- **preprocess_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.

  Deprecated since version 0.11: Implement it in the instrument's *read* method instead.
- **kwargs** – Valid keyword arguments for telnetlib.Telnet, currently this is only 'timeout'

ask(*command*)

Writes a command to the instrument and returns the resulting ASCII response

Deprecated since version 0.11: Call *Instrument.ask* instead.

Parameters **command** – command string to be sent to the instrument

Returns String ASCII response of the instrument

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary_values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

close()

Close the connection.

read(***kwargs*)

Read up to (excluding) *read_termination* or the whole read buffer.

Do not override in a subclass!

Parameters **kwargs** – Keyword arguments for the connection itself.

Returns str ASCII response of the instrument (excluding read_termination).

read_binary_values(*header_bytes=0*, *termination_bytes=None*, *dtype=<class 'numpy.float32'>*, ***kwargs*)

Returns a numpy array from a query for binary data

Parameters

- **header_bytes** (*int*) – Number of bytes to ignore in header.

- **termination_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.
- **kwargs** – Further arguments for the NumPy fromstring method.

Returns NumPy array of values

read_bytes(*count=- 1*, ***kwargs*)
Read a certain number of bytes from the instrument.

Do not override in a subclass!

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the connection itself.

Returns bytes Bytes response of the instrument (including termination).

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*)
Write a command to the instrument and returns a list of formatted values from the result.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string. If not specified, the Adapter default is used if available, otherwise no preprocessing is done.

Returns A list of the desired type, or strings where the casting fails

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Do not override in a subclass!

Parameters

- **command** (*str*) – Command string to be sent to the instrument (without termination).
- **kwargs** – Keyword arguments for the connection itself.

write_binary_values(*command*, *values*, *termination=''*, ***kwargs*)
Write binary data to the instrument, e.g. waveform for signal generators

Parameters

- **command** – command string to be sent to the instrument
- **values** – iterable representing the binary values
- **termination** – String added afterwards to terminate the message.
- **kwargs** – Key-word arguments to pass onto `Adapter._format_binary_values()`

Returns number of bytes written

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

Do not override in a subclass!

Parameters

- **content** (*bytes*) – The bytes to write to the instrument.
- **kwargs** – Keyword arguments for the connection itself.

4.7 Test adapters

These pieces are useful when writing tests.

pymeasure.test.**expected_protocol**(*instrument_cls*, *comm_pairs*, *connection_attributes={}*, *connection_methods={}*, ***kwargs*)
Context manager that checks sent/received instrument commands without a device connected.

Given an instrument class and a list of command-response pairs, this context manager confirms that the code in the context manager block produces the expected messages.

Terminators are excluded from the protocol definition, as those are typically a detail of the communication method (i.e. Adapter), and not the protocol itself.

Parameters

- **instrument_cls** (*pymeasure.Instrument*) – *Instrument* subclass to instantiate.
- **comm_pairs** (*list[2-tuples[str]]*) – List of command-response pairs, i.e. 2-tuples like *('VOLT?', '3.14')*. 'None' indicates that a pair member (command or response) does not exist, e.g. *(None, 'RESP1')*. Commands and responses are without termination characters.
- **connection_attributes** – Dictionary of connection attributes and their values.
- **connection_methods** – Dictionary of method names of the connection and their return values.
- ****kwargs** – Keyword arguments for the instantiation of the instrument.

class pymeasure.adapters.**ProtocolAdapter**(*comm_pairs=[]*, *preprocess_reply=None*, *connection_attributes={}*, *connection_methods={}*, ***kwargs*)
Bases: *pymeasure.adapters.adapter.Adapter*

Adapter class for testing the command exchange protocol without instrument hardware.

This adapter is primarily meant for use within *pymeasure.test.expected_protocol()*.

The `connection` attribute is a `unittest.mock.MagicMock` such that every call returns. If you want to set a return value, you can use `adapter.connection.some_method.return_value = 7`, such that a call to `adapter.connection.some_method()` will return *7*. Similarly, you can verify that this call to the connection method happened with `assert adapter.connection.some_method.called is True`. You can specify dictionaries with return values of attributes and methods.

Parameters

- **comm_pairs** (*list*) – List of "reference" message pair tuples. The first element is what is sent to the instrument, the second one is the returned message. 'None' indicates that a pair member (write or read) does not exist. The messages do **not** include the termination characters.
- **connection_attributes** – Dictionary of connection attributes and their values.

- **connection_methods** – Dictionary of method names of the connection and their return values.

class pymeasure.adapters.**FakeAdapter**(*preprocess_reply=None*, *log=None*, ***kwargs*)

Bases: *pymeasure.adapters.adapter.Adapter*

Provides a fake adapter for debugging purposes, which bounces back the command so that arbitrary values testing is possible.

```
a = FakeAdapter()
assert a.read() == ""
a.write("5")
assert a.read() == "5"
assert a.read() == ""
assert a.ask("10") == "10"
assert a.values("10") == [10]
```

ask(*command*)

Write the command to the instrument and returns the resulting ASCII response.

Deprecated since version 0.11: Call *Instrument.ask* instead.

Parameters **command** – SCPI command string to be sent to the instrument

Returns String ASCII response of the instrument

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary_values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

close()

Close the connection.

read(***kwargs*)

Read up to (excluding) *read_termination* or the whole read buffer.

Do not override in a subclass!

Parameters **kwargs** – Keyword arguments for the connection itself.

Returns str ASCII response of the instrument (excluding read_termination).

read_binary_values(*header_bytes=0*, *termination_bytes=None*, *dtype=<class 'numpy.float32'>*, ***kwargs*)

Returns a numpy array from a query for binary data

Parameters

- **header_bytes** (*int*) – Number of bytes to ignore in header.
- **termination_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.
- **kwargs** – Further arguments for the NumPy fromstring method.

Returns NumPy array of values

read_bytes(*count=- 1*, ***kwargs*)
Read a certain number of bytes from the instrument.

Do not override in a subclass!

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the connection itself.

Returns bytes Bytes response of the instrument (including termination).

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*)
Write a command to the instrument and returns a list of formatted values from the result.

Deprecated since version 0.11: Call *Instrument.values* instead.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string. If not specified, the Adapter default is used if available, otherwise no preprocessing is done.

Returns A list of the desired type, or strings where the casting fails

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Do not override in a subclass!

Parameters

- **command** (*str*) – Command string to be sent to the instrument (without termination).
- **kwargs** – Keyword arguments for the connection itself.

write_binary_values(*command*, *values*, *termination=''*, ***kwargs*)
Write binary data to the instrument, e.g. waveform for signal generators

Parameters

- **command** – command string to be sent to the instrument
- **values** – iterable representing the binary values
- **termination** – String added afterwards to terminate the message.
- **kwargs** – Key-word arguments to pass onto `Adapter._format_binary_values()`

Returns number of bytes written

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

Do not override in a subclass!

Parameters

- **content** (*bytes*) – The bytes to write to the instrument.

- **kwargs** – Keyword arguments for the connection itself.

CHAPTER

FIVE

PYMEASURE.EXPERIMENT

This section contains specific documentation on the classes and methods of the package.

5.1 Experiment class

The Experiment class is intended for use in the Jupyter notebook environment.

class pymeasure.experiment.experiment.**Experiment**(*title*, *procedure*, *analyse=<function Experiment.<lambda>>*)

Bases: object

Class which starts logging and creates/runs the results and worker processes.

```
procedure = Procedure()
experiment = Experiment(title, procedure)
experiment.start()
experiment.plot_live('x', 'y', style='.-')

for a multi-subplot graph:

import pylab as pl
ax1 = pl.subplot(121)
experiment.plot('x','y',ax=ax1)
ax2 = pl.subplot(122)
experiment.plot('x','z',ax=ax2)
experiment.plot_live()
```

Variables **value** – The value of the parameter

Parameters

- **title** – The experiment title
- **procedure** – The procedure object
- **analyse** – Post-analysis function, which takes a pandas dataframe as input and returns it with added (analysed) columns. The analysed results are accessible via experiment.data, as opposed to experiment.results.data for the 'raw' data.
- **_data_timeout** – Time limit for how long live plotting should wait for datapoints.

clear_plot()

Clear the figures and plot lists.

property data
Data property which returns analysed data, if an analyse function is defined, otherwise returns the raw data.

plot(**args*, ***kwargs*)
Plot the results from the experiment.data pandas dataframe. Store the plots in a plots list attribute.

plot_live(**args*, ***kwargs*)
Live plotting loop for jupyter notebook, which automatically updates (an) in-line matplotlib graph(s). Will create a new plot as specified by input arguments, or will update (an) existing plot(s).

start()
Start the worker

update_line(*ax*, *hl*, *xname*, *yname*)
Update a line in a matplotlib graph with new data.

update_plot()
Update the plots in the plots list with new data from the experiment.data pandas dataframe.

wait_for_data()
Wait for the data attribute to fill with datapoints.

pymeasure.experiment.experiment.**create_filename**(*title*)
Create a new filename according to the style defined in the config file. If no config is specified, create a temporary file.

pymeasure.experiment.experiment.**get_array**(*start*, *stop*, *step*)
Returns a numpy array from start to stop

pymeasure.experiment.experiment.**get_array_steps**(*start*, *stop*, *numsteps*)
Returns a numpy array from start to stop in numsteps

pymeasure.experiment.experiment.**get_array_zero**(*maxval*, *step*)
Returns a numpy array from 0 to maxval to -maxval to 0

5.2 Listener class

class pymeasure.experiment.listeners.**Listener**(*port*, *topic=''*, *timeout=0.01*)
Bases: pymeasure.thread.StoppableThread

Base class for Threads that need to listen for messages on a ZMQ TCP port and can be stopped by a thread-safe method call

message_waiting()
Check if we have a message, wait at most until timeout.

receive(*flags=0*)

class pymeasure.experiment.listeners.**Monitor**(*results*, *queue*)
Bases: pymeasure.log.QueueListener

class pymeasure.experiment.listeners.**Recorder**(*results*, *queue*, ***kwargs*)
Bases: pymeasure.log.QueueListener

Recorder loads the initial Results for a filepath and appends data by listening for it over a queue. The queue ensures that no data is lost between the Recorder and Worker.

stop()
Stop the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

5.3 Procedure class

class pymeasure.experiment.procedure.**Procedure**(***kwargs*)

Provides the base class of a procedure to organize the experiment execution. Procedures should be run by Workers to ensure that asynchronous execution is properly managed.

```
procedure = Procedure()
results = Results(procedure, data_filename)
worker = Worker(results, port)
worker.start()
```

Inheriting classes should define the startup, execute, and shutdown methods as needed. The shutdown method is called even with a software exception or abort event during the execute method.

If keyword arguments are provided, they are added to the object as attributes.

check_parameters()
Raises an exception if any parameter is missing before calling the associated function. Ensures that each value can be set and got, which should cast it into the right format. Used as a decorator @check_parameters on the startup method

evaluate_metadata()
Evaluates all Metadata objects, fixing their values to the current value

execute()
Preforms the commands needed for the measurement itself. During execution the shutdown method will always be run following this method. This includes when Exceptions are raised.

gen_measurement()
Create MEASURE and DATA_COLUMNS variables for get_datapoint method.

get_estimates()
Function that returns estimates that are to be displayed by the EstimatorWidget. Must be reimplemented by subclasses. Should return an int or float representing the duration in seconds, or a list with a tuple for each estimate. The tuple should consists of two strings: the first will be used as the label of the estimate, the second as the displayed estimate.

metadata_objects()
Returns a dictionary of all the Metadata objects

parameter_objects()
Returns a dictionary of all the Parameter objects and grabs any current values that are not in the default definitions

parameter_values()
Returns a dictionary of all the Parameter values and grabs any current values that are not in the default definitions

parameters_are_set()
Returns True if all parameters are set

refresh_parameters()
Enforces that all the parameters are re-cast and updated in the meta dictionary

set_parameters(*parameters*, *except_missing=True*)
Sets a dictionary of parameters and raises an exception if additional parameters are present if except_missing is True

shutdown()
Executes the commands necessary to shut down the instruments and leave them in a safe state. This method is always run at the end.

startup()
Executes the commands needed at the start-up of the measurement

class pymeasure.experiment.procedure.**UnknownProcedure**(*parameters*)
Handles the case when a *Procedure* object can not be imported during loading in the *Results* class

startup()
Executes the commands needed at the start-up of the measurement

5.4 Parameter classes

The parameter classes are used to define input variables for a *Procedure*. They each inherit from the *Parameter* base class.

class pymeasure.experiment.parameters.**BooleanParameter**(*name*, *default=None*, *ui_class=None*, *group_by=None*, *group_condition=True*)
Parameter sub-class that uses the boolean type to store the value.

Variables **value** – The boolean value of the parameter

Parameters

- **name** – The parameter name
- **default** – The default boolean value
- **ui_class** – A Qt class to use for the UI of this parameter

class pymeasure.experiment.parameters.**FloatParameter**(*name*, *units=None*, *minimum=-1000000000.0*, *maximum=1000000000.0*, *decimals=15*, *step=None*, ***kwargs*)
Parameter sub-class that uses the floating point type to store the value.

Variables **value** – The floating point value of the parameter

Parameters

- **name** – The parameter name
- **units** – The units of measure for the parameter
- **minimum** – The minimum allowed value (default: -1e9)
- **maximum** – The maximum allowed value (default: 1e9)
- **decimals** – The number of decimals considered (default: 15)
- **default** – The default floating point value
- **ui_class** – A Qt class to use for the UI of this parameter
- **step** – step size for parameter's UI spinbox. If None, spinbox will have step disabled

class pymeasure.experiment.parameters.**IntegerParameter**(*name*, *units=None*, *minimum=-1000000000.0*, *maximum=1000000000.0*, *step=None*, ***kwargs*)

Parameter sub-class that uses the integer type to store the value.

Variables **value** – The integer value of the parameter

Parameters

- **name** – The parameter name
- **units** – The units of measure for the parameter
- **minimum** – The minimum allowed value (default: -1e9)
- **maximum** – The maximum allowed value (default: 1e9)
- **default** – The default integer value
- **ui_class** – A Qt class to use for the UI of this parameter
- **step** – int step size for parameter's UI spinbox. If None, spinbox will have step disabled

class pymeasure.experiment.parameters.**ListParameter**(*name*, *choices=None*, *units=None*, ***kwargs*)

Parameter sub-class that stores the value as a list. String representation of choices must be unique.

Parameters

- **name** – The parameter name
- **choices** – An explicit list of choices, which is disregarded if None
- **units** – The units of measure for the parameter
- **default** – The default value
- **ui_class** – A Qt class to use for the UI of this parameter

property **choices**

Returns an immutable iterable of choices, or None if not set.

class pymeasure.experiment.parameters.**Measurable**(*name*, *fget=None*, *units=None*, *measure=True*, *default=None*, ***kwargs*)

Encapsulates the information for a measurable experiment parameter with information about the name, fget function and units if supplied. The value property is called when the procedure retrieves a datapoint and calls the fget function. If no fget function is specified, the value property will return the latest set value of the parameter (or default if never set).

Variables **value** – The value of the parameter

Parameters

- **name** – The parameter name
- **fget** – The parameter fget function (e.g. an instrument parameter)
- **default** – The default value

class pymeasure.experiment.parameters.**Metadata**(*name*, *fget=None*, *units=None*, *default=None*, *fmt='%s'*)

Encapsulates the information for metadata of the experiment with information about the name, the fget function and the units, if supplied. If no fget function is specified, the value property will return the latest set value of the parameter (or default if never set).

Variables **value** – The value of the parameter. This returns (if a value is set) the value obtained from the *fget* (after evaluation) or a manually set value. Returns *None* if no value has been set

Parameters

- **name** – The parameter name
- **fget** – The parameter fget function; can be provided as a callable, or as a string, in which case it is assumed to be the name of a method or attribute of the *Procedure* class in which the Metadata is defined. Passing a string also allows for nested attributes by separating them with a period (e.g. to access an attribute or method of an instrument) where only the last attribute can be a method.
- **units** – The parameter units
- **default** – The default value, in case no value is assigned or if no fget method is provided
- **fmt** – A string used to format the value upon writing it to a file. Default is "%s"

`is_set()`
Returns True if the Parameter value is set

`class pymeasure.experiment.parameters.Parameter(name, default=None, ui_class=None, group_by=None, group_condition=True)`
Encapsulates the information for an experiment parameter with information about the name, and units if supplied.

Variables **value** – The value of the parameter

Parameters

- **name** – The parameter name
- **default** – The default value
- **ui_class** – A Qt class to use for the UI of this parameter
- **group_by** – Defines the Parameter(s) that controls the visibility of the associated input; can be a string containting the Parameter name, a list of strings with multiple Parameter names, or a dict containing {"Parameter name": condition} pairs.
- **group_condition** – The condition for the group_by Parameter that controls the visibility of this parameter, provided as a value or a (lambda)function. If the group_by argument is provided as a list of strings, this argument can be either a single condition or a list of conditions. If the group_by argument is provided as a dict this argument is ignored.

`is_set()`
Returns True if the Parameter value is set

`class pymeasure.experiment.parameters.PhysicalParameter(name, uncertaintyType='absolute', **kwargs)`
VectorParameter sub-class of 2 dimensions to store a value and its uncertainty.

Variables **value** – The value of the parameter as a list of 2 floating point numbers

Parameters

- **name** – The parameter name
- **uncertainty_type** – Type of uncertainty, 'absolute', 'relative' or 'percentage'
- **units** – The units of measure for the parameter
- **default** – The default value
- **ui_class** – A Qt class to use for the UI of this parameter

`class pymeasure.experiment.parameters.VectorParameter(name, length=3, units=None, **kwargs)`
Parameter sub-class that stores the value in a vector format.

Variables **value** – The value of the parameter as a list of floating point numbers

Parameters

- **name** – The parameter name
- **length** – The integer dimensions of the vector
- **units** – The units of measure for the parameter
- **default** – The default value
- **ui_class** – A Qt class to use for the UI of this parameter

5.5 Worker class

class pymeasure.experiment.workers.**Worker**(*results*, *log_queue=None*, *log_level=20*, *port=None*)
Bases: pymeasure.thread.StoppableThread

Worker runs the procedure and emits information about the procedure and its status over a ZMQ TCP port. In a child thread, a Recorder is run to write the results to

emit(*topic*, *record*)
Emits data of some topic over TCP

handle_abort()

handle_error()

join(*timeout=0*)
Joins the current thread and forces it to stop after the timeout if necessary

Parameters **timeout** – Timeout duration in seconds

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

shutdown()

update_status(*status*)

5.6 Results class

class pymeasure.experiment.results.**CSVFormatter**(*columns*, *delimiter=','*)
Formatter of data results

format(*record*)
Formats a record as csv.

Parameters **record** (*dict*) – record to format.

Returns a string

class pymeasure.experiment.results.**Results**(*procedure*, *data_filename*)
The Results class provides a convenient interface to reading and writing data in connection with a *Procedure* object.

Variables

- **COMMENT** – The character used to identify a comment (default: #)
- **DELIMITER** – The character used to delimit the data (default: ,)
- **LINE_BREAK** – The character used for line breaks (default n)
- **CHUNK_SIZE** – The length of the data chuck that is read

Parameters

- **procedure** – Procedure object
- **data_filename** – The data filename where the data is or should be stored

format(*data*)
Returns a formatted string containing the data to be written to a file

header()
Returns a text header to accompany a datafile so that the procedure can be reconstructed

labels()
Returns the columns labels as a string to be written to the file

static load(*data_filename*, *procedure_class=None*)
Returns a Results object with the associated Procedure object and data

metadata()
Returns a text header for the metadata to write into the datafile

parse(*line*)
Returns a dictionary containing the data from the line

static parse_header(*header*, *procedure_class=None*)
Returns a Procedure object with the parameters as defined in the header text.

reload()
Preforms a full reloading of the file data, neglecting any changes in the comments

store_metadata()
Inserts the metadata header (if any) into the datafile

pymeasure.experiment.results.**replace_placeholders**(*string*, *procedure*, *date_format='%Y-%m-%d'*, *time_format='%H:%M:%S'*)
Replace placeholders in string with values from procedure parameters.

Replaces the placeholders in the provided string with the values of the associated parameters, as provided by the procedure. This uses the standard python string.format syntax. Apart from the parameter in the procedure (which should be called by their full names) "date" and "time" are also added as optional placeholders.

Parameters

- **string** – The string in which the placeholders are to be replaced. Python string.format syntax is used, e.g. "{Parameter Name}" to insert a FloatParameter called "Parameter Name", or "{Parameter Name:.2f}" to also specifically format the parameter.
- **procedure** – The procedure from which to get the parameter values.
- **date_format** – A string to represent how the additional placeholder "date" will be formatted.
- **time_format** – A string to represent how the additional placeholder "time" will be formatted.

pymeasure.experiment.results.**unique_filename**(*directory*, *prefix='DATA'*, *suffix=''*, *ext='csv'*, *dated_folder=False*, *index=True*, *datetimeformat='%Y-%m-%d'*, *procedure=None*)

Returns a unique filename based on the directory and prefix

CHAPTER

SIX

PYMEASURE.DISPLAY

This section contains specific documentation on the classes and methods of the package.

6.1 Browser classes

class pymeasure.display.browser.**Browser**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QTreeWidget

Graphical list view of *Experiment* objects allowing the user to view the status of queued Experiments as well as loading and displaying data from previous runs.

In order that different Experiments be displayed within the same Browser, they must have entries in *DATA_COLUMNS* corresponding to the *measured_quantities* of the Browser.

add(*experiment*)
Add a *Experiment* object to the Browser. This function checks to make sure that the Experiment measures the appropriate quantities to warrant its inclusion, and then adds a BrowserItem to the Browser, filling all relevant columns with Parameter data.

class pymeasure.display.browser.**BrowserItem**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QTreeWidgetItem

Represent a row in the *Browser* tree widget

6.2 Curves classes

class pymeasure.display.curves.**BufferCurve**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.PlotDataItem

Creates a curve based on a predefined buffer size and allows data to be added dynamically.

append(*x*, *y*)
Appends data to the curve with optional errors

prepare(*size*, *dtype=<class 'numpy.float32'>*)
Prepares the buffer based on its size, data type

class pymeasure.display.curves.**Crosshairs**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QObject

Attaches crosshairs to the a plot and provides a signal with the x and y graph coordinates

mouseMoved(*event=None*)
Updates the mouse position upon mouse movement

update()
Updates the mouse position based on the data in the plot. For dynamic plots, this is called each time the data changes to ensure the x and y values correspond to those on the display.

class pymeasure.display.curves.**ResultsCurve**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.PlotDataItem

Creates a curve loaded dynamically from a file through the Results object. The data can be forced to fully reload on each update, useful for cases when the data is changing across the full file instead of just appending.

update_data()
Updates the data by polling the results

class pymeasure.display.curves.**ResultsImage**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.ImageItem

Creates an image loaded dynamically from a file through the Results object.

colormap(*x*)
Return mapped color as 0.0-1.0 floats RGBA

find_img_index(*x*, *y*)
Finds the integer image indices corresponding to the closest x and y points of the data given some x and y data.

round_up(*x*)
Convenience function since numpy rounds to even

6.3 Inputs classes

class pymeasure.display.inputs.**BooleanInput**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.inputs.Input*, pyqtgraph.Qt.QtWidgets.QCheckBox

Checkbox for boolean values, connected to a BooleanParameter.

set_parameter(*parameter*)
Connects a new parameter to the input box, and initializes the box value.

Parameters **parameter** – parameter to connect.

class pymeasure.display.inputs.**Input**(*parameter*, ***kwargs*)
Bases: object

Mix-in class that connects a *Parameter* object to a GUI input box.

Parameters **parameter** – The parameter to connect to this input box.

Attr parameter Read-only property to access the associated parameter.

property parameter
The connected parameter object. Read-only property; see *set_parameter()*.

Note that reading this property will have the side-effect of updating its value from the GUI input box.

set_parameter(*parameter*)
Connects a new parameter to the input box, and initializes the box value.

Parameters **parameter** – parameter to connect.

update_parameter()
Update the parameter value with the Input GUI element's current value.

class pymeasure.display.inputs.**IntegerInput**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.inputs.Input*, pyqtgraph.Qt.QtWidgets.QSpinBox

Spin input box for integer values, connected to a `IntegerParameter`.

set_parameter(*parameter*)
Connects a new parameter to the input box, and initializes the box value.

Parameters **parameter** – parameter to connect.

class pymeasure.display.inputs.**ListInput**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.inputs.Input*, pyqtgraph.Qt.QtWidgets.QComboBox

Dropdown for list values, connected to a `ListParameter`.

set_parameter(*parameter*)
Connects a new parameter to the input box, and initializes the box value.

Parameters **parameter** – parameter to connect.

class pymeasure.display.inputs.**ScientificInput**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.inputs.Input*, pyqtgraph.Qt.QtWidgets.QDoubleSpinBox

Spinner input box for floating-point values, connected to a `FloatParameter`. This box will display and accept values in scientific notation when appropriate.

See also:

Class `FloatInput` For a non-scientific floating-point input box.

set_parameter(*parameter*)
Connects a new parameter to the input box, and initializes the box value.

Parameters **parameter** – parameter to connect.

class pymeasure.display.inputs.**StringInput**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.inputs.Input*, pyqtgraph.Qt.QtWidgets.QLineEdit

String input box connected to a `Parameter`. Parameter subclasses that are string-based may also use this input, but non-string parameters should use more specialised input classes.

6.4 Listeners classes

class pymeasure.display.listeners.**Monitor**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QThread

Monitor listens for status and progress messages from a Worker through a queue to ensure no messages are losts

class pymeasure.display.listeners.**QListener**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.thread.StoppableQThread*

Base class for QThreads that need to listen for messages on a ZMQ TCP port and can be stopped by a thread- and process-safe method call

6.5 Log classes

class pymeasure.display.log.**LogHandler**
Bases: logging.Handler

class Emitter(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QObject

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

6.6 Manager classes

class pymeasure.display.manager.**Experiment**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QObject

The Experiment class helps group the *Procedure*, *Results*, and their display functionality. Its function is only a convenient container.

Parameters

- **results** – *Results* object
- **curve_list** – *ResultsCurve* list. List of curves associated with an experiment. They could represent different views of the same experiment.
- **browser_item** – *BrowserItem* object

class pymeasure.display.manager.**ExperimentQueue**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QObject

Represents a Queue of Experiments and allows queries to be easily preformed

has_next()
Returns True if another item is on the queue

next()
Returns the next experiment on the queue

class pymeasure.display.manager.**Manager**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QObject

Controls the execution of *Experiment* classes by implementing a queue system in which Experiments are added, removed, executed, or aborted. When instantiated, the Manager is linked to a *Browser* and a PyQtGraph *PlotItem* within the user interface, which are updated in accordance with the execution status of the Experiments.

abort()
Aborts the currently running Experiment, but raises an exception if there is no running experiment

clear()
Remove all Experiments

is_running()
Returns True if a procedure is currently running

load(*experiment*)
Load a previously executed Experiment

next()
Initiates the start of the next experiment in the queue as long as no other experiments are currently running and there is a procedure in the queue.

queue(*experiment*)
Adds an experiment to the queue.

remove(*experiment*)
Removes an Experiment

resume()
Resume processing of the queue.

6.7 Plotter class

class pymeasure.display.plotter.**Plotter**(*results*, *refresh_time=0.1*, *linewidth=1*)
Bases: pymeasure.thread.StoppableThread

Plotter dynamically plots data from a file through the Results object.

See also:

Tutorial *Using the Plotter* A tutorial and example on using the Plotter and PlotterWindow.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setup_plot(*plot*)
This method does nothing by default, but can be overridden by the child class in order to set up custom options for the plot window, via its PlotItem.

Parameters **plot** – This window's PlotItem instance.

6.8 Qt classes

All Qt imports should reference pymeasure.display.Qt, for consistant importing from either PySide or PyQt4.

Qt.**fromUi**(***kwargs*)
Returns a Qt object constructed using loadUiType based on its arguments. All QWidget objects in the form class are set in the returned object for easy accessability.

6.9 Thread classes

class pymeasure.display.thread.**StoppableQThread**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QThread

Base class for QThreads which require the ability to be stopped by a thread-safe method call

join(*timeout=0*)
Joins the current thread and forces it to stop after the timeout if necessary

Parameters **timeout** – Timeout duration in seconds

6.10 Widget classes

class pymeasure.display.widgets.browser_widget.**BrowserWidget**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QWidget

Widget wrapper for *Browser* class

class pymeasure.display.widgets.directory_widget.**DirectoryLineEdit**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QLineEdit

Widget that allows to choose a directory path. A completer is implemented for quick completion. A browse button is available.

class pymeasure.display.widgets.estimator_widget.**EstimatorThread**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.thread.StoppableQThread*

class pymeasure.display.widgets.estimator_widget.**EstimatorWidget**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QWidget

Widget that allows to display up-front estimates of the measurement procedure.

This widget relies on a *get_estimates* method of the *Procedure* class. *get_estimates* is expected to return a list of tuples, where each tuple contains two strings: a label and the estimate.

If the *SequencerWidget* is also used, it is possible to ask for the current sequencer or its length by asking for two keyword arguments in the Implementation of the *get_estimates* function: *sequence* and *sequence_length*, respectively.

check_get_estimates_signature()
Method that checks the signature of the get_estimates function. It checks which input arguments are allowed and, if the output is correct for the EstimatorWidget, stores the number of estimates.

display_estimates(*estimates*)
Method that updates the shown estimates for the given set of estimates.

Parameters **estimates** – The set of estimates to be shown in the form of a list of tuples of (2) strings

get_estimates()
Method that makes a procedure with the currently entered parameters and returns the estimates for these parameters.

update_estimates()
Method that gets and displays the estimates. Implemented for connecting to the 'update'-button.

class pymeasure.display.widgets.image_frame.**ImageFrame**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.widgets.plot_frame.PlotFrame*

Extends *PlotFrame* to plot also axis Z using colors

ResultsClass
alias of *pymeasure.display.curves.ResultsImage*

class pymeasure.display.widgets.image_widget.**ImageWidget**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.widgets.tab_widget.TabWidget*, pyqtgraph.Qt.QtWidgets.QWidget

Extends the *ImageFrame* to allow different columns of the data to be dynamically chosen

load(*curve*)
Add curve to widget

new_curve(*results*, *color=pyqtgraph.intColor*, ***kwargs*)
Creates a new image

remove(*curve*)
Remove curve from widget

class pymeasure.display.widgets.inputs_widget.**InputsWidget**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QWidget

Widget wrapper for various *Inputs classes*

get_procedure()
Returns the current procedure

class pymeasure.display.widgets.log_widget.**LogWidget**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.widgets.tab_widget.TabWidget*, pyqtgraph.Qt.QtWidgets.QWidget

Widget to display logging information in GUI

It is recommended to include this widget in all subclasses of *ManagedWindowBase*

class pymeasure.display.widgets.plot_frame.**PlotFrame**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QFrame

Combines a PyQtGraph Plot with Crosshairs. Refreshes the plot based on the refresh_time, and allows the axes to be changed on the fly, which updates the plotted data

ResultsClass
alias of *pymeasure.display.curves.ResultsCurve*

parse_axis(*axis*)
Returns the units of an axis by searching the string

class pymeasure.display.widgets.plot_widget.**PlotWidget**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.widgets.tab_widget.TabWidget*, pyqtgraph.Qt.QtWidgets.QWidget

Extends *PlotFrame* to allow different columns of the data to be dynamically chosen

load(*curve*)
Add curve to widget

new_curve(*results*, *color=pyqtgraph.intColor*, ***kwargs*)
Create a new curve

remove(*curve*)
Remove curve from widget

set_color(*curve*, *color*)
Change the color of the pen of the curve

class pymeasure.display.widgets.results_dialog.**ResultsDialog**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QFileDialog

Widget that displays a dialog box for loading a past experiment run. It shows a preview of curves from the results file when selected in the dialog box.

This widget used by the *open_experiment* method in `ManagedWindowBase` class

class pymeasure.display.widgets.sequencer_widget.**ComboBoxDelegate**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QStyledItemDelegate

class pymeasure.display.widgets.sequencer_widget.**ExpressionValidator**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtGui.QValidator

class pymeasure.display.widgets.sequencer_widget.**LineEditDelegate**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QStyledItemDelegate

class pymeasure.display.widgets.sequencer_widget.**SequenceDialog**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QFileDialog

Widget that displays a dialog box for loading or saving a sequence tree.

It also shows a preview of sequence tree in the dialog box

Parameters **save** – True if we are saving a file. Default False.

class pymeasure.display.widgets.sequencer_widget.**SequencerTreeModel**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtCore.QAbstractItemModel

Model for sequencer data

Parameters

- **header** – List of string representing header data
- **data** – data associated with the model
- **parent** – A QWidget that QT will give ownership of this Widget to.

add_node(*parameter*, *parent=None*)
Add a row in the sequencer

columnCount(*parent*)
Return the number of columns in the model header.

The parent parameter exists only to support the signature of QAbstractItemModel.

data(*index*, *role*)
Return the data to display for the given index and the given role.

This method should not be called directly. This method is called implicitly by the QTreeView that is displaying us, as the way of finding out what to display where.

flags(*index*)
Set the flags for the item at the given QModelIndex.

Here, we just set all indexes to enabled, and selectable.

headerData(*section*, *orientation*, *role*)
Return the header data for the given section, orientation and role.

This method should not be called directly. This method is called implicitly by the QTreeView that is displaying us, as the way of finding out what to display where.

index(*row*, *col*, *parent*)
Return a QModelIndex instance pointing the row and column underneath the parent given. This method should not be called directly. This method is called implicitly by the QTreeView that is displaying us, as the way of finding out what to display where.

parent(*index=None*)
Return the index of the parent of a given index. If index is not supplied, return an invalid QModelIndex.

Parameters **index** – QModelIndex optional.

Returns

remove_node(*index*)
Remove a row in the sequencer

rowCount(*parent*)
Return the number of children of a given parent.

If an invalid QModelIndex is supplied, return the number of children under the root.

Parameters **parent** – QModelIndex

visit_tree(*parent*)
Return a generator to enumerate all the nodes in the tree

class pymeasure.display.widgets.sequencer_widget.**SequencerTreeView**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QTreeView

class pymeasure.display.widgets.sequencer_widget.**SequencerWidget**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QWidget

Widget that allows to generate a sequence of measurements

It allows sweeping parameters and moreover, one can write a simple text file to easily load a sequence. Sequences can also be saved

Currently requires a queue function of the *ManagedWindow* to have a "procedure" argument.

Parameters **inputs** – List of strings representing the parameters name

load_sequence(***, *filename=None*)
Load a sequence from a .txt file.

Parameters **filename** – Filename (string) of the to-be-loaded file.

queue_sequence()
Obtain a list of parameters from the sequence tree, enter these into procedures, and queue these procedures.

class pymeasure.display.widgets.tab_widget.**TabWidget**(*name*, **args*, ***kwargs*)
Bases: object

Utility class to define default implementation for some basic methods.

When defining a widget to be used in subclasses of *ManagedWindowBase*, users should inherit from this class and provide an implementation of these methods

load(*curve*)
Add curve to widget

new_curve(**args*, ***kwargs*)
Create a new curve

remove(*curve*)
Remove curve from widget

set_color(*curve*, *color*)
Set color for widget

class pymeasure.display.widgets.dock_widget.**DockWidget**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.widgets.tab_widget.TabWidget*, pyqtgraph.Qt.QtWidgets.QWidget

Widget that contains a DockArea with a number of Docks as determined by the length of the longest x_axis_labels or y_axis_labels list.

Parameters

- **name** – Name for the TabWidget
- **procedure_class** – procedure class describing the experiment (see *Procedure*)
- **x_axis_labels** – List of data column(s) for the x-axis of the plot. If the list is shorter than y_axis_labels the last item in the list to match y_axis_labels length.
- **y_axis_labels** – List of data column(s) for the y-axis of the plot. If the list is shorter than x_axis_labels the last item in the list to match x_axis_labels length.
- **linewidth** – line width for plots in *PlotWidget*
- **parent** – Passed on to QtWidgets.QWidget. Default is None

new_curve(*results*, *color=pyqtgraph.intColor*, ***kwargs*)
Create a new curve

6.11 Windows classes

class pymeasure.display.windows.managed_image_window.**ManagedImageWindow**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.windows.managed_window.ManagedWindow*

Display experiment output with an *ImageWidget* class.

Parameters

- **procedure_class** – procedure class describing the experiment (see *Procedure*)
- **x_axis** – the data-column for the x-axis of the plot, cannot be changed afterwards for the image-plot
- **y_axis** – the data-column for the y-axis of the plot, cannot be changed afterwards for the image-plot
- **z_axis** – the initial data-column for the z-axis of the plot, can be changed afterwards
- ****kwargs** – optional keyword arguments that will be passed to ManagedWindow

class pymeasure.display.windows.managed_window.**ManagedWindow**(**args: Any*, ***kwargs: Any*)
Bases: *pymeasure.display.windows.managed_window.ManagedWindowBase*

Display experiment output with an *PlotWidget* class.

See also:

Tutorial *Using the ManagedWindow* A tutorial and example on the basic configuration and usage of ManagedWindow.

Parameters

- **procedure_class** – procedure class describing the experiment (see *Procedure*)

- **x_axis** – the initial data-column for the x-axis of the plot
- **y_axis** – the initial data-column for the y-axis of the plot
- **linewidth** – linewidth for the displayed curves, default is 1
- ****kwargs** – optional keyword arguments that will be passed to *ManagedWindowBase*

class pymeasure.display.windows.managed_window.**ManagedWindowBase**(**args: Any*, ***kwargs: Any*)
Bases: pyqtgraph.Qt.QtWidgets.QMainWindow

Base class for GUI experiment management .

The ManagedWindowBase provides an interface for inputting experiment parameters, running several experiments (*Procedure*), plotting result curves, and listing the experiments conducted during a session.

The ManagedWindowBase uses a Manager to control Workers in a Queue, and provides a simple interface. The *queue()* method must be overridden by the child class.

The ManagedWindowBase allow user to define a set of widget that display information about the experiment. The information displayed may include: plots, tabular view, logging information,...

This class is not intended to be used directy, but it should be subclassed to provide some appropriate widget list. Example of classes usable as element of widget list are:

- *LogWidget*
- *PlotWidget*
- *ImageWidget*

Of course, users can define its own widget making sure that inherits from *TabWidget*.

Examples of ready to use classes inherited from ManagedWindowBase are:

- *ManagedWindow*
- *ManagedImageWindow*

See also:

Tutorial ***Using the ManagedWindow*** A tutorial and example on the basic configuration and usage of ManagedWindow.

Parameters for `__init__` constructor.

Parameters

- **procedure_class** – procedure class describing the experiment (see *Procedure*)
- **widget_list** – list of widget to be displayed in the GUI
- **inputs** – list of *Parameter* instance variable names, which the display will generate graphical fields for
- **displays** – list of *Parameter* instance variable names displayed in the browser window
- **log_channel** – `logging.Logger` instance to use for logging output
- **log_level** – logging level
- **parent** – Parent widget or `None`
- **sequencer** – a boolean stating whether or not the sequencer has to be included into the window

- **sequencer_inputs** – either None or a list of the parameter names to be scanned over. If no list of parameters is given, the parameters displayed in the manager queue are used.
- **sequence_file** – simple text file to quickly load a pre-defined sequence with the code:*Load sequence* button
- **inputs_in_scrollarea** – boolean that display or hide a scrollbar to the input area
- **directory_input** – specify, if present, where the experiment's result will be saved.
- **hide_groups** – a boolean controlling whether parameter groups are hidden (True, default) or disabled/grayed-out (False) when the group conditions are not met.

open_file_externally(*filename*)

Method to open the datafile using an external editor or viewer. Uses the default application to open a datafile of this filetype, but can be overridden by the child class in order to open the file in another application of choice.

queue(*procedure=None*)

Abstract method, which must be overridden by the child class.

Implementations must call `self.manager.queue(experiment)` and pass an `experiment` (*Experiment*) object which contains the *Results* and *Procedure* to be run.

The optional *procedure* argument is not required for a basic implementation, but is required when the *SequencerWidget* is used.

For example:

```
def queue(self):
    filename = unique_filename('results', prefix="data") # from pymeasure.
↪experiment

    procedure = self.make_procedure() # Procedure class was passed at␣
↪construction
    results = Results(procedure, filename)
    experiment = self.new_experiment(results)

    self.manager.queue(experiment)
```

set_parameters(*parameters*)

This method should be overwritten by the child class. The parameters argument is a dictionary of Parameter objects. The Parameters should overwrite the GUI values so that a user can click "Queue" to capture the same parameters.

class pymeasure.display.windows.plotter_window.**PlotterWindow**(**args: Any*, ***kwargs: Any*)

Bases: pyqtgraph.Qt.QtWidgets.QMainWindow

A window for plotting experiment results. Should not be instantiated directly, but only via the *Plotter* class.

See also:

Tutorial *Using the Plotter* A tutorial and example code for using the Plotter and PlotterWindow.

check_stop()

Checks if the Plotter should stop and exits the Qt main loop if so

class pymeasure.display.windows.managed_dock_window.**ManagedDockWindow**(**args: Any*, ***kwargs: Any*)

Bases: *pymeasure.display.windows.managed_window.ManagedWindowBase*

Display experiment output with multiple docking windows with *DockWidget* class.

Parameters

- **procedure_class** – procedure class describing the experiment (see *Procedure*)
- **x_axis** – the data column(s) for the x-axis of the plot. This may be a string or a list of strings from the data columns of the procedure. The list length determines the number of plots
- **y_axis** – the data column(s) for the y-axis of the plot. This may be a string or a list of strings from the data columns of the procedure. The list length determines the number of plots
- ****kwargs** – optional keyword arguments that will be passed to *ManagedWindowBase*

CHAPTER

SEVEN

PYMEASURE.INSTRUMENTS

This section contains documentation on the instrument classes.

7.1 Instrument classes

class pymeasure.instruments.common_base.**CommonBase**(***kwargs*)

Base class for instruments and channels.

This class contains everything needed for pymeasure's property creator *control()* and its derivatives *measurement()* and *setting()*.

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)

Add channels to the parent class.

The children will be added to the parent instance at instantiation with *CommonBase.add_child()*. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

ask(*command*, *query_delay=0*)

Write a command to the instrument and return the read response.

Parameters

- **command** – Command string to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.

Returns String returned by the device without read_termination.

binary_values(*command*, *query_delay=0*, ***kwargs*)

Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

static control(*get_command*, *set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, ***kwargs*)

Return a property for the class based on the supplied commands. This property may be set and read from the instrument. See also *measurement()* and *setting()*.

Parameters

- **get_command** – A string command that asks for the value, set to *None* if get is not supported (see also *setting()*).
- **set_command** – A string command that writes the value, set to *None* if set is not supported (see also *measurement()*).
- **docs** – A docstring that will be included in the documentation

- **validator** – A function that takes both a value and a group of valid values and returns a valid value, while it otherwise raises an exception
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **get_process** – A function that take a value and allows processing before value mapping, returning the processed value
- **set_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **command_process** – A function that takes a command and allows processing before executing the command
- **check_set_errors** – Toggles checking errors after setting
- **check_get_errors** – Toggles checking errors after getting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses.

Example of usage of dynamic parameter is as follows:

```
class GenericInstrument(Instrument):
    center_frequency = Instrument.control(
        ":SENS:FREQ:CENT?;", ":SENS:FREQ:CENT %e GHz;",
        " A floating point property that represents the frequency ... ",
        validator=strict_range,
        # Redefine this in subclasses to reflect actual instrument value:
        values=(1, 20),
        dynamic=True  # enable changing property parameters on-the-fly
    )

class SpecificInstrument(GenericInstrument):
    # Identical to GenericInstrument, except for frequency range
    # Override the "values" parameter of the "center_frequency" property
    center_frequency_values = (1, 10) # Redefined at subclass level

instrument = SpecificInstrument()
instrument.center_frequency_values = (1, 6e9) # Redefined at instance level
```

Warning: Unexpected side effects when using dynamic properties

Users must pay attention when using dynamic properties, since definition of class and/or instance attributes matching specific patterns could have unwanted side effect. The attribute name pattern *property_param*, where *property* is the name of the dynamic property (e.g. *center_frequency* in the example) and *param* is any of this method parameters name except *dynamic* and *docs* (e.g. *values* in the example) has to be considered reserved for dynamic property control.

static measurement(*get_command*, *docs*, *values=()*, *map_values=None*, *get_process=<function CommonBase.<lambda>>*, *command_process=<function CommonBase.<lambda>>*, *check_get_errors=False*, *dynamic=False*, ***kwargs*)

Return a property for the class based on the supplied commands. This is a measurement quantity that may only be read from the instrument, not set.

Parameters

- **get_command** – A string command that asks for the value
- **docs** – A docstring that will be included in the documentation
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **get_process** – A function that take a value and allows processing before value mapping, returning the processed value
- **command_process** – A function that take a command and allows processing before executing the command, for getting
- **check_get_errors** – Toggles checking errors after getting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses. See *control()* for an usage example.

remove_child(*child*)

Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

static setting(*set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *set_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *dynamic=False*, ***kwargs*)

Return a property for the class based on the supplied commands. This property may be set, but raises an exception when being read from the instrument.

Parameters

- **set_command** – A string command that writes the value
- **docs** – A docstring that will be included in the documentation
- **validator** – A function that takes both a value and a group of valid values and returns a valid value, while it otherwise raises an exception
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **set_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **check_set_errors** – Toggles checking errors after setting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses. See *control()* for an usage example.

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*, *maxsplit=-1*)

Write a command to the instrument and return a list of formatted values from the result.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result

- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string.
- **maxsplit** – At most *maxsplit* splits are done. -1 (default) indicates no limit.

Returns A list of the desired type, or strings where the casting fails

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Implement in subclass!

Parameters **query_delay** – Delay between writing and reading in seconds.

class pymeasure.instruments.**Instrument**(*adapter*, *name*, *includeSCPI=True*, ***kwargs*)
The base class for all Instrument definitions.

It makes use of one of the *Adapter* classes for communication with the connected hardware device. This decouples the instrument/command definition from the specific communication interface used.

When `adapter` is a string, this is taken as an appropriate resource name. Depending on your installed VISA library, this can be something simple like `COM1` or `ASRL2`, or a more complicated VISA resource name defining the target of your connection.

When `adapter` is an integer, a GPIB resource name is created based on that. In either case a *VISAAdapter* is constructed based on that resource name. Keyword arguments can be used to further configure the connection.

Otherwise, the passed *Adapter* object is used and any keyword arguments are discarded.

This class defines basic SCPI commands by default. This can be disabled with `includeSCPI` for instruments not compatible with the standard SCPI commands.

Parameters

- **adapter** – A string, integer, or *Adapter* subclass object
- **name** (*string*) – The name of the instrument. Often the model designation by default.
- **includeSCPI** – A boolean, which toggles the inclusion of standard SCPI commands
- ****kwargs** – In case `adapter` is a string or integer, additional arguments passed on to *VISAAdapter* (check there for details). Discarded otherwise.

check_errors()
Read all errors from the instrument.

Returns list of error entries

clear()
Clears the instrument status byte

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property id
Requests and returns the identification of the instrument.

property options
Requests and returns the device options installed.

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

reset()
Resets the instrument.

shutdown()
Brings the instrument to a safe and stable state

property status
Requests and returns the status byte and Master Summary Status bit.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Parameters

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

class pymeasure.instruments.**Channel**(*parent*, *id*)
The base class for channel definitions.

This class supports dynamic properties like *Instrument*, but requires an *Instrument* instance as a parent for communication.

insert_id() inserts the channel id into the command string sent to the instrument. The default implementation replaces the Channel's *placeholder* (default "ch") with the channel id in all command strings (e.g. "CHANnel{ch}:foo").

Parameters

- **parent** – The instrument (an instance of *Instrument*) to which the channel belongs.
- **id** – Identifier of the channel, as it is used for the communication.

check_errors()
Read all errors from the instrument.

Returns list of error entries

insert_id(*command*)
Insert the channel id in a command replacing *placeholder*.

Subclass this method if you want to do something else, like always prepending the channel id.

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

read_binary_values(***kwargs*)
Read binary values from the instrument.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Parameters

- **command** – command string to be sent to the instrument. '{ch}' is replaced by the channel id.
- **kwargs** – Keyword arguments for the adapter.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the instrument.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

class pymeasure.instruments.fakes.**FakeInstrument**(*adapter=None*, *name=None*, *includeSCPI=False*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Provides a fake implementation of the Instrument class for testing purposes.

static control(*get_command*, *set_command*, *docs*, *validator=<function FakeInstrument.<lambda>>*, *values=()*, *map_values=False*, *get_process=<function FakeInstrument.<lambda>>*, *set_process=<function FakeInstrument.<lambda>>*, *check_set_errors=False*, *check_get_errors=False*, ***kwargs*)

Fake Instrument.control.

Strip commands and only store and return values indicated by format strings to mimic many simple commands. This is analogous how the tests in test_instrument are handled.

class pymeasure.instruments.fakes.**SwissArmyFake**(*wait=0.1*, ***kwargs*)

Bases: *pymeasure.instruments.fakes.FakeInstrument*

Dummy instrument class useful for testing.

Like a Swiss Army knife, this class provides multi-tool functionality in the form of streams of multiple types of fake data. Data streams that can currently be generated by this class include 'voltages', sinusoidal 'waveforms', and mono channel 'image data'.

property frame

Get a new image frame.

property frame_format

Format for image data returned from the get_frame() method. Allowed values are: mono_8: single channel 8-bit image. mono_16: single channel 16-bit image.

property frame_height

Image frame height in pixels.

property frame_width

Image frame width in pixels.

property time

Float property for elapsed time.

property voltage

Get the voltage.

property wave

Return a waveform.

7.2 Validator functions

Validators are used in conjunction with the `Instrument.control` or `Instrument.setting` functions to allow properties with complex restrictions for valid values. They are described in more detail in the *Restricting values with validators* section.

pymeasure.instruments.validators.**discreteTruncate**(*number*, *discreteSet*)

Truncates the number to the closest element in the positive discrete set. Returns False if the number is larger than the maximum value or negative.

pymeasure.instruments.validators.**joined_validators**(**validators*)

Returns a validator function that represents a list of validators joined together.

A value passed to the validator is returned if it passes any validator (not all of them). Otherwise it raises a ValueError.

Note: the joined validator expects `values` to be a sequence of `values` appropriate for the respective validators (often sequences themselves).

Example

```
>>> from pymeasure.instruments.validators import strict_discrete_set, strict_range
>>> from pymeasure.instruments.validators import joined_validators
>>> joined_v = joined_validators(strict_discrete_set, strict_range)
>>> values = [['MAX','MIN'], range(10)]
>>> joined_v(5, values)
5
>>> joined_v('MAX', values)
'MAX'
>>> joined_v('NONSENSE', values)
Traceback (most recent call last):
...
ValueError: Value of NONSENSE does not match any of the joined validators
```

Parameters **validators** – an iterable of other validators

pymeasure.instruments.validators.**modular_range**(*value*, *values*)

Provides a validator function that returns the value if it is in the range. Otherwise it returns the value, modulo the max of the range.

Parameters

- **value** – a value to test
- **values** – A set of values that are valid

pymeasure.instruments.validators.**modular_range_bidirectional**(*value*, *values*)

Provides a validator function that returns the value if it is in the range. Otherwise it returns the value, modulo the max of the range. Allows negative values.

Parameters

- **value** – a value to test
- **values** – A set of values that are valid

pymeasure.instruments.validators.**strict_discrete_range**(*value*, *values*, *step*)

Provides a validator function that returns the value if its value is less than the maximum and greater than the minimum of the range and is a multiple of step. Otherwise it raises a ValueError.

Parameters

- **value** – A value to test
- **values** – A range of values (range, list, etc.)
- **step** – Minimum stepsize (resolution limit)

Raises ValueError if the value is out of the range

pymeasure.instruments.validators.**strict_discrete_set**(*value*, *values*)

Provides a validator function that returns the value if it is in the discrete set. Otherwise it raises a ValueError.

Parameters

- **value** – A value to test
- **values** – A set of values that are valid

Raises ValueError if the value is not in the set

pymeasure.instruments.validators.**strict_range**(*value*, *values*)

Provides a validator function that returns the value if its value is less than or equal to the maximum and greater than or equal to the minimum of `values`. Otherwise it raises a ValueError.

Parameters

- **value** – A value to test
- **values** – A range of values (range, list, etc.)

Raises ValueError if the value is out of the range

pymeasure.instruments.validators.**truncated_discrete_set**(*value*, *values*)
Provides a validator function that returns the value if it is in the discrete set. Otherwise, it returns the smallest value that is larger than the value.

Parameters

- **value** – A value to test
- **values** – A set of values that are valid

pymeasure.instruments.validators.**truncated_range**(*value*, *values*)
Provides a validator function that returns the value if it is in the range. Otherwise it returns the closest range bound.

Parameters

- **value** – A value to test
- **values** – A set of values that are valid

7.3 Comedi data acquisition

The Comedi libraries provide a convenient method for interacting with data acquisition cards, but are restricted to Linux compatible operating systems.

pymeasure.instruments.comedi.**getAI**(*device*, *channel*, *range=None*)
Returns the analog input channel as specified for a given device

pymeasure.instruments.comedi.**getAO**(*device*, *channel*, *range=None*)
Returns the analog output channel as specified for a given device

pymeasure.instruments.comedi.**readAI**(*device*, *channel*, *range=None*, *count=1*)
Reads a single measurement (count==1) from the analog input channel of the device specified. Multiple readings can be preformed with count not equal to one, which are seperated by an arbitrary time

pymeasure.instruments.comedi.**writeAO**(*device*, *channel*, *voltage*, *range=None*)
Writes a single voltage to the analog output channel of the device specified

7.4 Resource Manager

The list_resources function provides an interface to check connected instruments interactively.

pymeasure.instruments.**list_resources**()
Prints the available resources, and returns a list of VISA resource names

```
resources = list_resources()
#prints (e.g.)
    #0 : GPIB0::22::INSTR : Agilent Technologies,34410A,******
    #1 : GPIB0::26::INSTR : Keithley Instruments Inc., Model 2612, *****
dmm = Agilent34410(resources[0])
```

Instruments by manufacturer:

7.5 Active Technologies

This section contains specific documentation on the Active Technologies instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.5.1 Active Technologies AWG-401x 1.2GS/s Arbitrary Waveform Generator

class pymeasure.instruments.activetechnologies.**AWG401x_AFG**(*adapter*, ***kwargs*)

Bases: pymeasure.instruments.activetechnologies.AWG401x.AWG401x_base

Represents the Active Technologies AWG-401x Arbitrary Waveform Generator in AFG mode.

```
wfg = AWG401x_AFG("TCPIP::192.168.0.123::INSTR")

wfg.reset()                     # Reset the instrument at default state

wfg.ch[1].shape = "SINUSOID"    # Sets a sine waveform on CH1
wfg.ch[1].frequency = 4.7e3     # Sets the frequency to 4.7 kHz on CH1
wfg.ch[1].amplitude = 1         # Set amplitude of 1 V on CH1
wfg.ch[1].offset = 0            # Set the amplitude to 0 V on CH1
wfg.ch[1].enabled = True        # Enables the CH1

wfg.ch[2].shape = "SQUARE"      # Sets a square waveform on CH2
wfg.ch[2].frequency = 100e6     # Sets the frequency to 100 MHz on CH2
wfg.ch[2].amplitude = 0.5         # Set amplitude of 0.5 V on CH2
wfg.ch[2].offset = 0            # Set the amplitude to 0 V on CH2
wfg.ch[2].enabled = True        # Enables the CH2

wfg.enabled = True              # Enable output of waveform generator
wfg.beep()                      # "beep"

print(wfg.check_errors())       # Get the error queue
```

property enabled

A boolean property that enables the generation of signals.

class pymeasure.instruments.activetechnologies.**AWG401x_AWG**(*adapter*, ***kwargs*)

Bases: pymeasure.instruments.activetechnologies.AWG401x.AWG401x_base

Represents the Active Technologies AWG-401x Arbitrary Waveform Generator in AWG mode.

```
wfg = AWG401x_AWG("TCPIP::192.168.0.123::INSTR")

wfg.reset()             # Reset the instrument at default state

# Set a oscillating waveform
wfg.waveforms["MyWaveform"] = [1, 0] * 8

for i in range(1, wfg.num_ch + 1):
    wfg.entries[1].ch[i].voltage_high = 1       # Sets high voltage = 1
```

(continues on next page)

(continued from previous page)

```
    wfg.entries[1].ch[i].voltage_low = 0        # Sets low voltage = 1
    wfg.entries[1].ch[i].waveform = "SQUARE"    # Sets a square wave
    wfg.setting_ch[i].enabled = True            # Enable channel

wfg.entries.resize(2)           # Resize the number of entries to 2

wfg.entries[2].ch[1].waveform = "MyWaveform"   # Set custom waveform

wfg.enabled = True              # Enable output of waveform generator
wfg.beep()                      # "beep"

print(wfg.check_errors())       # Get the error queue
```

class DummyEntriesElements(*parent*, *number_of_channel*)

Bases: collections.abc.Sequence

Dummy List Class to list every sequencer entry. The content is loaded in real-time.

class WaveformsLazyDict(*parent*)

Bases: collections.abc.MutableMapping

This class inherit from MutableMapping in order to create a custom dict to lazy load, modify, delete and create instrument waveform.

reset()

Reset the class reloading the waveforms from instrument

property burst_count

This property sets or queries the burst count parameter.(dynamic)

property burst_count_max

This property queries the maximum burst count parameter.

property burst_count_min

This property queries the minimum burst count parameter.

property enabled

A boolean property that enables the generation of signals.

property entry_level_strategy

This property sets or or returns the Entry Length Strategy. This strategy manages the length of the sequencer entries in relationship with the length of the channel waveforms defined for each entry. The possible values are:

- ADAPTL<ONGER>: the length of an entry of the sequencer by default will be equal to the length of the longer channel waveform, among all analog channels, assigned to the entry.
- ADAPTS<HORTER>: the length of an entry of the sequencer by default will be equal to the length of the shorter channel waveform, among all analog channels, assigned to the entry.
- DEF<AULT>:the length of an entry of the sequencer by default will be equal to the value specified in the Sequencer Item Default Length [N] parameter

list_files(*path=None*)

Return a List of tuples with all file found in a directory. If the path is not specified the current directory will be used

property num_ch

This property queries the number of analog channels.

property num_dch
This property queries the number of digital channels.

remove_file(*file_name*, *path=None*)
Remove a specified file

property run_mode
This property sets or returns the AWG run mode. The possible values are:

- CONT<INUOUS>: each waveform will loop as written in the entry repetition parameter and the entire sequence is repeated circularly
- BURS<T>: the AWG waits for a trigger event. When the trigger event occurs each waveform will loop as written in the entry repetition parameter and the entire sequence will be repeated circularly many times as written in the Burst Count[N] parameter. If you set Burst Count[N]=1 the instrument is in Single mode and the sequence will be repeated only once.
- TCON<TINUOUS>: the AWG waits for a trigger event. When the trigger event occurs each waveform will loop as written in the entry repetition parameter and the entire sequence will be repeated circularly.
- STEP<PED>: the AWG, for each entry, waits for a trigger event before the execution of the sequencer entry. The waveform of the entry will loop as written in the entry repetition parameter. After the generation of an entry has completed, the last sample of the current entry or the first sample of the next entry is held until the next trigger is received. At the end of the entire sequence the execution will restart from the first entry.
- ADVA<NCED>: it enables the "Advanced" mode. In this mode the execution of the sequence can be changed by using conditional and unconditional jumps (JUMPTO and GOTO commands) and dynamic jumps (PATTERN JUMP commands).

The *RST command sets this parameter to CONTinuous.

property run_status
This property returns the run state of the AWG. The possible values are: STOPPED, WAITING_TRIGGER, RUNNING

property sample_decreasing_strategy
This property sets or returns the Sample Decreasing Strategy. The "Sample decreasing strategy" parameter defines the strategy used to adapt the waveform length to the sequencer entry length in the case where the original waveform length is longer than the sequencer entry length. Can be set to: DECIM<ATION>, CUTT<AIL>, CUTH<EAD>

property sample_increasing_strategy
This property sets or or returns the Sample Increasing Strategy. The "Sample increasing strategy" parameter defines the strategy used to adapt the waveform length to the sequencer entry length in the case where the original waveform length is shorter than the sequencer entry length. Can be set to: INTER<POLATION>, RETURN<ZERO>, HOLD<LAST>, SAMPLESM<ULTIPLICATION>

property sampling_rate
This property sets or queries the sample rate for the Sampling Clock.(dynamic)

property sampling_rate_max
This property queries the maximum sample rate for the Sampling Clock.

property sampling_rate_min
This property queries the minimum sample rate for the Sampling Clock.

save_file(*file_name*, *data*, *path=None*, *override_existing=False*)
Write a string in a file in the instrument

trigger()
Force a trigger event to occour.

property trigger_source
This property sets or returns the instrument trigger source. The possible values are:

- TIM<ER>: the trigger is sent at regular intervals.
- EXT<ERNAL>: the trigger come from the external BNC connector.
- MAN<UAL>: the trigger is sent via software or using the trigger button on front panel.

property waveforms
This property returns a dict with all the waveform present in the instrument system (Wave. List). It is possible to modify the values, delete them or create new waveforms

7.6 Advantest

This section contains specific documentation on the Advantest instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.6.1 Advantest R3767CG Vector Network Analyzer

class pymeasure.instruments.advantest.advantestR3767CG.**AdvantestR3767CG**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Advantest R3767CG VNA. Implements controls to change the analysis range and to retreve the data for the trace.

property center_frequency
Center Frequency in Hz

property id
Reads the instrument identification

property span_frequency
Span Frequency in Hz

property start_frequency
Starting frequency in Hz

property stop_frequency
Stoping frequency in Hz

property trace_1
Reads the Data array from trace 1 after formatting

7.7 Agilent

This section contains specific documentation on the Agilent instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.7.1 Agilent 8257D Signal Generator

class pymeasure.instruments.agilent.**Agilent8257D**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Agilent 8257D Signal Generator and provides a high-level interface for interacting with the instrument.

```
generator = Agilent8257D("GPIB::1")

generator.power = 0                     # Sets the output power to 0 dBm
generator.frequency = 5                 # Sets the output frequency to 5 GHz
generator.enable()                      # Enables the output
```

property amplitude_depth

A floating point property that controls the amplitude modulation in precent, which can take values from 0 to 100 %.

property amplitude_source

A string property that controls the source of the amplitude modulation signal, which can take the values: 'internal', 'internal 2', 'external', and 'external 2'.

property center_frequency

A floating point property that represents the center frequency in Hz. This property can be set.

config_amplitude_modulation(*frequency=1000.0*, *depth=100.0*, *shape='sine'*)

Configures the amplitude modulation of the output signal.

Parameters

- **frequency** – A modulation frequency for the internal oscillator
- **depth** – A linear depth precentage
- **shape** – A string that describes the shape for the internal oscillator

config_low_freq_out(*source='internal'*, *amplitude=3*)

Configures the low-frequency output signal.

Parameters

- **source** – The source for the low-frequency output signal.
- **amplitude** – Amplitude of the low-frequency output

config_pulse_modulation(*frequency=1000.0*, *input='square'*)

Configures the pulse modulation of the output signal.

Parameters

- **frequency** – A pulse rate frequency in Hertz
- **input** – A string that describes the internal pulse input

config_step_sweep()

Configures a step sweep through frequency

disable()

Disables the output of the signal.

disable_amplitude_modulation()

Disables amplitude modulation of the output signal.

disable_low_freq_out()
Disables low frequency output

disable_modulation()
Disables the signal modulation.

disable_pulse_modulation()
Disables pulse modulation of the output signal.

property dwell_time
A floating point property that represents the settling time in seconds at the current frequency or power setting. This property can be set.

enable()
Enables the output of the signal.

enable_amplitude_modulation()
Enables amplitude modulation of the output signal.

enable_low_freq_out()
Enables low frequency output

enable_pulse_modulation()
Enables pulse modulation of the output signal.

property frequency
A floating point property that represents the output frequency in Hz. This property can be set.

property has_amplitude_modulation
Reads a boolean value that is True if the amplitude modulation is enabled.

property has_modulation
Reads a boolean value that is True if the modulation is enabled.

property has_pulse_modulation
Reads a boolean value that is True if the pulse modulation is enabled.

property internal_frequency
A floating point property that controls the frequency of the internal oscillator in Hertz, which can take values from 0.5 Hz to 1 MHz.

property internal_shape
A string property that controls the shape of the internal oscillations, which can take the values: 'sine', 'triangle', 'square', 'ramp', 'noise', 'dual-sine', and 'swept-sine'.

property is_enabled
Reads a boolean value that is True if the output is on.

property low_freq_out_amplitude
A floating point property that controls the peak voltage (amplitude) of the low frequency output in volts, which can take values from 0-3.5V

property low_freq_out_source
A string property which controls the source of the low frequency output, which can take the values 'internal [2]' for the inernal source, or 'function [2]' for an internal function generator which can be configured.

property power
A floating point property that represents the output power in dBm. This property can be set.

property pulse_frequency
A floating point property that controls the pulse rate frequency in Hertz, which can take values from 0.1 Hz to 10 MHz.

property pulse_input
A string property that controls the internally generated modulation input for the pulse modulation, which can take the values: 'square', 'free-run', 'triggered', 'doublet', and 'gated'.

property pulse_source
A string property that controls the source of the pulse modulation signal, which can take the values: 'internal', 'external', and 'scalar'.

shutdown()
Shuts down the instrument by disabling any modulation and the output signal.

property start_frequency
A floating point property that represents the start frequency in Hz. This property can be set.

property start_power
A floating point property that represents the start power in dBm. This property can be set.

start_step_sweep()
Starts a step sweep.

property step_points
An integer number of points in a step sweep. This property can be set.

property stop_frequency
A floating point property that represents the stop frequency in Hz. This property can be set.

property stop_power
A floating point property that represents the stop power in dBm. This property can be set.

stop_step_sweep()
Stops a step sweep.

7.7.2 Agilent 8722ES Vector Network Analyzer

class pymeasure.instruments.agilent.**Agilent8722ES**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Agilent8722ES Vector Network Analyzer and provides a high-level interface for taking scans of the scattering parameters.

property averages
An integer representing the number of averages to take. Note that averaging must be enabled for this to take effect. This property can be set.

property averaging_enabled
A bool that indicates whether or not averaging is enabled. This property can be set.

property data
Returns the real and imaginary data from the last scan

property data_complex
Returns the complex power from the last scan

property data_log_magnitude
Returns the absolute magnitude values in dB from the last scan

property data_magnitude
Returns the absolute magnitude values from the last scan

property data_phase
Returns the phase in degrees from the last scan

disable_averaging()
Disables averaging

enable_averaging()
Enables averaging

property frequencies
Returns a list of frequencies from the last scan

is_averaging()
Returns True if averaging is enabled

log_magnitude(*real*, *imaginary*)
Returns the magnitude in dB from a real and imaginary number or numpy arrays

magnitude(*real*, *imaginary*)
Returns the magnitude from a real and imaginary number or numpy arrays

phase(*real*, *imaginary*)
Returns the phase in degrees from a real and imaginary number or numpy arrays

scan(*averages=None*, *blocking=None*, *timeout=None*, *delay=None*)
Initiates a scan with the number of averages specified and blocks until the operation is complete.

scan_continuous()
Initiates a continuous scan

property scan_points
Gets the number of scan points

scan_single()
Initiates a single scan

set_IF_bandwidth(*bandwidth*)
Sets the resolution bandwidth (IF bandwidth)

set_averaging(*averages*)
Sets the number of averages and enables/disables averaging. Should be between 1 and 999

set_fixed_frequency(*frequency*)
Sets the scan to be of only one frequency in Hz

property start_frequency
A floating point property that represents the start frequency in Hz. This property can be set.

property stop_frequency
A floating point property that represents the stop frequency in Hz. This property can be set.

property sweep_time
A floating point property that represents the sweep time in seconds. This property can be set.

7.7.3 Agilent E4408B Spectrum Analyzer

class pymeasure.instruments.agilent.**AgilentE4408B**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the AgilentE4408B Spectrum Analyzer and provides a high-level interface for taking scans of high-frequency spectrums

property center_frequency
A floating point property that represents the center frequency in Hz. This property can be set.

property frequencies
Returns a numpy array of frequencies in Hz that correspond to the current settings of the instrument.

property frequency_points
An integer property that represents the number of frequency points in the sweep. This property can take values from 101 to 8192.

property frequency_step
A floating point property that represents the frequency step in Hz. This property can be set.

property start_frequency
A floating point property that represents the start frequency in Hz. This property can be set.

property stop_frequency
A floating point property that represents the stop frequency in Hz. This property can be set.

property sweep_time
A floating point property that represents the sweep time in seconds. This property can be set.

trace(*number=1*)
Returns a numpy array of the data for a particular trace based on the trace number (1, 2, or 3).

trace_df(*number=1*)
Returns a pandas DataFrame containing the frequency and peak data for a particular trace, based on the trace number (1, 2, or 3).

7.7.4 Agilent E4980 LCR Meter

class pymeasure.instruments.agilent.**AgilentE4980**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents LCR meter E4980A/AL

property ac_current
AC current level, in Amps

property ac_voltage
AC voltage level, in Volts

aperture(*time=None*, *averages=1*)
Set and get aperture.

Parameters

- **time** – integration time as string: SHORT, MED, LONG (case insensitive); if None, get values
- **averages** – number of averages, numeric

freq_sweep(*freq_list*, *return_freq=False*)
Run frequency list sweep using sequential trigger.

Parameters

- **freq_list** – list of frequencies
- **return_freq** – if True, returns the frequencies read from the instrument

Returns values as configured with *mode*

property frequency
AC frequency (range depending on model), in Hertz

property impedance
Measured data A and B, according to *mode*

property mode
Select quantities to be measured:

- CPD: Parallel capacitance [F] and dissipation factor [number]
- CPQ: Parallel capacitance [F] and quality factor [number]
- CPG: Parallel capacitance [F] and parallel conductance [S]
- CPRP: Parallel capacitance [F] and parallel resistance [Ohm]
- CSD: Series capacitance [F] and dissipation factor [number]
- CSQ: Series capacitance [F] and quality factor [number]
- CSRS: Series capacitance [F] and series resistance [Ohm]
- LPD: Parallel inductance [H] and dissipation factor [number]
- LPQ: Parallel inductance [H] and quality factor [number]
- LPG: Parallel inductance [H] and parallel conductance [S]
- LPRP: Parallel inductance [H] and parallel resistance [Ohm]
- LSD: Series inductance [H] and dissipation factor [number]
- LSQ: Seriesinductance [H] and quality factor [number]
- LSRS: Series inductance [H] and series resistance [Ohm]
- RX: Resitance [Ohm] and reactance [Ohm]
- ZTD: Impedance, magnitude [Ohm] and phase [deg]
- ZTR: Impedance, magnitude [Ohm] and phase [rad]
- GB: Conductance [S] and susceptance [S]
- YTD: Admittance, magnitude [Ohm] and phase [deg]
- YTR: Admittance magnitude [Ohm] and phase [rad]

property trigger_source

Select trigger source; accept the values:

- HOLD: manual
- INT: internal
- BUS: external bus (GPIB/LAN/USB)
- EXT: external connector

7.7.5 Agilent 34410A Multimeter

class pymeasure.instruments.agilent.**Agilent34410A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represent the HP/Agilent/Keysight 34410A and related multimeters.

Implemented measurements: voltage_dc, voltage_ac, current_dc, current_ac, resistance, resistance_4w

property current_ac
AC current, in Amps

property current_dc
DC current, in Amps

property resistance
Resistance, in Ohms

property resistance_4w
Four-wires (remote sensing) resistance, in Ohms

property voltage_ac
AC voltage, in Volts

property voltage_dc
DC voltage, in Volts

7.7.6 HP/Agilent/Keysight 34450A Digital Multimeter

class pymeasure.instruments.agilent.**Agilent34450A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represent the HP/Agilent/Keysight 34450A and related multimeters.

```
dmm = Agilent34450A("USB0::...")
dmm.reset()
dmm.configure_voltage()
print(dmm.voltage)
dmm.shutdown()
```

beep()
Sounds a system beep.

property capacitance
Reads a capacitance measurement in Farads, based on the active `mode`.

property capacitance_auto_range
A boolean property that toggles auto ranging for capacitance.

property capacitance_range
A property that controls the capacitance range in Farads, which can take values 1E-9, 10E-9, 100E-9, 1E-6, 10E-6, 100E-6, 1E-3, 10E-3, as well as "MIN", "MAX", or "DEF" (1E-6). Auto-range is disabled when this property is set.

configure_capacitance(*capacitance_range='AUTO'*)
Configures the instrument to measure capacitance.

Parameters **capacitance_range** – A capacitance in Farads to set the capacitance range, can be 1E-9, 10E-9, 100E-9, 1E-6, 10E-6, 100E-6, 1E-3, 10E-3, as well as "MIN", "MAX", "DEF" (1E-6), or "AUTO".

configure_continuity()
Configures the instrument to measure continuity.

configure_current(*current_range='AUTO'*, *ac=False*, *resolution='DEF'*)
Configures the instrument to measure current.

Parameters

- **current_range** – A current in Amps to set the current range. DC values can be 100E-6, 1E-3, 10E-3, 100E-3, 1, 10, as well as "MIN", "MAX", "DEF" (100 mA), or "AUTO". AC values can be 10E-3, 100E-3, 1, 10, as well as "MIN", "MAX", "DEF" (100 mA), or "AUTO".
- **ac** – False for DC current, and True for AC current
- **resolution** – Desired resolution, can be 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

configure_diode()
Configures the instrument to measure diode voltage.

configure_frequency(*measured_from='voltage_ac'*, *measured_from_range='AUTO'*, *aperture='DEF'*)
Configures the instrument to measure frequency.

Parameters

- **measured_from** – "voltage_ac" or "current_ac"
- **measured_from_range** – range of measured_from. AC voltage can have ranges 100E-3, 1, 10, 100, 750, as well as "MIN", "MAX", "DEF" (10 V), or "AUTO". AC current can have ranges 10E-3, 100E-3, 1, 10, as well as "MIN", "MAX", "DEF" (100 mA), or "AUTO".
- **aperture** – Aperture time in Seconds, can be 100 ms, 1 s, as well as "MIN", "MAX", or "DEF" (1 s).

configure_resistance(*resistance_range='AUTO'*, *wires=2*, *resolution='DEF'*)
Configures the instrument to measure resistance.

Parameters

- **resistance_range** – A resistance in Ohms to set the resistance range, can be 100, 1E3, 10E3, 100E3, 1E6, 10E6, 100E6, as well as "MIN", "MAX", "DEF" (1E3), or "AUTO".
- **wires** – Number of wires used for measurement, can be 2 or 4.
- **resolution** – Desired resolution, can be 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

configure_temperature()
Configures the instrument to measure temperature.

configure_voltage(*voltage_range='AUTO'*, *ac=False*, *resolution='DEF'*)
Configures the instrument to measure voltage.

Parameters

- **voltage_range** – A voltage in Volts to set the voltage range. DC values can be 100E-3, 1, 10, 100, 1000, as well as "MIN", "MAX", "DEF" (10 V), or "AUTO". AC values can be 100E-3, 1, 10, 100, 750, as well as "MIN", "MAX", "DEF" (10 V), or "AUTO".
- **ac** – False for DC voltage, True for AC voltage
- **resolution** – Desired resolution, can be 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

property continuity
Reads a continuity measurement in Ohms, based on the active mode.

property current
Reads a DC current measurement in Amps, based on the active mode.

property current_ac
Reads an AC current measurement in Amps, based on the active mode.

property current_ac_auto_range
A boolean property that toggles auto ranging for AC current.

property current_ac_range
A property that controls the AC current range in Amps, which can take values 10E-3, 100E-3, 1, 10, as well as "MIN", "MAX", or "DEF" (100 mA). Auto-range is disabled when this property is set.

property current_ac_resolution
An property that controls the resolution in the AC current readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

property current_auto_range
A boolean property that toggles auto ranging for DC current.

property current_range
A property that controls the DC current range in Amps, which can take values 100E-6, 1E-3, 10E-3, 100E-3, 1, 10, as well as "MIN", "MAX", or "DEF" (100 mA). Auto-range is disabled when this property is set.

property current_resolution
A property that controls the resolution in the DC current readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", and "DEF" (3.00E-5).

property diode
Reads a diode measurement in Volts, based on the active mode.

property frequency
Reads a frequency measurement in Hz, based on the active mode.

property frequency_aperture
A property that controls the frequency aperture in seconds, which sets the integration period and measurement speed. Takes values 100 ms, 1 s, as well as "MIN", "MAX", or "DEF" (1 s).

property frequency_current_auto_range
Boolean property that toggles auto ranging for AC current in frequency measurements.

property frequency_current_range
A property that controls the current range in Amps for frequency on AC current measurements, which can take values 10E-3, 100E-3, 1, 10, as well as "MIN", "MAX", or "DEF" (100 mA). Auto-range is disabled when this property is set.

property frequency_voltage_auto_range
Boolean property that toggles auto ranging for AC voltage in frequency measurements.

property frequency_voltage_range
A property that controls the voltage range in Volts for frequency on AC voltage measurements, which can take values 100E-3, 1, 10, 100, 750, as well as "MIN", "MAX", or "DEF" (10 V). Auto-range is disabled when this property is set.

property resistance
Reads a resistance measurement in Ohms for 2-wire configuration, based on the active mode.

property resistance_4w
Reads a resistance measurement in Ohms for 4-wire configuration, based on the active `mode`.

property resistance_4w_auto_range
A boolean property that toggles auto ranging for 4-wire resistance.

property resistance_4w_range
A property that controls the 4-wire resistance range in Ohms, which can take values 100, 1E3, 10E3, 100E3, 1E6, 10E6, 100E6, as well as "MIN", "MAX", or "DEF" (1E3). Auto-range is disabled when this property is set.

property resistance_4w_resolution
A property that controls the resolution in the 4-wire resistance readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

property resistance_auto_range
A boolean property that toggles auto ranging for 2-wire resistance.

property resistance_range
A property that controls the 2-wire resistance range in Ohms, which can take values 100, 1E3, 10E3, 100E3, 1E6, 10E6, 100E6, as well as "MIN", "MAX", or "DEF" (1E3). Auto-range is disabled when this property is set.

property resistance_resolution
A property that controls the resolution in the 2-wire resistance readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

property temperature
Reads a temperature measurement in Celsius, based on the active `mode`.

property voltage
Reads a DC voltage measurement in Volts, based on the active `mode`.

property voltage_ac
Reads an AC voltage measurement in Volts, based on the active `mode`.

property voltage_ac_auto_range
A boolean property that toggles auto ranging for AC voltage.

property voltage_ac_range
A property that controls the AC voltage range in Volts, which can take values 100E-3, 1, 10, 100, 750, as well as "MIN", "MAX", or "DEF" (10 V). Auto-range is disabled when this property is set.

property voltage_ac_resolution
A property that controls the resolution in the AC voltage readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

property voltage_auto_range
A boolean property that toggles auto ranging for DC voltage.

property voltage_range
A property that controls the DC voltage range in Volts, which can take values 100E-3, 1, 10, 100, 1000, as well as "MIN", "MAX", or "DEF" (10 V). Auto-range is disabled when this property is set.

property voltage_resolution
A property that controls the resolution in the DC voltage readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as "MIN", "MAX", or "DEF" (1.50E-6).

7.7.7 Agilent 4155/4156 Semiconductor Parameter Analyzer

class pymeasure.instruments.agilent.agilent4156.**Agilent4156**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Agilent 4155/4156 Semiconductor Parameter Analyzer and provides a high-level interface for taking current-voltage (I-V) measurements.

```
from pymeasure.instruments.agilent import Agilent4156

# explicitly define r/w terminations; set sufficiently large timeout or None.
smu = Agilent4156("GPIB0::25", read_termination = '\n', write_termination = '\n',
                  timeout=None)

# reset the instrument
smu.reset()

# define configuration file for instrument and load config
smu.configure("configuration_file.json")

# save data variables, some or all of which are defined in the json config file.
smu.save(['VC', 'IC', 'VB', 'IB'])

# take measurements
status = smu.measure()

# measured data is a pandas dataframe and can be exported to csv.
data = smu.get_data(path='./t1.csv')
```

The JSON file is an ascii text configuration file that defines the settings of each channel on the instrument. The JSON file is used to configure the instrument using the convenience function *configure()* as shown in the example above. For example, the instrument setup for a bipolar transistor measurement is shown below.

```
{
       "SMU1": {
           "voltage_name" : "VC",
           "current_name" : "IC",
           "channel_function" : "VAR1",
           "channel_mode" : "V",
           "series_resistance" : "0OHM"
       },

       "SMU2": {
           "voltage_name" : "VB",
           "current_name" : "IB",
           "channel_function" : "VAR2",
           "channel_mode" : "I",
           "series_resistance" : "0OHM"
       },

       "SMU3": {
           "voltage_name" : "VE",
           "current_name" : "IE",
           "channel_function" : "CONS",
```

(continues on next page)

(continued from previous page)

```
            "channel_mode" : "V",
            "constant_value" : 0,
            "compliance" : 0.1
        },

         "SMU4": {
            "voltage_name" : "VS",
            "current_name" : "IS",
            "channel_function" : "CONS",
            "channel_mode" : "V",
            "constant_value" : 0,
            "compliance" : 0.1
        },

        "VAR1": {
            "start" : 1,
            "stop" : 2,
            "step" : 0.1,
            "spacing" : "LINEAR",
            "compliance" : 0.1
        },

        "VAR2": {
            "start" : 0,
            "step" : 10e-6,
            "points" : 3,
            "compliance" : 2

        }
    }
```

property analyzer_mode

A string property that controls the instrument operating mode.

- Values: SWEEP, SAMPLING

```
smu.analyzer_mode = "SWEEP"
```

configure(*config_file*)

Configure the channel setup and sweep using a JSON configuration file.

(JSON is the JavaScript Object Notation)

Parameters **config_file** – JSON file to configure instrument channels.

```
instr.configure('config.json')
```

property data_variables

Get a string list of data variables for which measured data is available.

This looks for all the variables saved by the *save()* and *save_var()* methods and returns it. This is useful for creation of dataframe headers.

Returns List

```
header = instr.data_variables
```

property delay_time
A floating point property that measurement delay time in seconds, which can take the values from 0 to 65s in 0.1s steps.

```
instr.delay_time = 1 # delay time of 1-sec
```

disable_all()
Disables all channels in the instrument.

```
instr.disable_all()
```

get_data(*path=None*)
Get the measurement data from the instrument after completion.

If the measurement period is set to INF in the *measure()* method, then the measurement must be stopped using *stop()* before getting valid data.

Parameters **path** – Path for optional data export to CSV.

Returns Pandas Dataframe

```
df = instr.get_data(path='./datafolder/data1.csv')
```

property hold_time
A floating point property that measurement hold time in seconds, which can take the values from 0 to 655s in 1s steps.

```
instr.hold_time = 2 # hold time of 2-secs.
```

property integration_time
A string property that controls the integration time.

- Values: SHORT, MEDIUM, LONG

```
instr.integration_time = "MEDIUM"
```

measure(*period='INF'*, *points=100*)
Performs a single measurement and waits for completion in sweep mode. In sampling mode, the measurement period and number of points can be specified.

Parameters

- **period** – Period of sampling measurement from 6E-6 to 1E11 seconds. Default setting is INF.
- **points** – Number of samples to be measured, from 1 to 10001. Default setting is 100.

save(*trace_list*)
Save the voltage or current in the instrument display list

Parameters **trace_list** – A list of channel variables whose measured data should be saved. A maximum of 8 variables are allowed. If only one variable is being saved, a string can be specified.

```
instr.save(['IC', 'IB', 'VC', 'VB']) #for list of variables
instr.save('IC')    #for single variable
```

save_var(*trace_list*)
Save the voltage or current in the instrument variable list.

This is useful if one or two more variables need to be saved in addition to the 8 variables allowed by *save()*.

Parameters **trace_list** – A list of channel variables whose measured data should be saved. A maximum of 2 variables are allowed. If only one variable is being saved, a string can be specified.

```
instr.save_var(['VA', 'VB'])
```

stop()
Stops the ongoing measurement

```
instr.stop()
```

class pymeasure.instruments.agilent.agilent4156.**SMU**(*adapter*, *channel*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

property channel_function
A string property that controls the SMU<n> channel function.

- Values: VAR1, VAR2, VARD or CONS.

```
instr.smu1.channel_function = "VAR1"
```

property channel_mode
A string property that controls the SMU<n> channel mode.

- Values: V, I or COMM

VPULSE AND IPULSE are not yet supported.

```
instr.smu1.channel_mode = "V"
```

property compliance
Sets the *constant* compliance value of SMU<n>.

If the SMU channel is setup as a variable (VAR1, VAR2, VARD) then compliance limits are set by the variable definition.

- Value: Voltage in (-200V, 200V) and current in (-1A, 1A) based on *channel_mode()*.

```
instr.smu1.compliance = 0.1
```

property constant_value
Set the constant source value of SMU<n>.

You use this command only if *channel_function()* is CONS and also *channel_mode()* should not be COMM.

Parameters **const_value** – Voltage in (-200V, 200V) and current in (-1A, 1A). Voltage or current depends on if *channel_mode()* is set to V or I.

```
instr.smu1.constant_value = 1
```

property current_name
Define the current name of the channel.

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.smu1.current_name = "Ibase"
```

property disable
Deletes the settings of SMU<n>.

```
instr.smu1.disable()
```

property series_resistance
Controls the series resistance of SMU<n>.

- Values: `0OHM`, `10KOHM`, `100KOHM`, or `1MOHM`

```
instr.smu1.series_resistance = "10KOHM"
```

property voltage_name
Define the voltage name of the channel.

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.smu1.voltage_name = "Vbase"
```

class pymeasure.instruments.agilent.agilent4156.**VAR1**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.agilent.agilent4156.VARX*

Class to handle all the specific definitions needed for VAR1. Most common methods are inherited from base class.

property spacing
Selects the sweep type of VAR1.

- Values: `LINEAR`, `LOG10`, `LOG25`, `LOG50`.

class pymeasure.instruments.agilent.agilent4156.**VAR2**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.agilent.agilent4156.VARX*

Class to handle all the specific definitions needed for VAR2. Common methods are imported from base class.

property points
Sets the number of sweep steps of VAR2. You use this command only if there is an SMU or VSU whose function (FCTN) is VAR2.

```
instr.var2.points = 10
```

class pymeasure.instruments.agilent.agilent4156.**VARD**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Class to handle all the definitions needed for VARD. VARD is always defined in relation to VAR1.

property compliance
Sets the sweep COMPLIANCE value of VARD.

```
instr.vard.compliance = 0.1
```

property offset
Sets the OFFSET value of VARD. For each step of sweep, the output values of VAR1' are determined by the following equation: VARD = VAR1 X RATio + OFFSet You use this command only if there is an SMU or VSU whose function is VARD.

```
instr.vard.offset = 1
```

property ratio
Sets the RATIO of VAR1'. For each step of sweep, the output values of VAR1' are determined by the following equation: VAR1' = VAR1 * RATio + OFFSet You use this command only if there is an SMU or VSU whose function (FCTN) is VAR1'.

```
instr.vard.ratio = 1
```

class pymeasure.instruments.agilent.agilent4156.**VARX**(*adapter*, *var_name*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Base class to define sweep variable settings

property compliance
Sets the sweep COMPLIANCE value.

```
instr.var1.compliance = 0.1
```

property start
Sets the sweep START value.

```
instr.var1.start = 0
```

property step
Sets the sweep STEP value.

```
instr.var1.step = 0.1
```

property stop
Sets the sweep STOP value.

```
instr.var1.stop = 3
```

class pymeasure.instruments.agilent.agilent4156.**VMU**(*adapter*, *channel*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

property channel_mode
A string property that controls the VMU<n> channel mode.

- Values: `V`, `DVOL`

property disable
Disables the settings of VMU<n>.

```
instr.vmu1.disable()
```

property voltage_name
Define the voltage name of the VMU channel.

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.vmu1.voltage_name = "Vanode"
```

class pymeasure.instruments.agilent.agilent4156.**VSU**(*adapter*, *channel*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

property channel_function
A string property that controls the VSU channel function.

- Value: VAR1, VAR2, VARD or CONS.

property channel_mode
Get channel mode of VSU<n>.

property constant_value
Sets the constant source value of VSU<n>.

```
instr.vsu1.constant_value = 0
```

property disable
Deletes the settings of VSU<n>.

```
instr.vsu1.disable()
```

property voltage_name
Define the voltage name of the VSU channel

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.vsu1.voltage_name = "Ve"
```

7.7.8 Agilent 33220A Arbitrary Waveform Generator

class pymeasure.instruments.agilent.**Agilent33220A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Agilent 33220A Arbitrary Waveform Generator.

```
# Default channel for the Agilent 33220A
wfg = Agilent33220A("GPIB::10")

wfg.shape = "SINUSOID"          # Sets a sine waveform
wfg.frequency = 4.7e3           # Sets the frequency to 4.7 kHz
wfg.amplitude = 1               # Set amplitude of 1 V
wfg.offset = 0                  # Set the amplitude to 0 V

wfg.burst_state = True          # Enable burst mode
wfg.burst_ncycles = 10          # A burst will consist of 10 cycles
wfg.burst_mode = "TRIGGERED"    # A burst will be applied on a trigger
wfg.trigger_source = "BUS"      # A burst will be triggered on TRG*

wfg.output = True               # Enable output of waveform generator
wfg.trigger()                   # Trigger a burst
wfg.wait_for_trigger()          # Wait until the triggering is finished
wfg.beep()                      # "beep"

print(wfg.check_errors())       # Get the error queue
```

property amplitude
A floating point property that controls the voltage amplitude of the output waveform in V, from 10e-3 V to 10 V. Can be set.

property amplitude_unit
A string property that controls the units of the amplitude. Valid values are Vpp (default), Vrms, and dBm. Can be set.

beep()
Causes a system beep.

property beeper_state
A boolean property that controls the state of the beeper. Can be set.

property burst_mode
A string property that controls the burst mode. Valid values are: TRIG<GERED>, GAT<ED>. This setting can be set.

property burst_ncycles
An integer property that sets the number of cycles to be output when a burst is triggered. Valid values are 1 to 50000. This can be set.

property burst_state
A boolean property that controls whether the burst mode is on (True) or off (False). Can be set.

property frequency
A floating point property that controls the frequency of the output waveform in Hz, from 1e-6 (1 uHz) to 20e+6 (20 MHz), depending on the specified function. Can be set.

property offset
A floating point property that controls the voltage offset of the output waveform in V, from 0 V to 4.995 V, depending on the set voltage amplitude (maximum offset = (10 - voltage) / 2). Can be set.

property output
A boolean property that turns on (True) or off (False) the output of the function generator. Can be set.

property pulse_dutycycle
A floating point property that controls the duty cycle of a pulse waveform function in percent. Can be set.

property pulse_hold
A string property that controls if either the pulse width or the duty cycle is retained when changing the period or frequency of the waveform. Can be set to: WIDT<H> or DCYC<LE>.

property pulse_period
A floating point property that controls the period of a pulse waveform function in seconds, ranging from 200 ns to 2000 s. Can be set and overwrites the frequency for *all* waveforms. If the period is shorter than the pulse width + the edge time, the edge time and pulse width will be adjusted accordingly.

property pulse_transition
A floating point property that controls the the edge time in seconds for both the rising and falling edges. It is defined as the time between 0.1 and 0.9 of the threshold. Valid values are between 5 ns to 100 ns. The transition time has to be smaller than 0.625 * the pulse width. Can be set.

property pulse_width
A floating point property that controls the width of a pulse waveform function in seconds, ranging from 20 ns to 2000 s, within a set of restrictions depending on the period. Can be set.

property ramp_symmetry
A floating point property that controls the symmetry percentage for the ramp waveform. Can be set.

property remote_local_state
A string property that controls the remote/local state of the function generator. Valid values are: LOC<AL>, REM<OTE>, RWL<OCK>. This setting can only be set.

property shape
A string property that controls the output waveform. Can be set to: SIN<USOID>, SQU<ARE>, RAMP, PULS<E>, NOIS<E>, DC, USER.

property square_dutycycle
A floating point property that controls the duty cycle of a square waveform function in percent. Can be set.

trigger()
Send a trigger signal to the function generator.

property trigger_source
A string property that controls the trigger source. Valid values are: IMM<EDIATE> (internal), EXT<ERNAL> (rear input), BUS (via trigger command). This setting can be set.

property trigger_state
A boolean property that controls whether the output is triggered (True) or not (False). Can be set.

property voltage_high
A floating point property that controls the upper voltage of the output waveform in V, from -4.990 V to 5 V (must be higher than low voltage). Can be set.

property voltage_low
A floating point property that controls the lower voltage of the output waveform in V, from -5 V to 4.990 V (must be lower than high voltage). Can be set.

wait_for_trigger(*timeout=3600*, *should_stop=<function Agilent33220A.<lambda>>*)
Wait until the triggering has finished or timeout is reached.

Parameters

- **timeout** – The maximum time the waiting is allowed to take. If timeout is exceeded, a TimeoutError is raised. If timeout is set to zero, no timeout will be used.
- **should_stop** – Optional function (returning a bool) to allow the waiting to be stopped before its end.

7.7.9 Agilent 33500 Function/Arbitrary Waveform Generator Family

class pymeasure.instruments.agilent.**Agilent33500**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Agilent 33500 Function/Arbitrary Waveform Generator family.

Individual devices are represented by subclasses. User can specify a channel to control, if no channel specified, a default channel is picked based on the device e.g. For Agilent33500B the default channel is channel 1. See reference manual for your device

```
generator = Agilent33500("GPIB::1")

generator.shape = 'SIN'                 # Sets default channel output signal shape␣
↪to sine
generator.ch[1].shape = 'SIN'           # Sets channel 1 output signal shape to sine
generator.frequency = 1e3               # Sets default channel output frequency to␣
↪1 kHz
generator.ch[1].frequency = 1e3         # Sets channel 1 output frequency to 1 kHz
generator.ch[2].amplitude = 5           # Sets channel 2 output amplitude to 5 Vpp
generator.ch[2].output = 'on'           # Enables channel 2 output
```

(continues on next page)

(continued from previous page)

```
generator.ch[1].shape = 'ARB'           # Set channel 1 shape to arbitrary
generator.ch[1].arb_srate = 1e6         # Set channel 1 sample rate to 1MSa/s

generator.ch[1].data_volatile_clear()   # Clear channel 1 volatile internal memory
generator.ch[1].data_arb(               # Send data of arbitrary waveform to␣
↪channel 1
    'test',
    range(-10000, 10000, +20),          # In this case a simple ramp
    data_format='DAC'                   # Data format is set to 'DAC'
 )
generator.ch[1].arb_file = 'test'       # Select the transmitted waveform 'test'
```

property amplitude
A floating point property that controls the voltage amplitude of the output waveform in V, from 10e-3 V to 10 V. Depends on the output impedance.

property amplitude_unit
A string property that controls the units of the amplitude. Valid values are VPP (default), VRMS, and DBM.

property arb_advance
A string property that selects how the device advances from data point to data point. Can be set to 'TRIG<GER>' or 'SRAT<E>' (default).

property arb_file
A string property that selects the arbitrary signal from the volatile memory of the device. String has to match an existing arb signal in volatile memory (set by *data_arb()*).

property arb_filter
A string property that selects the filter setting for arbitrary signals. Can be set to 'NORM<AL>', 'STEP' and 'OFF'.

property arb_srate
An floating point property that sets the sample rate of the currently selected arbitrary signal. Valid values are 1 µSa/s to 250 MSa/s (maximum range, can be lower depending on your device).

beep()
Causes a system beep.

property burst_mode
A string property that controls the burst mode. Valid values are: TRIG<GERED>, GAT<ED>.

property burst_ncycles
An integer property that sets the number of cycles to be output when a burst is triggered. Valid values are 1 to 100000. This can be set.

property burst_period
A floating point property that controls the period of subsequent bursts. Has to follow the equation burst_period > (burst_ncycles / frequency) + 1 µs. Valid values are 1 µs to 8000 s.

property burst_state
A boolean property that controls whether the burst mode is on (True) or off (False).

clear_display()
Removes a text message from the display.

data_arb(*arb_name*, *data_points*, *data_format='DAC'*)
Uploads an arbitrary trace into the volatile memory of the device.

The data_points can be given as: comma separated 16 bit DAC values (ranging from -32767 to +32767), as comma separated floating point values (ranging from -1.0 to +1.0) or as a binary data stream. Check the manual for more information. The storage depends on the device type and ranges from 8 Sa to 16 MSa (maximum).

Parameters

- **arb_name** – The name of the trace in the volatile memory. This is used to access the trace.
- **data_points** – Individual points of the trace. The format depends on the format parameter. format = 'DAC' (default): Accepts list of integer values ranging from -32767 to +32767. Minimum of 8 a maximum of 65536 points. format = 'float': Accepts list of floating point values ranging from -1.0 to +1.0. Minimum of 8 a maximum of 65536 points. format = 'binary': Accepts a binary stream of 8 bit data.
- **data_format** – Defines the format of data_points. Can be 'DAC' (default), 'float' or 'binary'. See documentation on parameter data_points above.

data_volatile_clear()

Clear all arbitrary signals from volatile memory.

This should be done if the same name is used continuously to load different arbitrary signals into the memory, since an error will occur if a trace is loaded which already exists in the memory.

property display

A string property which is displayed on the front panel of the device.

property ext_trig_out

A boolean property that controls whether the trigger out signal is active (True) or not (False). This signal is output from the Ext Trig connector on the rear panel in Burst and Wobbel mode.

property frequency

A floating point property that controls the frequency of the output waveform in Hz, from 1 uHz to 120 MHz (maximum range, can be lower depending on your device), depending on the specified function.

property offset

A floating point property that controls the voltage offset of the output waveform in V, from 0 V to 4.995 V, depending on the set voltage amplitude (maximum offset = (Vmax - voltage) / 2).

property output

A boolean property that turns on (True, 'on') or off (False, 'off') the output of the function generator.

property output_load

Sets the expected load resistance (should be the load impedance connected to the output. The output impedance is always 50 Ohm, this setting can be used to correct the displayed voltage for loads unmatched to 50 Ohm. Valid values are between 1 and 10 kOhm or INF for high impedance. No validator is used since both numeric and string inputs are accepted, thus a value outside the range will not return an error.

property phase

A floating point property that controls the phase of the output waveform in degrees, from -360 degrees to 360 degrees. Not available for arbitrary waveforms or noise.

property pulse_dutycycle

A floating point property that controls the duty cycle of a pulse waveform function in percent, from 0% to 100%.

property pulse_hold

A string property that controls if either the pulse width or the duty cycle is retained when changing the period or frequency of the waveform. Can be set to: WIDT<H> or DCYC<LE>.

property pulse_period

A floating point property that controls the period of a pulse waveform function in seconds, ranging from

33 ns to 1e6 s. Can be set and overwrites the frequency for *all* waveforms. If the period is shorter than the pulse width + the edge time, the edge time and pulse width will be adjusted accordingly.

property pulse_transition
A floating point property that controls the edge time in seconds for both the rising and falling edges. It is defined as the time between the 10% and 90% thresholds of the edge. Valid values are between 8.4 ns to 1 µs.

property pulse_width
A floating point property that controls the width of a pulse waveform function in seconds, ranging from 16 ns to 1 Ms, within a set of restrictions depending on the period.

property ramp_symmetry
A floating point property that controls the symmetry percentage for the ramp waveform, from 0.0% to 100.0%.

property shape
A string property that controls the output waveform. Can be set to: SIN<USOID>, SQU<ARE>, TRI<ANGLE>, RAMP, PULS<E>, PRBS, NOIS<E>, ARB, DC.

property square_dutycycle
A floating point property that controls the duty cycle of a square waveform function in percent, from 0.01% to 99.98%. The duty cycle is limited by the frequency and the minimal pulse width of 16 ns. See manual for more details.

trigger()
Send a trigger signal to the function generator.

property trigger_source
A string property that controls the trigger source. Valid values are: IMM<EDIATE> (internal), EXT<ERNAL> (rear input), BUS (via trigger command).

property voltage_high
A floating point property that controls the upper voltage of the output waveform in V, from -4.999 V to 5 V (must be higher than low voltage by at least 1 mV).

property voltage_low
A floating point property that controls the lower voltage of the output waveform in V, from -5 V to 4.999 V (must be lower than high voltage by at least 1 mV).

wait_for_trigger(*timeout=3600*, *should_stop=<function Agilent33500.<lambda>>*)
Wait until the triggering has finished or timeout is reached.

Parameters

- **timeout** – The maximum time the waiting is allowed to take. If timeout is exceeded, a TimeoutError is raised. If timeout is set to zero, no timeout will be used.
- **should_stop** – Optional function (returning a bool) to allow the waiting to be stopped before its end.

7.7.10 Agilent 33521A Function/Arbitrary Waveform Generator

class pymeasure.instruments.agilent.**Agilent33521A**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.agilent.agilent33500.Agilent33500*

Represents the Agilent 33521A Function/Arbitrary Waveform Generator.

This documentation page shows only methods different from the parent class *Agilent33500*.

property arb_srate

An floating point property that sets the sample rate of the currently selected arbitrary signal. Valid values are 1 µSa/s to 250 MSa/s. This can be set.

property frequency

A floating point property that controls the frequency of the output waveform in Hz, from 1 uHz to 30 MHz, depending on the specified function. Can be set.

7.7.11 Agilent B1500 Semiconductor Parameter Analyzer

Contents

General Information

This instrument driver does not support all configuration options of the B1500 mainframe yet. So far, it is possible to interface multiple SMU modules and source/measure currents and voltages, perform sampling and staircase sweep measurements. The implementation of further measurement functionalities is highly encouraged. Meanwhile the model is managed by Keysight, see the corresponding "Programming Guide" for details on the control methods and their parameters

Command Translation

Alphabetical list of implemented B1500 commands and their corresponding method/attribute names in this instrument driver.

Command	Property/Method
AAD	*SMU.adc_type()*
AB	*abort()*
AIT	*adc_setup()*
AV	*adc_averaging()*
AZ	*adc_auto_zero*
BC	*clear_buffer()*
CL	*SMU.disable()*
CM	*auto_calibration*
CMM	*SMU.meas_op_mode()*
CN	*SMU.enable()*
DI	*SMU.force()* mode: 'CURRENT'
DV	*SMU.force()* mode: 'VOLTAGE'
DZ	*force_gnd()*, *SMU.force_gnd()*
ERRX?	*check_errors()*
FL	*SMU.filter*
FMT	*data_format()*
*IDN?	id()
*LRN?	*query_learn()*, multiple methods to read/format settings directly
MI	*SMU.sampling_source()* mode: 'CURRENT'
ML	*sampling_mode*
MM	*meas_mode()*
MSC	*sampling_auto_abort()*
MT	*sampling_timing()*
MV	*SMU.sampling_source()* mode: 'VOLTAGE'
*OPC?	*check_idle()*
PA	*pause()*
PAD	*parallel_meas*
RI	*meas_range_current*
RM	*SMU.meas_range_current_auto()*
*RST	*reset()*
RV	*meas_range_voltage*
SSR	*series_resistor*
TSC	*time_stamp*
TSR	*clear_timer()*
UNT?	*query_modules()*
WAT	*wait_time()*
WI	*SMU.staircase_sweep_source()* mode: 'CURRENT'
WM	*sweep_auto_abort()*
WSI	*SMU.synchronous_sweep_source()* mode: 'CURRENT'
WSV	*SMU.synchronous_sweep_source()* mode: 'VOLTAGE'
WT	*sweep_timing()*
WV	*SMU.staircase_sweep_source()* mode: 'VOLTAGE'
XE	*send_trigger()*

Examples

Initialization of the Instrument

```
from pymeasure.instruments.agilent import AgilentB1500

# explicitly define r/w terminations; set sufficiently large timeout in milliseconds or␣
↪None.
b1500=AgilentB1500("GPIB0::17::INSTR", read_termination='\r\n', write_termination='\r\n',
↪ timeout=600000)
# query SMU config from instrument and initialize all SMU instances
b1500.initialize_all_smus()
# set data output format (required!)
b1500.data_format(21, mode=1) #call after SMUs are initialized to get names for the␣
↪channels
```

IV measurement with 4 SMUs

```
# choose measurement mode
b1500.meas_mode('STAIRCASE_SWEEP', *b1500.smu_references) #order in smu_references␣
↪determines order of measurement

# settings for individual SMUs
for smu in b1500.smu_references:
    smu.enable() #enable SMU
    smu.adc_type = 'HRADC' #set ADC to high-resoultion ADC
    smu.meas_range_current = '1 nA'
    smu.meas_op_mode = 'COMPLIANCE_SIDE' # other choices: Current, Voltage, FORCE_SIDE,␣
↪COMPLIANCE_AND_FORCE_SIDE

# General Instrument Settings
# b1500.adc_averaging = 1
# b1500.adc_auto_zero = True
b1500.adc_setup('HRADC','AUTO',6)
#b1500.adc_setup('HRADC','PLC',1)

#Sweep Settings
b1500.sweep_timing(0,5,step_delay=0.1) #hold,delay
b1500.sweep_auto_abort(False,post='STOP') #disable auto abort, set post measurement␣
↪output condition to stop value of sweep
# Sweep Source
nop = 11
b1500.smu1.staircase_sweep_source('VOLTAGE','LINEAR_DOUBLE','Auto Ranging',0,1,nop,0.
↪001) #type, mode, range, start, stop, steps, compliance
# Synchronous Sweep Source
b1500.smu2.synchronous_sweep_source('VOLTAGE','Auto Ranging',0,1,0.001) #type, range,␣
↪start, stop, comp
# Constant Output (could also be done using synchronous sweep source with start=stop,␣
↪but then the output is not ramped up)
b1500.smu3.ramp_source('VOLTAGE','Auto Ranging',-1,stepsize=0.1,pause=20e-3) #output␣
↪starts immediately! (compared to sweeps)
```

(continues on next page)

(continued from previous page)

```
b1500.smu4.ramp_source('VOLTAGE','Auto Ranging',0,stepsize=0.1,pause=20e-3)

#Start Measurement
b1500.check_errors()
b1500.clear_buffer()
b1500.clear_timer()
b1500.send_trigger()

# read measurement data all at once
b1500.check_idle() #wait until measurement is finished
data = b1500.read_data(2*nop) #Factor 2 beacuse of double sweep

#alternatively: read measurement data live
meas = []
for i in range(nop*2):
    read_data = b1500.read_channels(4+1) # 4 measurement channels, 1 sweep source␣
↪(returned due to mode=1 of data_format)
    # process live data for plotting etc.
    # data format for every channel (status code, channel name e.g. 'SMU1', data name e.g
↪'Current Measurement (A)', value)
    meas.append(read_data)

#sweep constant sources back to 0V
b1500.smu3.ramp_source('VOLTAGE','Auto Ranging',0,stepsize=0.1,pause=20e-3)
b1500.smu4.ramp_source('VOLTAGE','Auto Ranging',0,stepsize=0.1,pause=20e-3)
```

Sampling measurement with 4 SMUs

```
# choose measurement mode
b1500.meas_mode('SAMPLING', *b1500.smu_references) #order in smu_references determines␣
↪order of measurement
number_of_channels = len(b1500.smu_references)

# settings for individual SMUs
for smu in b1500.smu_references:
    smu.enable() #enable SMU
    smu.adc_type = 'HSADC' #set ADC to high-speed ADC
    smu.meas_range_current = '1 nA'
    smu.meas_op_mode = 'COMPLIANCE_SIDE' # other choices: Current, Voltage, FORCE_SIDE,␣
↪COMPLIANCE_AND_FORCE_SIDE

b1500.sampling_mode = 'LINEAR'
# b1500.adc_averaging = 1
# b1500.adc_auto_zero = True
b1500.adc_setup('HSADC','AUTO',1)
#b1500.adc_setup('HSADC','PLC',1)
nop=11
b1500.sampling_timing(2,0.005,nop) #MT: bias hold time, sampling interval, number of␣
↪points
b1500.sampling_auto_abort(False,post='BIAS') #MSC: BASE/BIAS
```

(continues on next page)

(continued from previous page)

```
b1500.time_stamp = True

# Sources
b1500.smu1.sampling_source('VOLTAGE','Auto Ranging',0,1,0.001) #MV/MI: type, range, base,
↪ bias, compliance
b1500.smu2.sampling_source('VOLTAGE','Auto Ranging',0,1,0.001)
b1500.smu3.ramp_source('VOLTAGE','Auto Ranging',-1,stepsize=0.1,pause=20e-3) #output␣
↪starts immediately! (compared to sweeps)
b1500.smu4.ramp_source('VOLTAGE','Auto Ranging',-1,stepsize=0.1,pause=20e-3)

#Start Measurement
b1500.check_errors()
b1500.clear_buffer()
b1500.clear_timer()
b1500.send_trigger()

meas=[]
for i in range(nop):
    read_data = b1500.read_channels(1+2*number_of_channels) #Sampling Index + (time␣
↪stamp + measurement value) * number of channels
    # process live data for plotting etc.
    # data format for every channel (status code, channel name e.g. 'SMU1', data name e.g
↪'Current Measurement (A)', value)
    meas.append(read_data)

#sweep constant sources back to 0V
b1500.smu3.ramp_source('VOLTAGE','Auto Ranging',0,stepsize=0.1,pause=20e-3)
b1500.smu4.ramp_source('VOLTAGE','Auto Ranging',0,stepsize=0.1,pause=20e-3)
```

Main Classes

Classes to communicate with the instrument:

- *AgilentB1500*: Main instrument class
- *SMU*: Instantiated by main instrument class for every SMU

All *query* commands return a human readable dict of settings. These are intended for debugging/logging/file headers, not for passing to the accompanying setting commands.

class pymeasure.instruments.agilent.agilentB1500.**AgilentB1500**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Agilent B1500 Semiconductor Parameter Analyzer and provides a high-level interface for taking different kinds of measurements.

property smu_references

Returns all SMU instances.

property smu_names

Returns all SMU names.

query_learn(*query_type*)

Queries settings from the instrument (*LRN?). Returns dict of settings.

Parameters **query_type** (*int or str*) – Query type (number according to manual)

query_learn_header(*query_type*, ***kwargs*)
Queries settings from the instrument (*LRN?). Returns dict of settings in human readable format for debugging or file headers. For optional arguments check the underlying definition of *QueryLearn.query_learn_header()*.

Parameters **query_type** (*int or str*) – Query type (number according to manual)

reset()
Resets the instrument to default settings (*RST)

query_modules()
Queries module models from the instrument. Returns dictionary of channel and module type.

Returns Channel:Module Type

Return type dict

initialize_smu(*channel*, *smu_type*, *name*)
Initializes SMU instance by calling *SMU*.

Parameters

- **channel** (*int*) – SMU channel
- **smu_type** (*str*) – SMU type, e.g. 'HRSMU'
- **name** (*str*) – SMU name for pymeasure (data output etc.)

Returns SMU instance

Return type *SMU*

initialize_all_smus()
Initialize all SMUs by querying available modules and creating a SMU class instance for each. SMUs are accessible via attributes .smu1 etc.

pause(*pause_seconds*)
Pauses Command Excecution for given time in seconds (PA)

Parameters **pause_seconds** (*int*) – Seconds to pause

abort()
Aborts the present operation but channels may still output current/voltage (AB)

force_gnd()
Force 0V on all channels immediately. Current Settings can be restored with RZ. (DZ)

check_errors()
Check for errors (ERRX?)

check_idle()
Check if instrument is idle (*OPC?)

clear_buffer()
Clear output data buffer (BC)

clear_timer()
Clear timer count (TSR)

send_trigger()
Send trigger to start measurement (except High Speed Spot) (XE)

property auto_calibration
Enable/Disable SMU auto-calibration every 30 minutes. (CM)

Type bool

data_format(*output_format*, *mode=0*)
Specifies data output format. Check Documentation for parameters. Should be called once per session to set the data format for interpreting the measurement values read from the instrument. (FMT)

Currently implemented are format 1, 11, and 21.

Parameters

- **output_format** (*str*) – Output format string, e.g. FMT21
- **mode** (*int, optional*) – Data output mode, defaults to 0 (only measurement data is returned)

property parallel_meas

Enable/Disable parallel measurements. Effective for SMUs using HSADC and measurement modes 1,2,10,18. (PAD)

Type bool

query_meas_settings()
Read settings for TM, AV, CM, FMT and MM commands (31) from the instrument.

query_meas_mode()
Read settings for MM command (part of 31) from the instrument.

meas_mode(*mode*, **args*)
Set Measurement mode of channels. Measurements will be taken in the same order as the SMU references are passed. (MM)

Parameters

- **mode** (*MeasMode*) – Measurement mode
  - Spot
  - Staircase Sweep
  - Sampling
- **args** (*SMU*) – SMU references

query_adc_setup()
Read ADC settings (55, 56) from the intrument.

adc_setup(*adc_type*, *mode*, *N=''*)
Set up operation mode and parameters of ADC for each ADC type. (AIT) Defaults:

- HSADC: Auto N=1, Manual N=1, PLC N=1, Time N=0.000002(s)
- HRADC: Auto N=6, Manual N=3, PLC N=1

Parameters

- **adc_type** (*ADCType*) – ADC type
- **mode** (*ADCMode*) – ADC mode
- **N** (*str, optional*) – additional parameter, check documentation, defaults to ''

adc_averaging(*number*, *mode='Auto'*)
Set number of averaging samples of the HSADC. (AV)

Defaults: N=1, Auto

Parameters

- **number** (*int*) – Number of averages
- **mode** (*AutoManual*, optional) – Mode (`'Auto'`,`'Manual'`), defaults to 'Auto'

property adc_auto_zero
Enable/Disable ADC zero function. Halfs the integration time, if off. (AZ)

Type bool

property time_stamp
Enable/Disable Time Stamp function. (TSC)

Type bool

query_time_stamp_setting()
Read time stamp settings (60) from the instrument.

wait_time(*wait_type*, *N*, *offset=0*)
Configure wait time. (WAT)

Parameters

- **wait_type** (*WaitTimeType*) – Wait time type
- **N** (*float*) – Coefficient for initial wait time, default: 1
- **offset** (*int, optional*) – Offset for wait time, defaults to 0

query_staircase_sweep_settings()
Reads Staircase Sweep Measurement settings (33) from the instrument.

sweep_timing(*hold*, *delay*, *step_delay=0*, *step_trigger_delay=0*, *measurement_trigger_delay=0*)
Sets Hold Time, Delay Time and Step Delay Time for staircase or multi channel sweep measurement. (WT)
If not set, all parameters are 0.

Parameters

- **hold** (*float*) – Hold time
- **delay** (*float*) – Delay time
- **step_delay** (*float, optional*) – Step delay time, defaults to 0
- **step_trigger_delay** (*float, optional*) – Trigger delay time, defaults to 0
- **measurement_trigger_delay** (*float, optional*) – Measurement trigger delay time, defaults to 0

sweep_auto_abort(*abort*, *post='START'*)
Enables/Disables the automatic abort function. Also sets the post measurement condition. (WM)

Parameters

- **abort** (*bool*) – Enable/Disable automatic abort
- **post** (*StaircaseSweepPostOutput*, optional) – Output after measurement, defaults to 'Start'

query_sampling_settings()
Reads Sampling Measurement settings (47) from the instrument.

property sampling_mode
Set linear or logarithmic sampling mode. (ML)

Type *SamplingMode*

sampling_timing(*hold_bias*, *interval*, *number*, *hold_base=0*)
Sets Timing Parameters for the Sampling Measurement (MT)

Parameters

- **hold_bias** (*float*) – Bias hold time
- **interval** (*float*) – Sampling interval
- **number** (*int*) – Number of Samples
- **hold_base** (*float, optional*) – Base hold time, defaults to 0

sampling_auto_abort(*abort*, *post='Bias'*)
Enables/Disables the automatic abort function. Also sets the post measurement condition. (MSC)

Parameters

- **abort** (*bool*) – Enable/Disable automatic abort
- **post** (*SamplingPostOutput*, optional) – Output after measurement, defaults to 'Bias'

read_data(*number_of_points*)
Reads all data from buffer and returns Pandas DataFrame. Specify number of measurement points for correct splitting of the data list.

Parameters **number_of_points** (*int*) – Number of measurement points

Returns Measurement Data

Return type pd.DataFrame

read_channels(*nchannels*)
Reads data for 1 measurement point from the buffer. Specify number of measurement channels + sweep sources (depending on data output setting).

Parameters **nchannels** (*int*) – Number of channels which return data

Returns Measurement data

Return type tuple

query_series_resistor()
Read series resistor status (53) for all SMUs.

query_meas_range_current_auto()
Read auto ranging mode status (54) for all SMUs.

query_meas_op_mode()
Read SMU measurement operation mode (46) for all SMUs.

query_meas_ranges()
Read measruement ranging status (32) for all SMUs.

class pymeasure.instruments.agilent.agilentB1500.**SMU**(*parent*, *channel*, *smu_type*, *name*, ***kwargs*)
Bases: object

Provides specific methods for the SMUs of the Agilent B1500 mainframe

Parameters

- **parent** (*AgilentB1500*) – Instance of the B1500 mainframe class
- **channel** (*int*) – Channel number of the SMU
- **smu_type** (*str*) – Type of the SMU
- **name** (*str*) – Name of the SMU

write(*string*)
Wraps *Instrument.write()* method of B1500.

ask(*string*)
Wraps `ask()` method of B1500.

query_learn(*query_type*, *command*)
Wraps *query_learn()* method of B1500.

check_errors()
Wraps *check_errors()* method of B1500.

property status
Query status of the SMU.

enable()
Enable Source/Measurement Channel (`CN`)

disable()
Disable Source/Measurement Channel (`CL`)

force_gnd()
Force 0V immediately. Current Settings can be restored with `RZ` (not implemented). (`DZ`)

property filter
Enables/Disables SMU Filter. (`FL`)

Type bool

property series_resistor
Enables/Disables 1MOhm series resistor. (`SSR`)

Type bool

property meas_op_mode
Set SMU measurement operation mode. (`CMM`)

Type *MeasOpMode*

property adc_type
ADC type of individual measurement channel. (`AAD`)

Type *ADCType*

force(*source_type*, *source_range*, *output*, *comp=''*, *comp_polarity=''*, *comp_range=''*)
Applies DC Current or Voltage from SMU immediately. (`DI`, `DV`)

Parameters

- **source_type** (*str*) – Source type (`'Voltage','Current'`)
- **source_range** (*int or str*) – Output range index or name
- **output** – Source output value in A or V
- **comp** (*float, optional*) – Compliance value, defaults to previous setting
- **comp_polarity** (*CompliancePolarity*) – Compliance polairty, defaults to auto
- **comp_range** (*int or str, optional*) – Compliance ranging type, defaults to auto

ramp_source(*source_type*, *source_range*, *target_output*, *comp=''*, *comp_polarity=''*, *comp_range=''*, *stepsize=0.001*, *pause=0.02*)
Ramps to a target output from the set value with a given step size, each separated by a pause.

Parameters

- **source_type** (*str*) – Source type (`'Voltage'` or `'Current'`)
- **target_output** – Target output voltage or current
- **irange** (*int*) – Output range index
- **comp** (*float, optional*) – Compliance, defaults to previous setting
- **comp_polarity** (*CompliancePolarity*) – Compliance polairty, defaults to auto
- **comp_range** (*int or str, optional*) – Compliance ranging type, defaults to auto
- **stepsize** – Maximum size of steps
- **pause** – Duration in seconds to wait between steps

Type target_output: float

property meas_range_current
Current measurement range index. (`RI`)

Possible settings depend on SMU type, e.g. `0` for Auto Ranging: *SMUCurrentRanging*

property meas_range_voltage
Voltage measurement range index. (`RV`)

Possible settings depend on SMU type, e.g. `0` for Auto Ranging: *SMUVoltageRanging*

meas_range_current_auto(*mode*, *rate=50*)
Specifies the auto range operation. Check Documentation. (`RM`)

Parameters

- **mode** (*int*) – Range changing operation mode
- **rate** (*int, optional*) – Parameter used to calculate the *current* value, defaults to 50

staircase_sweep_source(*source_type*, *mode*, *source_range*, *start*, *stop*, *steps*, *comp*, *Pcomp=''*)
Specifies Staircase Sweep Source (Current or Voltage) and its parameters. (`WV` or `WI`)

Parameters

- **source_type** (*str*) – Source type (`'Voltage','Current'`)
- **mode** (*SweepMode*) – Sweep mode
- **source_range** (*int*) – Source range index
- **start** (*float*) – Sweep start value
- **stop** (*float*) – Sweep stop value
- **steps** (*int*) – Number of sweep steps
- **comp** (*float*) – Compliance value
- **Pcomp** (*float, optional*) – Power compliance, defaults to not set

synchronous_sweep_source(*source_type*, *source_range*, *start*, *stop*, *comp*, *Pcomp=''*)
Specifies Synchronous Staircase Sweep Source (Current or Voltage) and its parameters. (`WSV` or `WSI`)

Parameters

- **source_type** (*str*) – Source type (`'Voltage','Current'`)
- **source_range** (*int*) – Source range index
- **start** (*float*) – Sweep start value
- **stop** (*float*) – Sweep stop value

- **comp** (*float*) – Compliance value
- **Pcomp** (*float, optional*) – Power compliance, defaults to not set

sampling_source(*source_type*, *source_range*, *base*, *bias*, *comp*)
Sets DC Source (Current or Voltage) for sampling measurement. DV/DI commands on the same channel overwrite this setting. (MV or MI)

Parameters

- **source_type** (*str*) – Source type ('Voltage','Current')
- **source_range** (*int*) – Source range index
- **base** (*float*) – Base voltage/current
- **bias** (*float*) – Bias voltage/current
- **comp** (*float*) – Compliance value

Supporting Classes

Classes that provide additional functionalities:

- *QueryLearn*: Process read out of instrument settings
- *SMUCurrentRanging*, *SMUVoltageRanging*: Allowed ranges for different SMU types and transformation of range names to indices (base: *Ranging*)

class pymeasure.instruments.agilent.agilentB1500.**QueryLearn**
Bases: object

Methods to issue and process *LRN? (learn) command and response.

static query_learn(*ask*, *query_type*)
Issues *LRN? (learn) command to the instrument to read configuration. Returns dictionary of commands and set values.

Parameters **query_type** (*int*) – Query type according to the programming guide

Returns Dictionary of command and set values

Return type dict

classmethod query_learn_header(*ask*, *query_type*, *smu_references*, *single_command=False*)
Issues *LRN? (learn) command to the instrument to read configuration. Processes information to human readable values for debugging purposes or file headers.

Parameters

- **ask** (*Instrument.ask*) – ask method of the instrument
- **query_type** (*int or str*) – Number according to Programming Guide
- **smu_references** (*dict*) – SMU references by channel
- **single_command** (*str*) – if only a single command should be returned, defaults to False

Returns Read configuration

Return type dict

static to_dict(*parameters*, *names*, **args*)
Takes parameters returned by *query_learn()* and ordered list of corresponding parameter names (optional function) and returns dict of parameters including names.

Parameters

- **parameters** (*dict*) – Parameters for one command returned by *query_learn()*
- **names** (*list*) – list of names or (name, function) tuples, ordered

Returns Parameter name and (processed) parameter

Return type dict

class pymeasure.instruments.agilent.agilentB1500.**Ranging**(*supported_ranges*, *ranges*, *fixed_ranges=False*)

Bases: object

Possible Settings for SMU Current/Voltage Output/Measurement ranges. Transformation of available Voltage/Current Range Names to Index and back.

Parameters

- **supported_ranges** (*list*) – Ranges which are supported (list of range indizes)
- **ranges** (*dict*) – All range names {Name: Indizes}
- **fixed_ranges** – add fixed ranges (negative indizes); defaults to False

__call__(*input_value*)

Gives named tuple (name/index) of given Range. Throws error if range is not supported by this SMU.

Parameters **input** (*str or int*) – Range name or index

Returns named tuple (name/index) of range

Return type namedtuple

class pymeasure.instruments.agilent.agilentB1500.**SMUCurrentRanging**(*smu_type*)

Bases: object

Provides Range Name/Index transformation for current measurement/sourcing. Validity of ranges is checked against the type of the SMU.

Omitting the 'limited auto ranging'/'range fixed' specification in the range string for current measurement defaults to 'limited auto ranging'.

Full specification: '1 nA range fixed' or '1 nA limited auto ranging'

'1 nA' defaults to '1 nA limited auto ranging'

class pymeasure.instruments.agilent.agilentB1500.**SMUVoltageRanging**(*smu_type*)

Bases: object

Provides Range Name/Index transformation for voltage measurement/sourcing. Validity of ranges is checked against the type of the SMU.

Omitting the 'limited auto ranging'/'range fixed' specification in the range string for voltage measurement defaults to 'limited auto ranging'.

Full specification: '2 V range fixed' or '2 V limited auto ranging'

'2 V' defaults to '2 V limited auto ranging'

Enumerations

Enumerations are used for easy selection of the available parameters (where it is applicable). Methods accept member name or number as input, but name is recommended for readability reasons. The member number is passed to the instrument. Converting an enumeration member into a string gives a title case, whitespace separated string (*__str__()*) which cannot be used to select an enumeration member again. It's purpose is only logging or documentation.

class pymeasure.instruments.agilent.agilentB1500.**CustomIntEnum**(*value*)
Bases: enum.IntEnum

Provides additional methods to IntEnum:

- Conversion to string automatically replaces '_' with ' ' in names and converts to title case
- get classmethod to get enum reference with name or integer

__str__()
Gives title case string of enum value

classmethod get(*input_value*)
Gives Enum member by specifying name or value.

Parameters **input_value** (*str or int*) – Enum name or value

Returns Enum member

class pymeasure.instruments.agilent.agilentB1500.**ADCType**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

ADC Type

HSADC = 0
High-speed ADC

HRADC = 1
High-resolution ADC

HSADC_PULSED = 2
High-resolution ADC for pulsed measurements

class pymeasure.instruments.agilent.agilentB1500.**ADCMode**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

ADC Mode

AUTO = 0

MANUAL = 1

PLC = 2

TIME = 3

class pymeasure.instruments.agilent.agilentB1500.**AutoManual**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Auto/Manual selection

AUTO = 0

MANUAL = 1

class pymeasure.instruments.agilent.agilentB1500.**MeasMode**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Measurement Mode

SPOT = 1

STAIRCASE_SWEEP = 2

SAMPLING = 10

class pymeasure.instruments.agilent.agilentB1500.**MeasOpMode**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Measurement Operation Mode

COMPLIANCE_SIDE = 0

CURRENT = 1

VOLTAGE = 2

FORCE_SIDE = 3

COMPLIANCE_AND_FORCE_SIDE = 4

class pymeasure.instruments.agilent.agilentB1500.**SweepMode**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Sweep Mode

LINEAR_SINGLE = 1

LOG_SINGLE = 2

LINEAR_DOUBLE = 3

LOG_DOUBLE = 4

class pymeasure.instruments.agilent.agilentB1500.**SamplingMode**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Sampling Mode

LINEAR = 1

LOG_10 = 2
Logarithmic 10 data points/decade

LOG_25 = 3
Logarithmic 25 data points/decade

LOG_50 = 4
Logarithmic 50 data points/decade

LOG_100 = 5
Logarithmic 100 data points/decade

LOG_250 = 6
Logarithmic 250 data points/decade

LOG_5000 = 7
Logarithmic 5000 data points/decade

class pymeasure.instruments.agilent.agilentB1500.**SamplingPostOutput**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Output after sampling

BASE = 1

BIAS = 2

class pymeasure.instruments.agilent.agilentB1500.**StaircaseSweepPostOutput**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Output after staircase sweep

START = 1

STOP = 2

class pymeasure.instruments.agilent.agilentB1500.**CompliancePolarity**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Compliance polarity

AUTO = 0

MANUAL = 1

class pymeasure.instruments.agilent.agilentB1500.**WaitTimeType**(*value*)
Bases: *pymeasure.instruments.agilent.agilentB1500.CustomIntEnum*

Wait time type

SMU_SOURCE = 1

SMU_MEASUREMENT = 2

CMU_MEASUREMENT = 3

7.8 Ametek

This section contains specific documentation on the Ametek instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.8.1 Ametek 7270 DSP Lockin Amplifier

class pymeasure.instruments.ametek.**Ametek7270**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

This is the class for the Ametek DSP 7270 lockin amplifier

property adc1
Reads the input value of ADC1 in Volts

property adc2
Reads the input value of ADC2 in Volts

property adc3
Reads the input value of ADC3 in Volts

property adc4
Reads the input value of ADC4 in Volts

property dac1
A floating point property that represents the output value on DAC1 in Volts. This property can be set.

property dac2
A floating point property that represents the output value on DAC2 in Volts. This property can be set.

property dac3
A floating point property that represents the output value on DAC3 in Volts. This property can be set.

property dac4
A floating point property that represents the output value on DAC4 in Volts. This property can be set.

property frequency
A floating point property that represents the lock-in frequency in Hz. This property can be set.

property harmonic
An integer property that represents the reference harmonic mode control, taking values from 1 to 127. This property can be set.

property id
Reads the instrument identification

property mag
Reads the magnitude in Volts

property phase
A floating point property that represents the reference harmonic phase in degrees. This property can be set.

property sensitivity
A floating point property that controls the sensitivity range in Volts, which can take discrete values from 2 nV to 1 V. This property can be set.

set_channel_A_mode()
Sets instrument to channel A mode – assuming it is in voltage mode

set_differential_mode(*lineFiltering=True*)
Sets instrument to differential mode – assuming it is in voltage mode

set_voltage_mode()
Sets instrument to voltage control mode

shutdown()
Ensures the instrument in a safe state

property slope
A integer property that controls the filter slope in dB/octave, which can take the values 6, 12, 18, or 24 dB/octave. This property can be set.

property time_constant
A floating point property that controls the time constant in seconds, which takes values from 10 microseconds to 100,000 seconds. This property can be set.

property voltage
A floating point property that represents the voltage in Volts. This property can be set.

property x
Reads the X value in Volts

property x1
Reads the first harmonic X value in Volts

property x2
Reads the second harmonic X value in Volts

property xy
Reads both the X and Y values in Volts

property y
Reads the Y value in Volts

property y1
Reads the first harmonic Y value in Volts

property y2
Reads the second harmonic Y value in Volts

7.9 AMI

This section contains specific documentation on the AMI instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.9.1 AMI 430 Power Supply

class pymeasure.instruments.ami.**AMI430**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the AMI 430 Power supply and provides a high-level for interacting with the instrument.

```
magnet = AMI430("TCPIP::web.address.com::7180::SOCKET")


magnet.coilconst = 1.182                # kGauss/A
magnet.voltage_limit = 2.2              # Sets the voltage limit in V

magnet.target_current = 10              # Sets the target current to 10 A
magnet.target_field = 1                 # Sets target field to 1 kGauss

magnet.ramp_rate_current = 0.0357       # Sets the ramp rate in A/s
magnet.ramp_rate_field = 0.0422         # Sets the ramp rate in kGauss/s
magnet.ramp                             # Initiates the ramping
magnet.pause                            # Pauses the ramping
magnet.status                           # Returns the status of the magnet

magnet.ramp_to_current(5)             # Ramps the current to 5 A

magnet.shutdown()                     # Ramps the current to zero and disables␣
↪output
```

property coilconst
A floating point property that sets the coil contant in kGauss/A.

disable_persistent_switch()
Disables the persistent switch.

enable_persistent_switch()
Enables the persistent switch.

property field
Reads the field in kGauss of the magnet.

has_persistent_switch_enabled()
Returns a boolean if the persistent switch is enabled.

property magnet_current
Reads the current in Amps of the magnet.

pause()
Pauses the ramping of the magnetic field.

ramp()
Initiates the ramping of the magnetic field to set current/field with ramping rate previously set.

property ramp_rate_current
A floating point property that sets the current ramping rate in A/s.

property ramp_rate_field
A floating point property that sets the field ramping rate in kGauss/s.

ramp_to_current(*current*, *rate*)
Heats up the persistent switch and ramps the current with set ramp rate.

ramp_to_field(*field*, *rate*)
Heats up the persistent switch and ramps the current with set ramp rate.

shutdown(*ramp_rate=0.0357*)
Turns on the persistent switch, ramps down the current to zero, and turns off the persistent switch.

property state
Reads the field in kGauss of the magnet.

property supply_current
Reads the current in Amps of the power supply.

property target_current
A floating point property that sets the target current in A for the magnet.

property target_field
A floating point property that sets the target field in kGauss for the magnet.

property voltage_limit
A floating point property that sets the voltage limit for charging/discharging the magnet.

wait_for_holding(*should_stop=<function AMI430.<lambda>>*, *timeout=800*, *interval=0.1*)

zero()
Initiates the ramping of the magnetic field to zero current/field with ramping rate previously set.

7.10 Anaheim Automation

This section contains specific documentation on the Anaheim Automation instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.10.1 DP-Series Step Motor Controller

The DPSeriesMotorController class implements a base driver class for Anaheim-Automation DP Series stepper motor controllers. There are many controllers sold in this series, all of which implement the same core command set. Some controllers, like the DPY50601, implement additional functionality that is not included in this driver. If these additional features are desired, they should be implemented in a subclass.

class pymeasure.instruments.anaheimautomation.**DPSeriesMotorController**(*adapter*, *address=0*, *encoder_enabled=False*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Base class to interface with Anaheim Automation DP series stepper motor controllers.

This driver has been tested with the DPY50601 and DPE25601 motor controllers.

property absolute_position
Float property representing the value of the motor position measured in absolute units. Note that in DP series motor controller instrument manuals, *absolute position* refers to the 'step_position' property rather than this property. Also note that use of this property relies on steps_to_absolute() and absolute_to_steps() being implemented in a subclass. In this way, the user can define the conversion from a motor step position into any desired absolute unit. Absolute units could be the position in meters of a linear stage or the angular position of a gimbal mount, etc. This property can be set.

absolute_to_steps(*pos*)
Convert an absolute position to a number of steps to move. This must be implemented in subclasses.

Parameters pos – Absolute position in the units determined by the subclassed absolute_to_steps() method.

property address
Integer property representing the address that the motor controller uses for serial communications.

property basespeed
Integer property that represents the motor controller's starting/homing speed. This property can be set.

property busy
Query to see if the controller is currently moving a motor.

check_errors()
Method to read the error codes register and log when an error is detected.

Return error_code one byte with the error codes register contents

property direction
A string property that represents the direction in which the stepper motor will rotate upon subsequent step commands. This property can be set. 'CW' corresponds to clockwise rotation and 'CCW' corresponds to counter-clockwise rotation.

property encoder_autocorrect
A boolean property to enable or disable the encoder auto correct function. This property can be set.

property encoder_delay
An integer property that represents the wait time in ms. after a move is finished before the encoder is read for a potential encoder auto-correct action to take place. This property can be set.

property encoder_enabled
A boolean property to represent whether an external encoder is connected and should be used to set the step_position property.

property encoder_motor_ratio
An integer property that represents the ratio of the number of encoder pulses per motor step. This property can be set.

property encoder_retries
An integer property that represents the number of times the motor controller will try the encoder auto correct function before setting an error flag. This property can be set.

property encoder_window
An integer property that represents the allowable error in encoder pulses from the desired position before the encoder auto-correct function runs. This property can be set.

property error_reg
Reads the current value of the error codes register.

home(*home_mode*)
Send command to the motor controller to 'home' the motor.

Parameters `home_mode` – 0 or 1 specifying which homing mode to run.

0 will perform a homing operation where the controller moves the motor until a soft limit is reached, then will ramp down to base speed and continue motion until a home limit is reached.

In mode 1, the controller will move the motor until a limit is reached, then will ramp down to base speed, change direction, and run until the limit is released.

`property maxspeed`
Integer property that represents the motor controller's maximum (running) speed. This property can be set.

`move`(*direction*)
Move the stepper motor continuously in the given direction until a stop command is sent or a limit switch is reached. This method corresponds to the 'slew' command in the DP series instrument manuals.

Parameters `direction` – value to set on the direction property before moving the motor.

`reset_position`()
Reset position as counted by the motor controller and an externally connected encoder to 0.

`property step_position`
Integer property representing the value of the motor position measured in steps counted by the motor controller or, if encoder_enabled is set, the steps counted by an externally connected encoder. Note that in the DP series motor controller instrument manuals, this property would be referred to as the 'absolute position' while this driver implements a conversion between steps and absolute units for the *absolute position* property. This property can be set.

`steps_to_absolute`(*steps*)
Convert a position measured in steps to an absolute position.

Parameters `steps` – Position in steps to be converted to an absolute position.

`stop`()
Method that stops all motion on the motor controller.

`wait_for_completion`(*interval=0.5*)
Block until the controller is not "busy" (i.e. block until the motor is no longer moving.)

Parameters `interval` – (float) seconds between queries to the "busy" flag.

Returns None

`write`(*command*)
Override the instrument base write method to add the motor controller's address to the command string.

Parameters `command` – command string to be sent to the motor controller.

7.11 Anapico

This section contains specific documentation on the Anapico instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.11.1 Anapico APSIN12G Signal Generator

class pymeasure.instruments.anapico.**APSIN12G**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Anapico APSIN12G Signal Generator with option 9K, HP and GPIB.

property blanking
A string property that represents the blanking of output power when frequency is changed. ON makes the output to be blanked (off) while changing frequency. This property can be set.

disable_rf()
Disables the RF output.

enable_rf()
Enables the RF output.

property frequency
A floating point property that represents the output frequency in Hz. This property can be set.

property power
A floating point property that represents the output power in dBm. This property can be set.

property reference_output
A string property that represents the 10MHz reference output from the synth. This property can be set.

7.12 Andeen Hagerling

This section contains specific documentation on the Andeen Hagerling instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.12.1 Andeen Hagerling AH2500A capacitance bridge

class pymeasure.instruments.andeenhagerling.**AH2500A**(*adapter*, *name=None*, *timeout=3000*, *write_termination='\n'*, *read_termination='\n'*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Andeen Hagerling 2500A Precision Capacitance Bridge implementation

property caplossvolt
Perform a single capacitance, loss measurement and return the values in units of pF and nS. The used measurement voltage is returned as third value.

property config
Read out configuration

trigger()
Triggers a new measurement without blocking and waiting for the return value.

triggered_caplossvolt()
reads the measurement value after the device was triggered by the trigger function.

property vhighest
maximum RMS value of the used measurement voltage. Values of up to 15 V are allowed. The device will select the best suiting range below the given value.

7.12.2 Andeen Hagerling AH2700A capacitance bridge

class pymeasure.instruments.andeenhagerling.**AH2700A**(*adapter*, *name='Andeen Hagerling 2700A Precision Capacitance Bridge'*, *timeout=5000*, ***kwargs*)

Bases: *pymeasure.instruments.andeenhagerling.ah2500a.AH2500A*

Andeen Hagerling 2700A Precision Capacitance Bridge implementation

property caplossvolt
Perform a single capacitance, loss measurement and return the values in units of pF and nS. The used measurement voltage is returned as third value.

property config
Read out configuration

property frequency
test frequency used for the measurements. Allowed are values between 50 and 20000 Hz. The device selects the closest possible frequency to the given value.

property id
Reads the instrument identification

reset()
Resets the instrument.

trigger()
Triggers a new measurement without blocking and waiting for the return value.

triggered_caplossvolt()
reads the measurement value after the device was triggered by the trigger function.

property vhighest
maximum RMS value of the used measurement voltage. Values of up to 15 V are allowed. The device will select the best suiting range below the given value.

7.13 Anritsu

This section contains specific documentation on the Anritsu instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.13.1 Anritsu MG3692C Signal Generator

class pymeasure.instruments.anritsu.**AnritsuMG3692C**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Anritsu MG3692C Signal Generator

disable()
Disables the signal output.

enable()
Enables the signal output.

property frequency
A floating point property that represents the output frequency in Hz. This property can be set.

property output
A boolean property that represents the signal output state. This property can be set to control the output.

property power
A floating point property that represents the output power in dBm. This property can be set.

shutdown()
Shuts down the instrument, putting it in a safe state.

7.13.2 Anritsu MS9710C Optical Spectrum Analyzer

class pymeasure.instruments.anritsu.**AnritsuMS9710C**(*adapter*, *name='Anritsu MS9710C Optical Spectrum Analyzer'*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Anritsu MS9710C Optical Spectrum Analyzer.

property analysis
Analysis Control

property analysis_result
Read back anaysis result from current scan.

property average_point
Number of averages to take on each point (2-1000), or OFF

property average_sweep
Number of averages to make on a sweep (2-1000) or OFF

center_at_peak(***kwargs*)
Center the spectrum at the measured peak.

property data_memory_a_condition
Returns the data condition of data memory register A. Starting wavelength, and a sampling point (l1, l2, n).

property data_memory_a_size
Returns the number of points sampled in data memory register A.

property data_memory_a_values
Reads the binary data from memory register A.

property data_memory_b_condition
Returns the data condition of data memory register B. Starting wavelength, and a sampling point (l1, l2, n).

property data_memory_b_size
Returns the number of points sampled in data memory register B.

property data_memory_b_values
Reads the binary data from memory register B.

property data_memory_select
Memory Data Select.

property dip_search
Dip Search Mode

property ese2
Extended Event Status Enable Register 2

property esr2
Extended Event Status Register 2

property level_lin
Level Linear Scale (/div)

property level_log
Level Log Scale (/div)

property level_opt_attn
Optical Attenuation Status (ON/OFF)

property level_scale
Current Level Scale

property measure_mode
Returns the current Measure Mode the OSA is in.

measure_peak()
Measure the peak and return the trace marker.

property peak_search
Peak Search Mode

read_memory(*slot='A'*)
Read the scan saved in a memory slot.

property resolution
Resolution (nm)

property resolution_actual
Resolution Actual (ON/OFF)

property resolution_vbw
Video Bandwidth Resolution

property sampling_points
Number of sampling points

single_sweep(***kwargs*)
Perform a single sweep and wait for completion.

property trace_marker
Sets the trace marker with a wavelength. Returns the trace wavelength and power.

property trace_marker_center
Trace Marker at Center. Set to 1 or True to initiate command

wait(*n=3*, *delay=1*)
Query OPC Command and waits for appropriate response.

wait_for_sweep(*n=20*, *delay=0.5*)
Wait for a sweep to stop.

This is performed by checking bit 1 of the ESR2.

property wavelength_center
Center Wavelength of Spectrum Scan in nm.

property wavelength_marker_value
Wavelength Marker Value (wavelength or freq.?)

property wavelength_span
Wavelength Span of Spectrum Scan in nm.

property wavelength_start
Wavelength Start of Spectrum Scan in nm.

property wavelength_stop
Wavelength Stop of Spectrum Scan in nm.

property wavelength_value_in
Wavelength value in Vacuum or Air

property wavelengths
Return a numpy array of the current wavelengths of scans.

7.13.3 Anritsu MS9740A Optical Spectrum Analyzer

class pymeasure.instruments.anritsu.**AnritsuMS9740A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.anritsu.anritsuMS9710C.AnritsuMS9710C*

Anritsu MS9740A Optical Spectrum Analyzer.

property average_sweep
Nr. of averages to make on a sweep (1-1000), with 1 being a single (non-averaged) sweep

property data_memory_select
Memory Data Select.

repeat_sweep(*n=20*, *delay=0.5*)
Perform a single sweep and wait for completion.

property resolution
Resolution (nm)

property resolution_vbw
Video Bandwidth Resolution

property sampling_points
Number of sampling points

7.13.4 Anritsu MS2090A Handheld Spectrum Analyzer

class pymeasure.instruments.anritsu.**AnritsuMS2090A**(*adapter*, *name='Anritsu MS2090A Handheld Spectrum Analyzer'*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Anritsu MS2090A Handheld Spectrum Analyzer.

abort()
Initiate a sweep/measurement.

property active_state
The "set" state indicates that the instrument is used by someone.

property external_current
This command queries the actual bias current in A

property fetch_control
Returns the Control Channel measurement in json format.

property fetch_density
Returns the most recent channel density measurement

property fetch_eirpower
Returns the current EIRP, Max EIRP, Horizontal EIRP, Vertical and Sum EIRP results in dBm.

property fetch_eirpower_data
This command returns the current EIRP measurement result in dBm.

property fetch_eirpower_max
This command returns the Max EIRP measurement result in dBm.

property fetch_emf
Return the current EMF measurement data. JSON format.

property fetch_emf_meter
Return the live EMF measurement data. JSON format.

property fetch_emf_meter_sample
Return the EMF measurement data for a specified sample number. JSON format.

property fetch_interference_power
Fetch Interference Finder Integrated Power.

property fetch_mimo_antenas
Returns the sync power measurement in json format.

property fetch_ocupied_bw
Returns the different set of measurement information depending on the suffix.

property fetch_ota_mapping
Returns the most recent Coverage Mapping measurement result.

property fetch_pan
Return the current Pulse Analyzer measurement data. JSON format

property fetch_pbch_constellation
Get the latest Physical Broadcast Channel constellation hitmap

property fetch_pci
Returns PCI measurements

property fetch_pdsch
Returns the Data Channel Measurements in JSON format.

property fetch_pdsch_constellation
Get the latest Physical Downlink Shared Channel constellation

property fetch_peak
Returns a pair of peak amplitude in current sweep.

property fetch_power
Returns the most recent channel power measurement.

property fetch_rrm
Returns the Radio Resource Management in JSON format.

property fetch_scan
Returns the cell scanner measurements in JSON format

property fetch_semask
This command returns the current Spectral Emission Mask measurement result.

property fetch_ssb
Returns the SSB measurement

property fetch_sync_evm
Returns the Sync EVM measurement in JSON format.

property fetch_sync_power
Returns the sync power measurements in JSON format

property fetch_tae
Returns the Time Alignment Error in JSON format.

property frequency_center
Sets the center frequency in Hz

property frequency_offset
Sets the frequency offset in Hz

property frequency_span
Sets the frequency span in Hz

property frequency_span_full
Sets the frequency span to full span

property frequency_span_last
Sets the frequency span to the previous span value.

property frequency_start
Sets the start frequency in Hz

property frequency_step
Set or query the step size to gradually increase or decrease frequency values in Hz

property frequency_stop
Sets the start frequency in Hz

property gps
Returns the timestamp, latitude, and longitude of the device.

property gps_all
Returns the fix timestamp, latitude, longitude, altitude and information on the sat used.

property gps_full
Returns the timestamp, latitude, longitude, altitude, and satellite count of the device.

property gps_last
Returns the timestamp, latitude, longitude, and altitude of the last fixed GPS result.

init_all_sweep()
Initiate all sweep/measurement.

property init_continuous
Specified whether the sweep/measurement is triggered continuously

property init_spa_self
Perform a self-test and return the results.

init_sweep()
Initiate a sweep/measurement.

property meas_acpower
Sets the active measurement to adjacent channel power ratio, sets the default measurement parameters, triggers a new measurement and returns the main channel power, lower adjacent, upper adjacent, lower alternate and upper alternate channel power results.

property meas_emf_meter_clear_all
Clear the EMF measurement data of all samples. Sampling state will be turned off if it was on.

property meas_emf_meter_clear_sample
Clear the EMF measurement data for a specified sample number. Sampling state will be turned off if the specified sample is currently active.

property meas_emf_meter_sample
Start or Stop applying the measurement results to the currently selected sample

property meas_int_power
Sets the active measurement to interference finder, sets the default measurement parameters, triggers a new measurement and returns integrated power as the result. It is a combination of the commands :CONFigure:INTerference; :READ:INTerference:POWer?

property meas_iq_capture
This set command is used to start the IQ capture measurement.

property meas_iq_capture_fail
Sets or queries whether the instrument will automatically save an IQ capture when losing sync

property meas_ota_mapp
Sets the active measurement to OTA Coverage Mapping, sets the default measurement parameters, triggers a new measurement, and returns the measured values.

property meas_ota_run
Turn on/off OTA Coverage Mapping Data Collection. The instrument must be in Coverage Mapping measurement for the command to be effective

property meas_power
Sets the active measurement to channel power, sets the default measurement parameters, triggers a new measurement and returns channel power as the result. It is a combination of the commands :CONFigure:CHPower; :READ:CHPower:CHPower?

property meas_power_all
Sets the active measurement to channel power, sets the default measurement parameters, triggers a new measurement and returns the channel power and channel power density results. It is a combination of the commands :CONFigure:CHPower; :READ:CHPower?

property power_density
Sets the active measurement to channel power, sets the default measurement parameters, triggers a new measurement and returns channel power density as the result. It is a combination of the commands :CONFigure:CHPower; :READ:CHPower:DENSity?

property preamp
Sets the state of the preamp. Note that this may cause a change in the reference level and/or attenuation.

property sense_mode
Set the operational mode of the Spa app.

property view_sense_modes
Returns a list of available modes for the Spa application. The response is a comma-separated list of mode names. See command [:SENSe]:MODE for the mode name specification.

7.14 Attocube

This section contains specific documentation on the Attocube instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.14.1 Attocube Adapters

class pymeasure.instruments.attocube.adapters.**AttocubeConsoleAdapter**(*host*, *port*, *passwd*, ***kwargs*)

Bases: *pymeasure.adapters.telnet.TelnetAdapter*

Adapter class for connecting to the Attocube Standard Console. This console is a Telnet prompt with password authentication.

Parameters

- **host** – host address of the instrument
- **port** – TCPIP port
- **passwd** – password required to open the connection
- **kwargs** – Any valid key-word argument for TelnetAdapter

ask(*command*)

Writes a command to the instrument and returns the resulting ASCII response

Parameters **command** – command string to be sent to the instrument

Returns String ASCII response of the instrument

extract_value(*reply*)

preprocess_reply function for the Attocube console. This function tries to extract <value> from 'name = <value> [unit]'. If <value> can not be identified the original string is returned.

Parameters **reply** – reply string

Returns string with only the numerical value, or the original string

7.14.2 Attocube ANC300 Motion Controller

class pymeasure.instruments.attocube.anc300.**ANC300Controller**(*host*, *axisnames*, *passwd*, *query_delay=0.05*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Attocube ANC300 Piezo stage controller with several axes

Parameters

- **host** – host address of the instrument
- **axisnames** – a list of axis names which will be used to create properties with these names
- **passwd** – password for the attocube standard console
- **query_delay** – delay between sending and reading (default 0.05 sec)
- **kwargs** – Any valid key-word argument for TelnetAdapter

check_errors()

Read after setting a value.

property controllerBoardVersion
Serial number of the controller board

ground_all()
Grounds all axis of the controller.

stop_all()
Stop all movements of the axis.

property version
Version number and instrument identification

class pymeasure.instruments.attocube.anc300.Axis(*controller*, *axis*)
Bases: object

Represents a single open loop axis of the Attocube ANC350

Parameters

- **axis** – axis identifier, integer from 1 to 7
- **controller** – ANC300Controller instance used for the communication

property capacity
Saved capacity value in nF of the axis.

check_errors()
Read after setting a setting or control.

property frequency
Frequency of the stepping motion in Hertz from 1 to 10000 Hz. This property can be set.

measure_capacity()
Obtains a new measurement of the capacity. The mode of the axis returns to 'gnd' after the measurement.

Returns capacity the freshly measured capacity in nF.

property mode
Axis mode. This can be 'gnd', 'inp', 'cap', 'stp', 'off', 'stp+', 'stp-'. Available modes depend on the actual axis model

move(*steps*, *gnd=True*)
Move 'steps' steps in the direction given by the sign of the argument. This method will change the mode of the axis automatically and ground the axis on the end if 'gnd' is True. The method returns only when the movement is finished.

Parameters

- **steps** – finite integer value of steps to be performed. A positive sign corresponds to upwards steps, a negative sign to downwards steps.
- **gnd** – bool, flag to decide if the axis should be grounded after completion of the movement

property offset_voltage
Offset voltage in Volts from 0 to 150 V. This property can be set.

property output_voltage
Output voltage in volts.

property pattern_down
step down pattern of the piezo drive. 256 values ranging from 0 to 255 representing the the sequence of output voltages within one step of the piezo drive. This property can be set, the set value needs to be an array with 256 integer values.

property pattern_up
step up pattern of the piezo drive. 256 values ranging from 0 to 255 representing the the sequence of output voltages within one step of the piezo drive. This property can be set, the set value needs to be an array with 256 integer values.

property serial_nr
Serial number of the axis

property stepd
Step downwards for N steps. Mode must be 'stp' and N must be positive.

property stepu
Step upwards for N steps. Mode must be 'stp' and N must be positive.

stop()
Stop any motion of the axis

property voltage
Amplitude of the stepping voltage in volts from 0 to 150 V. This property can be set.

7.15 BK Precision

This section contains specific documentation on the BK Precision instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.15.1 BK Precision 9130B DC Power Supply

class pymeasure.instruments.bkprecision.**BKPrecision9130B**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the BK Precision 9130B DC Power Supply interface for interacting with the instrument.

property channel
An integer property used to control which channel is selected. Can only take values [1, 2, 3].

property current
Floating point property used to control current of the selected channel.

property source_enabled
A boolean property that controls whether the source is enabled, takes values True or False.

property voltage
Floating point property used to control voltage of the selected channel.

7.16 Danfysik

This section contains specific documentation on the Danfysik instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.16.1 Danfysik 8500 Power Supply

class pymeasure.instruments.danfysik.**Danfysik8500**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Danfysik 8500 Electromanget Current Supply and provides a high-level interface for interacting with the instrument

To allow user access to the Prolific Technology PL2303 Serial port adapter in Linux, create the file: `/etc/udev/rules.d/50-danfysik.rules`, with contents:

```
SUBSYSTEMS=="usb",ATTRS{idVendor}=="067b",ATTRS{idProduct}=="2303",MODE="0666",
↪SYMLINK+="danfysik"
```

Then reload the udev rules with:

```
sudo udevadm control --reload-rules
sudo udevadm trigger
```

The device will be accessible through the port `/dev/danfysik`.

add_ramp_step(*current*)

Adds a current step to the ramp set.

Parameters **current** – A current in Amps

clear_ramp_set()

Clears the ramp set.

clear_sequence(*stack*)

Clears the sequence by the stack number.

Parameters **stack** – A stack number between 0-15

property current

The actual current in Amps. This property can be set through *current_ppm*.

property current_ppm

The current in parts per million. This property can be set.

property current_setpoint

The setpoint for the current, which can deviate from the actual current (*current*) while the supply is in the process of setting the value.

disable()

Disables the flow of current.

enable()

Enables the flow of current.

property id

Reads the idenfitication information.

is_current_stable()

Returns True if the current is within 0.02 A of the setpoint value.

is_enabled()

Returns True if the current supply is enabled.

is_ready()

Returns True if the instrument is in the ready state.

is_sequence_running(*stack*)
Returns True if a sequence is running with a given stack number

Parameters **stack** – A stack number between 0-15

local()
Sets the instrument in local mode, where the front panel can be used.

property polarity
The polarity of the current supply, being either -1 or 1. This property can be set by suppling one of these values.

ramp_to_current(*current*, *points*, *delay_time=1*)
Executes *set_ramp_to_current()* and starts the ramp.

read()
Read the device and raise exceptions if errors are reported by the instrument.

Returns String ASCII response of the instrument

Raises An `Exception` if the Danfysik raises an error

remote()
Sets the instrument in remote mode, where the the front panel is disabled.

reset_interlocks()
Resets the instrument interlocks.

set_ramp_delay(*time*)
Sets the ramp delay time in seconds.

Parameters **time** – The time delay time in seconds

set_ramp_to_current(*current*, *points*, *delay_time=1*)
Sets up a linear ramp from the initial current to a different current, with a number of points, and delay time.

Parameters

- **current** – The final current in Amps
- **points** – The number of linear points to traverse
- **delay_time** – A delay time in seconds

set_sequence(*stack*, *currents*, *times*, *multiplier=999999*)
Sets up an arbitrary ramp profile with a list of currents (Amps) and a list of interval times (seconds) on the specified stack number (0-15)

property slew_rate
The slew rate of the current sweep.

start_ramp()
Starts the current ramp.

start_sequence(*stack*)
Starts a sequence by the stack number.

Parameters **stack** – A stack number between 0-15

property status
A list of human-readable strings that contain the instrument status information, based on *status_hex*.

property status_hex
The status in hexadecimal. This value is parsed in *status* into a human-readable list.

stop_ramp()
Stops the current ramp.

stop_sequence()
Stops the currently running sequence.

sync_sequence(*stack*, *delay=0*)
Arms the ramp sequence to be triggered by a hardware input to pin P33 1&2 (10 to 24 V) or a TS command. If a delay is provided, the sequence will start after the delay.

Parameters

- **stack** – A stack number between 0-15
- **delay** – A delay time in seconds

wait_for_current(*has_aborted=<function Danfysik8500.<lambda>>*, *delay=0.01*)
Blocks the process until the current has stabilized. A provided function has_aborted can be supplied, which is checked after each delay time (in seconds) in addition to the stability check. This allows an abort feature to be integrated.

Parameters

- **has_aborted** – A function that returns True if the process should stop waiting
- **delay** – The delay time in seconds between each check for stability

wait_for_ready(*has_aborted=<function Danfysik8500.<lambda>>*, *delay=0.01*)
Blocks the process until the instrument is ready. A provided function has_aborted can be supplied, which is checked after each delay time (in seconds) in addition to the readiness check. This allows an abort feature to be integrated.

Parameters

- **has_aborted** – A function that returns True if the process should stop waiting
- **delay** – The delay time in seconds between each check for readiness

7.17 Delta Elektronika

This section contains specific documentation on the Delta Elektronika instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.17.1 Delta Elektronica SM7045D Power source

class pymeasure.instruments.deltaelektronika.**SM7045D**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

This is the class for the SM 70-45 D power supply.

```
source = SM7045D("GPIB::8")

source.ramp_to_zero(1)              # Set output to 0 before enabling
source.enable()                     # Enables the output
source.current = 1                  # Sets a current of 1 Amps
```

property current
A floating point property that represents the output current of the power supply in Amps. This property can be set.

disable()
Enables remote shutdown, hence input will be disabled.

enable()
Disable remote shutdown, hence output will be enabled.

property max_current
A floating point property that represents the maximum output current of the power supply in Amps. This property can be set.

property max_voltage
A floating point property that represents the maximum output voltage of the power supply in Volts. This property can be set.

property measure_current
Measures the actual output current of the power supply in Amps.

property measure_voltage
Measures the actual output voltage of the power supply in Volts.

ramp_to_current(*target_current*, *current_step=0.1*)
Gradually increase/decrease current to target current.

Parameters

- **target_current** – Float that sets the target current (in A)
- **current_step** – Optional float that sets the current steps / ramp rate (in A/s)

ramp_to_zero(*current_step=0.1*)
Gradually decrease the current to zero.

Parameters **current_step** – Optional float that sets the current steps / ramp rate (in A/s)

property rsd
Check whether remote shutdown is enabled/disabled and thus if the output of the power supply is disabled/enabled.

shutdown()
Set the current to 0 A and disable the output of the power source.

property voltage
A floating point property that represents the output voltage setting of the power supply in Volts. This property can be set.

7.18 Edwards

This section contains specific documentation on the Edwards instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.18.1 Edwards nxds vacuum pump

pymeasure.instruments.edwards.**nxds**
alias of <module 'pymeasure.instruments.edwards.nxds' from '/home/docs/checkouts/readthedocs.org/user_builds/pymeasure/chec

7.19 EURO TEST

This section contains specific documentation on the EURO TEST instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.19.1 Euro Test HPP120256 High Voltage Power Supply

class pymeasure.instruments.eurotest.**EurotestHPP120256**(*adapter*, *query_delay=0.1*, *write_delay=0.4*, *timeout=5000*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Euro Test High Voltage DC Source model HPP-120-256 and provides a high-level interface for interacting with the instrument using the Euro Test command set (Not SCPI command set).

hpp120256 = EurotestHPP120256("GPIB0::20::INSTR")

print(hpp120256.id) print(hpp120256.lam_status) print(hpp120256.status)

hpp120256.ramp_to_zero(100.0)

hpp120256.voltage_ramp = 50.0 # V/s hpp120256.current_limit = 2.0 # mA inst.kill_enabled = True # Enable over-current protection time.sleep(1.0) # Give time to enable kill inst.output_enabled = True time.sleep(1.0) # Give time to output on

abs_output_voltage_error = 0.02 # kV

hpp120256.wait_for_output_voltage_reached(abs_output_voltage_error, 1.0, 40.0)

Here voltage HV output should be at 0.0 kV

print("Setting the output voltage to 1.0kV...") hpp120256.voltage_setpoint = 1.0 # kV

Now HV output should be rising to reach the 1.0kV at 50.0 V/s

hpp120256.wait_for_output_voltage_reached(abs_output_voltage_error, 1.0, 40.0)

Here voltage HV output should be at 1.0 kV

hpp120256.shutdown()

hpp120256.wait_for_output_voltage_reached(abs_output_voltage_error, 1.0, 60.0)

Here voltage HV output should be at 0.0 kV

inst.output_enabled = False

Now the HV voltage source is in safe state

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

class EurotestHPP120256Status(*value*)

Bases: `enum.IntFlag`

Auxiliary class create for translating the instrument 16bits_status_string into an Enum_IntFlag that will help to the user to understand such status.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

binary_values(*command*, *query_delay=0*, ***kwargs*)

Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

`check_errors()`
Read all errors from the instrument.

Returns list of error entries

`property complete`
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

`property current`
Measure the actual output current in mAmps (float).

`property current_limit`
Control the current limit in mAmps (float strictly from 0 to 25).

`property current_range`
Measure the actual output current range in mAmps (float).

`emergency_off()`
The output of the HV source will be switched OFF permanently and the values of the voltage and current settings set to zero

`property id`
Return the identification of the instrument (string)

`property kill_enabled`
Control the instrument kill enable (boolean).

`property lam_status`
Return the instrument lam status (string).

`property options`
Requests and returns the device options installed.

`property output_enabled`
Control the instrument output enable (boolean).

`ramp_to_zero`(*voltage_rate=200.0*)
Sets the voltage output setting to zero and the ramp setting to a value determined by the voltage_rate parameter. In summary, the method conducts (ramps) the voltage output to zero at a determinated voltage changing rate (ramp in V/s). :param voltage_rate: Is the changing rate (ramp in V/s) for the ramp setting

`read_binary_values`(***kwargs*)
Read binary values from the device.

`read_bytes`(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown(*voltage_rate=200.0*)
Change the output voltage setting (V) to zero and the ramp speed - voltage_rate (V/s) of the output voltage. After calling shutdown, if the HV voltage output > 0 it should drop to zero at a certain rate given by the voltage_rate parameter. :param voltage_rate: indicates the changing rate (V/s) of the voltage output

property status
Return the instrument status (EurotestHPP120256Status).

property voltage
Measure the actual output voltage in kVolts (float).

property voltage_ramp
Control the voltage ramp in Volts/second (int strictly from 10 to 3000).

property voltage_range
Measure the actual output voltage range in kVolts (float).

property voltage_setpoint
Control the voltage set-point in kVolts (float strictly from 0 to 12).

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

wait_for_output_voltage_reached(*voltage_setpoint*, *abs_output_voltage_error=0.03*, *check_period=1.0*, *timeout=60.0*)
Wait until HV voltage output reaches the voltage setpoint.

Checks the voltage output every check_period seconds and raises an exception if the voltage output doesn't reach the voltage setting until the timeout time. :param voltage_setpoint: the voltage in kVolts setted in the HV power supply which should be present at the output after some time (depends on the ramp setting). :param abs_output_voltage_error: absolute error in kVolts for being considered an output voltage reached. :param check_period: voltage output will be measured every check_period (seconds) time. :param timeout: time (seconds) give to the voltage output to reach the voltage setting. :return: None :raises: Exception if the voltage output can't reach the voltage setting before the timeout completes (seconds).

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.20 Fluke

This section contains specific documentation on the Fluke instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.20.1 Fluke 7341 Temperature bath

class pymeasure.instruments.fluke.**Fluke7341**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the compact constant temperature bath from Fluke

property id
Read the instrument model

property set_point
A *float* property to set the bath temperature set-point. Valid values are in the range -40 to 150 °C. The unit is as defined in property *unit*. This property can be read

property temperature
Read the current bath temperature. The unit is as defined in property *unit*.

property unit
A string property that controls the temperature unit. Possible values are *c* for Celsius and *f* for Fahrenheit`.

7.21 F.W. Bell

This section contains specific documentation on the F.W. Bell instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.21.1 F.W. Bell 5080 Handheld Gaussmeter

class pymeasure.instruments.fwbell.**FWBell5080**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the F.W. Bell 5080 Handheld Gaussmeter and provides a high-level interface for interacting with the instrument

Parameters **port** – The serial port of the instrument

```
meter = FWBell5080('/dev/ttyUSB0')  # Connects over serial port /dev/ttyUSB0 (Linux)

meter.units = 'gauss'               # Sets the measurement units to Gauss
meter.range = 1                     # Sets the range to 3 kG
print(meter.field)                  # Reads and prints a field measurement in G

fields = meter.fields(100)          # Samples 100 field measurements
print(fields.mean(), fields.std())  # Prints the mean and standard deviation of the␣
↪samples
```

auto_range()
Enables the auto range functionality.

property field
Reads a floating point value of the field in the appropriate units.

fields(*samples=1*)
Returns a numpy array of field samples for a given sample number.

Parameters **samples** – The number of samples to preform

property range
An integer property that controls the maximum field range in the active units. The range unit is dependent on the current units mode (gauss, tesla, amp-meter). Value sets an equivalent range across units that increases in magnitude (1, 10, 100).

Value	gauss	tesla	amp-meter
0	300 G	30 mT	23.88 kAm
1	3 kG	300 mT	238.8 kAm
2	30 kG	3 T	2388 kAm

read()
Overwrites the *Instrument.read* method to remove semicolons and replace spaces with colons.

reset()
Resets the instrument.

property units
A string property that controls the field units, which can take the values: 'gauss', 'gauss ac', 'tesla', 'tesla ac', 'amp-meter', and 'amp-meter ac'. The AC versions configure the instrument to measure AC.

7.22 Heidenhain

This section contains specific documentation on the Heidenhain instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.22.1 Heidenhain ND287 Position Display Unit

pymeasure.instruments.heidenhain.**nd287**
alias of <module 'pymeasure.instruments.heidenhain.nd287' from '/home/docs/checkouts/readthedocs.org/user_builds/pymeasure/

7.23 HC Photonics

This section contains specific documentation on the HC Photonics instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.23.1 HCP TC038 crystal oven

class pymeasure.instruments.hcp.**TC038**(*adapter*, *address=1*, *timeout=1000*, *includeSCPI=False*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Communication with the HCP TC038 oven.

This is the older version with an AC power supply and AC heater.

It has parity or framing errors from time to time. Handle them in your application.

The oven always responds with an "OK" to all valid requests or commands.

Parameters

- **adapter** (*str*) – Name of the COM-Port.
- **address** (*int*) – Address of the device. Should be between 1 and 99.
- **timeout** (*int*) – Timeout in ms.

check_errors()
Read the error from the instrument and return a list of errors.

property information
The information about the device and its capabilites.

property monitored_value
The currently monitored value. For default it is the current temperature in °C.

read()
Do error checking on reading.

set_monitored_quantity(*quantity='temperature'*)
Configure the oven to monitor a certain *quantity*.

quantity may be any key of *registers*. Default is the current temperature in °C.

property setpoint
The current setpoint of the temperature controller in °C.

property temperature
The currently measured temperature in °C.

write(*command*)
Send a *command* in its own protocol.

7.23.2 HCP TC038D crystal oven

class pymeasure.instruments.hcp.**TC038D**(*adapter*, *name='TC038D'*, *address=1*, *timeout=1000*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Communication with the HCP TC038D oven.

This is the newer version with DC heating.

The oven expects raw bytes written, no ascii code, and sends raw bytes. For the variables are two or four-byte modes available. We use the four-byte mode addresses. In that case element count has to be double the variables read.

check_errors()
To be called from the property setters to read the acknowledgment.

ping(*test_data=0*)
Test the connection sending an integer up to 65535, checks the response.

read()
Read response and interpret the number, returning it as a string.

property setpoint
The setpoint of the oven in °C.

property temperature
The current oven temperature in °C.

write(*command*)
Write a command to the device.

Parameters **command** (*str*) – comma separated string of: - the function: read ('R') or write ('W') or 'echo', - the address to write to (e.g. '0x106' or '262'), - the values (comma separated) to write - or the number of elements to read (defaults to 1).

7.24 Hewlett Packard

This section contains specific documentation on the Hewlett Packard instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.24.1 HP 33120A Arbitrary Waveform Generator

class pymeasure.instruments.hp.**HP33120A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Hewlett Packard 33120A Arbitrary Waveform Generator and provides a high-level interface for interacting with the instrument.

property amplitude
A floating point property that controls the voltage amplitude of the output signal. The default units are in peak-to-peak Volts, but can be controlled by *amplitude_units*. The allowed range depends on the waveform shape and can be queried with *max_amplitude* and *min_amplitude*.

property amplitude_units
A string property that controls the units of the amplitude, which can take the values Vpp, Vrms, dBm, and default.

beep()
Causes a system beep.

property frequency
A floating point property that controls the frequency of the output in Hz. The allowed range depends on the waveform shape and can be queried with *max_frequency* and *min_frequency*.

property max_amplitude
Reads the maximum *amplitude* in Volts for the given shape

property max_frequency
Reads the maximum *frequency* in Hz for the given shape

property max_offset
Reads the maximum *offset* in Volts for the given shape

property min_amplitude
Reads the minimum *amplitude* in Volts for the given shape

property min_frequency
Reads the minimum *frequency* in Hz for the given shape

property min_offset
Reads the minimum *offset* in Volts for the given shape

property offset
A floating point property that controls the amplitude voltage offset in Volts. The allowed range depends on the waveform shape and can be queried with *max_offset* and *min_offset*.

property shape
A string property that controls the shape of the wave, which can take the values: sinusoid, square, triangle, ramp, noise, dc, and user.

7.24.2 HP 34401A Multimeter

class pymeasure.instruments.hp.**HP34401A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the HP 34401A instrument.

property current_ac
AC current, in Amps

property current_dc
DC current, in Amps

property resistance
Resistance, in Ohms

property resistance_4w
Four-wires (remote sensing) resistance, in Ohms

property voltage_ac
AC voltage, in Volts

property voltage_dc
DC voltage, in Volts

7.24.3 HP 3437A System-Voltmeter

class pymeasure.instruments.hp.**HP3437A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.hp.hplegacyinstrument.HPLegacyInstrument*

Represents the Hewlett Packard 3737A system voltmeter and provides a high-level interface for interacting with the instrument.

class SRQ(*value*)
Bases: enum.IntFlag

Enum element for SRQ mask bit decoding

property SRQ_mask
Return current SRQ mask, this property can be set,

bit assigment for SRQ:

Bit (dec)	Description
1	SRQ when invalid program
2	SRQ when trigger is ignored
4	SRQ when data ready

check_errors()
As this instrument does not have a error indication bit, this function alwyas returns an empty list.

property delay
Return the value (float) for the delay between two measurements, this property can be set,

valid range: 100ns - 0.999999s

property number_readings
Return value (int) for the number of consecutive measurements, this property can be set, valid range: 0 - 9999

pb_desc
alias of `pymeasure.instruments.hp.hp3437A.PackedBits`

property range
Return the current measurement voltage range.

This property can be set, valid values: 0.1, 1, 10 (V).

Note: This instrument does not have autorange capability.

Overrange will be in indicated as 0.99,9.99 or 99.9

---

read_data()
Reads measured data from instrument, returns a np.array.

(This function also takes care of unpacking the data if required)

Return data np.array containing the data

status_desc
alias of `pymeasure.instruments.hp.hp3437A.Status`

property talk_ascii
A boolean property, True if the instrument is set to ASCII-based communication. This property can be set.

property trigger
Return current selected trigger mode, this property can be set,

Possibe values are:

Value	Explanation
internal	automatic trigger (internal)
external	external trigger (connector on back or GET)
hold/manual	holds the measurement/issues a manual trigger

7.24.4 HP 3478A Multimeter

class pymeasure.instruments.hp.**HP3478A**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.hp.hplegacyinstrument.HPLegacyInstrument*

Represents the Hewlett Packard 3478A 5 1/2 digit multimeter and provides a high-level interface for interacting with the instrument.

class ERRORS(*value*)

Bases: enum.IntFlag

Enum element for errror bit decoding

property SRQ_mask

Return current SRQ mask, this property can be set,

bit assigment for SRQ:

Bit (dec)	Description
1	SRQ when Data ready
4	SRQ when Syntax error
8	SRQ when internal error
16	front panel SQR button
32	SRQ by invalid calibration

property active_connectors

Return selected connectors (“front”/”back”), based on front-panel selector switch

property auto_range_enabled

Property describing the auto-ranging status

Value	Status
True	auto-range function activated
False	manual range selection / auto-range disabled

The range can be set with the *range* property

property auto_zero_enabled

Return auto-zero status, this property can be set

Value	Status
True	auto-zero active
False	auto-zero disabled

property calibration_data

Read or write the calibration data as an array of 256 values between 0 and 15.

The calibration data of an HP 3478A is stored in a 256x4 SRAM that is permanently powered by a 3v Lithium battery. When the battery runs out, the calibration data is lost, and recalibration is required.

When read, this property fetches and returns the calibration data so that it can be backed up.

When assigned a value, it similarly expects an array of 256 values between 0 and 15, and writes the values back to the instrument.

When writing, exceptions are raised for the following conditions:

- The CAL ENABLE switch at the front of the instrument is not set to ON.

- The array with values does not contain exactly 256 elements.
- The array with values does not pass a verification check.

IMPORTANT: changing the calibration data results in permanent loss of the previous data. Use with care!

property calibration_enabled
Return calibration enable switch setting, based on front-panel selector switch

Value	Status
True	calbration possible
False	calibration locked

check_errors()
Method to read the error status register

Return error_status one byte with the error status register content

Rtype error_status int

display_reset()
Reset the display of the instrument.

property display_text
Displays up to 12 upper-case ASCII characters on the display.

property display_text_no_symbol
Displays up to 12 upper-case ASCII characters on the display and disables all symbols on the display.

property error_status
Checks the error status register

property measure_ACI
Returns the measured value for AC current as a float in A.

property measure_ACV
Returns the measured value for AC Voltage as a float in V.

property measure_DCI
Returns the measured value for DC current as a float in A.

property measure_DCV
Returns the measured value for DC Voltage as a float in V.

property measure_R2W
Returns the measured value for 2-wire resistance as a float in Ohm.

property measure_R4W
Returns the measured value for 4-wire resistance as a float in Ohm.

property measure_Rext
Returns the measured value for extended resistance mode (>30M, 2-wire) resistance as a float in Ohm.

property mode
Return current selected measurement mode, this propery can be set. Allowed values are

Mode	Function
ACI	AC current
ACV	AC voltage
DCI	DC current
DCV	DC voltage
R2W	2-wire resistance
R4W	4-wire resistance
Rext	extended resistance method (requires additional 10 M resistor)

property range

Returns the current measurement range, this property can be set.

Valid values are :

Mode	Range
ACI	0.3, 3, auto
ACV	0.3, 3, 30, 300, auto
DCI	0.3, 3, auto
DCV	0.03, 0.3, 3, 30, 300, auto
R2W	30, 300, 3000, 3E4, 3E5, 3E6, 3E7, auto
R4W	30, 300, 3000, 3E4, 3E5, 3E6, 3E7, auto
Rext	3E7, auto

property resolution

Returns current selected resolution, this property can be set.

Possible values are 3,4 or 5 (for 3 1/2, 4 1/2 or 5 1/2 digits of resolution)

status_desc

alias of `pymeasure.instruments.hp.hp3478A.Status`

property trigger

Return current selected trigger mode, this property can be set

Possibe values are:

Value	Meaning
auto	automatic trigger (internal)
internal	automatic trigger (internal)
external	external trigger (connector on back or GET)
hold	holds the measurement
fast	fast trigger for AC measurements

verify_calibration_data(*cal_data*)

Verify the checksums of all calibration entries.

Expects an array of 256 values with calibration data.

Return calibration_correct True when all checksums are correct.

Rtype calibration_correct boolean

verify_calibration_entry(*cal_data*, *entry_nr*)

Verify the checksum of one calibration entry.

Expects an array of 256 values with calibration data, and an entry number from 0 to 18.

Returns True when the checksum of the specified calibration entry is correct.

write_calibration_data(*cal_data*, *verify_calibration_data=True*)
Method to write calibration data.

The cal_data parameter format is the same as the 'calibration_data' property.

Verification of the cal_data array can be bypassed by setting 'verify_calibration_data' to False.

7.24.5 HP 8116A 50 MHz Pulse/Function Generator

class pymeasure.instruments.hp.**HP8116A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Hewlett-Packard 8116A 50 MHz Pulse/Function Generator and provides a high-level interface for interacting with the instrument. The resolution for all floating point instrument parameters is 3 digits.

class Digit(*value*)
Bases: enum.Enum

Enum of the digits used with the autovernier (see *HP8116A.start_autovernier()*).

class Direction(*value*)
Bases: enum.Enum

Enum of the directions used with the autovernier (see *HP8116A.start_autovernier()*).

GPIB_trigger()
Initate trigger via low-level GPIB-command (aka GET - group execute trigger).

property amplitude
A floating point value that controls the amplitude of the output in V. The allowed amplitude range generally is 10 mV to 16 V, but it is also limited by the current offset.

ask(*command*, *num_bytes=None*)
Write a command to the instrument, read the response, and return the response as ASCII text.

Parameters

- **command** – The command to send to the instrument.
- **num_bytes** – The number of bytes to read from the instrument. If not specified, the number of bytes is automatically determined by the command.

property autovernier_enabled
A boolean property that controls whether the autovernier is enabled.

property burst_number
An integer value that controls the number of periods generated in a burst. The allowed range is 1 to 1999. It is only valid for units with Option 001 in one of the burst modes.

check_errors()
Check for errors in the 8116A.

Returns list of error entries or empty list if no error occurred.

property complement_enabled
A boolean property that controls whether the complement of the signal is generated.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property control_mode
A string property that controls the control mode of the instrument. Possible values are 'off', 'FM', 'AM', 'PWM', 'VCO'.

property duty_cycle
An integer value that controls the duty cycle of the output in percent. The allowed range generally is 10 % to 90 %, but it also depends on the current frequency. It is valid for all shapes except 'pulse', where *pulse_width* is used instead.

property frequency
A floating point value that controls the frequency of the output in Hz. The allowed frequency range is 1 mHz to 52.5 MHz.

property haversine_enabled
A boolean property that controls whether a haversine/havertriangle signal is generated when in 'triggered', 'internal_burst' or 'external_burst' operating mode.

property high_level
A floating point value that controls the high level of the output in V. The allowed high level range generally is -7.9 V to 8 V, but it must be at least 10 mV greater than the low level.

property limit_enabled
A boolean property that controls whether parameter limiting is enabled.

property low_level
A floating point value that controls the low level of the output in V. The allowed low level range generally is -8 V to 7.9 V, but it must be at least 10 mV less than the high level.

property offset
A floating point value that controls the offset of the output in V. The allowed offset range generally is -7.95 V to 7.95 V, but it is also limited by the amplitude.

property operating_mode
A string property that controls the operating mode of the instrument. Possible values (without Option 001) are: 'normal', 'triggered', 'gate', 'external_width'. With Option 001, 'internal_sweep', 'external_sweep', 'external_width', 'external_pulse' are also available.

property options
Return the device options installed. The only possible option is 001.

property output_enabled
A boolean property that controls whether the output is enabled.

property pulse_width
A floating point value that controls the pulse width. The allowed pulse width range is 8 ns to 999 ms. The pulse width may not be larger than the period.

property repetition_rate
A floating point value that controls the repetition rate (= the time between bursts) in 'internal_burst' mode. The allowed range is 20 ns to 999 ms.

reset()
Initatiate a reset (like a power-on reset) of the 8116A.

property shape
A string property that controls the shape of the output waveform. Possible values are: 'dc', 'sine', 'triangle', 'square', 'pulse'.

shutdown()
Gracefully close the connection to the 8116A.

start_autovernier(*control*, *digit*, *direction*, *start_value=None*)
Start the autovernier on the specified control.

Parameters

- **control** – The control to change, pass as `HP8116A.some_control`. Allowed controls are frequency, amplitude, offset, duty_cycle, and pulse_width
- **digit** – The digit to change, type: *HP8116A.Digit*.
- **direction** – The direction in which to change the control, type: *HP8116A.Direction*.
- **start_value** – An optional value to start the autovernier at. If not specified, the current value of the control is used.

property status
Returns the status byte of the 8116A as an IntFlag-type enum.

property sweep_marker_frequency
A floating point value that controls the frequency marker in both sweep modes. At this frequency, the marker output switches from low to high. The allowed range is 1 mHz to 52.5 MHz.

property sweep_start
A floating point value that controls the start frequency in both sweep modes. The allowed range is 1 mHz to 52.5 MHz.

property sweep_stop
A floating point value that controls the stop frequency in both sweep modes. The allowed range is 1 mHz to 52.5 MHz.

property sweep_time
A floating point value that controls the sweep time per decade in both sweep modes. The sweep time is selectable in a 1-2-5 sequence between 10 ms and 500 s.

property trigger_slope
A string property that controls the slope the trigger triggers on. Possible values are: 'off', 'positive', 'negative'.

write(*command*)
Write a command to the instrument and wait until the 8116A has interpreted it.

7.24.6 HP Signal generator HP8657B

Note:

- This instrument does not support reading back values, as it is a listen-only GPIB device.
- Other instruments of this family could be implemented using the dynamic ranges feature.
- Optional pulse modulation feature is not supported yet.

Glossary:

Abbreviation	Explanation
AM	Amplitude Modulation
FM	Frequency Modulation
dBm	power level in dB referenced to 1mW

class pymeasure.instruments.hp.**HP8657B**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Hewlett Packard 8657B signal generator and provides a high-level interface for interacting with the instrument.

class Modulation(*value*)
Bases: enum.IntEnum

IntEnum for the different modulation sources

property am_depth
Set the modulation depth for AM, usable range 0-99.9%

property am_source
Set the source for the AM function with *Modulation* enumeration.

Value	Meaning
OFF	no modulation active
INT_400HZ	internal 400 Hz modulation source
INT_1000HZ	internal 1000 Hz modulation source
EXTERNAL	External source, AC coupling

Note:

- AM & FM can be active at the same time
- only one internal source can be active at the time
- use "OFF" to deactivate AM

usage example:

```
sig_gen = HP8657B("GPIB::7")
...
sig_gen.am_source = sig_gen.Modulation.INT_400HZ     #  Enable int. 400 Hz␣
↪source for AM
sig_gen.am_depth = 50                                #  Set AM modulation depth␣
↪to 50%
...
sig_gen.am_source = sig_gen.Modulation.OFF           #  Turn AM off
```

check_errors()
Method to read the error status register as the 8657B does not support any readout of values, this will return 0 and log a warning

clear()
Reset the instrument to power-on default settings

property fm_deviation
Set the peak deviation in kHz for the FM function, useable range 0.1 - 400 kHz

***NOTE*:** the maximum usable deviation is depending on the output frequency, refer to the instrument documentation for further detail.

property fm_source
Set the source for the FM function with *Modulation* enumeration.

Value	Meaning
OFF	no modulation active
INT_400HZ	internal 400 Hz modulation source
INT_1000HZ	internal 1000 Hz modulation source
EXTERNAL	External source, AC coupling
DC_FM	External source, DC coupling (FM only)

Note:

- AM & FM can be active at the same time
- only one internal source can be active at the time
- use "OFF" to deactivate FM
- refer to the documentation rearding details on use of DC FM mode

usage example:

```
sig_gen = HP8657B("GPIB::7")
...
sig_gen.fm_source = sig_gen.Modulation.EXTERNAL     #  Enable external source␣
↪for FM
sig_gen.fm_deviation = 15                           #  Set FM peak deviation to␣
↪15 kHz
...
sig_gen.fm_source = sig_gen.Modulation.OFF          #  Turn FM off
```

property frequency
Set the output frequency of the instrument in Hz.

For the 8567B the valid range is 100 kHz to 2060 MHz.

id = 'HP,8657B,N/A,N/A'
Manual ID entry

property level
Set the output level in dBm.

For the 8657B the range is -143.5 to +17 dBm/

property level_offset
Set the output offset in dB, usable range -199 to +199 dB.

property output_enabled
Control whether the output is enabled.

reset()
Resets the instrument.

shutdown()
Brings the instrument to a safe and stable state

7.24.7 Support class for HP legacy devices

Currently this implementation is used for the following instruments which do not support SCPI:

- HP3437A System-Voltmeter
- HP3478A Digital Multimeter
- HP6632/33/34A System power supply

class pymeasure.instruments.hp.**HPLegacyInstrument**(*adapter*, *name='HP legacy instrument'*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Class for legacy HP instruments from the era before SPCI, based on *pymeasure.Instrument*

GPIB_trigger()
Initate trigger via low-level GPIB-command (aka GET - group execute trigger)

reset()
Initatiates a reset (like a power-on reset) of the HP3478A

shutdown()
provides a way to gracefully close the connection to the HP3478A

property status
Returns an object representing the current status of the unit.

status_desc
alias of pymeasure.instruments.hp.hplegacyinstrument.StatusBitsBase

values(*command*, ***kwargs*)
Write a command to the instrument and return a list of formatted values from the result.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string.
- **maxsplit** – At most *maxsplit* splits are done. -1 (default) indicates no limit.

Returns A list of the desired type, or strings where the casting fails

write(*command*)
Write a string command to the instrument appending *write_termination*.

Parameters

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

7.24.8 HP System Power Supplies HP663XA

Currently supported models are:

Model	Voltage	Current	Power
6632A	0..20 V	0..5.0 A	100 W
6633A	0..50 V	0..2.5 A	100 W
6634A	0..100 V	0..1.0 A	100 W

Note:

- The multi-channel system power supplies HP 6621A, 6622A, 6623A, 6624A, 6625A, 6626A, 6627A & 6628A share some of the command syntax and could probably be incorporated in this implementation
- The B-version of these models (6632B, 6633B & 6634B) are SPCI-compliant and could be implemented in a similiar manner

class pymeasure.instruments.hp.**HP6632A**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.hp.hplegacyinstrument.HPLegacyInstrument*

Represents the Hewlett Packard 6632A system power supply and provides a high-level interface for interacting with the instrument.

class ERRORS(*value*)

Bases: enum.Enum

Enum class for error messages

property OCP_enabled

A bool property which controls if the OCP (OverCurrent Protection) is enabled

property SRQ_enabled

A bool property which controls if the SRQ (ServiceReQuest) is enabled

class ST_ERRORS(*value*)

Bases: enum.Enum

Enum class for selftest errors

check_errors()

Method to read the error status register

Return error_status one byte with the error status register content

Rtype error_status int

check_selftest_errors()

Method to read the error status register

Return error_status one byte with the error status register content

Rtype error_status int

clear()

Resets the instrument to power-on default settings

property current

A floating point property that controls the output current of the device.

(dynamic)

property delay

A float propery that changes the reprogamming delay Default values: 8 ms in FAST mode 80 ms in NORM mode

Values will be rounded to the next 4 ms by the instrument

property display_active
A boot property which controls if the display is enabled

property id
Reads the ID of the instrument and returns this value for now

property output_enabled
A bool property which controls if the output is enabled

property over_voltage_limit
A floationg point property that sets the OVP threshold.

(dynamic)

reset_OVP_OCP()
Resets Overvoltage and Overcurrent protections

property rom_version
Reads the ROM id (software version) of the instrument and returns this value for now

property status
Returns an object representing the current status of the unit.

status_desc
alias of `pymeasure.instruments.hp.hpsystempsu.Status`

property voltage
A floating point proptery that controls the output voltage of the device.

(dynamic)

class pymeasure.instruments.hp.**HP6633A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.hp.hpsystempsu.HP6632A*

Represents the Hewlett Packard 6633A system power supply and provides a high-level interface for interacting with the instrument.

class pymeasure.instruments.hp.**HP6634A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.hp.hpsystempsu.HP6632A*

Represents the Hewlett Packard 6634A system power supply and provides a high-level interface for interacting with the instrument.

7.25 Keithley

This section contains specific documentation on the Keithley instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.25.1 Keithley 2000 Multimeter

class pymeasure.instruments.keithley.**Keithley2000**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*, pymeasure.instruments.keithley.buffer.KeithleyBuffer

Represents the Keithley 2000 Multimeter and provides a high-level interface for interacting with the instrument.

```
meter = Keithley2000("GPIB::1")
meter.measure_voltage()
print(meter.voltage)
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)

Bases: object

Add channels to the parent class.

The children will be added to the parent instance at instantiation with CommonBase.add_child(). The variable name (e.g. channels) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use channels as variable and leave the prefix at the default "ch_".

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

acquire_reference(*mode=None*)

Sets the active value as the reference for the active mode, or can set another mode by its name.

Parameters **mode** – A valid *mode* name, or None for the active mode

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: instrument.channels["F"] == instrument.ch_F

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

auto_range(*mode=None*)
Sets the active mode to use auto-range, or can set another mode by its name.

Parameters **mode** – A valid *mode* name, or None for the active mode

beep(*frequency*, *duration*)
Sounds a system beep.

Parameters

- **frequency** – A frequency in Hz between 65 Hz and 2 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

property beep_state
A string property that enables or disables the system status beeper, which can take the values: :code:'enabled' and :code:'disabled'.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

property buffer_data
Returns a numpy array of values from the buffer.

property buffer_points
An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

check_errors()
Read all errors from the instrument.

Returns list of error entries

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

config_buffer(*points=64*, *delay=0*)
Configures the measurement buffer for a number of points, to be taken with a specified delay.

Parameters

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

property current
Reads a DC or AC current measurement in Amps, based on the active *mode*.

property current_ac_bandwidth
A floating point property that sets the AC current detector bandwidth in Hz, which can take the values 3, 30, and 300 Hz.

property current_ac_digits
An integer property that controls the number of digits in the AC current readings, which can take values from 4 to 7.

property current_ac_nplc
A floating point property that controls the number of power line cycles (NPLC) for the AC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property current_ac_range
A floating point property that controls the AC current range in Amps, which can take values from 0 to 3.1 A. Auto-range is disabled when this property is set.

property current_ac_reference
A floating point property that controls the AC current reference value in Amps, which can take values from -3.1 to 3.1 A.

property current_digits
An integer property that controls the number of digits in the DC current readings, which can take values from 4 to 7.

property current_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property current_range
A floating point property that controls the DC current range in Amps, which can take values from 0 to 3.1 A. Auto-range is disabled when this property is set.

property current_reference
A floating point property that controls the DC current reference value in Amps, which can take values from -3.1 to 3.1 A.

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process.

disable_filter(*mode=None*)
Disables the averaging filter for the active mode, or can set another mode by its name.

Parameters **mode** – A valid *mode* name, or None for the active mode

disable_reference(*mode=None*)
Disables the reference for the active mode, or can set another mode by its name.

Parameters **mode** – A valid *mode* name, or None for the active mode

enable_filter(*mode=None*, *type='repeat'*, *count=1*)
Enables the averaging filter for the active mode, or can set another mode by its name.

Parameters

- **mode** – A valid *mode* name, or None for the active mode
- **type** – The type of averaging filter, either 'repeat' or 'moving'.
- **count** – A number of averages, which can take take values from 1 to 100

enable_reference(*mode=None*)
Enables the reference for the active mode, or can set another mode by its name.

Parameters **mode** – A valid *mode* name, or None for the active mode

property frequency
Reads a frequency measurement in Hz, based on the active *mode*.

property frequency_aperature
A floating point property that controls the frequency aperature in seconds, which sets the integration period and measurement speed. Takes values from 0.01 to 1.0 s.

property frequency_digits
An integer property that controls the number of digits in the frequency readings, which can take values from 4 to 7.

property frequency_reference
A floating point property that controls the frequency reference value in Hz, which can take values from 0 to 15 MHz.

property frequency_threshold
A floating point property that controls the voltage signal threshold level in Volts for the frequency measurement, which can take values from 0 to 1010 V.

property id
Requests and returns the identification of the instrument.

is_buffer_full()
Returns True if the buffer is full of measurements.

local()
Returns control to the instrument panel, and enables the panel if disabled.

measure_continuity()
Configures the instrument to perform continuity testing.

measure_current(*max_current=0.01*, *ac=False*)
Configures the instrument to measure current, based on a maximum current to set the range, and a boolean flag to determine if DC or AC is required.

Parameters

- **max_current** – A current in Volts to set the current range
- **ac** – False for DC current, and True for AC current

measure_diode()
Configures the instrument to perform diode testing.

measure_frequency()
Configures the instrument to measure the frequency.

measure_period()
Configures the instrument to measure the period.

measure_resistance(*max_resistance=10000000.0*, *wires=2*)
Configures the instrument to measure voltage, based on a maximum voltage to set the range, and a boolean flag to determine if DC or AC is required.

Parameters

- **max_voltage** – A voltage in Volts to set the voltage range
- **ac** – False for DC voltage, and True for AC voltage

measure_temperature()
Configures the instrument to measure the temperature.

measure_voltage(*max_voltage=1*, *ac=False*)
Configures the instrument to measure voltage, based on a maximum voltage to set the range, and a boolean flag to determine if DC or AC is required.

Parameters

- **max_voltage** – A voltage in Volts to set the voltage range
- **ac** – False for DC voltage, and True for AC voltage

property mode
A string property that controls the configuration mode for measurements, which can take the values: :code:'current' (DC), :code:'current ac', :code:'voltage' (DC), :code:'voltage ac', :code:'resistance' (2-wire), :code:'resistance 4W' (4-wire), :code:'period', :code:'frequency', :code:'temperature', :code:'diode', and :code:'frequency'.

property options
Requests and returns the device options installed.

property period
Reads a period measurement in seconds, based on the active *mode*.

property period_aperature
A floating point property that controls the period aperature in seconds, which sets the integration period and measurement speed. Takes values from 0.01 to 1.0 s.

property period_digits
An integer property that controls the number of digits in the period readings, which can take values from 4 to 7.

property period_reference
A floating point property that controls the period reference value in seconds, which can take values from 0 to 1 s.

property period_threshold
A floating point property that controls the voltage signal threshold level in Volts for the period measurement, which can take values from 0 to 1010 V.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remote()
Places the instrument in the remote state, which is does not need to be explicity called in general.

remote_lock()
Disables and locks the front panel controls to prevent changes during remote operations. This is disabled by calling *local()*.

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument state.

reset_buffer()
Resets the buffer.

property resistance
Reads a resistance measurement in Ohms for both 2-wire and 4-wire configurations, based on the active *mode*.

property resistance_4W_digits
An integer property that controls the number of digits in the 4-wire resistance readings, which can take values from 4 to 7.

property resistance_4W_nplc
A floating point property that controls the number of power line cycles (NPLC) for the 4-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property resistance_4W_range
A floating point property that controls the 4-wire resistance range in Ohms, which can take values from 0 to 120 MOhms. Auto-range is disabled when this property is set.

property resistance_4W_reference
A floating point property that controls the 4-wire resistance reference value in Ohms, which can take values from 0 to 120 MOhms.

property resistance_digits
An integer property that controls the number of digits in the 2-wire resistance readings, which can take values from 4 to 7.

property resistance_nplc
A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property resistance_range
A floating point property that controls the 2-wire resistance range in Ohms, which can take values from 0 to 120 MOhms. Auto-range is disabled when this property is set.

property resistance_reference
A floating point property that controls the 2-wire resistance reference value in Ohms, which can take values from 0 to 120 MOhms.

shutdown()
Brings the instrument to a safe and stable state

start_buffer()
Starts the buffer.

property status
Requests and returns the status byte and Master Summary Status bit.

stop_buffer()
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

property temperature
Reads a temperature measurement in Celsius, based on the active *mode*.

property temperature_digits
An integer property that controls the number of digits in the temperature readings, which can take values from 4 to 7.

property temperature_nplc
A floating point property that controls the number of power line cycles (NPLC) for the temperature measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property temperature_reference
A floating point property that controls the temperature reference value in Celsius, which can take values from -200 to 1372 C.

property trigger_count
An integer property that controls the trigger count, which can take values from 1 to 9,999.

property trigger_delay
A floating point property that controls the trigger delay in seconds, which can take values from 1 to 9,999,999.999 s.

property voltage
Reads a DC or AC voltage measurement in Volts, based on the active *mode*.

property voltage_ac_bandwidth
A floating point property that sets the AC voltage detector bandwidth in Hz, which can take the values 3, 30, and 300 Hz.

property voltage_ac_digits
An integer property that controls the number of digits in the AC voltage readings, which can take values from 4 to 7.

property voltage_ac_nplc
A floating point property that controls the number of power line cycles (NPLC) for the AC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property voltage_ac_range
A floating point property that controls the AC voltage range in Volts, which can take values from 0 to 757.5 V. Auto-range is disabled when this property is set.

property voltage_ac_reference
A floating point property that controls the AC voltage reference value in Volts, which can take values from -757.5 to 757.5 Volts.

property voltage_digits
An integer property that controls the number of digits in the DC voltage readings, which can take values from 4 to 7.

property voltage_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property voltage_range
A floating point property that controls the DC voltage range in Volts, which can take values from 0 to 1010 V. Auto-range is disabled when this property is set.

property voltage_reference
A floating point property that controls the DC voltage reference value in Volts, which can take values from -1010 to 1010 V.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)
Blocks the program, waiting for a full buffer. This function returns early if the should_stop function returns True or the timeout is reached before the buffer is full.

Parameters

- **should_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.2 Keithley 2260B DC Power Supply

class pymeasure.instruments.keithley.**Keithley2260B**(*adapter*, *read_termination='\n'*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Keithley 2260B Power Supply (minimal implementation) and provides a high-level interface for interacting with the instrument.

For a connection through tcpip, the device only accepts connections at port 2268, which cannot be configured otherwise. example connection string: 'TCPIP::xxx.xxx.xxx.xxx::2268::SOCKET' the read termination for this interface is

```
source = Keithley2260B("GPIB::1")
source.voltage = 1
print(source.voltage)
print(source.current)
```

(continues on next page)

(continued from previous page)

```
print(source.power)
print(source.applied)
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)

Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

property applied
Simultaneous control of voltage (volts) and current (amps). Values need to be supplied as tuple of (voltage, current). Depending on whether the instrument is in constant current or constant voltage mode, the values achieved by the instrument will differ from the ones set.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

check_errors()
Logs any system errors reported by the instrument.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property current
Reads the current (in Ampere) the dc power supply is putting out.

property current_limit
A floating point property that controls the source current in amps. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual current achieved.

property error
Returns a tuple of an error code and message from a single error.

property id
Requests and returns the identification of the instrument.

property options
Requests and returns the device options installed.

property output_enabled
A boolean property that controls whether the source is enabled, takes values True or False.

property power
Reads the power (in Watt) the dc power supply is putting out.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown()
Disable output, call parent function

property status
Requests and returns the status byte and Master Summary Status bit.

property voltage
Reads the voltage (in Volt) the dc power supply is putting out.

property voltage_setpoint
A floating point property that controls the source voltage in volts. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual voltage achieved.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.3 Keithley 2306 Dual Channel Battery/Charger Simulator

class pymeasure.instruments.keithley.**Keithley2306**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Keithley 2306 Dual Channel Battery/Charger Simulator.

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
```

(continues on next page)

(continued from previous page)

```
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

binary_values(*command*, *query_delay=0*, ***kwargs*)

Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

property both_channels_enabled

A boolean setting that controls whether both channel outputs are enabled, takes values of True or False.

ch(*channel_number*)

Get a channel from this instrument.

Param channel_number: int: the number of the channel to be selected

Type *Channel*

check_errors()
Read all errors from the instrument.

Returns list of error entries

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property display_brightness
A floating point property that controls the display brightness, takes values beteween 0.0 and 1.0. A blank display is 0.0, 1/4 brightness is for values less or equal to 0.25, otherwise 1/2 brightness for values less than or equal to 0.5, otherwise 3/4 brightness for values less than or equal to 0.75, otherwise full brightness.

property display_channel
An integer property that controls the display channel, takes values 1 or 2.

property display_enabled
A boolean property that controls whether the display is enabled, takes values True or False.

property display_text_data
A string property that control text to be displayed, takes strings up to 32 characters.

property display_text_enabled
A boolean property that controls whether display text is enabled, takes values True or False.

property id
Requests and returns the identification of the instrument.

property options
Requests and returns the device options installed.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

relay(*relay_number*)
Get a relay channel from this instrument.

Param relay_number: int: the number of the relay to be selected

Type Relay

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown()
Brings the instrument to a safe and stable state

property status
Requests and returns the status byte and Master Summary Status bit.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.4 Keithley 2400 SourceMeter

class pymeasure.instruments.keithley.**Keithley2400**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*, pymeasure.instruments.keithley.buffer.KeithleyBuffer

Represents the Keithely 2400 SourceMeter and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley2400("GPIB::1")

keithley.apply_current()                # Sets up to source current
keithley.source_current_range = 10e-3   # Sets the source current range to 10 mA
keithley.compliance_voltage = 10        # Sets the compliance voltage to 10 V
keithley.source_current = 0             # Sets the source current to 0 mA
keithley.enable_source()                # Enables the source output

keithley.measure_voltage()              # Sets up to measure voltage

keithley.ramp_to_current(5e-3)          # Ramps the current to 5 mA
print(keithley.voltage)                 # Prints the voltage in Volts

keithley.shutdown()                     # Ramps the current to 0 mA and disables␣
↪output
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: object

Add channels to the parent class.

The children will be added to the parent instance at instantiation with CommonBase.add_child(). The variable name (e.g. channels) will be used as the *collection* of the children. You may define the attribute

prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

apply_current(*current_range=None*, *compliance_voltage=0.1*)

Configures the instrument to apply a source current, and uses an auto range unless a current range is specified. The compliance voltage is also set.

Parameters

- **compliance_voltage** – A float in the correct range for a *compliance_voltage*
- **current_range** – A *current_range* value or None

apply_voltage(*voltage_range=None*, *compliance_current=0.1*)
Configures the instrument to apply a source voltage, and uses an auto range unless a voltage range is specified. The compliance current is also set.

Parameters

- **compliance_current** – A float in the correct range for a *compliance_current*
- **voltage_range** – A *voltage_range* value or None

property auto_output_off
A boolean property that enables or disables the auto output-off. Valid values are True (output off after measurement) and False (output stays on after measurement).

auto_range_source()
Configures the source to use an automatic range.

property auto_zero
A property that controls the auto zero option. Valid values are True (enabled) and False (disabled) and 'ONCE' (force immediate).

beep(*frequency*, *duration*)
Sounds a system beep.

Parameters

- **frequency** – A frequency in Hz between 65 Hz and 2 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

property buffer_data
Returns a numpy array of values from the buffer.

property buffer_points
An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

check_errors()
Logs any system errors reported by the instrument.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property compliance_current
A floating point property that controls the compliance current in Amps.

property compliance_voltage
A floating point property that controls the compliance voltage in Volts.

config_buffer(*points=64*, *delay=0*)
Configures the measurement buffer for a number of points, to be taken with a specified delay.

Parameters

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

property current
Reads the current in Amps, if configured for this reading.

property current_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property current_range
A floating point property that controls the measurement current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process.

disable_output_trigger()
Disables the output trigger for the Trigger layer

disable_source()
Disables the source of current or voltage depending on the configuration of the instrument.

property display_enabled
A boolean property that controls whether or not the display of the sourcemeter is enabled. Valid values are True and False.

enable_source()
Enables the source of current or voltage depending on the configuration of the instrument.

property error
Returns a tuple of an error code and message from a single error.

property filter_count
A integer property that controls the number of readings that are acquired and stored in the filter buffer for the averaging

property filter_state
A string property that controls if the filter is active.

property filter_type
A String property that controls the filter's type. REP : Repeating filter MOV : Moving filter

property id
Requests and returns the identification of the instrument.

is_buffer_full()
Returns True if the buffer is full of measurements.

property line_frequency
An integer property that controls the line frequency in Hertz. Valid values are 50 and 60.

property line_frequency_auto
A boolean property that enables or disables auto line frequency. Valid values are True and False.

property max_current
Returns the maximum current from the buffer

property max_resistance
Returns the maximum resistance from the buffer

property max_voltage
Returns the maximum voltage from the buffer

property maximums
Returns the calculated maximums for voltage, current, and resistance from the buffer data as a list.

property mean_current
Returns the mean current from the buffer

property mean_resistance
Returns the mean resistance from the buffer

property mean_voltage
Returns the mean voltage from the buffer

property means
Reads the calculated means (averages) for voltage, current, and resistance from the buffer data as a list.

property measure_concurent_functions
A boolean property that enables or disables the ability to measure more than one function simultaneously. When disabled, volts function is enabled. Valid values are True and False.

measure_current(*nplc=1*, *current=0.000105*, *auto_range=True*)
Configures the measurement of current.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **current** – Upper limit of current in Amps, from -1.05 A to 1.05 A
- **auto_range** – Enables auto_range if True, else uses the set current

measure_resistance(*nplc=1*, *resistance=210000.0*, *auto_range=True*)
Configures the measurement of resistance.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **resistance** – Upper limit of resistance in Ohms, from -210 MOhms to 210 MOhms
- **auto_range** – Enables auto_range if True, else uses the set resistance

measure_voltage(*nplc=1*, *voltage=21.0*, *auto_range=True*)
Configures the measurement of voltage.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **voltage** – Upper limit of voltage in Volts, from -210 V to 210 V
- **auto_range** – Enables auto_range if True, else uses the set voltage

property min_current
Returns the minimum current from the buffer

property min_resistance
Returns the minimum resistance from the buffer

property min_voltage
Returns the minimum voltage from the buffer

property minimums
Returns the calculated minimums for voltage, current, and resistance from the buffer data as a list.

property options
Requests and returns the device options installed.

property output_off_state
Select the output-off state of the SourceMeter. HIMP : output relay is open, disconnects external circuitry. NORM : V-Source is selected and set to 0V, Compliance is set to 0.5% full scale of the present current range. ZERO : V-Source is selected and set to 0V, compliance is set to the programmed Source I value or to 0.5% full scale of the present current range, whichever is greater. GUAR : I-Source is selected and set to 0A

output_trigger_on_external(*line=1*, *after='DEL'*)
Configures the output trigger on the specified trigger link line number, with the option of supplying the part of the measurement after which the trigger should be generated (default to delay, which is right before the measurement)

Parameters

- **line** – A trigger line from 1 to 4
- **after** – An event string that determines when to trigger

ramp_to_current(*target_current*, *steps=30*, *pause=0.02*)
Ramps to a target current from the set current value over a certain number of linear steps, each separated by a pause duration.

Parameters

- **target_current** – A current in Amps
- **steps** – An integer number of steps
- **pause** – A pause duration in seconds to wait between steps

ramp_to_voltage(*target_voltage*, *steps=30*, *pause=0.02*)
Ramps to a target voltage from the set voltage value over a certain number of linear steps, each separated by a pause duration.

Parameters

- **target_voltage** – A voltage in Amps
- **steps** – An integer number of steps
- **pause** – A pause duration in seconds to wait between steps

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters `child` – Instance of the child to delete.

`reset()`
Resets the instrument and clears the queue.

`reset_buffer()`
Resets the buffer.

`property resistance`
Reads the resistance in Ohms, if configured for this reading.

`property resistance_nplc`
A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

`property resistance_range`
A floating point property that controls the resistance range in Ohms, which can take values from 0 to 210 MOhms. Auto-range is disabled when this property is set.

`sample_continuously()`
Causes the instrument to continuously read samples and turns off any buffer or output triggering

`set_timed_arm(`*interval*`)`
Sets up the measurement to be taken with the internal trigger at a variable sampling rate defined by the interval in seconds between sampling points

`set_trigger_counts(`*arm*, *trigger*`)`
Sets the number of counts for both the sweeps (arm) and the points in those sweeps (trigger), where the total number of points can not exceed 2500

`shutdown()`
Ensures that the current or voltage is turned to zero and disables the output.

`property source_current`
A floating point property that controls the source current in Amps.

`property source_current_range`
A floating point property that controls the source current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

`property source_delay`
A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 0 [seconds] and 999.9999 [seconds].

`property source_delay_auto`
A boolean property that enables or disables auto delay. Valid values are True and False.

`property source_enabled`
A boolean property that controls whether the source is enabled, takes values True or False. The convenience methods *enable_source()* and *disable_source()* can also be used.

`property source_mode`
A string property that controls the source mode, which can take the values 'current' or 'voltage'. The convenience methods *apply_current()* and *apply_voltage()* can also be used.

`property source_voltage`
A floating point property that controls the source voltage in Volts.

property source_voltage_range
A floating point property that controls the source voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

property standard_devs
Returns the calculated standard deviations for voltage, current, and resistance from the buffer data as a list.

start_buffer()
Starts the buffer.

status()
Requests and returns the status byte and Master Summary Status bit.

property std_current
Returns the current standard deviation from the buffer

property std_resistance
Returns the resistance standard deviation from the buffer

property std_voltage
Returns the voltage standard deviation from the buffer

stop_buffer()
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

triad(*base_frequency*, *duration*)
Sounds a musical triad using the system beep.

Parameters

- **base_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

trigger()
Executes a bus trigger, which can be used when *trigger_on_bus()* is configured.

property trigger_count
An integer property that controls the trigger count, which can take values from 1 to 9,999.

property trigger_delay
A floating point property that controls the trigger delay in seconds, which can take values from 0 to 999.9999 s.

trigger_immediately()
Configures measurements to be taken with the internal trigger at the maximum sampling rate.

trigger_on_bus()
Configures the trigger to detect events based on the bus trigger, which can be activated by *trigger()*.

trigger_on_external(*line=1*)
Configures the measurement trigger to be taken from a specific line of an external trigger

Parameters **line** – A trigger line from 1 to 4

use_front_terminals()
Enables the front terminals for measurement, and disables the rear terminals.

use_rear_terminals()
Enables the rear terminals for measurement, and disables the front terminals.

property voltage
Reads the voltage in Volts, if configured for this reading.

property voltage_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property voltage_range
A floating point property that controls the measurement voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)
Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

Parameters

- **should_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

property wires
An integer property that controls the number of wires in use for resistance measurements, which can take the value of 2 or 4.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.5 Keithley 2450 SourceMeter

class pymeasure.instruments.keithley.**Keithley2450**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*, pymeasure.instruments.keithley.buffer.KeithleyBuffer

Represents the Keithely 2450 SourceMeter and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley2450("GPIB::1")

keithley.apply_current()                # Sets up to source current
keithley.source_current_range = 10e-3   # Sets the source current range to 10 mA
keithley.compliance_voltage = 10        # Sets the compliance voltage to 10 V
keithley.source_current = 0             # Sets the source current to 0 mA
keithley.enable_source()                # Enables the source output
```

(continues on next page)

(continued from previous page)

```
keithley.measure_voltage()              # Sets up to measure voltage

keithley.ramp_to_current(5e-3)          # Ramps the current to 5 mA
print(keithley.voltage)                 # Prints the voltage in Volts

keithley.shutdown()                     # Ramps the current to 0 mA and disables␣
↪output
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)

Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.

- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

apply_current(*current_range=None*, *compliance_voltage=0.1*)
Configures the instrument to apply a source current, and uses an auto range unless a current range is specified. The compliance voltage is also set.

Parameters

- **compliance_voltage** – A float in the correct range for a *compliance_voltage*
- **current_range** – A *current_range* value or None

apply_voltage(*voltage_range=None*, *compliance_current=0.1*)
Configures the instrument to apply a source voltage, and uses an auto range unless a voltage range is specified. The compliance current is also set.

Parameters

- **compliance_current** – A float in the correct range for a *compliance_current*
- **voltage_range** – A *voltage_range* value or None

auto_range_source()
Configures the source to use an automatic range.

beep(*frequency*, *duration*)
Sounds a system beep.

Parameters

- **frequency** – A frequency in Hz between 65 Hz and 2 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

property buffer_data
Returns a numpy array of values from the buffer.

property buffer_points
An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

check_errors()
Logs any system errors reported by the instrument.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property compliance_current
A floating point property that controls the compliance current in Amps.

property compliance_voltage
A floating point property that controls the compliance voltage in Volts.

config_buffer(*points=64*, *delay=0*)
Configures the measurement buffer for a number of points, to be taken with a specified delay.

Parameters

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

property current
Reads the current in Amps, if configured for this reading.

property current_filter_count
A integer property that controls the number of readings that are acquired and stored in the filter buffer for the averaging

property current_filter_state
A string property that controls if the filter is active.

property current_filter_type
A String property that controls the filter's type for the current. REP : Repeating filter MOV : Moving filter

property current_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property current_output_off_state
Select the output-off state of the SourceMeter. HIMP : output relay is open, disconnects external circuitry. NORM : V-Source is selected and set to 0V, Compliance is set to 0.5% full scale of the present current range. ZERO : V-Source is selected and set to 0V, compliance is set to the programmed Source I value or to 0.5% full scale of the present current range, whichever is greater. GUAR : I-Source is selected and set to 0A

property current_range
A floating point property that controls the measurement current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process.

disable_source()
Disables the source of current or voltage depending on the configuration of the instrument.

enable_source()
Enables the source of current or voltage depending on the configuration of the instrument.

property error
Returns a tuple of an error code and message from a single error.

property id
Requests and returns the identification of the instrument.

is_buffer_full()
Returns True if the buffer is full of measurements.

property max_current
Returns the maximum current from the buffer

property max_resistance
Returns the maximum resistance from the buffer

property max_voltage
Returns the maximum voltage from the buffer

property maximums
Returns the calculated maximums for voltage, current, and resistance from the buffer data as a list.

property mean_current
Returns the mean current from the buffer

property mean_resistance
Returns the mean resistance from the buffer

property mean_voltage
Returns the mean voltage from the buffer

property means
Reads the calculated means (averages) for voltage, current, and resistance from the buffer data as a list.

measure_current(*nplc=1*, *current=0.000105*, *auto_range=True*)
Configures the measurement of current.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **current** – Upper limit of current in Amps, from -1.05 A to 1.05 A
- **auto_range** – Enables auto_range if True, else uses the set current

measure_resistance(*nplc=1*, *resistance=210000.0*, *auto_range=True*)
Configures the measurement of resistance.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **resistance** – Upper limit of resistance in Ohms, from -210 MOhms to 210 MOhms
- **auto_range** – Enables auto_range if True, else uses the set resistance

measure_voltage(*nplc=1*, *voltage=21.0*, *auto_range=True*)
Configures the measurement of voltage.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **voltage** – Upper limit of voltage in Volts, from -210 V to 210 V
- **auto_range** – Enables auto_range if True, else uses the set voltage

property min_current
Returns the minimum current from the buffer

property min_resistance
Returns the minimum resistance from the buffer

property min_voltage
Returns the minimum voltage from the buffer

property minimums
Returns the calculated minimums for voltage, current, and resistance from the buffer data as a list.

property options
Requests and returns the device options installed.

ramp_to_current(*target_current*, *steps=30*, *pause=0.02*)
Ramps to a target current from the set current value over a certain number of linear steps, each separated by a pause duration.

Parameters

- **target_current** – A current in Amps
- **steps** – An integer number of steps
- **pause** – A pause duration in seconds to wait between steps

ramp_to_voltage(*target_voltage*, *steps=30*, *pause=0.02*)
Ramps to a target voltage from the set voltage value over a certain number of linear steps, each separated by a pause duration.

Parameters

- **target_voltage** – A voltage in Amps
- **steps** – An integer number of steps
- **pause** – A pause duration in seconds to wait between steps

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument and clears the queue.

reset_buffer()
Resets the buffer.

property resistance
Reads the resistance in Ohms, if configured for this reading.

property resistance_nplc
A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property resistance_range
A floating point property that controls the resistance range in Ohms, which can take values from 0 to 210 MOhms. Auto-range is disabled when this property is set.

shutdown()
Ensures that the current or voltage is turned to zero and disables the output.

property source_current
A floating point property that controls the source current in Amps.

property source_current_delay
A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 0 [seconds] and 999.9999 [seconds].

property source_current_delay_auto
A boolean property that enables or disables auto delay. Valid values are True and False.

property source_current_range
A floating point property that controls the source current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

property source_enabled
Reads a boolean value that is True if the source is enabled.

property source_mode
A string property that controls the source mode, which can take the values 'current' or 'voltage'. The convenience methods *apply_current()* and *apply_voltage()* can also be used.

property source_voltage
A floating point property that controls the source voltage in Volts.

property source_voltage_delay
A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 0 [seconds] and 999.9999 [seconds].

property source_voltage_delay_auto
A boolean property that enables or disables auto delay. Valid values are True and False.

property source_voltage_range
A floating point property that controls the source voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

property standard_devs
Returns the calculated standard deviations for voltage, current, and resistance from the buffer data as a list.

start_buffer()
Starts the buffer.

property status
Requests and returns the status byte and Master Summary Status bit.

property std_current
Returns the current standard deviation from the buffer

property std_resistance
Returns the resistance standard deviation from the buffer

property std_voltage
Returns the voltage standard deviation from the buffer

stop_buffer()
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

triad(*base_frequency*, *duration*)
Sounds a musical triad using the system beep.

Parameters

- **base_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

trigger()
Executes a bus trigger.

use_front_terminals()
Enables the front terminals for measurement, and disables the rear terminals.

use_rear_terminals()
Enables the rear terminals for measurement, and disables the front terminals.

property voltage
Reads the voltage in Volts, if configured for this reading.

property voltage_filter_count
A integer property that controls the number of readings that are acquired and stored in the filter buffer for the averaging

property voltage_filter_type
A String property that controls the filter's type for the current. REP : Repeating filter MOV : Moving filter

property voltage_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property voltage_output_off_state
Select the output-off state of the SourceMeter. HIMP : output relay is open, disconnects external circuitry. NORM : V-Source is selected and set to 0V, Compliance is set to 0.5% full scale of the present current range. ZERO : V-Source is selected and set to 0V, compliance is set to the programmed Source I value or to 0.5% full scale of the present current range, whichever is greater. GUAR : I-Source is selected and set to 0A

property voltage_range
A floating point property that controls the measurement voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)
Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

Parameters

- **should_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

property wires
An integer property that controls the number of wires in use for resistance measurements, which can take the value of 2 or 4.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.6 Keithley 2700 MultiMeter/Switch System

class pymeasure.instruments.keithley.**Keithley2700**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*, pymeasure.instruments.keithley.buffer.KeithleyBuffer

Represents the Keithely 2700 Multimeter/Switch System and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley2700("GPIB::1")
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: object

Add channels to the parent class.

The children will be added to the parent instance at instantiation with CommonBase.add_child(). The variable name (e.g. channels) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use channels as variable and leave the prefix at the default "ch_".

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)
Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

beep(*frequency*, *duration*)
Sounds a system beep.

Parameters

- **frequency** – A frequency in Hz between 65 Hz and 2 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

property buffer_data
Returns a numpy array of values from the buffer.

property buffer_points
An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

channels_from_rows_columns(*rows*, *columns*, *slot=None*)
Determine the channel numbers between column(s) and row(s) of the 7709 connection matrix. Returns a list of channel numbers. Only one of the parameters 'rows' or 'columns' can be "all"

Parameters

- **rows** – row number or list of numbers; can also be "all"
- **columns** – column number or list of numbers; can also be "all"

- **slot** – slot number (1 or 2) of the 7709 card to be used

check_errors()
Logs any system errors reported by the instrument.

close_rows_to_columns(*rows*, *columns*, *slot=None*)
Closes (connects) the channels between column(s) and row(s) of the 7709 connection matrix. Only one of the parameters 'rows' or 'columns' can be "all"

Parameters

- **rows** – row number or list of numbers; can also be "all"
- **columns** – column number or list of numbers; can also be "all"
- **slot** – slot number (1 or 2) of the 7709 card to be used

property closed_channels
Parameter that controls the opened and closed channels. All mentioned channels are closed, other channels will be opened.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

config_buffer(*points=64*, *delay=0*)
Configures the measurement buffer for a number of points, to be taken with a specified delay.

Parameters

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

determine_valid_channels()
Determine what cards are installed into the Keithley 2700 and from that determine what channels are valid.

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process.

display_closed_channels()
Show the presently closed channels on the display of the Keithley 2700.

property display_text
A string property that controls the text shown on the display of the Keithley 2700. Text can be up to 12 ASCII characters and must be enabled to show.

property error
Returns a tuple of an error code and message from a single error.

get_state_of_channels(*channels*)
Get the open or closed state of the specified channels

Parameters **channels** – a list of channel numbers, or single channel number

property id
Requests and returns the identification of the instrument.

is_buffer_full()
Returns True if the buffer is full of measurements.

open_all_channels()
Open all channels of the Keithley 2700.

property open_channels
A parameter that opens the specified list of channels. Can only be set.

open_rows_to_columns(*rows*, *columns*, *slot=None*)
Opens (disconnects) the channels between column(s) and row(s) of the 7709 connection matrix. Only one of the parameters 'rows' or 'columns' can be "all"

Parameters

- **rows** – row number or list of numbers; can also be "all"
- **columns** – column number or list of numbers; can also be "all"
- **slot** – slot number (1 or 2) of the 7709 card to be used

property options
Property that lists the installed cards in the Keithley 2700. Returns a dict with the integer card numbers on the position.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument and clears the queue.

reset_buffer()
Resets the buffer.

shutdown()
Brings the instrument to a safe and stable state

start_buffer()
Starts the buffer.

property status
Requests and returns the status byte and Master Summary Status bit.

stop_buffer()
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

property text_enabled
A boolean property that controls whether a text message can be shown on the display of the Keithley 2700.

triad(*base_frequency*, *duration*)
Sounds a musical triad using the system beep.

Parameters

- **base_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)
Blocks the program, waiting for a full buffer. This function returns early if the should_stop function returns True or the timeout is reached before the buffer is full.

Parameters

- **should_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.7 Keithley 6221 AC and DC Current Source

class pymeasure.instruments.keithley.**Keithley6221**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*, pymeasure.instruments.keithley.buffer.KeithleyBuffer

Represents the Keithely 6221 AC and DC current source and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley6221("GPIB::1")
keithley.clear()

# Use the keithley as an AC source
keithley.waveform_function = "square"   # Set a square waveform
keithley.waveform_amplitude = 0.05      # Set the amplitude in Amps
keithley.waveform_offset = 0            # Set zero offset
keithley.source_compliance = 10         # Set compliance (limit) in V
keithley.waveform_dutycycle = 50        # Set duty cycle of wave in %
keithley.waveform_frequency = 347       # Set the frequency in Hz
keithley.waveform_ranging = "best"      # Set optimal output ranging
keithley.waveform_duration_cycles = 100 # Set duration of the waveform

# Link end of waveform to Service Request status bit
keithley.operation_event_enabled = 128  # OSB listens to end of wave
keithley.srq_event_enabled = 128        # SRQ listens to OSB
```

(continues on next page)

(continued from previous page)

```
keithley.waveform_arm()                 # Arm (load) the waveform

keithley.waveform_start()               # Start the waveform

keithley.adapter.wait_for_srq()         # Wait for the pulse to finish

keithley.waveform_abort()               # Disarm (unload) the waveform

keithley.shutdown()                     # Disables output
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)

Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.

- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

beep(*frequency*, *duration*)
Sounds a system beep.

Parameters

- **frequency** – A frequency in Hz between 65 Hz and 2 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

property buffer_data
Returns a numpy array of values from the buffer.

property buffer_points
An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

check_errors()
Logs any system errors reported by the instrument.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

config_buffer(*points=64*, *delay=0*)
Configures the measurement buffer for a number of points, to be taken with a specified delay.

Parameters

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

define_arbitary_waveform(*datapoints*, *location=1*)
Define the data points for the arbitrary waveform and copy the defined waveform into the given storage location.

Parameters

- **datapoints** – a list (or numpy array) of the data points; all values have to be between -1 and 1; 100 points maximum.

- **location** – integer storage location to store the waveform in. Value must be in range 1 to 4.

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process.

disable_output_trigger()
Disables the output trigger for the Trigger layer

disable_source()
Disables the source of current or voltage depending on the configuration of the instrument.

property display_enabled
A boolean property that controls whether or not the display of the sourcemeter is enabled. Valid values are True and False.

enable_source()
Enables the source of current or voltage depending on the configuration of the instrument.

property error
Returns a tuple of an error code and message from a single error.

property id
Requests and returns the identification of the instrument.

is_buffer_full()
Returns True if the buffer is full of measurements.

property measurement_event_enabled
An integer value that controls which measurement events are registered in the Measurement Summary Bit (MSB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

property measurement_events
An integer value that reads which measurement events have been registered in the Measurement event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

property operation_event_enabled
An integer value that controls which operation events are registered in the Operation Summary Bit (OSB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

property operation_events
An integer value that reads which operation events have been registered in the Operation event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

property options
Requests and returns the device options installed.

property output_low_grounded
A boolean property that controls whether the low output of the triax connection is connected to earth ground (True) or is floating (False).

output_trigger_on_external(*line=1*, *after='DEL'*)
Configures the output trigger on the specified trigger link line number, with the option of supplying the part of the measurement after which the trigger should be generated (default to delay, which is right before the measurement)

Parameters

- **line** – A trigger line from 1 to 4
- **after** – An event string that determines when to trigger

property questionable_event_enabled
An integer value that controls which questionable events are registered in the Questionable Summary Bit (QSB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

property questionable_events
An integer value that reads which questionable events have been registered in the Questionable event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters child – Instance of the child to delete.

reset()
Resets the instrument and clears the queue.

reset_buffer()
Resets the buffer.

set_timed_arm(*interval*)
Sets up the measurement to be taken with the internal trigger at a variable sampling rate defined by the interval in seconds between sampling points

shutdown()
Disables the output.

property source_auto_range
A boolean property that controls the auto range of the current source. Valid values are True or False.

property source_compliance
A floating point property that controls the compliance of the current source in Volts. valid values are in range 0.1 [V] to 105 [V].

property source_current
A floating point property that controls the source current in Amps.

property source_delay
A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 1e-3 [seconds] and 999999.999 [seconds].

property source_enabled
A boolean property that controls whether the source is enabled, takes values True or False. The convenience methods *enable_source()* and *disable_source()* can also be used.

property source_range
A floating point property that controls the source current range in Amps, which can take values between -0.105 A and +0.105 A. Auto-range is disabled when this property is set.

property srq_event_enabled
An integer value that controls which event registers trigger the Service Request (SRQ) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

property standard_event_enabled
An integer value that controls which standard events are registered in the Event Summary Bit (ESB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

property standard_events
An integer value that reads which standard events have been registered in the Standard event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

start_buffer()
Starts the buffer.

property status
Requests and returns the status byte and Master Summary Status bit.

stop_buffer()
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

triad(*base_frequency*, *duration*)
Sounds a musical triad using the system beep.

Parameters

- **base_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

trigger()
Executes a bus trigger, which can be used when *trigger_on_bus()* is configured.

trigger_immediately()
Configures measurements to be taken with the internal trigger at the maximum sampling rate.

trigger_on_bus()
Configures the trigger to detect events based on the bus trigger, which can be activated by *trigger()*.

trigger_on_external(*line=1*)
Configures the measurement trigger to be taken from a specific line of an external trigger

Parameters **line** – A trigger line from 1 to 4

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)
Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

Parameters

- **should_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

waveform_abort()
Abort the waveform output and disarm the waveform function.

property waveform_amplitude
A floating point property that controls the (peak) amplitude of the waveform in Amps. Valid values are in range 2e-12 to 0.105.

waveform_arm()
Arm the current waveform function.

property waveform_duration_cycles
A floating point property that controls the duration of the waveform in cycles. Valid values are in range 1e-3 to 99999999900.

waveform_duration_set_infinity()
Set the waveform duration to infinity.

property waveform_duration_time
A floating point property that controls the duration of the waveform in seconds. Valid values are in range 100e-9 to 999999.999.

property waveform_dutycycle
A floating point property that controls the duty-cycle of the waveform in percent for the square and ramp waves. Valid values are in range 0 to 100.

property waveform_frequency
A floating point property that controls the frequency of the waveform in Hertz. Valid values are in range 1e-3 to 1e5.

property waveform_function
A string property that controls the selected wave function. Valid values are "sine", "ramp", "square", "arbitrary1", "arbitrary2", "arbitrary3" and "arbitrary4".

property waveform_offset
A floating point property that controls the offset of the waveform in Amps. Valid values are in range -0.105 to 0.105.

property waveform_phasemarker_line
A numerical property that controls the line of the phase marker.

property waveform_phasemarker_phase
A numerical property that controls the phase of the phase marker.

property waveform_ranging
A string property that controls the source ranging of the waveform. Valid values are "best" and "fixed".

waveform_start()
Start the waveform output. Must already be armed

property waveform_use_phasemarker
A boolean property that controls whether the phase marker option is turned on or of. Valid values True (on) or False (off). Other settings for the phase marker have not yet been implemented.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.8 Keithley 6517B Electrometer

class pymeasure.instruments.keithley.**Keithley6517B**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*, pymeasure.instruments.keithley.buffer.KeithleyBuffer

Represents the Keithely 6517B ElectroMeter and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley6517B("GPIB::1")

keithley.apply_voltage()              # Sets up to source current
keithley.source_voltage_range = 200   # Sets the source voltage
                                      # range to 200 V
keithley.source_voltage = 20          # Sets the source voltage to 20 V
keithley.enable_source()              # Enables the source output

keithley.measure_resistance()         # Sets up to measure resistance

keithley.ramp_to_voltage(50)          # Ramps the voltage to 50 V
print(keithley.resistance)            # Prints the resistance in Ohms

keithley.shutdown()                   # Ramps the voltage to 0 V
                                      # and disables output
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: object

Add channels to the parent class.

The children will be added to the parent instance at instantiation with CommonBase.add_child(). The variable name (e.g. channels) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use channels as variable and leave the prefix at the default "ch_".

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)
Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

apply_voltage(*voltage_range=None*)
Configures the instrument to apply a source voltage, and uses an auto range unless a voltage range is specified.

Parameters **voltage_range** – A *voltage_range* value or None (activates auto range)

auto_range_source()
Configures the source to use an automatic range.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

property buffer_data
Returns a numpy array of values from the buffer.

property buffer_points
An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

check_errors()
Logs any system errors reported by the instrument.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

config_buffer(*points=64*, *delay=0*)
Configures the measurement buffer for a number of points, to be taken with a specified delay.

Parameters

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

property current
Reads the current in Amps, if configured for this reading.

property current_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property current_range
A floating point property that controls the measurement current range in Amps, which can take values between -20 and +20 mA. Auto-range is disabled when this property is set.

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process.

disable_source()
Disables the source of current or voltage depending on the configuration of the instrument.

enable_source()
Enables the source of current or voltage depending on the configuration of the instrument.

property error
Returns a tuple of an error code and message from a single error.

property id
Requests and returns the identification of the instrument.

is_buffer_full()
Returns True if the buffer is full of measurements.

measure_current(*nplc=1*, *current=0.000105*, *auto_range=True*)
Configures the measurement of current.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **current** – Upper limit of current in Amps, from -21 mA to 21 mA
- **auto_range** – Enables auto_range if True, else uses the current_range attribut

measure_resistance(*nplc=1*, *resistance=210000.0*, *auto_range=True*)
Configures the measurement of resistance.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10

- **resistance** – Upper limit of resistance in Ohms, from -210 POhms to 210 POhms
- **auto_range** – Enables auto_range if True, else uses the resistance_range attribut

measure_voltage(*nplc=1*, *voltage=21.0*, *auto_range=True*)
Configures the measurement of voltage.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **voltage** – Upper limit of voltage in Volts, from -1000 V to 1000 V
- **auto_range** – Enables auto_range if True, else uses the voltage_range attribut

property options
Requests and returns the device options installed.

ramp_to_voltage(*target_voltage*, *steps=30*, *pause=0.02*)
Ramps to a target voltage from the set voltage value over a certain number of linear steps, each separated by a pause duration.

Parameters

- **target_voltage** – A voltage in Volts
- **steps** – An integer number of steps
- **pause** – A pause duration in seconds to wait between steps

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters child – Instance of the child to delete.

reset()
Resets the instrument and clears the queue.

reset_buffer()
Resets the buffer.

property resistance
Reads the resistance in Ohms, if configured for this reading.

property resistance_nplc
A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property resistance_range
A floating point property that controls the resistance range in Ohms, which can take values from 0 to 100e18 Ohms. Auto-range is disabled when this property is set.

shutdown()
Ensures that the current or voltage is turned to zero and disables the output.

property source_current_resistance_limit
Boolean property which enables or disables resistance current limit

property source_enabled
Reads a boolean value that is True if the source is enabled.

property source_voltage
A floating point property that controls the source voltage in Volts.

property source_voltage_range
A floating point property that controls the source voltage range in Volts, which can take values from -1000 to 1000 V. Auto-range is disabled when this property is set.

start_buffer()
Starts the buffer.

property status
Requests and returns the status byte and Master Summary Status bit.

stop_buffer()
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

trigger()
Executes a bus trigger, which can be used when *trigger_on_bus()* is configured.

trigger_immediately()
Configures measurements to be taken with the internal trigger at the maximum sampling rate.

trigger_on_bus()
Configures the trigger to detect events based on the bus trigger, which can be activated by *trigger()*.

property voltage
Reads the voltage in Volts, if configured for this reading.

property voltage_nplc
A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

property voltage_range
A floating point property that controls the measurement voltage range in Volts, which can take values from -1000 to 1000 V. Auto-range is disabled when this property is set.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)
Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

Parameters

- **should_stop** – A function that returns True when this function should return early

- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.9 Keithley 2750 Multimeter/Switch System

class pymeasure.instruments.keithley.**Keithley2750**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Keithley2750 multimeter/switch system and provides a high-level interface for interacting with the instrument.

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)
Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

check_errors()
Read all errors from the instrument.

Returns list of error entries

close(*channel*)
Closes (connects) the specified channel.

Parameters **channel** (*int*) – 3-digit number for the channel

Returns None

property closed_channels
Reads the list of closed channels

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property id
Requests and returns the identification of the instrument.

open(*channel*)
Opens (disconnects) the specified channel.

Parameters **channel** (*int*) – 3-digit number for the channel

Returns None

open_all()
Opens (disconnects) all the channels on the switch matrix.

Returns None

property options
Requests and returns the device options installed.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown()
Brings the instrument to a safe and stable state

property status
Requests and returns the status byte and Master Summary Status bit.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.25.10 Keithley 2600 SourceMeter

class pymeasure.instruments.keithley.**Keithley2600**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Keithley 2600 series (channel A and B) SourceMeter

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)

Bases: object

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

check_errors()
Logs any system errors reported by the instrument.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property error
Returns a tuple of an error code and message from a single error.

property id
Requests and returns the identification of the instrument.

property options
Requests and returns the device options installed.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown()
Brings the instrument to a safe and stable state

property status
Requests and returns the status byte and Master Summary Status bit.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.26 Keysight

This section contains specific documentation on the keysight instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.26.1 Keysight DSOX1102G Oscilloscope

class pymeasure.instruments.keysight.**KeysightDSOX1102G**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Keysight DSOX1102G Oscilloscope interface for interacting with the instrument.

Refer to the Keysight DSOX1102G Oscilloscope Programmer's Guide for further details about using the lower-level methods to interact directly with the scope.

```
scope = KeysightDSOX1102G(resource)
scope.autoscale()
ch1_data_array, ch1_preamble = scope.download_data(source="channel1", points=2000)
# ...
scope.shutdown()
```

Known issues:

- The digitize command will be completed before the operation is. May lead to VI_ERROR_TMO (timeout) occuring when sending commands immediately after digitize. Current fix: if deemed necessary, add delay between digitize and follow-up command to scope.

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
```

(continues on next page)

(continued from previous page)

```
                                    ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

property acquisition_mode
A string parameter that sets the acquisition mode. Can be "realtime" or "segmented".

property acquisition_type
A string parameter that sets the type of data acquisition. Can be "normal", "average", "hresolution", or "peak".

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)
Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

ask(*command*, *query_delay=0*)
Write a command to the instrument and return the read response.

Parameters

- **command** – Command string to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.

Returns String returned by the device without read_termination.

autoscale()
Autoscale displayed channels.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

check_errors()
Read all errors from the instrument.

Returns list of error entries

clear()
Clears the instrument status byte

clear_status()
Clear device status.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

static control(*get_command*, *set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, ***kwargs*)
Return a property for the class based on the supplied commands. This property may be set and read from the instrument. See also *measurement()* and *setting()*.

Parameters

- **get_command** – A string command that asks for the value, set to *None* if get is not supported (see also *setting()*).
- **set_command** – A string command that writes the value, set to *None* if set is not supported (see also *measurement()*).
- **docs** – A docstring that will be included in the documentation
- **validator** – A function that takes both a value and a group of valid values and returns a valid value, while it otherwise raises an exception
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **get_process** – A function that take a value and allows processing before value mapping, returning the processed value
- **set_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **command_process** – A function that takes a command and allows processing before executing the command
- **check_set_errors** – Toggles checking errors after setting

- **check_get_errors** – Toggles checking errors after getting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses.

Example of usage of dynamic parameter is as follows:

```
class GenericInstrument(Instrument):
    center_frequency = Instrument.control(
        ":SENS:FREQ:CENT?;", ":SENS:FREQ:CENT %e GHz;",
        " A floating point property that represents the frequency ... ",
        validator=strict_range,
        # Redefine this in subclasses to reflect actual instrument value:
        values=(1, 20),
        dynamic=True  # enable changing property parameters on-the-fly
    )

class SpecificInstrument(GenericInstrument):
    # Identical to GenericInstrument, except for frequency range
    # Override the "values" parameter of the "center_frequency" property
    center_frequency_values = (1, 10) # Redefined at subclass level

instrument = SpecificInstrument()
instrument.center_frequency_values = (1, 6e9) # Redefined at instance level
```

Warning: Unexpected side effects when using dynamic properties

Users must pay attention when using dynamic properties, since definition of class and/or instance attributes matching specific patterns could have unwanted side effect. The attribute name pattern *property_param*, where *property* is the name of the dynamic property (e.g. *center_frequency* in the example) and *param* is any of this method parameters name except *dynamic* and *docs* (e.g. *values* in the example) has to be considered reserved for dynamic property control.

default_setup()
Default setup, some user settings (like preferences) remain unchanged.

digitize(*source: str*)
Acquire waveforms according to the settings of the :ACQuire commands. Ensure a delay between the digitize operation and further commands, as timeout may be reached before digitize has completed. :param source: “channel1”, “channel2”, “function”, “math”, “fft”, “abus”, or “ext”.

download_data(*source*, *points=62500*)
Get data from specified source of oscilloscope. Returned objects are a np.ndarray of data values (no temporal axis) and a dict of the waveform preamble, which can be used to build the corresponding time values for all data points.

Multimeter will be stopped for proper acquisition.

Parameters

- **source** – measurement source, can be “channel1”, “channel2”, “function”, “fft”, “wmemory1”, “wmemory2”, or “ext”.
- **points** – integer number of points to acquire. Note that oscilloscope may return fewer points than specified, this is not an issue of this library. Can be 100, 250, 500, 1000, 2000, 5000, 10000, 20000, 50000, or 62500.

Return data_ndarray, waveform_preamble_dict see waveform_preamble property for dict format.

download_image(*format_='png'*, *color_palette='color'*)
Get image of oscilloscope screen in bytearray of specified file format.

Parameters

- **format** – "bmp", "bmp8bit", or "png"
- **color_palette** – "color" or "grayscale"

factory_reset()
Factory default setup, no user settings remain unchanged.

property id
Requests and returns the identification of the instrument.

static measurement(*get_command*, *docs*, *values=()*, *map_values=None*, *get_process=<function CommonBase.<lambda>>*, *command_process=<function CommonBase.<lambda>>*, *check_get_errors=False*, *dynamic=False*, ***kwargs*)
Return a property for the class based on the supplied commands. This is a measurement quantity that may only be read from the instrument, not set.

Parameters

- **get_command** – A string command that asks for the value
- **docs** – A docstring that will be included in the documentation
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **get_process** – A function that take a value and allows processing before value mapping, returning the processed value
- **command_process** – A function that take a command and allows processing before executing the command, for getting
- **check_get_errors** – Toggles checking errors after getting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses. See *control()* for an usage example.

property options
Requests and returns the device options installed.

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

run()
Starts repetitive acquisitions.

This is the same as pressing the Run key on the front panel.

static setting(*set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *set_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *dynamic=False*, ***kwargs*)
Return a property for the class based on the supplied commands. This property may be set, but raises an exception when being read from the instrument.

Parameters

- **set_command** – A string command that writes the value
- **docs** – A docstring that will be included in the documentation
- **validator** – A function that takes both a value and a group of valid values and returns a valid value, while it otherwise raises an exception
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **set_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **check_set_errors** – Toggles checking errors after setting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses. See *control()* for an usage example.

shutdown()
Brings the instrument to a safe and stable state

single()
Causes the instrument to acquire a single trigger of data. This is the same as pressing the Single key on the front panel.

property status
Requests and returns the status byte and Master Summary Status bit.

stop()
Stops the acquisition. This is the same as pressing the Stop key on the front panel.

property system_setup
A string parameter that sets up the oscilloscope. Must be in IEEE 488.2 format. It is recommended to only set a string previously obtained from this command.

property timebase
Read timebase setup as a dict containing the following keys: - “REF”: position on screen of timebase reference (str) - “MAIN:RANG”: full-scale timebase range (float) - “POS”: interval between trigger and reference point (float) - “MODE”: mode (str)

property timebase_mode
A string parameter that sets the current time base. Can be “main”, “window”, “xy”, or “roll”.

property timebase_offset
A float parameter that sets the time interval in seconds between the trigger event and the reference position (at center of screen by default).

property timebase_range
A float parameter that sets the full-scale horizontal time in seconds for the main window.

property timebase_scale
A float parameter that sets the horizontal scale (units per division) in seconds for the main window.

timebase_setup(*mode=None*, *offset=None*, *horizontal_range=None*, *scale=None*)
Set up timebase. Unspecified parameters are not modified. Modifying a single parameter might impact other parameters. Refer to oscilloscope documentation and make multiple consecutive calls to channel_setup if needed.

Parameters

- **mode** – Timebase mode, can be “main”, “window”, “xy”, or “roll”.
- **offset** – Offset in seconds between trigger and center of screen.
- **horizontal_range** – Full-scale range in seconds.
- **scale** – Units-per-division in seconds.

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*, *maxsplit=-1*)
Write a command to the instrument and return a list of formatted values from the result.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string.
- **maxsplit** – At most *maxsplit* splits are done. -1 (default) indicates no limit.

Returns A list of the desired type, or strings where the casting fails

wait_for(*query_delay=0*)
Wait for some time. Used by ‘ask’ to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

property waveform_data
Get the binary block of sampled data points transmitted using the IEEE 488.2 arbitrary block data format.

property waveform_format
A string parameter that controls how the data is formatted when sent from the oscilloscope. Can be “ascii”, “word” or “byte”. Words are transmitted in big endian by default.

property waveform_points
An integer parameter that sets the number of waveform points to be transferred with the waveform_data method. Can be any of the following values: 100, 250, 500, 1000, 2 000, 5 000, 10 000, 20 000, 50 000, 62 500.

Note that the oscilloscope may provide less than the specified nb of points.

property waveform_points_mode
A string parameter that sets the data record to be transferred with the waveform_data method. Can be "normal", "maximum", or "raw".

property waveform_preamble
Get preamble information for the selected waveform source as a dict with the following keys: - "format": byte, word, or ascii (str) - "type": normal, peak detect, or average (str) - "points": nb of data points transferred (int) - "count": always 1 (int) - "xincrement": time difference between data points (float) - "xorigin": first data point in memory (float) - "xreference": data point associated with xorigin (int) - "yincrement": voltage difference between data points (float) - "yorigin": voltage at center of screen (float) - "yreference": data point associated with yorigin (int)

property waveform_source
A string parameter that selects the analog channel, function, or reference waveform to be used as the source for the waveform methods. Can be "channel1", "channel2", "function", "fft", "wmemory1", "wmemory2", or "ext".

write(*command*, ***kwargs*)
Write a string command to the instrument appending *write_termination*.

Parameters

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.26.2 Keysight N5767A Power Supply

class pymeasure.instruments.keysight.**KeysightN5767A**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Keysight N5767A Power supply interface for interacting with the instrument.

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: object

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

binary_values(*command*, *query_delay=0*, ***kwargs*)

Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

check_errors()

Read all errors from the instrument.

Returns list of error entries

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property current
Reads a setting current in Amps.

property current_range
A floating point property that controls the DC current range in Amps, which can take values from 0 to 25 A. Auto-range is disabled when this property is set.

disable()
Disables the flow of current.

enable()
Enables the flow of current.

property id
Requests and returns the identification of the instrument.

is_enabled()
Returns True if the current supply is enabled.

property options
Requests and returns the device options installed.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown()
Brings the instrument to a safe and stable state

property status
Requests and returns the status byte and Master Summary Status bit.

property voltage
Reads a DC voltage measurement in Volts.

property voltage_range
A floating point property that controls the DC voltage range in Volts, which can take values from 0 to 60 V. Auto-range is disabled when this property is set.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.26.3 Keysight N5767A Power Supply

class pymeasure.instruments.keysight.**KeysightN7776C**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

This represents the Keysight N7776C Tunable Laser Source interface.

```
laser = N7776C(address)
laser.sweep_wl_start = 1550
laser.sweep_wl_stop = 1560
laser.sweep_speed = 1
laser.sweep_mode = 'CONT'
laser.output_enabled = 1
while laser.sweep_state == 1:
    log.info('Sweep in progress.')
laser.output_enabled = 0
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.

- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)
Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

check_errors()
Read all errors from the instrument.

Returns list of error entries

close()
Fully closes the connection to the instrument through the adapter connection.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

get_wl_data()
Function returning the wavelength data logged in the internal memory of the laser

property id
Requests and returns the identification of the instrument.

property locked
Boolean property controlling the lock state (True/False) of the laser source

next_step()
Performs the next sweep step in stepped sweep if it is paused or in manual mode.

property options
Requests and returns the device options installed.

property output_enabled
Boolean Property that controls the state (on/off) of the laser source

previous_step()
Performs one sweep step backwards in stepped sweep if its paused or in manual mode.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown()
Brings the instrument to a safe and stable state

property status
Requests and returns the status byte and Master Summary Status bit.

property sweep_mode
Sweep mode of the swept laser source

property sweep_points
Returns the number of datapoints that the :READout:DATA? command will return.

property sweep_speed
Speed of the sweep (in nanometers per second).

property sweep_state
State of the wavelength sweep. Stops, starts, pauses or continues a wavelength sweep. Possible state values are 0 (not running), 1 (running) and 2 (paused). Refer to the N7776C user manual for exact usage of the paused option.

property sweep_step
Step width of the sweep (in nanometers).

property sweep_twoway
Sets the repeat mode. Applies in stepped,continuous and manual sweep mode.

property sweep_wl_start
Start Wavelength (in nanometers) for a sweep.

property sweep_wl_stop
End Wavelength (in nanometers) for a sweep.

property trigger_in
Sets the incoming trigger response and arms the module.

property trigger_out
Specifies if and at which point in a sweep cycle an output trigger is generated and arms the module.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

property wavelength
Absolute wavelength of the output light (in nanometers)

property wl_logging
State (on/off) of the lambda logging feature of the laser source.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.26.4 Keysight E36312A Triple Output Power Supply

pymeasure.instruments.keysight.**keysightE36312A**
alias of <module 'pymeasure.instruments.keysight.keysightE36312A' from '/home/docs/checkouts/readthedocs.org/user_builds/pymeasure/checkouts/latest/pymeasure/instruments/keysight/keysightE36312

7.27 Lake Shore Cryogenics

This section contains specific documentation on the Lake Shore Cryogenics instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.27.1 Lake Shore 331 Temperature Controller

class pymeasure.instruments.lakeshore.**LakeShore331**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Lake Shore 331 Temperature Controller and provides a high-level interface for interacting with the instrument.

```
controller = LakeShore331("GPIB::1")

print(controller.setpoint_1)        # Print the current setpoint for loop 1
controller.setpoint_1 = 50          # Change the setpoint to 50 K
controller.heater_range = 'low'     # Change the heater range to Low
controller.wait_for_temperature()   # Wait for the temperature to stabilize
print(controller.temperature_A)     # Print the temperature at sensor A
```

disable_heater()

Turns the *heater_range* to `off` to disable the heater.

property heater_range

A string property that controls the heater range, which can take the values: off, low, medium, and high. These values correlate to 0, 0.5, 5 and 50 W respectively.

property setpoint_1

A floating point property that controls the setpoint temperature in Kelvin for Loop 1.

property setpoint_2

A floating point property that controls the setpoint temperature in Kelvin for Loop 2.

property temperature_A

Reads the temperature of the sensor A in Kelvin.

property temperature_B

Reads the temperature of the sensor B in Kelvin.

wait_for_temperature(*accuracy=0.1*, *interval=0.1*, *sensor='A'*, *setpoint=1*, *timeout=360*, *should_stop=<function LakeShore331.<lambda>>*)

Blocks the program, waiting for the temperature to reach the setpoint within the accuracy (%), checking this each interval time in seconds.

Parameters

- **accuracy** – An acceptable percentage deviation between the setpoint and temperature
- **interval** – A time in seconds that controls the refresh rate
- **sensor** – The desired sensor to read, either A or B
- **setpoint** – The desired setpoint loop to read, either 1 or 2
- **timeout** – A timeout in seconds after which an exception is raised
- **should_stop** – A function that returns True if waiting should stop, by default this always returns False

7.27.2 Lake Shore 421 Gaussmeter

class pymeasure.instruments.lakeshore.**LakeShore421**(*adapter*, *baud_rate=9600*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Lake Shore 421 Gaussmeter and provides a high-level interface for interacting with the instrument.
.. code-block:: python

gaussmeter = LakeShore421("COM1") gaussmeter.unit = "T" # Set units to Tesla gaussmeter.auto_range = True # Turn on auto-range gaussmeter.fast_mode = True # Turn on fast-mode

A delay of 50 ms is ensured between subsequent writes, as the instrument cannot correctly handle writes any faster.

property alarm_active
A boolean property that returns whether the alarm is triggered.

property alarm_audible
A boolean property that enables or disables the audible alarm beeper.

property alarm_high
Property that controls the upper setpoint for the alarm mode in the current units. This takes into account the field multiplier.

property alarm_high_multiplier
Returns the multiplier for the upper alarm setpoint field.

property alarm_high_raw
ALMH %g

property alarm_in_out
A string property that controls whether an active alarm is caused when the field reading is inside ("Inside") or outside ("Outside") of the high and low setpoint values.

property alarm_low
Property that controls the lower setpoint for the alarm mode in the current units. This takes into account the field multiplier.

property alarm_low_multiplier
Returns the multiplier for the lower alarm setpoint field.

property alarm_low_raw
ALML %g

property alarm_mode_enabled
A boolean property that enables or disables the alarm mode.

property alarm_sort_enabled
A boolean property that enables or disables the alarm Sort Pass/Fail function.

property auto_range
A boolean property that controls the auto-range option of the meter. Valid values are True and False. Note that the auto-range is relatively slow and might not suffice for rapid measurements.

property display_filter_enabled
A boolean property that controls the display filter to make it more readable when the probe is exposed to a noisy field. The filter function makes a linear average of 8 readings and settles in approximately 2 seconds.

property fast_mode
A boolean property that controls the fast-mode option of the meter. Valid values are True and False. When enabled, the relative mode, Max Hold mode, alarms, and autorange are disabled.

property field
: Returns the field in the current units. This property takes into account the field multiplier. Returns np.nan if field is out of range.

property field_mode
: A string property that controls whether the gaussmeter measures AC or DC magnetic fields. Valid values are “AC” and “DC”.

property field_multiplier
: Returns the field multiplier for the returned magnetic field.

property field_range
: A floating point property that controls the field range of the meter in the current unit (G or T). Valid values are 30e3, 3e3, 300, 30 (when in Gauss), or 0.003, 0.03, 0.3, and 3 (when in Tesla).

property field_range_raw
: A integer property that controls the field range of the meter. Valid values are 0 (highest) to 3 (lowest).

property field_raw
: Returns the field in the current units and multiplier

property front_panel_brightness
: An integer property that controls the brightness of the from panel display. Valid values are 0 (dimmest) to 7 (brightest).

property front_panel_locked
: A boolean property that locks or unlocks all front panel entries except pressing the Alarm key to silence alarms.

property max_hold_enabled
: A boolean property that enables or disables the Max Hold function to store the largest field since the last reset (with max_hold_reset).

property max_hold_field
: Returns the largest field since the last reset in the current units. This property takes into account the field multiplier. Returns np.nan if field is out of range.

property max_hold_field_raw
: Returns the largest field since the last reset in the current units and multiplier.

property max_hold_multiplier
: Returns the multiplier for the returned max hold field.

max_hold_reset()
: Clears the stored Max Hold value.

property probe_type
: Returns type of field-probe used with the gaussmeter. Possible values are High Sensitivity, High Stability, or Ultra-High Sensitivity.

property relative_field
: Returns the relative field in the current units. This property takes into account the field multiplier. Returns np.nan if field is out of range.

property relative_field_raw
: Returns the relative field in the current units and the current multiplier.

property relative_mode_enabled
: A boolean property that enables or disables the relative mode to see small variations with respect to a given setpoint.

property relative_multiplier
Returns the relative field multiplier for the returned magnetic field.

property relative_setpoint
Property that controls the setpoint for the relative field mode in the current units. This takes into account the field multiplier.

property relative_setpoint_multiplier
Returns the multiplier for the setpoint field.

property relative_setpoint_raw
Property that controls the setpoint for the relative field mode in the current units and multiplier.

property serial_number
Returns the serial number of the probe.

shutdown()
Closes the serial connection to the system.

property unit
A string property that controls the units used by the gaussmeter. Valid values are G (Gauss), T (Tesla).

write(*command*)
Write a string command to the instrument appending *write_termination*.

Parameters

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

zero_probe(*wait=True*)
Reset the probe value to 0. It is normally used with a zero gauss chamber, but may also be used with an open probe to cancel the Earth magnetic field. To cancel larger magnetic fields, the relative mode should be used.

Parameters **wait** (*bool*) – Wait for 20 seconds after issuing the command to allow the resetting to finish.

7.27.3 Lake Shore 425 Gaussmeter

class pymeasure.instruments.lakeshore.**LakeShore425**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the LakeShore 425 Gaussmeter and provides a high-level interface for interacting with the instrument

To allow user access to the LakeShore 425 Gaussmeter in Linux, create the file: `/etc/udev/rules.d/52-lakeshore425.rules`, with contents:

```
SUBSYSTEMS=="usb",ATTRS{idVendor}=="1fb9",ATTRS{idProduct}=="0401",MODE="0666",
↪SYMLINK+="lakeshore425"
```

Then reload the udev rules with:

```
sudo udevadm control --reload-rules
sudo udevadm trigger
```

The device will be accessible through `/dev/lakeshore425`.

ac_mode(*wideband=True*)
Sets up a measurement of an oscillating (AC) field

auto_range()
Sets the field range to automatically adjust

dc_mode(*wideband=True*)
Sets up a steady-state (DC) measurement of the field

property field
Returns the field in the current units

measure(*points*, *has_aborted=<function LakeShore425.<lambda>>*, *delay=0.001*)
Returns the mean and standard deviation of a given number of points while blocking

property range
A floating point property that controls the field range in units of Gauss, which can take the values 35, 350, 3500, and 35,000 G.

property unit
A string property that controls the units of the instrument, which can take the values of G, T, Oe, or A/m.

zero_probe()
Initiates the zero field sequence to calibrate the probe

7.28 LeCroy

This section contains specific documentation on the LeCroy instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.28.1 LeCroy T3DSO1204 Oscilloscope

class pymeasure.instruments.lecroy.**LeCroyT3DSO1204**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the LeCroy T3DSO1204 Oscilloscope interface for interacting with the instrument.

Refer to the LeCroy T3DSO1204 Oscilloscope Programmer's Guide for further details about using the lower-level methods to interact directly with the scope.

Attributes: WRITE_INTERVAL_S: minimum time between two commands. If a command is received less than WRITE_INTERVAL_S after the previous one, the code blocks until at least WRITE_INTERVAL_S seconds have passed. Because the oscilloscope takes a non neglibile time to perform some operations, it might be needed for the user to tweak the sleep time between commands. The WRITE_INTERVAL_S is set to 10ms as default however its optimal value heavily depends on the actual commands and on the connection type, so it is impossible to give a unique value to fit all cases. An interval between 10ms and 500ms second proved to be good, depending on the commands and connection latency.

```
scope = LeCroyT3DSO1204(resource)
scope.autoscale()
ch1_data_array, ch1_preamble = scope.download_waveform(source="C1", points=2000)
# ...
scope.shutdown()
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)
Bases: object

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

property acquisition_average
A integer parameter that selects the average times of average acquisition.

acquisition_sample_size(*source*)
Get acquisition sample size for a certain channel. Used mainly for waveform acquisition. If the source is MATH, the SANU? MATH query does not seem to work, so I return the memory size instead.

Parameters **source** – channel number of channel name.

Returns acquisition sample size of that channel.

property acquisition_sample_size_c1
A integer parameter that returns the number of data points that the hardware will acquire from the input signal of channel 1. Note. Channel 2 and channel 1 share the same ADC, so the sample is the same too.

property acquisition_sample_size_c2
A integer parameter that returns the number of data points that the hardware will acquire from the input signal of channel 2. Note. Channel 2 and channel 1 share the same ADC, so the sample is the same too.

property acquisition_sample_size_c3
A integer parameter that returns the number of data points that the hardware will acquire from the input signal of channel 3. Note. Channel 3 and channel 4 share the same ADC, so the sample is the same too.

property acquisition_sample_size_c4
A integer parameter that returns the number of data points that the hardware will acquire from the input signal of channel 4. Note. Channel 3 and channel 4 share the same ADC, so the sample is the same too.

property acquisition_sampling_rate
A integer parameter that returns the sample rate of the scope.

property acquisition_status
A string parameter that defines the acquisition status of the scope.

property acquisition_type
A string parameter that sets the type of data acquisition. Can be "normal", "peak", "average", "highres".

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)
Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.
- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

arm_acquisition()
Causes the instrument to acquire a single trigger of data. This is the same as pressing the Single key on the front panel.

ask(*command*, *query_delay=0*)
Write a command to the instrument and return the read response.

Parameters

- **command** – Command string to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.

Returns String returned by the device without read_termination.

autoscale()
Autoscale displayed channels.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

center_trigger()
This command automatically sets the trigger levels to center of the trigger source waveform.

ch(*source*)
Get channel object from its index or its name. Or if source is "math", just return the scope object.

Parameters **source** – can be 1, 2, 3, 4 or C1, C2, C3, C4, MATH

Returns handle to the selected source.

check_errors()
Read all errors from the instrument.

Returns list of error entries

clear()
Clears the instrument status byte

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

static control(*get_command*, *set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, ***kwargs*)
Return a property for the class based on the supplied commands. This property may be set and read from the instrument. See also *measurement()* and *setting()*.

Parameters

- **get_command** – A string command that asks for the value, set to *None* if get is not supported (see also *setting()*).
- **set_command** – A string command that writes the value, set to *None* if set is not supported (see also *measurement()*).
- **docs** – A docstring that will be included in the documentation
- **validator** – A function that takes both a value and a group of valid values and returns a valid value, while it otherwise raises an exception
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **get_process** – A function that take a value and allows processing before value mapping, returning the processed value
- **set_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **command_process** – A function that takes a command and allows processing before executing the command
- **check_set_errors** – Toggles checking errors after setting
- **check_get_errors** – Toggles checking errors after getting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses.

Example of usage of dynamic parameter is as follows:

```
class GenericInstrument(Instrument):
    center_frequency = Instrument.control(
        ":SENS:FREQ:CENT?;", ":SENS:FREQ:CENT %e GHz;",
        " A floating point property that represents the frequency ... ",
```

(continues on next page)

(continued from previous page)

```
        validator=strict_range,
        # Redefine this in subclasses to reflect actual instrument value:
        values=(1, 20),
        dynamic=True  # enable changing property parameters on-the-fly
    )

class SpecificInstrument(GenericInstrument):
    # Identical to GenericInstrument, except for frequency range
    # Override the "values" parameter of the "center_frequency" property
    center_frequency_values = (1, 10) # Redefined at subclass level

instrument = SpecificInstrument()
instrument.center_frequency_values = (1, 6e9) # Redefined at instance level
```

Warning: Unexpected side effects when using dynamic properties

Users must pay attention when using dynamic properties, since definition of class and/or instance attributes matching specific patterns could have unwanted side effect. The attribute name pattern *property_param*, where *property* is the name of the dynamic property (e.g. *center_frequency* in the example) and *param* is any of this method parameters name except *dynamic* and *docs* (e.g. *values* in the example) has to be considered reserved for dynamic property control.

default_setup()
Set up the oscilloscope for remote operation.

The COMM_HEADER command controls the way the oscilloscope formats response to queries. This command does not affect the interpretation of messages sent to the oscilloscope. Headers can be sent in their long or short form regardless of the CHDR setting. By setting the COMM_HEADER to OFF, the instrument is going to reply with minimal information, and this makes the response message much easier to parse. The user should not be fiddling with the COMM_HEADER during operation, because if the communication header is anything other than OFF, the whole driver breaks down.

display_parameter(*parameter*, *channel*)
Same as the display_parameter method in the Channel subclass

download_image()
Get a BMP image of oscilloscope screen in bytearray of specified file format.

download_waveform(*source*, *requested_points=None*, *sparsing=None*)
Get data points from the specified source of the oscilloscope. The returned objects are two np.ndarray of data and time points and a dict with the waveform preamble, that contains metadata about the waveform. Note. :param source: measurement source. It can be “C1”, “C2”, “C3”, “C4”, “MATH”. :param requested_points: number of points to acquire. If 0, all available points will be returned. :param sparsing: interval between data points. For example if sparsing = 4, only one point every 4 points is read. :return: data_ndarray, time_ndarray, waveform_preamble_dict: see waveform_preamble property for dict format.

property grid_display
Select the type of the grid which is used to display (FULL, HALF, OFF)

property id
Requests and returns the identification of the instrument.

property intensity
Sets the intensity level of the grid or the trace in percent

property math_define
A string parameter that sets the desired waveform math operation between two channels. Three parameters must be passed as a tuple: 1. source1 : source channel on the left 2. operation : operator must be "*", "/", "+", "-" 3. source2 : source channel on the right

property math_vdiv
A float parameter that sets the vertical scale of the selected math operation. This command is only valid in add, subtract, multiply and divide operation. Note: Legal values for the scale depend on the selected operation.

property math_vpos
A integer parameter that sets the vertical position of the math waveform with specified source. Note: The point represents the screen pixels and is related to the screen center. For example, if the point is 50. The math waveform will be displayed 1 grid above the vertical center of the screen. Namely one grid is 50.

property measure_delay
The MEASURE_DELY command places the instrument in the continuous measurement mode and starts a type of delay measurement. The MEASURE_DELY? query returns the measured value of delay type. The command accepts three arguments with the following syntax: measure_delay = (<type>,<sourceA>,<sourceB>) <type> := {PHA,FRR,FRF,FFR,FFF,LRR,LRF,LFR,LFF,SKEW} <sourceA>,<sourceB> := {C1,C2,C3,C4} where if sourceA=CX and sourceB=CY, then X < Y Type Description PHA The phase difference between two channels. (rising edge - rising edge) FRR Delay between two channels. (first rising edge - first rising edge) FRF Delay between two channels. (first rising edge - first falling edge) FFR Delay between two channels. (first falling edge - first rising edge) FFF Delay between two channels. (first falling edge - first falling edge) LRR Delay between two channels. (first rising edge - last rising edge) LRF Delay between two channels. (first rising edge - last falling edge) LFR Delay between two channels. (first falling edge - last rising edge) LFF Delay between two channels. (first falling edge - last falling edge) Skew Delay between two channels. (edge – edge of the same type)

measure_parameter(*parameter*, *channel*)
Same as the measure_parameter method in the Channel subclass

static measurement(*get_command*, *docs*, *values=()*, *map_values=None*, *get_process=<function CommonBase.<lambda>>*, *command_process=<function CommonBase.<lambda>>*, *check_get_errors=False*, *dynamic=False*, ***kwargs*)
Return a property for the class based on the supplied commands. This is a measurement quantity that may only be read from the instrument, not set.

Parameters

- **get_command** – A string command that asks for the value
- **docs** – A docstring that will be included in the documentation
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **get_process** – A function that take a value and allows processing before value mapping, returning the processed value
- **command_process** – A function that take a command and allows processing before executing the command, for getting
- **check_get_errors** – Toggles checking errors after getting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses. See *control()* for an usage example.

property memory_size
A float parameter that selects the maximum depth of memory. <size>:={7K,70K,700K,7M} for non-interleaved mode. Non-interleaved means a single channel is active per A/D converter. Most oscilloscopes feature two channels per A/D converter. <size>:={14K,140K,1.4M,14M} for interleave mode. Interleave mode means multiple active channels per A/D converter.

property menu
Control the bottom menu enabled state. (strict bool)

property options
Requests and returns the device options installed.

read(***kwargs*)
Read up to (excluding) *read_termination* or the whole read buffer.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

run()
Starts repetitive acquisitions.

This is the same as pressing the Run key on the front panel.

static setting(*set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *set_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *dynamic=False*, ***kwargs*)
Return a property for the class based on the supplied commands. This property may be set, but raises an exception when being read from the instrument.

Parameters

- **set_command** – A string command that writes the value
- **docs** – A docstring that will be included in the documentation
- **validator** – A function that takes both a value and a group of valid values and returns a valid value, while it otherwise raises an exception
- **values** – A list, tuple, range, or dictionary of valid values, that can be used as to map values if `map_values` is True.
- **map_values** – A boolean flag that determines if the values should be interpreted as a map
- **set_process** – A function that takes a value and allows processing before value mapping, returning the processed value

- **check_set_errors** – Toggles checking errors after setting
- **dynamic** – Specify whether the property parameters are meant to be changed in instances or subclasses. See *control()* for an usage example.

shutdown()
Brings the instrument to a safe and stable state

property status
Requests and returns the status byte and Master Summary Status bit.

stop()
Stops the acquisition. This is the same as pressing the Stop key on the front panel.

property timebase
Read timebase setup as a dict containing the following keys: - "timebase_scale": horizontal scale in seconds/div (float) - "timebase_offset": interval in seconds between the trigger and the reference position (float) - "timebase_hor_magnify": horizontal scale in the zoomed window in seconds/div (float) - "timebase_hor_position": horizontal position in the zoomed window in seconds (float)

property timebase_hor_magnify
A float parameter that sets the zoomed (delayed) window horizontal scale (seconds/div). The main sweep scale determines the range for this command.

property timebase_hor_position
A string parameter that sets the horizontal position in the zoomed (delayed) view of the main sweep. The main sweep range and the main sweep horizontal position determine the range for this command. The value for this command must keep the zoomed view window within the main sweep range.

property timebase_offset
A float parameter that sets the time interval in seconds between the trigger event and the reference position (at center of screen by default).

property timebase_scale
A float parameter that sets the horizontal scale (units per division) in seconds (S), for the main window.

timebase_setup(*scale=None*, *offset=None*, *hor_magnify=None*, *hor_position=None*)
Set up timebase. Unspecified parameters are not modified. Modifying a single parameter might impact other parameters. Refer to oscilloscope documentation and make multiple consecutive calls to timebase_setup if needed.

Parameters

- **scale** – interval in seconds between the trigger event and the reference position.
- **offset** – horizontal scale per division in seconds/div.
- **hor_magnify** – horizontal scale in the zoomed window in seconds/div.
- **hor_position** – horizontal position in the zoomed window in seconds.

property trigger
Read trigger setup as a dict containing the following keys: - "mode": trigger sweep mode [auto, normal, single, stop] - "trigger_type": condition that will trigger the acquisition of waveforms [edge, slew,glit,intv,runt,drop] - "source": trigger source [c1,c2,c3,c4] - "hold_type": hold type (refer to page 172 of programing guide) - "hold_value1": hold value1 (refer to page 172 of programing guide) - "hold_value2": hold value2 (refer to page 172 of programing guide) - "coupling": input coupling for the selected trigger sources - "level": trigger level voltage for the active trigger source - "level2": trigger lower level voltage for the active trigger source (only slew/runt trigger) - "slope": trigger slope of the specified trigger source

property trigger_mode

A string parameter that selects the trigger sweep mode. <mode>:= {AUTO,NORM,SINGLE,STOP} • auto : When AUTO sweep mode is selected, the oscilloscope begins to search for the trigger signal that meets the conditions. If the trigger signal is satisfied, the running state on the top left corner of the user interface shows Trig'd, and the interface shows stable waveform. Otherwise, the running state always shows Auto, and the interface shows unstable waveform. • normal : When NORMAL sweep mode is selected, the oscilloscope enters the wait trigger state and begins to search for trigger signals that meet the conditions. If the trigger signal is satisfied, the running state shows Trig'd, and the interface shows stable waveform. Otherwise, the running state shows Ready, and the interface displays the last triggered waveform (previous trigger) or does not display the waveform (no previous trigger). • single : When SINGLE sweep mode is selected, the backlight of SINGLE key lights up, the oscilloscope enters the waiting trigger state and begins to search for the trigger signal that meets the conditions. If the trigger signal is satisfied, the running state shows Trig'd, and the interface shows stable waveform. Then, the oscilloscope stops scanning, the RUN/STOP key is red light, and the running status shows Stop. Otherwise, the running state shows Ready, and the interface does not display the waveform. • stopped : STOP is a part of the option of this command, but not a trigger mode of the oscilloscope.

property trigger_select

A string parameter that selects the condition that will trigger the acquisition of waveforms. Depending on the trigger type, additional parameters must be specified. These additional parameters are grouped in pairs. The first in the pair names the variable to be modified, while the second gives the new value to be assigned. Pairs may be given in any order and restricted to those variables to be changed. There are five parameters that can be specified. Parameters 1. 2. 3. are always mandatory. Parameters 4. 5. are required only for certain combinations of the previous parameters. 1. <trig_type>:={edge, slew, glit, intv, runt, drop} 2. <source>:={c1, c2, c3, c4, line} 3. - <hold_type>:={ti, off} for edge trigger. - <hold_type>:={ti} for drop trigger. - <hold_type>:={ps, pl, p2, p1} for glit/runt trigger. - <hold_type>:={is, il, i2, i1} for slew/intv trigger. 4. <hold_value1>:= a time value with unit. 5. <hold_value2>:= a time value with unit. Note: • "line" can only be selected when the trigger type is "edge". • All time arguments should be given in multiples of seconds. Use the scientific notation if necessary. • The range of hold_values varies from trigger types. [80nS, 1.5S] for "edge" trigger, and [2nS, 4.2S] for others. • The trigger_select command is switched automatically between the short, normal and extended version depending on the number of expected parameters.

trigger_setup(*mode=None*, *source=None*, *trigger_type=None*, *hold_type=None*, *hold_value1=None*, *hold_value2=None*, *coupling=None*, *level=None*, *level2=None*, *slope=None*)

Set up trigger. Unspecified parameters are not modified. Modifying a single parameter might impact other parameters. Refer to oscilloscope documentation and make multiple consecutive calls to trigger_setup and channel_setup if needed. :param mode: trigger sweep mode [auto, normal, single, stop] :param source: trigger source [c1, c2, c3, c4, line] :param trigger_type: condition that will trigger the acquisition of waveforms [edge,slew,glit,intv,runt,drop] :param hold_type: hold type (refer to page 172 of programing guide) :param hold_value1: hold value1 (refer to page 172 of programing guide) :param hold_value2: hold value2 (refer to page 172 of programing guide) :param coupling: input coupling for the selected trigger sources :param level: trigger level voltage for the active trigger source :param level2: trigger lower level voltage for the active trigger source (only slew/runt trigger) :param slope: trigger slope of the specified trigger source

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*, *maxsplit=-1*)

Write a command to the instrument and return a list of formatted values from the result.

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result
- **preprocess_reply** – optional callable used to preprocess values received from the instrument. The callable returns the processed string.

- **maxsplit** – At most *maxsplit* splits are done. -1 (default) indicates no limit.

Returns A list of the desired type, or strings where the casting fails

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

property waveform_first_point
An integer parameter that specifies the address of the first data point to be sent. For waveforms acquired in sequence mode, this refers to the relative address in the given segment. The first data point starts at zero and is strictly positive.

property waveform_points
An integer parameter that sets the number of waveform points to be transferred with the digitize method. NP = 0 sends all data points.

Note that the oscilloscope may provide less than the specified nb of points.

property waveform_preamble
Get preamble information for the selected waveform source as a dict with the following keys: - "type": normal, peak detect, average, high resolution (str) - "requested_points": number of data points requested by the user (int) - "sampled_points": number of data points sampled by the oscilloscope (int) - "transmitted_points": number of data points actually transmitted (optional) (int) - "memory_size": size of the oscilloscope internal memory in bytes (int) - "sparsing": sparse point. It defines the interval between data points. (int) - "first_point": address of the first data point to be sent (int) - "source": source of the data : "C1", "C2", "C3", "C4", "MATH". - "unit": Physical units of the Y-axis - "type": type of data acquisition. Can be "normal", "peak", "average", "highres" - "average": average times of average acquisition - "sampling_rate": sampling rate (it is a read-only property) - "grid_number": number of horizontal grids (it is a read-only property) - "status": acquisition status of the scope. Can be "stopped", "triggered", "ready", "auto", "armed" - "xdiv": horizontal scale (units per division) in seconds - "xoffset": time interval in seconds between the trigger event and the reference position - "ydiv": vertical scale (units per division) in Volts - "yoffset": value that is represented at center of screen in Volts

property waveform_sparsing
An integer parameter that defines the interval between data points. For example: SP = 0 sends all data points. SP = 4 sends 1 point every 4 data points.

write(*command*, ***kwargs*)
Writes the command to the instrument through the adapter. Note. If the last command was sent less than WRITE_INTERVAL_S before, this method blocks for the remaining time so that commands are never sent with rate more than 1/WRITE_INTERVAL_S Hz.

Parameters **command** – command string to be sent to the instrument

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.29 MKS Instruments

This section contains specific documentation on the MKS Instruments devices that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.29.1 MKS Instruments 937B Vacuum Gauge Controller

class pymeasure.instruments.mksinst.mks937b.**MKS937B**(*adapter*, *name='MKS 937B vacuum gauge controller'*, *address=253*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

MKS 937B vacuum gauge controller

Connection to the device is made through an RS232/RS485 serial connection. The communication protocol of this device is as follows:

Query: '@<aaa><Command>?;FF' with the response '@<aaa>ACK<Response>;FF' Set command: '@<aaa><Command>!<parameter>;FF' with the response '@<aaa>ACK<Response>;FF' Above <aaa> is an address from 001 to 254 which can be specified upon initialization. Since ';FF' is not supported by pyvisa as terminator this class overloads the device communication methods.

Parameters

- **adapter** – pyvisa resource name of the instrument or adapter instance
- **name** (*string*) – The name of the instrument.
- **address** – device address included in every message to the instrument (default=253)
- **kwargs** – Any valid key-word argument for Instrument

property all_pressures
Read pressures on all channels in selected units

check_errors()
check reply string for acknowledgement string

property combined_pressure1
Read pressure on channel 1 and its combination sensor

property combined_pressure2
Read pressure on channel 2 and its combination sensor

read()
Reads from the instrument including the correct termination characters

property serial
Serial number of the instrument

property unit
Pressure unit used for all pressure readings from the instrument

write(*command*)
Writes to the instrument including the device address

Parameters **command** – command string to be sent to the instrument

7.30 Newport

This section contains specific documentation on the Newport instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.30.1 ESP 300 Motion Controller

class pymeasure.instruments.newport.**ESP300**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Newport ESP 300 Motion Controller and provides a high-level for interacting with the instrument.

By default this instrument is constructed with x, y, and phi attributes that represent axes 1, 2, and 3. Custom implementations can overwrite this depending on the avalible axes. Axes are controlled through an *Axis* class.

property axes
A list of the *Axis* objects that are present.

clear_errors()
Clears the error messages by checking until a 0 code is recived.

disable()
Disables all of the axes associated with this controller.

enable()
Enables all of the axes associated with this controller.

property error
Reads an error code from the motion controller.

property errors
Returns a list of error Exceptions that can be later raised, or used to diagnose the situation.

shutdown()
Shuts down the controller by disabling all of the axes.

class pymeasure.instruments.newport.esp300.**Axis**(*axis*, *controller*)

Bases: object

Represents an axis of the Newport ESP300 Motor Controller, which can have independent parameters from the other axes.

define_position(*position*)
Overwrites the value of the current position with the given value.

disable()
Disables motion for the axis.

enable()
Enables motion for the axis.

property enabled
Returns a boolean value that is True if the motion for this axis is enabled.

home(*type=1*)
Drives the axis to the home position, which may be the negative hardware limit for some actuators (e.g. LTA-HS). type can take integer values from 0 to 6.

property left_limit
A floating point property that controls the left software limit of the axis.

property motion_done
Returns a boolean that is True if the motion is finished.

property position
A floating point property that controls the position of the axis. The units are defined based on the actuator. Use the *wait_for_stop()* method to ensure the position is stable.

property right_limit
A floating point property that controls the right software limit of the axis.

property units
A string property that controls the displacement units of the axis, which can take values of: enconder count, motor step, millimeter, micrometer, inches, milli-inches, micro-inches, degree, gradient, radian, milliradian, and microradian.

wait_for_stop(*delay=0*, *interval=0.05*)
Blocks the program until the motion is completed. A further delay can be specified in seconds.

zero()
Resets the axis position to be zero at the current poisiton.

class pymeasure.instruments.newport.esp300.**AxisError**(*code*)
Bases: Exception

Raised when a particular axis causes an error for the Newport ESP300.

class pymeasure.instruments.newport.esp300.**GeneralError**(*code*)
Bases: Exception

Raised when the Newport ESP300 has a general error.

7.31 National Instruments

This section contains specific documentation on the National Instruments instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.31.1 NI Virtual Bench

General Information

The armstrap/pyvirtualbench Python wrapper for the VirtualBench C-API is required. This Instrument driver only interfaces the pyvirtualbench Python wrapper.

Examples

To be documented. Check the examples in the pyvirtualbench repository to get an idea.

Simple Example to switch digital lines of the DIO module.

```
from pymeasure.instruments.ni import VirtualBench

vb = VirtualBench(device_name='VB8012-3057E1C')
line = 'dig/2'  # may be list of lines
# initialize DIO module -> available via vb.dio
vb.acquire_digital_input_output(line, reset=False)
```

(continues on next page)

(continued from previous page)

```
vb.dio.write(self.line, {True})
sleep(1000)
vb.dio.write(self.line, {False})

vb.shutdown()
```

Instrument Class

class pymeasure.instruments.ni.virtualbench.**VirtualBench**(*device_name=''*, *name='VirtualBench'*)

Bases: object

Represents National Instruments Virtual Bench main frame.

Subclasses implement the functionalities of the different modules:

- Mixed-Signal-Oscilloscope (MSO)
- Digital Input Output (DIO)
- Function Generator (FGEN)
- Power Supply (PS)
- Serial Peripheral Interface (SPI) -> not implemented for pymeasure yet
- Inter Integrated Circuit (I2C) -> not implemented for pymeasure yet

For every module exist methods to save/load the configuration to file. These methods are not wrapped so far, checkout the pyvirtualbench file.

All calibration methods and classes are not wrapped so far, since these are not required on a very regular basis. Also the connections via network are not yet implemented. Check the pyvirtualbench file, if you need the functionality.

Parameters

- **device_name** (*str*) – Full unique device name
- **name** (*str*) – Name for display in pymeasure

class DigitalInputOutput(*virtualbench*, *lines*, *reset*, *vb_name=''*)

Bases: pymeasure.instruments.ni.virtualbench.VirtualBench.VirtualBenchInstrument

Represents Digital Input Output (DIO) Module of Virtual Bench device. Allows to read/write digital channels and/or set channels to export the start signal of FGEN module or trigger of MSO module.

export_signal(*line*, *digitalSignalSource*)

Exports a signal to the specified line.

Parameters

- **line** (*str*) – Line string
- **digitalSignalSource** (*int*) – 0 for FGEN start or 1 for MSO trigger

query_export_signal(*line*)

Indicates the signal being exported on the specified line.

Parameters **line** (*str*) – Line string
Returns Exported signal (FGEN start or MSO trigger)
Return type enum

query_line_configuration()
Indicates the current line configurations. Tristate Lines, Static Lines, and Export Lines contain comma-separated range_data and/or colon-delimited lists of all acquired lines

read(*lines*)
Reads the current state of the specified lines.
Parameters **lines** (*str*) – Line string, requires full name specification e.g. `'VB8012-xxxxxxx/dig/0:7'` since instrument_handle is not required (only library_handle)
Returns List of line states (HIGH/LOW)
Return type list

reset_instrument()
Resets the session configuration to default values, and resets the device and driver software to a known state.

shutdown()
Removes the session and deallocates any resources acquired during the session. If output is enabled on any channels, they remain in their current state.

tristate_lines(*lines*)
Sets all specified lines to a high-impedance state. (Default)

validate_lines(*lines*, *return_single_lines=False*, *validate_init=False*)
Validate lines string Allowed patterns (case sensitive):
- `'VBxxxx-xxxxxxx/dig/0:7'`
- `'VBxxxx-xxxxxxx/dig/0'`
- `'dig/0'`
- `'VBxxxx-xxxxxxx/trig'`
- `'trig'`

Allowed Line Numbers: 0-7 or trig

Parameters
- **lines** (*str*) – Line string to test
- **return_single_lines** (*bool, optional*) – Return list of line numbers as well, defaults to False
- **validate_init** (*bool, optional*) – Check if lines are initialized (in `self._line_numbers`), defaults to False

Returns Line string, optional list of single line numbers
Return type str, optional (str, list)

write(*lines*, *data*)
Writes data to the specified lines.
Parameters
- **lines** (*str*) – Line string
- **data** (*list or tuple*) – List of data, (`True` = High, `False` = Low)

class DigitalMultimeter(*virtualbench*, *reset*, *vb_name=''*)
Bases: `pymeasure.instruments.ni.virtualbench.VirtualBench.VirtualBenchInstrument`

Represents Digital Multimeter (DMM) Module of Virtual Bench device. Allows to measure either DC/AC voltage or current, Resistance or Diodes.

configure_ac_current(*auto_range_terminal*)
Configure auto rage terminal for AC current measurement
Parameters **auto_range_terminal** – Terminal to perform auto ranging (`'LOW'` or `'HIGH'`)

configure_dc_current(*auto_range_terminal*)
Configure auto rage terminal for DC current measurement

Parameters **auto_range_terminal** – Terminal to perform auto ranging (`'LOW'` or `'HIGH'`)

configure_dc_voltage(*dmm_input_resistance*)
Configure DC voltage input resistance

Parameters **dmm_input_resistance** (*int or str*) – Input resistance (`'TEN_MEGA_OHM'` or `'TEN_GIGA_OHM'`)

configure_measurement(*dmm_function*, *auto_range=True*, *manual_range=1.0*)
Configure Instrument to take a DMM measurement

Parameters

- **name** (*dmm_function:DMM function index or*) –
  - `'DC_VOLTS'`, `'AC_VOLTS'`
  - `'DC_CURRENT'`, `'AC_CURRENT'`
  - `'RESISTANCE'`
  - `'DIODE'`
- **auto_range** (*bool*) – Enable/Disable auto ranging
- **manual_range** (*float*) – Manually set measurement range

query_ac_current()
Indicates auto range terminal for AC current measurement

query_dc_current()
Indicates auto range terminal for DC current measurement

query_dc_voltage()
Indicates input resistance setting for DC voltage measurement

query_measurement()
Query DMM measurement settings from the instrument

Returns Auto range, range data

Return type (bool, float)

read()
Read measurement value from the instrument

Returns Measurement value

Return type float

reset_instrument()
Reset the DMM module to defaults

shutdown()
Removes the session and deallocates any resources acquired during the session. If output is enabled on any channels, they remain in their current state.

validate_auto_range_terminal(*auto_range_terminal*)
Check value for choosing the auto range terminal for DC current measurement

Parameters **auto_range_terminal** (*int or str*) – Terminal to perform auto ranging (`'LOW'` or `'HIGH'`)

Returns Auto range terminal to pass to the instrument

Return type int

validate_dmm_function(*dmm_function*)
Check if DMM function *dmm_function* exists

Parameters **dmm_function** (*int or str*) – DMM function index or name:

- 'DC_VOLTS', 'AC_VOLTS'
- 'DC_CURRENT', 'AC_CURRENT'
- 'RESISTANCE'
- 'DIODE'

Returns DMM function index to pass to the instrument

Return type int

static validate_range(*dmm_function*, *range*)

Checks if range is valid for the chosen dmm_function

Parameters

- **dmm_function** (*int*) – DMM Function
- **range** (*int or float*) – Range value, e.g. maximum value to measure

Returns Range value to pass to instrument

Return type int

class FunctionGenerator(*virtualbench*, *reset*, *vb_name=''*)

Bases: pymeasure.instruments.ni.virtualbench.VirtualBench.VirtualBenchInstrument

Represents Function Generator (FGEN) Module of Virtual Bench device.

configure_arbitrary_waveform(*waveform*, *sample_period*)

Configures the instrument to output a waveform. The waveform is output either after the end of the current waveform if output is enabled, or immediately after output is enabled.

Parameters

- **waveform** (*list*) – Waveform as list of values
- **sample_period** (*float*) – Time between two waveform points (maximum of 125MS/s, which equals 80ns)

configure_arbitrary_waveform_gain_and_offset(*gain*, *dc_offset*)

Configures the instrument to output an arbitrary waveform with a specified gain and offset value. The waveform is output either after the end of the current waveform if output is enabled, or immediately after output is enabled.

Parameters

- **gain** (*float*) – Gain, multiplier of waveform values
- **dc_offset** (*float*) – DC offset in volts

configure_standard_waveform(*waveform_function*, *amplitude*, *dc_offset*, *frequency*, *duty_cycle*)

Configures the instrument to output a standard waveform. Check instrument manual for maximum ratings which depend on load.

Parameters

- **waveform_function** (*int or str*) – Waveform function ("SINE", "SQUARE", "TRIANGLE/RAMP", "DC")
- **amplitude** (*float*) – Amplitude in volts
- **dc_offset** (*float*) – DC offset in volts
- **frequency** (*float*) – Frequency in Hz

- **duty_cycle** (*int*) – Duty cycle in %

property filter
Enables or disables the filter on the instrument.

Parameters **enable_filter** (*bool*) – Enable/Disable filter

query_arbitrary_waveform()
Returns the samples per second for arbitrary waveform generation.

Returns Samples per second

Return type int

query_arbitrary_waveform_gain_and_offset()
Returns the settings for arbitrary waveform generation that includes gain and offset settings.

Returns Gain, DC offset

Return type (float, float)

query_generation_status()
Returns the status of waveform generation on the instrument.

Returns Status

Return type enum

query_standard_waveform()
Returns the settings for a standard waveform generation.

Returns Waveform function, amplitude, dc_offset, frequency, duty_cycle

Return type (enum, float, float, float, int)

query_waveform_mode()
Indicates whether the waveform output by the instrument is a standard or arbitrary waveform.

Returns Waveform mode

Return type enum

reset_instrument()
Resets the session configuration to default values, and resets the device and driver software to a known state.

run()
Transitions the session from the Stopped state to the Running state.

self_calibrate()
Performs offset nulling calibration on the device. You must run FGEN Initialize prior to running this method.

shutdown()
Removes the session and deallocates any resources acquired during the session. If output is enabled on any channels, they remain in their current state.

stop()
Transitions the acquisition from either the Triggered or Running state to the Stopped state.

class MixedSignalOscilloscope(*virtualbench*, *reset*, *vb_name=''*)
Bases: `pymeasure.instruments.ni.virtualbench.VirtualBench.VirtualBenchInstrument`

Represents Mixed Signal Oscilloscope (MSO) Module of Virtual Bench device. Allows to measure oscilloscope data from analog and digital channels.

Methods from pyvirtualbench not implemented in pymeasure yet:

- enable_digital_channels
- configure_digital_threshold
- configure_advanced_digital_timing
- configure_state_mode
- configure_digital_edge_trigger
- configure_digital_pattern_trigger
- configure_digital_glitch_trigger
- configure_digital_pulse_width_trigger
- query_digital_channel
- query_enabled_digital_channels
- query_digital_threshold
- query_advanced_digital_timing
- query_state_mode
- query_digital_edge_trigger
- query_digital_pattern_trigger
- query_digital_glitch_trigger
- query_digital_pulse_width_trigger
- read_digital_u64

auto_setup()
Automatically configure the instrument

configure_analog_channel(*channel*, *enable_channel*, *vertical_range*, *vertical_offset*, *probe_attenuation*, *vertical_coupling*)
Configure analog measurement channel

Parameters

- **channel** (*str*) – Channel string
- **enable_channel** (*bool*) – Enable/Disable channel
- **vertical_range** (*float*) – Vertical measurement range (0V - 20V), the instrument discretizes to these ranges: `[20, 10, 5, 2, 1, 0.5, 0.2, 0.1, 0.05]` which are 5x the values shown in the native UI.
- **vertical_offset** (*float*) – Vertical offset to correct for (inverted compared to VB native UI, -20V - +20V, resolution 0.1mV)
- **probe_attenuation** (*int or str*) – Probe attenuation (`'ATTENUATION_10X'` or `'ATTENUATION_1X'`)
- **vertical_coupling** (*int or str*) – Vertical coupling (`'AC'` or `'DC'`)

configure_analog_channel_characteristics(*channel*, *input_impedance*, *bandwidth_limit*)
Configure electrical characteristics of the specified channel

Parameters

- **channel** (*str*) – Channel string

- **input_impedance** (*int or str*) – Input Impedance (`'ONE_MEGA_OHM'` or `'FIFTY_OHMS'`)
- **bandwidth_limit** (*int*) – Bandwidth limit (100MHz or 20MHz)

configure_analog_edge_trigger(*trigger_source*, *trigger_slope*, *trigger_level*, *trigger_hysteresis*, *trigger_instance*)

Configures a trigger to activate on the specified source when the analog edge reaches the specified levels.

Parameters

- **trigger_source** (*str*) – Channel string
- **trigger_slope** (*int or str*) – Trigger slope (`'RISING'`, `'FALLING'` or `'EITHER'`)
- **trigger_level** (*float*) – Trigger level
- **trigger_hysteresis** (*float*) – Trigger hysteresis
- **trigger_instance** (*int or str*) – Trigger instance

configure_analog_pulse_width_trigger(*trigger_source*, *trigger_polarity*, *trigger_level*, *comparison_mode*, *lower_limit*, *upper_limit*, *trigger_instance*)

Configures a trigger to activate on the specified source when the analog edge reaches the specified levels within a specified window of time.

Parameters

- **trigger_source** (*str*) – Channel string
- **trigger_polarity** (*int or str*) – Trigger slope (`'POSITIVE'` or `'NEGATIVE'`)
- **trigger_level** (*float*) – Trigger level
- **comparison_mode** (*int or str*) – Mode of compariosn (`'GREATER_THAN_UPPER_LIMIT'`, `'LESS_THAN_LOWER_LIMIT'`, `'INSIDE_LIMITS'` or `'OUTSIDE_LIMITS'`)
- **lower_limit** (*float*) – Lower limit
- **upper_limit** (*float*) – Upper limit
- **trigger_instance** (*int or str*) – Trigger instance

configure_immediate_trigger()

Configures a trigger to immediately activate on the specified channels after the pretrigger time has expired.

configure_timing(*sample_rate*, *acquisition_time*, *pretrigger_time*, *sampling_mode*)

Configure timing settings of the MSO

Parameters

- **sample_rate** (*int*) – Sample rate (15.26kS - 1GS)
- **acquisition_time** (*float*) – Acquisition time (1ns - 68.711s)
- **pretrigger_time** (*float*) – Pretrigger time (0s - 10s)
- **sampling_mode** – Sampling mode (`'SAMPLE'` or `'PEAK_DETECT'`)

configure_trigger_delay(*trigger_delay*)

Configures the amount of time to wait after a trigger condition is met before triggering.

param float trigger_delay Trigger delay (0s - 17.1799s)

`force_trigger()`
Causes a software-timed trigger to occur after the pretrigger time has expired.

`query_acquisition_status()`
Returns the status of a completed or ongoing acquisition.

`query_analog_channel(channel)`
Indicates the vertical configuration of the specified channel.

Returns Channel enabled, vertical range, vertical offset, probe attenuation, vertical coupling

Return type (bool, float, float, enum, enum)

`query_analog_channel_characteristics(channel)`
Indicates the properties that control the electrical characteristics of the specified channel. This method returns an error if too much power is applied to the channel.

return Input impedance, bandwidth limit

rtype (enum, float)

`query_analog_edge_trigger(trigger_instance)`
Indicates the analog edge trigger configuration of the specified instance.

Returns Trigger source, trigger slope, trigger level, trigger hysteresis

Return type (str, enum, float, float)

`query_analog_pulse_width_trigger(trigger_instance)`
Indicates the analog pulse width trigger configuration of the specified instance.

Returns Trigger source, trigger polarity, trigger level, comparison mode, lower limit, upper limit

Return type (str, enum, float, enum, float, float)

`query_enabled_analog_channels()`
Returns String of enabled analog channels.

Returns Enabled analog channels

Return type str

`query_timing()`
Indicates the timing configuration of the MSO. Call directly before measurement to read the actual timing configuration and write it to the corresponding class variables. Necessary to interpret the measurement data, since it contains no time information.

Returns Sample rate, acquisition time, pretrigger time, sampling mode

Return type (float, float, float, enum)

`query_trigger_delay()`
Indicates the trigger delay setting of the MSO.

Returns Trigger delay

Return type float

`query_trigger_type(trigger_instance)`
Indicates the trigger type of the specified instance.

Parameters `trigger_instance` – Trigger instance (`'A'` or `'B'`)

Returns Trigger type

Return type str

`read_analog_digital_dataframe()`
Transfers data from the instrument and returns a pandas dataframe of the analog measurement data, including time coordinates

Returns Dataframe with time and measurement data

Return type pd.DataFrame

`read_analog_digital_u64()`
Transfers data from the instrument as long as the acquisition state is Acquisition Complete. If the state is either Running or Triggered, this method will wait until the state transitions to Acquisition Complete. If the state is Stopped, this method returns an error.

Returns Analog data out, analog data stride, analog t0, digital data out, digital timestamps out, digital t0, trigger timestamp, trigger reason

Return type (list, int, pyvb.Timestamp, list, list, pyvb.Timestamp, pyvb.Timestamp, enum)

`reset_instrument()`
Resets the session configuration to default values, and resets the device and driver software to a known state.

`run(autoTrigger=True)`
Transitions the acquisition from the Stopped state to the Running state. If the current state is Triggered, the acquisition is first transitioned to the Stopped state before transitioning to the Running state. This method returns an error if too much power is applied to any enabled channel.

Parameters **autoTrigger** (*bool*) – Enable/Disable auto triggering

`shutdown()`
Removes the session and deallocates any resources acquired during the session. If output is enabled on any channels, they remain in their current state.

`stop()`
Transitions the acquisition from either the Triggered or Running state to the Stopped state.

`validate_channel(channel)`
Check if `channel` is a correct specification

Parameters **channel** (*str*) – Channel string

Returns Channel string

Return type str

`static validate_trigger_instance(trigger_instance)`
Check if `trigger_instance` is a valid choice

Parameters **trigger_instance** (*int or str*) – Trigger instance (`'A'` or `'B'`)

Returns Trigger instance

Return type int

`class PowerSupply(virtualbench, reset, vb_name='')`
Bases: `pymeasure.instruments.ni.virtualbench.VirtualBench.VirtualBenchInstrument`

Represents Power Supply (PS) Module of Virtual Bench device

configure_current_output(*channel*, *current_level*, *voltage_limit*)
Configures a current output on the specified channel. This method should be called once for every channel you want to configure to output current.

configure_voltage_output(*channel*, *voltage_level*, *current_limit*)
Configures a voltage output on the specified channel. This method should be called once for every channel you want to configure to output voltage.

property outputs_enabled
Enables or disables all outputs on all channels of the instrument.

Parameters enable_outputs (*bool*) – Enable/Disable outputs

query_current_output(*channel*)
Indicates the current output settings on the specified channel.

query_voltage_output(*channel*)
Indicates the voltage output settings on the specified channel.

read_output(*channel*)
Reads the voltage and current levels and outout mode of the specified channel.

reset_instrument()
Resets the session configuration to default values, and resets the device and driver software to a known state.

shutdown()
Removes the session and deallocates any resources acquired during the session. If output is enabled on any channels, they remain in their current state.

property tracking
Enables or disables tracking between the positive and negative 25V channels. If enabled, any configuration change on the positive 25V channel is mirrored to the negative 25V channel, and any writes to the negative 25V channel are ignored.

Parameters enable_tracking (*bool*) – Enable/Disable tracking

validate_channel(*channel*, *current=False*, *voltage=False*)
Check if channel string is valid and if output current/voltage are within the output ranges of the channel

Parameters

- **channel** (*str*) – Channel string ("ps/+6V","ps/+25V","ps/-25V")
- **current** (*bool, optional*) – Current output, defaults to False
- **voltage** (*bool, optional*) – Voltage output, defaults to False

Returns channel or channel, current & voltage

Return type str or (str, float, float)

acquire_digital_input_output(*lines*, *reset=False*)
Establishes communication with the DIO module. This method should be called once per session.

Parameters

- **lines** (*str*) – Lines to acquire, reading is possible on all lines
- **reset** (*bool, optional*) – Reset DIO module, defaults to False

acquire_digital_multimeter(*reset=False*)
Establishes communication with the DMM module. This method should be called once per session.

Parameters **reset** (*bool, optional*) – Reset the DMM module, defaults to False

acquire_function_generator(*reset=False*)
Establishes communication with the FGEN module. This method should be called once per session.

Parameters **reset** (*bool, optional*) – Reset the FGEN module, defaults to False

acquire_mixed_signal_oscilloscope(*reset=False*)
Establishes communication with the MSO module. This method should be called once per session.

Parameters **reset** (*bool, optional*) – Reset the MSO module, defaults to False

acquire_power_supply(*reset=False*)
Establishes communication with the PS module. This method should be called once per session.

Parameters **reset** (*bool, optional*) – Reset the PS module, defaults to False

collapse_channel_string(*names_in*)
Collapses a channel string into a comma and colon-delimited equivalent. Last element is the number of channels.

Parameters **names_in** (*str*) – Channel string

Returns Channel string with colon notation where possible, number of channels

Return type (str, int)

convert_timestamp_to_values(*timestamp*)
Converts a timestamp to seconds and fractional seconds

Parameters **timestamp** (*pyvb.Timestamp*) – VirtualBench timestamp

Returns (seconds_since_1970, fractional seconds)

Return type (int, float)

convert_values_to_datetime(*timestamp*)
Converts timestamp to datetime object

Parameters **timestamp** (*pyvb.Timestamp*) – VirtualBench timestamp

Returns Timestamp as DateTime object

Return type DateTime

convert_values_to_timestamp(*seconds_since_1970*, *fractional_seconds*)
Converts seconds and fractional seconds to a timestamp

Parameters

- **seconds_since_1970** (*int*) – Date/Time in seconds since 1970
- **fractional_seconds** (*float*) – Fractional seconds

Returns VirtualBench timestamp

Return type pyvb.Timestamp

expand_channel_string(*names_in*)
Expands a channel string into a comma-delimited (no colon) equivalent. Last element is the number of channels. `'dig/0:2'` -> `('dig/0, dig/1, dig/2',3)`

Parameters **names_in** (*str*) – Channel string

Returns Channel string with all channels separated by comma, number of channels

Return type (str, int)

get_calibration_information()
Returns calibration information for the specified device, including the last calibration date and calibration interval.

Returns Calibration date, recommended calibration interval in months, calibration interval in months

Return type (pyvb.Timestamp, int, int)

get_library_version()
Return the version of the VirtualBench runtime library

shutdown()
Finalize the VirtualBench library.

class pymeasure.instruments.ni.virtualbench.**VirtualBench_Direct**(**args: Any*, ***kwargs: Any*)
Bases: pyvirtualbench.PyVirtualBench

Represents National Instruments Virtual Bench main frame. This class provides direct access to the armstrap/pyvirtualbench Python wrapper.

7.32 Oxford Instruments

This section contains specific documentation on the Oxford Instruments instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.32.1 Oxford Instruments VISA Adapter

class pymeasure.instruments.oxfordinstruments.**OxfordInstrumentsAdapter**(*resource_name*, *max_attempts=5*, ***kwargs*)
Bases: *pymeasure.adapters.visa.VISAAdapter*

Adapter class for the VISA library using PyVISA to communicate with instruments. Checks the replies from instruments for validity.

Parameters

- **resource_name** – VISA resource name that identifies the address
- **max_attempts** – Integer that sets how many attempts at getting a valid response to a query can be made
- **kwargs** – key-word arguments for constructing a PyVISA Adapter

ask(*command*)
Write the command to the instrument and return the resulting ASCII response. Also check the validity of the response before returning it; if the response is not valid, another attempt is made at getting a valid response, until the maximum amount of attempts is reached.

Parameters **command** – ASCII command string to be sent to the instrument

Returns String ASCII response of the instrument

Raises *OxfordVISAError* if the maximum number of attempts is surpassed without getting a valid response

is_valid_response(*response*, *command*)
Check if the response received from the instrument after a command is valid and understood by the instrument.

Parameters

- **response** – String ASCII response of the device
- **command** – command used in the initial query

Returns True if the response is valid and the response indicates the instrument recognised the command

class pymeasure.instruments.oxfordinstruments.adapters.**OxfordVISAError**
Bases: Exception

7.32.2 Oxford Instruments Intelligent Temperature Controller 503

class pymeasure.instruments.oxfordinstruments.**ITC503**(*adapter*, *name='Oxford ITC503'*, *clear_buffer=True*, *min_temperature=0*, *max_temperature=1677.7*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Oxford Intelligent Temperature Controller 503.

```
itc = ITC503("GPIB::24")        # Default channel for the ITC503

itc.control_mode = "RU"         # Set the control mode to remote
itc.heater_gas_mode = "AUTO"    # Turn on auto heater and flow
itc.auto_pid = True             # Turn on auto-pid

print(itc.temperature_setpoint) # Print the current set-point
itc.temperature_setpoint = 300  # Change the set-point to 300 K
itc.wait_for_temperature()      # Wait for the temperature to stabilize
print(itc.temperature_1)        # Print the temperature at sensor 1
```

class FLOW_CONTROL_STATUS(*value*)
Bases: enum.IntFlag

IntFlag class for decoding the flow control status. Contains the following flags:

bit	flag	meaning
4	HEATER_ERROR_SIGN	Sign of heater-error; True means negative
3	TEMPERATURE_ERROR_SIGN	Sign of temperature-error; True means negative
2	SLOW_VALVE_ACTION	Slow valve action occurring
1	COOLDOWN_TERMINATION	Cooldown-termination occurring
0	FAST_COOLDOWN	Fast-cooldown occurring

property auto_pid
A boolean property that sets the Auto-PID mode on (True) or off (False).

property auto_pid_table
A property that controls values in the auto-pid table. Relies on *ITC503.x_pointer* and *ITC503.y_pointer* (or *ITC503.pointer*) to point at the location in the table that is to be set or read.

The x-pointer selects the table entry (1 to 16); the y-pointer selects the parameter:

y-pointer	parameter
1	upper temperature limit
2	proportional band
3	integral action time
4	derivative action time

property control_mode
A string property that sets the ITC in *local* or *remote* and *locked* or *unlocked*, locking the LOC/REM button. Allowed values are:

value	state
LL	local & locked
RL	remote & locked
LU	local & unlocked
RU	remote & unlocked

property derivative_action_time
A floating point property that controls the derivative action time for the PID controller in minutes. Can be set if the PID controller is in manual mode. Valid values are 0 [min.] to 273 [min.].

property front_panel_display
A string property that controls what value is displayed on the front panel of the ITC. Valid values are: 'temperature setpoint', 'temperature 1', 'temperature 2', 'temperature 3', 'temperature error', 'heater', 'heater voltage', 'gasflow', 'proportional band', 'integral action time', 'derivative action time', 'channel 1 freq/4', 'channel 2 freq/4', 'channel 3 freq/4'.

property gasflow
A floating point property that controls gas flow when in manual mode. The value is expressed as a percentage of the maximum gas flow. Valid values are in range 0 [off] to 99.9 [%].

property gasflow_configuration_parameter
A property that controls the gas flow configuration parameters. Relies on the *ITC503.x_pointer* to select which parameter is set or read:

x-pointer	parameter
1	valve gearing
2	target table & features configuration
3	gas flow scaling
4	temperature error sensitivity
5	heater voltage error sensitivity
6	minimum gas valve in auto

property gasflow_control_status
A property that reads the gas-flow control status. Returns the status in the form of a *ITC503.FLOW_CONTROL_STATUS* IntFlag.

property heater
A floating point property that represents the heater output power as a percentage of the maximum voltage. Can be set if the heater is in manual mode. Valid values are in range 0 [off] to 99.9 [%].

property heater_gas_mode
A string property that sets the heater and gas flow control to *auto* or *manual*. Allowed values are:

value	state
MANUAL	heater & gas manual
AM	heater auto, gas manual
MA	heater manual, gas auto
AUTO	heater & gas auto

property heater_voltage
A floating point property that represents the heater output power in volts. For controlling the heater, use the *ITC503.heater* property.

property integral_action_time
A floating point property that controls the integral action time for the PID controller in minutes. Can be set if the PID controller is in manual mode. Valid values are 0 [min.] to 140 [min.].

property pointer
A tuple property to set pointers into tables for loading and examining values in the table, of format (x, y). The significance and valid values for the pointer depends on what property is to be read or set. The value for x and y can be in the range 0 to 128.

program_sweep(*temperatures*, *sweep_time*, *hold_time*, *steps=None*)
Program a temperature sweep in the controller. Stops any running sweep. After programming the sweep, it can be started using OxfordITC503.sweep_status = 1.

Parameters

- **temperatures** – An array containing the temperatures for the sweep
- **sweep_time** – The time (or an array of times) to sweep to a set-point in minutes (between 0 and 1339.9).
- **hold_time** – The time (or an array of times) to hold at a set-point in minutes (between 0 and 1339.9).
- **steps** – The number of steps in the sweep, if given, the temperatures, sweep_time and hold_time will be interpolated into (approximately) equal segments

property proportional_band
A floating point property that controls the proportional band for the PID controller in Kelvin. Can be set if the PID controller is in manual mode. Valid values are 0 [K] to 1677.7 [K].

property sweep_status
An integer property that sets the sweep status. Values are:

value	meaning
0	Sweep not running
1	Start sweep / sweeping to first set-point
2P - 1	Sweeping to set-point P
2P	Holding at set-point P

property sweep_table
A property that controls values in the sweep table. Relies on *ITC503.x_pointer* and *ITC503.y_pointer* (or *ITC503.pointer*) to point at the location in the table that is to be set or read.

The x-pointer selects the step of the sweep (1 to 16); the y-pointer selects the parameter:

y-pointer	parameter
1	set-point temperature
2	sweep-time to set-point
3	hold-time at set-point

property target_voltage
A float property that reads the current heater target voltage with which the actual heater voltage is being compared. Only valid if gas-flow in auto mode.

property target_voltage_table
A property that controls values in the target heater voltage table. Relies on the *ITC503.x_pointer* to select the entry in the table that is to be set or read (1 to 64).

property temperature_1
Reads the temperature of the sensor 1 in Kelvin.

property temperature_2
Reads the temperature of the sensor 2 in Kelvin.

property temperature_3
Reads the temperature of the sensor 3 in Kelvin.

property temperature_error
Reads the difference between the set-point and the measured temperature in Kelvin. Positive when set-point is larger than measured.

property temperature_setpoint
A floating point property that controls the temperature set-point of the ITC in kelvin. (dynamic)

property valve_scaling
A float property that reads the valve scaling parameter. Only valid if gas-flow in auto mode.

property version
A string property that returns the version of the IPS.

wait_for_temperature(*error=0.01*, *timeout=3600*, *check_interval=0.5*, *stability_interval=10*, *thermalize_interval=300*, *should_stop=<function ITC503.<lambda>>*)
Wait for the ITC to reach the set-point temperature.

Parameters

- **error** – The maximum error in Kelvin under which the temperature is considered at set-point
- **timeout** – The maximum time the waiting is allowed to take. If timeout is exceeded, a TimeoutError is raised. If timeout is None, no timeout will be used.
- **check_interval** – The time between temperature queries to the ITC.
- **stability_interval** – The time over which the temperature_error is to be below error to be considered stable.
- **thermalize_interval** – The time to wait after stabilizing for the system to thermalize.
- **should_stop** – Optional function (returning a bool) to allow the waiting to be stopped before its end.

wipe_sweep_table()
Wipe the currently programmed sweep table.

property x_pointer
An integer property to set pointers into tables for loading and examining values in the table. The significance and valid values for the pointer depends on what property is to be read or set.

property y_pointer
An integer property to set pointers into tables for loading and examining values in the table. The significance and valid values for the pointer depends on what property is to be read or set.

7.32.3 Oxford Instruments Intelligent Power Supply 120-10 for superconducting magnets

class pymeasure.instruments.oxfordinstruments.**IPS120_10**(*adapter*, *name='Oxford IPS'*, *clear_buffer=True*, *switch_heater_heating_delay=None*, *switch_heater_cooling_delay=None*, *field_range=None*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Oxford Superconducting Magnet Power Supply IPS 120-10.

```
ips = IPS120_10("GPIB::25")  # Default channel for the IPS

ips.enable_control()         # Enables the power supply and remote control

ips.train_magnet([           # Train the magnet after it has been cooled-down
    (11.8, 1.0),
    (13.9, 0.4),
    (14.9, 0.2),
    (16.0, 0.1),
])

ips.set_field(12)           # Bring the magnet to 12 T. The switch heater will
                            # be turned off when the field is reached and the
                            # current is ramped back to 0 (i.e. persistent mode).

print(self.field)           # Print the current field (whether in persistent or
                            # non-persistent mode)

ips.set_field(0)            # Bring the magnet to 0 T. The persistent mode will be
                            # turned off first (i.e. current back to set-point and
                            # switch-heater on); afterwards the switch-heater will
                            # again be turned off.

ips.disable_control()       # Disables the control of the supply, turns off the
                            # switch-heater and clamps the output.
```

Parameters

- **clear_buffer** – A boolean property that controls whether the instrument buffer is clear upon initialisation.
- **switch_heater_heating_delay** – The time in seconds (default is 20s) to wait after the switch-heater is turned on before the heater is expected to be heated.

- **switch_heater_cooling_delay** – The time in seconds (default is 20s) to wait after the switch-heater is turned off before the heater is expected to be cooled down.
- **field_range** – A numeric value or a tuple of two values to indicate the lowest and highest allowed magnetic fields. If a numeric value is provided the range is expected to be from `-field_range` to `+field_range`. The default range is -7 to +7 Tesla.

property activity
A string property that controls the activity of the IPS. Valid values are "hold", "to setpoint", "to zero" and "clamp"

property control_mode
A string property that sets the IPS in *local* or *remote* and *locked* or *unlocked*, locking the LOC/REM button. Allowed values are:

value	state
LL	local & locked
RL	remote & locked
LU	local & unlocked
RU	remote & unlocked

property current_measured
A floating point property that returns the measured magnet current of the IPS in amps. (dynamic)

property current_setpoint
A floating point property that controls the magnet current set-point of the IPS in ampere. (dynamic)

property demand_current
A floating point property that returns the demand magnet current of the IPS in amps. (dynamic)

property demand_field
A floating point property that returns the demand magnetic field of the IPS in Tesla. (dynamic)

disable_control()
Disable active control of the IPS (if at 0T) by turning off the switch heater, clamping the output and setting control to local. Raise a *MagnetError* if field not at 0T.

disable_persistent_mode()
Disable the persistent magnetic field mode. Raise a *MagnetError* if the magnet is not at rest.

enable_control()
Enable active control of the IPS by setting control to remote and turning off the clamp.

enable_persistent_mode()
Enable the persistent magnetic field mode. Raise a *MagnetError* if the magnet is not at rest.

property field
Property that returns the current magnetic field value in Tesla.

property field_setpoint
A floating point property that controls the magnetic field set-point of the IPS in Tesla. (dynamic)

property persistent_field
A floating point property that returns the persistent magnetic field of the IPS in Tesla. (dynamic)

set_field(*field*, *sweep_rate=None*, *persistent_mode_control=True*)
Change the applied magnetic field to a new specified magnitude. If allowed (via *persistent_mode_control*) the persistent mode will be turned off if needed and turned on when the magnetic field is reached. When the new field set-point is 0, the set-point of the instrument will not be changed but rather the *to zero*

functionality will be used. Also, the persistent mode will not turned on upon reaching the 0T field in this case.

Parameters

- **field** – The new set-point for the magnetic field in Tesla.
- **sweep_rate** – A numeric value that controls the rate with which to change the magnetic field in Tesla/minute.
- **persistent_mode_control** – A boolean that controls whether the persistent mode may be turned off (if needed before sweeping) and on (when the field is reached); if set to `False` but the system is in persistent mode, a *MagnetError* will be raised and the magnetic field will not be changed.

property sweep_rate
A floating point property that controls the sweep-rate of the IPS in Tesla/minute. (dynamic)

property sweep_status
A string property that returns the current sweeping mode of the IPS.

property switch_heater_enabled
A boolean property that controls whether the switch heater is enabled or not. When the switch heater is enabled (`True`), the switch is closed and the switch is open and the current in the magnet can be controlled; when the switch heater is disabled (`False`) the switch is closed and the current in the magnet cannot be controlled.

When turning on the switch heater with `True`, the switch heater is only activated if the current of the power supply matches the last recorded current in the magnet.

> **Warning:** These checks can be omitted by using `"Force"` in stead of `True`. Caution: Not performing these checks can cause serious damage to both the power supply and the magnet.

After turning on the switch heater it is necessary to wait several seconds for the switch the respond.

Raises a *SwitchHeaterError* if the system reports a 'heater fault' or if no switch is fitted on the system upon getting the status.

property switch_heater_status
An integer property that returns the switch heater status of the IPS. Use the *switch_heater_enabled* property for controlling and reading the switch heater. When using this property, the user is referred to the IPS120-10 manual for the meaning of the integer values.

train_magnet(*training_scheme*)
Train the magnet after cooling down. Afterwards, set the field back to 0 tesla (at last-used ramp-rate).

Parameters **training_scheme** – The training scheme as a list of tuples; each tuple should consist of a (field [T], ramp-rate [T/min]) pair.

property version
A string property that returns the version of the IPS.

wait_for_idle(*delay=1*, *max_wait_time=None*, *should_stop=<function IPS120_10.<lambda>>*)
Wait until the system is at rest (i.e. current of field not ramping).

Parameters

- **delay** – Time in seconds between each query into the state of the instrument.

- **max_wait_time** – Maximum time in seconds to wait before is at rest. If the system is not at rest within this time a `TimeoutError` is raised. `None` is interpreted as no maximum time.
- **should_stop** – A function that returns `True` when this function should return early.

class pymeasure.instruments.oxfordinstruments.ips120_10.**MagnetError**
Bases: `ValueError`

Exception that is raised for issues regarding the state of the magnet or power supply.

class pymeasure.instruments.oxfordinstruments.ips120_10.**SwitchHeaterError**
Bases: `ValueError`

Exception that is raised for issues regarding the state of the superconducting switch.

7.32.4 Oxford Instruments Power Supply 120-10 for superconducting magnets

class pymeasure.instruments.oxfordinstruments.**PS120_10**(*adapter*, *name='Oxford PS'*, ***kwargs*)
Bases: *pymeasure.instruments.oxfordinstruments.ips120_10.IPS120_10*

Represents the Oxford Superconducting Magnet Power Supply PS 120-10.

```
ps = PS120_10("GPIB::25")   # Default channel for the IPS

ps.enable_control()         # Enables the power supply and remote control

ps.train_magnet([           # Train the magnet after it has been cooled-down
    (11.8, 1.0),
    (13.9, 0.4),
    (14.9, 0.2),
    (16.0, 0.1),
])

ps.set_field(12)            # Bring the magnet to 12 T. The switch heater will
                            # be turned off when the field is reached and the
                            # current is ramped back to 0 (i.e. persistent mode).

print(self.field)           # Print the current field (whether in persistent or
                            # non-persistent mode)

ps.set_field(0)             # Bring the magnet to 0 T. The persistent mode will be
                            # turned off first (i.e. current back to set-point and
                            # switch-heater on); afterwards the switch-heater will
                            # again be turned off.

ps.disable_control()        # Disables the control of the supply, turns off the
                            # switch-heater and clamps the output.
```

Parameters

- **clear_buffer** – A boolean property that controls whether the instrument buffer is clear upon initialisation.
- **switch_heater_heating_delay** – The time in seconds (default is 20s) to wait after the switch-heater is turned on before the heater is expected to be heated.

- **switch_heater_cooling_delay** – The time in seconds (default is 20s) to wait after the switch-heater is turned off before the heater is expected to be cooled down.
- **field_range** – A numeric value or a tuple of two values to indicate the lowest and highest allowed magnetic fields. If a numeric value is provided the range is expected to be from `-field_range` to `+field_range`.

class pymeasure.instruments.oxfordinstruments.ips120_10.**MagnetError**
Bases: `ValueError`

Exception that is raised for issues regarding the state of the magnet or power supply.

class pymeasure.instruments.oxfordinstruments.ips120_10.**SwitchHeaterError**
Bases: `ValueError`

Exception that is raised for issues regarding the state of the superconducting switch.

7.33 Parker

This section contains specific documentation on the Parker instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.33.1 Parker GV6 Servo Motor Controller

class pymeasure.instruments.parker.**ParkerGV6**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Parker Gemini GV6 Servo Motor Controller and provides a high-level interface for interacting with the instrument

property angle
Returns the angle in degrees based on the position and whether relative or absolute positioning is enabled, returning None on error

property angle_error
Returns the angle error in degrees based on the position error, or returns None on error

disable()
Disables the motor from moving

enable()
Enables the motor to move

is_moving()
Returns True if the motor is currently moving

kill()
Stops the motor

move()
Initiates the motor to move to the setpoint

property position
Returns an integer number of counts that correspond to the angular position where 1 revolution equals 4000 counts

property position_error
Returns the error in the number of counts that corresponds to the error in the angular position where 1 revolution equals 4000 counts

read()
Overwrites the Instrument.read command to provide the correct functionality

reset()
Resets the motor controller while blocking and (CAUTION) resets the absolute position value of the motor

set_defaults()
Sets up the default values for the motor, which is run upon construction

set_hardware_limits(*positive=True*, *negative=True*)
Enables (True) or disables (False) the hardware limits for the motor

set_software_limits(*positive*, *negative*)
Sets the software limits for motion based on the count unit where 4000 counts is 1 revolution

property status
Returns a list of the motor status in readable format

stop()
Stops the motor during movement

use_absolute_position()
Sets the motor to accept setpoints from an absolute zero position

use_relative_position()
Sets the motor to accept setpoints that are relative to the last position

7.34 Pendulum

This section contains specific documentation on the Pendulum instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.34.1 Pendulum CNT91 frequency counter

class pymeasure.instruments.pendulum.cnt91.**CNT91**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents a Pendulum CNT-91 frequency counter.

property batch_size
Maximum number of buffer entries that can be transmitted at once.

buffer_frequency_time_series(*channel*, *n_samples*, *sample_rate*, *trigger_source=None*)
Record a time series to the buffer and read it out after completion.

Parameters

- **channel** – Channel that should be used
- **n_samples** – The number of samples
- **sample_rate** – Sample rate in Hz
- **trigger_source** – Optionally specify a trigger source to start the measurement

configure_frequency_array_measurement(*n_samples*, *channel*)
Configure the counter for an array of measurements.

Parameters

- **n_samples** – The number of samples

- **channel** – Measurment channel (A, B, C, E, INTREF)

property continuous
Controls whether to perform continuous measurements.

property external_arming_start_slope
Set slope for the start arming condition.

property external_start_arming_source
Select arming input or switch off the start arming function. Options are 'A', 'B' and 'E' (rear). 'IMM' turns trigger off.

property format
Reponse format (ASCII or REAL).

property interpolator_autocalibrated
Controls if interpolators should be calibrated automatically.

property measurement_time
Gate time for one measurement in s.

read_buffer(*expected_length=0*)
Read out the entire buffer.

Parameters **expected_length** – The expected length of the buffer. If more data is read, values at the end are removed. Defaults to 0, which means that the entire buffer is returned independent of its length.

Returns Frequency values from the buffer.

7.35 Razorbill

This section contains specific documentation on the Razorbill instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.35.1 Razorbill RP100 custrom power supply for Razorbill Instrums stress & strain cells

class pymeasure.instruments.razorbill.**razorbillRP100**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents Razorbill RP100 strain cell controller

```
scontrol = razorbillRP100("ASRL/dev/ttyACM0::INSTR")

scontrol.output_1 = True      # turns output on
scontrol.slew_rate_1 = 1      # sets slew rate to 1V/s
scontrol.voltage_1 = 10       # sets voltage on output 1 to 10V
```

property contact_current_1
Returns the current in amps present at the front panel output of channel 1

property contact_current_2
Returns the current in amps present at the front panel output of channel 2

property contact_voltage_1
Returns the Voltage in volts present at the front panel output of channel 1

property contact_voltage_2
Returns the Voltage in volts present at the front panel output of channel 2

property instant_voltage_1
Returns the instantaneous output of source one in volts

property instant_voltage_2
Returns the instanteneous output of source two in volts

property output_1
Turns output of channel 1 on or off

property output_2
Turns output of channel 2 on or off

property slew_rate_1
Sets or queries the source slew rate in volts/sec of channel 1

property slew_rate_2
Sets or queries the source slew rate in volts/sec of channel 2

property voltage_1
Sets or queries the output voltage of channel 1

property voltage_2
Sets or queries the output voltage of channel 2

7.36 Rohde & Schwarz

This section contains specific documentation on the Rohde & Schwarz instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.36.1 R&S SFM TV test transmitter

class pymeasure.instruments.rohdeschwarz.sfm.**SFM**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Rohde&Schwarz SFM TV test transmitter interface for interacting with the instrument.

Note: The current implementation only works with the first system in this unit.

Further source extension for system 2-6 would be required.

The intermodulation subsystem is also not yet implmented.

---

property R75_out
A bool property that controls the use of the 75R output (if installed)

Value	Meaning
False	50R output active (N)
True	75R output active (BNC)

refer also to chapter 3.6.5 of the manual

property TV_country

A string property that controls the country specifics of the video/sound system to be used

Possible values are:

Value	Meaning
BG_G	BG General
DK_G	DK General
I_G	I General
L_G	L General
GERM	Germany
BELG	Belgium
NETH	Netherlands
FIN	Finland
AUST	Australia
BG_T	BG Th
DENM	Denmark
NORW	Norway
SWED	Sweden
GUS	Russia
POL1	Poland
POL2	Poland
HUNG	Hungary
CHEC	Czech Republic
CHINA1	China
CHINA2	China
GRE	Great Britain
SAFR	South Africa
FRAN	France
USA	United States
KOR	Korea
JAP	Japan
CAN	Canada
SAM	South America

Please confirm with the manual about the details for these settings.

property TV_standard

A string property that controls the type of video standard

Possible values are:

Value	Lines	System
BG	625	PAL
DK	625	SECAM
I	625	PAL
K1	625	SECAM
L	625	SECAM
M	525	NTSC
N	625	NTSC

Please confirm with the manual about the details for these settings.

property basic_info
A String property containing infomation about the hardware modules installed in the unit

property beeper_enabled
A bool property that controls the beeper status,

refer also to chapter 3.6.8 of the manual

calibration(*number=1*, *subsystem=None*)
Function to either calibrate the whole modulator, when subsystem parameter is omitted, or calibrate a subsystem of the modulator.

Valid subsystem selections: "NICam, VISion, SOUNd1, SOUNd2, CODer"

channel_down_relative()
Decreases the output frequency to the next low channel/special channel based on the current country settings

property channel_sweep_start
A float property controlling the start frequency for channel sweep in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property channel_sweep_step
A float property controlling the start frequency for channel sweep in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property channel_sweep_stop
A float property controlling the start frequency for channel sweep in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property channel_table
A string property controlling which channel table is used

Possible selections are:

Value	Meaning
DEF	Default channel table
USR1	User table No. 1
USR2	User table No. 2
USR3	User table No. 3
USR4	User table No. 4
USR5	User table No. 5

refer also to chapter 3.6.6.1 of the manual

channel_up_relative()
Increases the output frequency to the next higher channel/special channel based on the current country settings

coder_adjust()
Starts the automatic setting of the differential deviation

refer also to chapter 3.6.6.4 of the manual

property coder_id_frequency
A int property that controls the frequency of the identification of the coder

valid range 0 .. 200 Hz

property coder_modulation_degree
A float property that controls the modulation degree of the identification of the coder

valid range: 0 .. 0.9

property coder_pilot_deviation
A int property that controls deviation of the pilot frequency of the coder

valid range: 1 .. 4 kHz

property coder_pilot_frequency
A int property that controls the pilot frequency of the coder

valid range: 40 .. 60 kHz

property cw_frequency
A float property controlling the CW-frequency in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property date
A list property for the date of the RTC in the unit

property event_reg
Content of the event register of the Status Operation Register refer also to chapter 3.6.7 of the manual

property ext_ref_base_unit
A bool property for the external reference for the basic unit

Value	Meaning
False	Internal 10 MHz is used
True	External 10 MHz is used

property ext_ref_extension
A bool property for the external reference for the extension frame

Value	Meaning
False	Internal 10 MHz is used
True	External 10 MHz is used

property ext_vid_connector
A string property controlling which connector is used as the input of the video source

Possible selections are:

Value	Meaning
HIGH	Front connector - Hi-Z
LOW	Front connector - 75R
REAR1	Rear connector 1
REAR2	Rear connector 2
AUTO	Automatic assignment

property external_modulation_frequency
A int property that controls the setting for the external modulator frequency

valid range: 32 .. 46 MHz

property external_modulation_power
A int property that controls the setting for the external modulator output power

valid range: -7..0 dBm

refer also to chapter 3.6.6.5 of the manual

property external_modulation_source
A bool property for the modulation source selection

refer also to chapter 3.6.6.8 of the manual

property frequency
A float property controlling the frequency in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property frequency_mode
A string property controlling which the unit is used in

Possible selections are:

Value	Meaning
CW	Continous wave mode
FIXED	fixed frequency mode
CHSW	Channel sweep
RFSW	Frequency sweep

Note: selecting the sweep mode, will start the sweep imemdiately!

property gpib_address
A int property that controls the GPIB address of the unit

valid range: 0..30

property high_frequency_resolution
A property that controls the frequency resolution,

Possible selections are:

Value	Meaning
False	Low resolution (1000Hz)
True	High resolution (1Hz)

property level
A float property controlling the output level in dBm,

- Minimum -99dBm
- Maximum 10dBm (depending on output mode)

refer also to chapter 3.6.6.2 of the manual

property level_mode
A string property controlling the output attenuator and linearity mode

Possible selections are:

Value	Meaning	max. output level
NORM	Normal mode	+6 dBm
LOWN	low noise mode	+10 dBm
CONT	continous mode	+10 dBm
LOWD	low distortion mode	+0 dBm

Contiuous mode allows up to 14 dB of level setting without use of the mechanical attenuator.

property lower_sideband_enabled
A bool property that controls the use of the lower sideband

refer also to chapter 3.6.6.10 of the manual

property modulation_enabled
A bool property that controls the modulation status

property nicam_IQ_inverted
A bool property that controls if the NICAM IQ signals are inverted or not

Value	Meaning
False	normal (IQ)
True	inverted (QI)

property nicam_additional_bits
A int property that controls the additional data in the NICAM modulator

valid range: 0 .. 2047

property nicam_audio_frequency
A int property that controls the frequency of the internal sound generator

valid range: 0 Hz .. 15 kHz

property nicam_audio_volume
A float property that controls the audio volume in the NICAM modulator in dB

valid range: 0..60 dB

property nicam_bit_error_enabled
A bool property that controls the status of an artifical bit error rate to be applied

property nicam_bit_error_rate
A float property that controls the artifical bit error rate.

valid range: 1.2E-7 .. 2E-3

property nicam_carrier_enabled
A bool property that controls if the NICAM carrier is switched on or off

property nicam_carrier_frequency
A float property that controls the frequency of the NICAM carrier

valid range: 33.05 MHz +/- 0.2 Mhz

property nicam_carrier_level
A float property that controls the value of the NICAM carrier

valid range: -40 .. -13 dB

property nicam_control_bits
A int property that controls the additional data in the NICAM modulator

valid range: 0 .. 3

property nicam_data
A int property that controls the data in the NICAM modulator

valid range: 0 .. 2047

property nicam_intercarrier_frequency
A float property that controls the inter-carrier frequency of the NICAM carrier

valid range: 5 .. 9 MHz

property nicam_mode
A string property that controls the signal type to be sent via NICAM

Possible values are:

Value	Meaning
MON	Mono sound + NICAM data
STER	Stereo sound
DUAL	Dual channel sound
DATA	NICAM data only

refer also to chapter 3.6.6.6 of the manual

property nicam_preemphasis_enabled
A bool property that controls the status of the J17 preemphasis

property nicam_source
A string property that controls the signal source for NICAM

Possible values are:

Value	Meaning
INT	Internal audio generator(s)
EXT	External audio source
CW	Continous wave signal
RAND	Random data stream
TEST	Test signal

property nicam_test_signal
A int property that controls the selection of the test signal applied

Value	Meaning
1	Test signal 1 (91 kHz square wave, I&Q 90deg apart)
2	Test signal 2 (45.5 kHz square wave, I&Q 90deg apart)
3	Test signal 3 (182 kHz sine wave, I&Q in phase)

property normal_channel
A int property controlling the current selected regular/normal channel number valid selections are based on the country settings.

property operation_enable_reg
Content of the enable register of the Status Operation Register

Valid range: 0…32767

property output_voltage
A float property controlling the output level in Volt,

Minimum 2.50891e-6, Maximum 0.707068 (depending on output mode) refer also to chapter 3.6.6.12 of the manual

property questionable_event_reg
Content of the event register of the Status Questionable Operation Register

property questionable_operation_enable_reg
Content of the enable register of the Status Questionable Operation Register

Valid range 0…32767

property questionanble_status_reg
Content of the condition register of the Status Questionable Operation Register

property remote_interfaces
A string property controlling the selection of interfaces for remote control

Possible selections are:

Value	Meaning
OFF	no remote control
GPIB	GPIB only enabled
SER	RS232 only enabled
BOTH	GPIB & RS232 enabled

property rf_out_enabled
A bool property that controls the status of the RF-output

property rf_sweep_center
A float property controlling the center frequency for sweep in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property rf_sweep_span
A float property controlling the sweep span in Hz,

- Minimum 1 kHz
- Maximum 1 GHz

property rf_sweep_start
A float property controlling the start frequency for sweep in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property rf_sweep_step
A float property controlling the stepwidth for sweep in Hz,

- Minimum 1 kHz
- Maximum 1 GHz

property rf_sweep_stop
A float property controlling the stop frequency for sweep in Hz

- Minimum 5 MHz
- Maximum 1 GHz

property scale_volt
A string property that controls the unit to be used for voltage entries on the unit

Possible values are: AV,FV, PV, NV, UV, MV, V, KV, MAV, GV, TV, PEV, EV, DBAV, DBFV, DBPV, DBNV, DBUV, DBMV, DBV, DBKV, DBMAv, DBGV, DBTV, DBPEv, DBEV

refer also to chapter 3.6.9 of the manual

property serial_baud
A int property that controls the serial communication speed ,

Possible values are: 110,300,600,1200,4800,9600,19200

property serial_bits
A int property that controls the number of bits used in serial communication

Possible values are: 7 or 8

property serial_flowcontrol
A string property that controls the serial handshake type used in serial communication

Possible values are:

Value	Meaning
NONE	no flow-control/handshake
XON	XON/XOFF flow-control
ACK	hardware handshake with RTS&CTS

property serial_parity
A string property that controls the parity type used for serial communication

Possible values are:

Value	Meaning
NONE	no parity
EVEN	even parity
ODD	odd parity
ONE	parity bit fixed to 1
ZERO	parity bit fixed to 0

property serial_stopbits
A int property that controls the number of stop-bits used in serial communication,

Possible values are: 1 or 2

property sound_mode
A string property that controls the type of audio signal

Possible values are:

Value	Meaning
MONO	MOnoaural sound
PIL	pilot-carrier + mono
BTSC	BTSC + mono
STER	Stereo sound
DUAL	Dual channel sound
NIC	NICAM + Mono

property special_channel
A int property controlling the current selected special channel number valid selections are based on the country settings.

property status_info_shown
A bool property that controls if the display shows infomation during remote control

status_preset()
partly resets the SCPI status reporting structures

property status_reg
Content of the condition register of the Status Operation Register

property subsystem_info
A String property containing infomation about the system configuration

property system_number
A int property for the selected systems (if more than 1 available)

- Minimum 1
- Maximum 6

property time
A list property for the time of the RTC in the unit

property vision_average_enabled
A bool property that controls the average mode for the vision system

property vision_balance
A float property that controls the balance of the vision modulator

valid range: -0.5 .. 0.5

property vision_carrier_enabled
A bool property that controls the vision carrier status

refer also to chapter 3.6.6.9 of the manual

property vision_carrier_frequency
A float property that controls the frequency of the vision carrier

valid range: 32 .. 46 MHz

property vision_clamping_average
A float property that controls the operation point of the vision modulator

valid range: -0.5 .. 0.5

property vision_clamping_enabled
A bool property that controls the clamping behavior of the vision modulator

property vision_clamping_mode
A string property that controls the clamping mode of the vision modulator

Possible selections are HARD or SOFT

property vision_precorrection_enabled
A bool property that controls the precorrection behavior of the vision modulator

property vision_residual_carrier_level
A float property that controls the value of the residual carrier

valid range: 0 .. 0.3 (30%)

property vision_sideband_filter_enabled
A bool property that controls the use of the VSBF (vestigal sideband filter) in the vision modulator

property vision_videosignal_enabled
A bool property that controls if the video signal is switched on or off

class pymeasure.instruments.rohdeschwarz.sfm.**Sound_Channel**(*instrument*, *number*)
Bases: object

Class object for the two sound channels

refere also to chapter 3.6.6.7 of the user manual

property carrier_enabled
A bool property that controls if the audio carrier is switched on or off

property carrier_frequency
A float property that controls the frequency of the sound carrier

valid range: 32 .. 46 MHz

property carrier_level
A float property that controls the level of the audio carrier in dB relative to the vision carrier (0dB)

valid range: -34 .. -6 dB

property deviation
A int property that controls deviation of the selected audio signal

valid range: 0 .. 110 kHz

property frequency
A int property that controls the frequency of the internal sound generator

valid range: 300 Hz .. 15 kHz

property modulation_degree
A float property that controls the modulation depth for the audio signal (Note: only for the use of AM in Standard L)

valid range: 0 .. 1 (100%)

property modulation_enabled
A bool property that controls the audio modulation status

Value	Meaning
False	modulation disabled
True	modulation enabled

property preemphasis_enabled
A bool property that controls if the preemphasis for the audio is switched on or off

property preemphasis_time
A int property that controls if the mode of the preemphasis for the audio signal

Value	Meaning
50	50 us preemphasis
75	75 us preemphasis

property use_external_source
A bool property for the audio source selection

Value	Meaning
False	Internal audio generator(s)
True	External signal source

values(*command*, ***kwargs*)
Reads a set of values from the instrument through the adapter, passing on any keyword arguments.

7.36.2 R&S FSL spectrum analyzer

Connecting to the instrument via network

Once connected to the network, the instrument's IP address can be found by clicking the "Setup" button and navigating to "General Settings" -> "Network Address".

It can then be connected like this:

```
from pymeasure.instruments.rohdeschwarz import FSL
fsl = FSL("TCPIP::192.168.1.123::INSTR")
```

Getting and setting parameters

Most parameters are implemented as properties, which means they can be read and written (getting and setting) in a consistent and simple way. If numerical values are provided, base units are used (s, Hz, dB, ...). Alternatively, the values can also be provided with a unit, e.g. `"1.5 GHz"` or `"1.5GHz"`. Return values are always numerical.

```
# Getting the current center frequency
fsl.freq_center

9000000000.0
```

```
# Changing it to 10 MHz by providing the numerical value
fsl.freq_center = 10e6
```

```
# Verifying:
fsl.freq_center

10000000.0
```

```
# Changing it to 9 GHz by providing a string and verifying the result
fsl.freq_center = '9GHz'
fsl.freq_center

9000000000.0
```

```
# Setting the span to maximum
fsl.freq_span = '7 GHz'
```

Reading a trace

We will read the current trace

```
x, y = fsl.read_trace()
```

Markers

Markers are implemented as their own class. You can create them like this:

```
m1 = fsl.create_marker()
```

Set peak exursion:

```
m1.peak_excursion = 3
```

Set marker to a specific position:

```
m1.x = 10e9
```

Find the next peak to the left and get the level:

```
m1.to_next_peak('left')
m1.y

-34.9349060059
```

Delta markers

Delta markers can be created by setting the appropriate keyword.

```
d2 = fsl.create_marker(is_delta_marker=True)
d2.name

'DELT2'
```

Example program

Here is an example of a simple script for recording the peak of a signal.

```
m1 = fsl.create_marker() # create marker 1

# Set standard settings, set to full span
fsl.continuous_sweep = False
fsl.freq_span = '18 GHz'
fsl.res_bandwidth = "AUTO"
fsl.video_bandwidth = "AUTO"
fsl.sweep_time = "AUTO"

# Perform a sweep on full span, set the marker to the peak and some to that marker
fsl.single_sweep()
m1.to_peak()
m1.zoom('20 MHz')

# take data from the zoomed-in region
fsl.single_sweep()
x, y = fsl.read_trace()
```

class pymeasure.instruments.rohdeschwarz.fsl.**FSL**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents a Rohde&Schwarz FSL spectrum analyzer.

All physical values that can be set can either be as a string of a value and a unit (e.g. "1.2 GHz") or as a float value in the base units (Hz, dBm, etc.).

property attenuation
Attenuation in dB.

continue_single_sweep()
Continue with single sweep with synchronization.

property continuous_sweep
Continuous (True) or single sweep (False)

create_marker(*num=1*, *is_delta_marker=False*)
Create a marker.

Parameters

- **num** – The marker number (1-4)
- **is_delta_marker** – True if the marker is a delta marker, default is False.

Returns The marker object.

property freq_center
Center frequency in Hz.

property freq_span
Frequency span in Hz.

property freq_start
Start frequency in Hz.

property freq_stop
Stop frequency in Hz.

read_trace(*n_trace=1*)
Read trace data.

Parameters **n_trace** – The trace number (1-6). Default is 1.

Returns 2d numpy array of the trace data, [[frequency], [amplitude]].

property res_bandwidth
Resolution bandwidth in Hz. Can be set to 'AUTO'

single_sweep()
Perform a single sweep with synchronization.

property sweep_time
Sweep time in s. Can be set to 'AUTO'.

property trace_mode
Trace mode ('WRIT', 'MAXH', 'MINH', 'AVER' or 'VIEW')

property video_bandwidth
Video bandwidth in Hz. Can be set to 'AUTO'

7.36.3 R&S HMP4040 Power Supply

class pymeasure.instruments.rohdeschwarz.hmp.**HMP4040**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents a Rohde&Schwarz HMP4040 power supply.

beep()
Emit a single beep from the instrument.

clear_sequence(*channel*)
Clear the sequence of the selected channel.

property control_method
Enables manual front panel ('LOC'), remote ('REM') or manual/remote control('MIX') control or locks the the front panel control ('RWL').

property current
Output current in A. Range depends on instrument type.

property current_step
Current step in A.

current_to_max()
Set current of the selected channel to its maximum value.

current_to_min()
Set current of the selected channel to its minimum value.

load_sequence(*slot*)
Load a saved waveform from internal memory (slot 1, 2 or 3).

property max_current
Maximum current in A.

property max_voltage
Maximum voltage in V.

property measured_current
Measured current in A.

property measured_voltage
Measured voltage in V.

property min_current
Minimum current in A.

property min_voltage
Minimum voltage in V.

property output_enabled
Set the output on or off or check the output status.

property repetitions
Number of repetitions (0…255). If 0 is entered, the sequence isrepeated indefinitely.

save_sequence(*slot*)
Save the sequence defined in the sequence property to internal memory (slot 1, 2 or 3).

property selected_channel
Selected channel.

property selected_channel_active
Set the selected channel to active or inactive or check its status.

property sequence
Define sequence of triplets of voltage (V), current (A) and dwell time (s).

set_channel_state(*channel*, *state*)
Set the state of the channel to active or inactive.

Parameters

- **channel** (*int*) – Channel number to set the state of.
- **state** (*bool*) – State of the channel, i.e. True for active, False for inactive.

start_sequence(*channel*)
Start the sequence of the selected channel.

step_current_down()
Decreases current by one step.

step_current_up()
Increase current by one step.

step_voltage_down()
Decrease voltage by one step.

step_voltage_up()
Increase voltage by one step.

stop_sequence(*channel*)
Stop the sequence defined in the sequence property of the selected channel.

transfer_sequence(*channel*)
Transfer the sequence defined in the sequence property to the selected channel.

property version
The SCPI version the instrument's command set complies with.

property voltage
Output voltage in V. Increment 0.001 V.

property voltage_and_current
Output voltage (V) and current (A).

property voltage_step
Voltage step in V. Default 1 V.

voltage_to_max()
Set voltage of the selected channel to its maximum value.

voltage_to_min()
Set voltage of the selected channel to its minimum value.

7.37 Siglent Technologies

This section contains specific documentation on the Siglent Technologies instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.37.1 Siglent Technologies Base Class

class pymeasure.instruments.siglenttechnologies.siglent_spdbase.**SPDBase**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

The base class for Siglent SPDxxxxX instruments.

enable_local_interface(*enable: bool = True*)
Configure the availability of the local interface.

Type bool `True`: enables the local interface `False`: disables it.

property error
Read the error code and information of the instrument.

Type string

property fw_version
Read the software version of the instrument.

Type string

recall_config(*index*)
Recall a config from memory.

Parameters **index** – int: index of the location from which to recall the configuration

save_config(*index*)
Save the current config to memory.

Parameters **index** – int: index of the location to save the configuration

property selected_channel
Control the selected channel of the instrument.

:type : int (dynamic)

shutdown()
Ensure that the voltage is turned to zero and disable the output.

property system_status_code
Read the system status register.

Type *SystemStatusCode*

class pymeasure.instruments.siglenttechnologies.siglent_spdbase.**SPDSingleChannelBase**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDBase*

enable_4W_mode(*enable: bool = True*)
Enable 4-wire mode.

Type bool `True`: enables 4-wire mode `False`: disables it.

class pymeasure.instruments.siglenttechnologies.siglent_spdbase.**SPDChannel**(*parent*, *id*, *voltage_range: list = [0, 16]*, *current_range: list = [0, 8]*)

Bases: *pymeasure.instruments.channel.Channel*

The channel class for Siglent SPDxxxxX instruments.

configure_timer(*step*, *voltage*, *current*, *duration*)
Configure the timer step.

Parameters

- **step** – int: index of the step to save the configuration
- **voltage** – float: voltage setpoint of the step
- **current** – float: current limit of the step
- **duration** – int: duration of the step in seconds

property current
Measure the channel output current.

Type float

property current_limit
Control the output current configuration of the channel.

:type : float (dynamic)

enable_output(*enable: bool = True*)
Enable the channel output.

Type bool `True`: enables the output `False`: disables it

enable_timer(*enable: bool = True*)
Enable the channel timer.

Type bool `True`: enables the timer `False`: disables it

property power
Measure the channel output power.

Type float

property voltage
Measure the channel output voltage.

Type float

property voltage_setpoint
Control the output voltage configuration of the channel.

:type : float (dynamic)

class pymeasure.instruments.siglenttechnologies.siglent_spdbase.**SystemStatusCode**(*value*)
System status enums based on IntFlag

Used in conjunction with *system_status_code*.

Value	Enum
256	WAVEFORM_DISPLAY
64	TIMER_ENABLED
32	FOUR_WIRE
16	OUTPUT_ENABLED
1	CONSTANT_CURRENT
0	CONSTANT_VOLTAGE

7.37.2 Siglent SPD1168X Power Supply

class pymeasure.instruments.siglenttechnologies.**SPD1168X**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDSingleChannelBase*

Represent the Siglent SPD1168X Power Supply.

7.37.3 Siglent SPD1305X Power Supply

class pymeasure.instruments.siglenttechnologies.**SPD1305X**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDSingleChannelBase*

Represent the Siglent SPD1305X Power Supply.

7.38 Signal Recovery

This section contains specific documentation on the Signal Recovery instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.38.1 DSP 7265 Lock-in Amplifier

class pymeasure.instruments.signalrecovery.**DSP7265**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

This is the class for the DSP 7265 lockin amplifier

property adc1
Reads the input value of ADC1 in Volts

property adc2
Reads the input value of ADC2 in Volts

buffer_to_float(*buffer_data*, *sensitivity=None*, *sensitivity2=None*, *raise_error=True*)
Method that converts fixed-point buffer data to floating point data.

The provided data is converted as much as possible, but there are some requirements to the data if all provided columns are to be converted; if a key in the provided data cannot be converted it will be omitted in the returned data or an exception will be raised, depending on the value of raise_error.

The requirements for converting the data are as follows:

- Converting X, Y, magnitude and noise requires sensitivity data, which can either be part of the provided data or can be provided via the sensitivity argument
- The same holds for X2, Y2 and magnitude2 with sensitivity2.
- Converting the frequency requires both 'frequency part 1' and 'frequency part 2'.

Parameters

- **buffer_data** (*dict*) – The data to be converted. Must be in the format as returned by the *get_buffer* method: a dict of numpy arrays.
- **sensitivity** – If provided, the sensitivity used to convert X, Y, magnitude and noise. Can be provided as a float or as an array that matches the length of elements in *buffer_data*. If both a sensitivity is provided and present in the buffer_data, the provided value is used for the conversion, but the sensitivity in the buffer_data is stored in the returned dict.
- **sensitivity2** – Same as the first sensitivity argument, but for X2, Y2, magnitude2 and noise2.
- **raise_error** (*bool*) – Determines whether an exception is raised in case not all keys provided in buffer_data can be converted. If False, the columns that cannot be converted are omitted in the returned dict.

Returns Floating-point buffer data

Return type dict

property curve_buffer_bits
An integer property that controls which data outputs are stored in the curve buffer. Valid values are values between 1 and 65,535 (or 2,097,151 in dual reference mode).

property curve_buffer_interval
An integer property that controls Sets the time interval between successive points being acquired in the curve buffer. The time interval is specified in ms with a resolution of 5 ms; input values are rounded up to a multiple of 5. Valid values are values between 0 and 1,000,000,000 (corresponding to 12 days). The interval may be set to 0, which sets the rate of data storage to the curve buffer to 1.25 ms/point (800 Hz). However this only allows storage of the X and Y channel outputs. There is no need to issue a CBD 3 command to set this up since it happens automatically when acquisition starts.

property curve_buffer_length
An integer property that controls the length of the curve buffer. Valid values are values between 1 and 32,768, but the actual maximum amount of points is determined by the amount of curves that are stored, as set via the curve_buffer_bits property (32,768 / n)

property curve_buffer_status
A property that represents the status of the curve buffer acquisition with four values: the first value represents the status with 5 possibilities (0: no activity, 1: acquisition via TD command running, 2: acquisition bya TDC command running, 5: acquisition via TD command halted, 6: acquisition bia TDC command halted); the second value is the number of sweeps that is acquired; the third value is the decimal representation of the status byte (the same response as the ST command; the fourth value is the number of points acquired in the curve buffer.

property dac1
A floating point property that represents the output value on DAC1 in Volts. This property can be set.

property dac2
A floating point property that represents the output value on DAC2 in Volts. This property can be set.

property dac3
A floating point property that represents the output value on DAC3 in Volts. This property can be set.

property dac4
A floating point property that represents the output value on DAC4 in Volts. This property can be set.

property frequency
A floating point property that represents the lock-in frequency in Hz. This property can be set.

get_buffer(*quantity=None*, *convert_to_float=True*, *wait_for_buffer=True*)
Method that retrieves the buffer after it has been filled. The data retrieved from the lock-in is in a fixed-point format, which requires translation before it can be interpreted as meaningful data. When *convert_to_float* is True the conversion is performed (if possible) before returning the data.

Parameters

- **quantity** (*str*) – If provided, names the quantity that is to be retrieved from the curve buffer; can be any of: 'x', 'y', 'magnitude', 'phase', 'sensitivity', 'adc1', 'adc2', 'adc3', 'dac1', 'dac2', 'noise', 'ratio', 'log ratio', 'event', 'frequency part 1' and 'frequency part 2'; for both dual modes, additional options are: 'x2', 'y2', 'magnitude2', 'phase2', 'sensitivity2'. If no quantity is provided, all available data is retrieved.
- **convert_to_float** (*bool*) – Bool that determines whether to convert the fixed-point buffer-data to meaningful floating point values via the *buffer_to_float* method. If True, this method tries to convert all the available data to meaningful values; if this is not possible, an exception will be raised. If False, this conversion is not performed and the raw buffer-data is returned.
- **wait_for_buffer** (*bool*) – Bool that determines whether to wait for the data acquisition to finished if this method is called before the acquisition is finished. If True, the method waits until the buffer is filled before continuing; if False, the method raises an exception if the acquisition is not finished when the method is called.

property harmonic
An integer property that represents the reference harmonic mode control, taking values from 1 to 65535. This property can be set.

property id
Reads the instrument identification

property imode
Property that controls the voltage/current mode. can be 'voltage mode', 'current mode', or 'low noise current mode'

init_curve_buffer()
Initializes the curve storage memory and status variables. All record of previously taken curves is removed.

property log_ratio
Reads the log ratio output, defined as log(X/ADC1)

property mag
Reads the magnitude in Volts

property phase
Reads the phase in degrees

property ratio
Reads the ratio output, defined as X/ADC1

property reference
Controls the oscillator reference. Can be "internal", "external rear" or "external front"

property reference_phase
A floating point property that represents the reference harmonic phase in degrees. This property can be set.

property sensitivity
A floating point property that controls the sensitivity range in Volts (for voltage mode) or Amps (for current modes). When in Volts it takes discrete values from 2 nV to 1 V. When in Amps it takes discrete values from 2 fA to 1 µA (for normal current mode) or up to 10 nA (for low noise current mode). This property can be set.

setDifferentialMode(*lineFiltering=True*)
Sets lockin to differential mode, measuring A-B

set_buffer(*points*, *quantities=None*, *interval=0.01*)
Method that prepares the curve buffer for a measurement.

Parameters

- **points** (*int*) – Number of points to be recorded in the curve buffer
- **quantities** (*list*) – List containing the quantities (strings) that are to be recorded in the curve buffer, can be any of: 'x', 'y', 'magnitude', 'phase', 'sensitivity', 'adc1', 'adc2', 'adc3', 'dac1', 'dac2', 'noise', 'ratio', 'log ratio', 'event', 'frequency' (or 'frequency part 1' and 'frequency part 2'); for both dual modes, additional options are: 'x2', 'y2', 'magnitude2', 'phase2', 'sensitivity2'. Default is 'x' and 'y'.
- **interval** (*float*) – The interval between two subsequent points stored in the curve buffer in s. Default is 10 ms.

shutdown()
Brings the instrument to a safe and stable state

property slope
A integer property that controls the filter slope in dB/octave, which can take the values 6, 12, 18, or 24 dB/octave. This property can be set.

start_buffer()
Initiates data acquisition. Acquisition starts at the current position in the curve buffer and continues at the rate set by the STR command until the buffer is full.

property time_constant
A floating point property that controls the time constant in seconds, which takes values from 10 microseconds to 50,000 seconds. This property can be set.

property voltage
A floating point property that represents the voltage in Volts. This property can be set.

wait_for_buffer(*timeout=None*, *delay=0.1*)
Method that waits until the curve buffer is filled

property x
Reads the X value in Volts

property xy
Reads both the X and Y values in Volts

property y
Reads the Y value in Volts

7.39 Stanford Research Systems

This section contains specific documentation on the Stanford Research Systems (SRS) instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.39.1 SR510 Lock-in Amplifier

class pymeasure.instruments.srs.**SR510**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

property frequency
A float property representing the SR510 input reference frequency

property output
A float property that represents the SR510 output voltage in Volts.

property phase
A float property that represents the SR510 reference to input phase offset in degrees. Queries return values between -180 and 180 degrees. This property can be set with a range of values between -999 to 999 degrees. Set values are mapped internal in the lockin to -180 and 180 degrees.

property sensitivity
A float property that represents the SR510 sensitivity value. This property can be set.

property status
A string property representing the bits set within the SR510 status byte

property time_constant
A float property that represents the SR510 PRE filter time constant. This property can be set.

7.39.2 SR570 Lock-in Amplifier

class pymeasure.instruments.srs.**SR570**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

property bias_enabled
Boolean that turns the bias on or off. Allowed values are: True (bias on) and False (bias off)

property bias_level
A floating point value in V that sets the bias voltage level of the amplifier, in the [-5V,+5V] limits. The values are up to 1 mV precision level.

blank_front()
“Blanks the frontend output of the device

clear_overload()
“Reset the filter capacitors to clear an overload condition

disable_bias()
Turns the bias voltage off

disable_offset_current()
“Disables the offset current

enable_bias()
Turns the bias voltage on

enable_offset_current()
"Enables the offset current

property filter_type
A string that sets the filter type. Allowed values are: ['6dB Highpass', '12dB Highpass', '6dB Bandpass', '6dB Lowpass', '12dB Lowpass', 'none']

property front_blanked
Boolean that blanks(True) or un-blanks (False) the front panel

property gain_mode
A string that sets the gain mode. Allowed values are: ['Low Noise', 'High Bandwidth', 'Low Drift']

property high_freq
A floating point value that sets the highpass frequency of the amplifier, which takes a discrete value in a 1-3 sequence. Values are truncated to the closest allowed value if not exact. Allowed values range from 0.03 Hz to 1 MHz.

property invert_signal_sign
An boolean sets the signal invert sense. Allowed values are: True (inverted) and False (not inverted).

property low_freq
A floating point value that sets the lowpass frequency of the amplifier, which takes a discrete value in a 1-3 sequence. Values are truncated to the closest allowed value if not exact. Allowed values range from 0.03 Hz to 1 MHz.

property offset_current
A floating point value in A that sets the absolute value of the offset current of the amplifier, in the [1pA,5mA] limits. The offset current takes discrete values in a 1-2-5 sequence. Values are truncated to the closest allowed value if not exact.

property offset_current_enabled
Boolean that turns the offset current on or off. Allowed values are: True (current on) and False (current off).

property offset_current_sign
An string that sets the offset current sign. Allowed values are: 'positive' and 'negative'.

property sensitivity
A floating point value that sets the sensitivity of the amplifier, which takes discrete values in a 1-2-5 sequence. Values are truncated to the closest allowed value if not exact. Allowed values range from 1 pA/V to 1 mA/V.

property signal_inverted
Boolean that inverts the signal if True

unblank_front()
Un-blanks the frontend output of the device

7.39.3 SR830 Lock-in Amplifier

class pymeasure.instruments.srs.**SR830**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

property adc1
Reads the Aux input 1 value in Volts with 1/3 mV resolution.

property adc2
Reads the Aux input 2 value in Volts with 1/3 mV resolution.

property adc3
Reads the Aux input 3 value in Volts with 1/3 mV resolution.

property adc4
Reads the Aux input 4 value in Volts with 1/3 mV resolution.

auto_offset(*channel*)
Offsets the channel (X, Y, or R) to zero

property aux_in_1
Reads the Aux input 1 value in Volts with 1/3 mV resolution.

property aux_in_2
Reads the Aux input 2 value in Volts with 1/3 mV resolution.

property aux_in_3
Reads the Aux input 3 value in Volts with 1/3 mV resolution.

property aux_in_4
Reads the Aux input 4 value in Volts with 1/3 mV resolution.

property aux_out_1
A floating point property that controls the output of Aux output 1 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property aux_out_2
A floating point property that controls the output of Aux output 2 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property aux_out_3
A floating point property that controls the output of Aux output 3 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property aux_out_4
A floating point property that controls the output of Aux output 4 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property channel1
A string property that represents the type of Channel 1, taking the values X, R, X Noise, Aux In 1, or Aux In 2. This property can be set.

property channel2
A string property that represents the type of Channel 2, taking the values Y, Theta, Y Noise, Aux In 3, or Aux In 4. This property can be set.

property dac1
A floating point property that controls the output of Aux output 1 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dac2
A floating point property that controls the output of Aux output 2 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dac3
A floating point property that controls the output of Aux output 3 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dac4
A floating point property that controls the output of Aux output 4 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property err_status
Reads the value of the lockin error (ERR) status byte. Returns an IntFlag type with positions within the string corresponding to different error flags: bit 0: unused bit 1: backup error bit 2: RAM error bit 3: unused bit 4: ROM error bit 5: GPIB error bit 6: DSP error bit 7: Math error

property filter_slope
An integer property that controls the filter slope, which can take on the values 6, 12, 18, and 24 dB/octave. Values are truncated to the next highest level if they are not exact.

property filter_synchronous
A boolean property that controls the synchronous filter. This property can be set. Allowed values are: True or False

property frequency
A floating point property that represents the lock-in frequency in Hz. This property can be set.

get_buffer(*channel=1*, *start=0*, *end=None*)
Aquires the 32 bit floating point data through binary transfer

get_scaling(*channel*)
Returns the offset precent and the exapnsion term that are used to scale the channel in question

property harmonic
An integer property that controls the harmonic that is measured. Allowed values are 1 to 19999. Can be set.

property input_config
An string property that controls the input configuration. Allowed values are: ['A', 'A - B', 'I (1 MOhm)', 'I (100 MOhm)']

property input_coupling
An string property that controls the input coupling. Allowed values are: ['AC', 'DC']

property input_grounding
An string property that controls the input shield grounding. Allowed values are: ['Float', 'Ground']

property input_notch_config
An string property that controls the input line notch filter status. Allowed values are: ['None', 'Line', '2 x Line', 'Both']

is_out_of_range()
Returns True if the magnitude is out of range

property lia_status
Reads the value of the lockin amplifier (LIA) status byte. Returns a binary string with positions within the string corresponding to different status flags: bit 0: Input/Amplifier overload bit 1: Time constant filter overload bit 2: Output overload bit 3: Reference unlock bit 4: Detection frequency range switched bit 5: Time constant changed indirectly bit 6: Data storage triggered bit 7: unused

property magnitude
Reads the magnitude in Volts.

output_conversion(*channel*)
Returns a function that can be used to determine the signal from the channel output (X, Y, or R)

property phase
A floating point property that represents the lock-in phase in degrees. This property can be set.

quick_range()
While the magnitude is out of range, increase the sensitivity by one setting

property reference_source
An string property that controls the reference source. Allowed values are: [‘External’, ‘Internal’]

property reference_source_trigger
A string property that controls the reference source triggering. Allowed values are: [‘SINE’, ‘POS EDGE’, ‘NEG EDGE’]

property sample_frequency
Gets the sample frequency in Hz

property sensitivity
A floating point property that controls the sensitivity in Volts, which can take discrete values from 2 nV to 1 V. Values are truncated to the next highest level if they are not exact.

set_scaling(*channel*, *precent*, *expand=0*)
Sets the offset of a channel (X=1, Y=2, R=3) to a certain precent (-105% to 105%) of the signal, with an optional expansion term (0, 10=1, 100=2)

property sine_voltage
A floating point property that represents the reference sine-wave voltage in Volts. This property can be set.

snap(*val1='X'*, *val2='Y'*, **vals*)
Method that records and retrieves 2 to 6 parameters at a single instant. The parameters can be one of: X, Y, R, Theta, Aux In 1, Aux In 2, Aux In 3, Aux In 4, Frequency, CH1, CH2. Default is “X” and “Y”.

Parameters

- **val1** – first parameter to retrieve
- **val2** – second parameter to retrieve
- **vals** – other parameters to retrieve (optional)

property theta
Reads the theta value in degrees.

property time_constant
A floating point property that controls the time constant in seconds, which can take discrete values from 10 microseconds to 30,000 seconds. Values are truncated to the next highest level if they are not exact.

wait_for_buffer(*count*, *has_aborted=<function SR830.<lambda>>*, *timeout=60*, *timestep=0.01*)
Wait for the buffer to fill a certain count

property x
Reads the X value in Volts.

property xy
Reads the X and Y values in Volts.

property y
Reads the Y value in Volts.

7.39.4 SR860 Lock-in Amplifier

class pymeasure.instruments.srs.SR860(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

property adc1
Reads the Aux input 1 value in Volts with 1/3 mV resolution.

property adc2
Reads the Aux input 2 value in Volts with 1/3 mV resolution.

property adc3
Reads the Aux input 3 value in Volts with 1/3 mV resolution.

property adc4
Reads the Aux input 4 value in Volts with 1/3 mV resolution.

property aux_in_1
Reads the Aux input 1 value in Volts with 1/3 mV resolution.

property aux_in_2
Reads the Aux input 2 value in Volts with 1/3 mV resolution.

property aux_in_3
Reads the Aux input 3 value in Volts with 1/3 mV resolution.

property aux_in_4
Reads the Aux input 4 value in Volts with 1/3 mV resolution.

property aux_out_1
A floating point property that controls the output of Aux output 1 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property aux_out_2
A floating point property that controls the output of Aux output 2 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property aux_out_3
A floating point property that controls the output of Aux output 3 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property aux_out_4
A floating point property that controls the output of Aux output 4 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dac1
A floating point property that controls the output of Aux output 1 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dac2
A floating point property that controls the output of Aux output 2 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dac3
A floating point property that controls the output of Aux output 3 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dac4
A floating point property that controls the output of Aux output 4 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

property dcmode
A string property that represents the sine out dc mode. This property can be set. Allowed values are:[‘COM’, ‘DIF’, ‘common’, ‘difference’]

property detectedfrequency
Returns the actual detected frequency in HZ.

property extfreqency
Returns the external frequency in Hz.

property filer_synchronous
A string property that represents the synchronous filter. This property can be set. Allowed values are:[‘Off’, ‘On’]

property filter_advanced
A string property that represents the advanced filter. This property can be set. Allowed values are:[‘Off’, ‘On’]

property filter_slope
A integer property that sets the filter slope to 6 dB/oct(i=0), 12 DB/oct(i=1), 18 dB/oct(i=2), 24 dB/oct(i=3).

property frequency
A floating point property that represents the lock-in frequency in Hz. This property can be set.

property frequencypreset1
A floating point property that represents the preset frequency for the F1 preset button. This property can be set.

property frequencypreset2
A floating point property that represents the preset frequency for the F2 preset button. This property can be set.

property frequencypreset3
A floating point property that represents the preset frequency for the F3 preset button. This property can be set.

property frequencypreset4
A floating point property that represents the preset frequency for the F4 preset button. This property can be set.

property front_panel
Turns the front panel blanking on(i=0) or off(i=1).

property get_noise_bandwidth
Returns the equivalent noise bandwidth, in hertz.

property get_signal_strength_indicator
Returns the signal strength indicator.

property gettimebase
Returns the current 10 MHz timebase source.

property harmonic
An integer property that controls the harmonic that is measured. Allowed values are 1 to 99. Can be set.

property harmonicdual
An integer property that controls the harmonic in dual reference mode that is measured. Allowed values are 1 to 99. Can be set.

property horizontal_time_div
A integer property for the horizontal time/div according to the following table:[‘0=0.5s’, ‘1=1s’, ‘2=2s’,

'3=5s', '4=10s', '5=30s', '6=1min', '7=2min', '8=5min', '9=10min', '10=30min', '11=1hour', '12=2hour', '13=6hour', '14=12hour', '15=1day', '16=2days']

property input_coupling
A string property that represents the input coupling. This property can be set. Allowed values are:['AC', 'DC']

property input_current_gain
A string property that represents the current input gain. This property can be set. Allowed values are:['1MEG', '100MEG']

property input_range
A string property that represents the input range. This property can be set. Allowed values are:['1V', '300M', '100M', '30M', '10M']

property input_shields
A string property that represents the input shield grounding. This property can be set. Allowed values are:['Float', 'Ground']

property input_signal
A string property that represents the signal input. This property can be set. Allowed values are:['VOLT', 'CURR', 'voltage', 'current']

property input_voltage_mode
A string property that represents the voltage input mode. This property can be set. Allowed values are:['A', 'A-B']

property internalfrequency
A floating property that represents the internal lock-in frequency in Hz This property can be set.

property magnitude
Reads the magnitude in Volts.

property parameter_DAT1
A integer property that assigns a parameter to data channel 1(green). This parameters can be set. Allowed values are:['i=', '0=Xoutput', '1=Youtput', '2=Routput', 'Thetaoutput', '4=Aux IN1', '5=Aux IN2', '6=Aux IN3', '7=Aux IN4', '8=Xnoise', '9=Ynoise', '10=AUXOut1', '11=AuxOut2', '12=Phase', '13=Sine Out amplitude', '14=DCLevel', '15I=nt.referenceFreq', '16=Ext.referenceFreq']

property parameter_DAT2
A integer property that assigns a parameter to data channel 2(blue). This parameters can be set. Allowed values are:['i=', '0=Xoutput', '1=Youtput', '2=Routput', 'Thetaoutput', '4=Aux IN1', '5=Aux IN2', '6=Aux IN3', '7=Aux IN4', '8=Xnoise', '9=Ynoise', '10=AUXOut1', '11=AuxOut2', '12=Phase', '13=Sine Out amplitude', '14=DCLevel', '15I=nt.referenceFreq', '16=Ext.referenceFreq']

property parameter_DAT3
A integer property that assigns a parameter to data channel 3(yellow). This parameters can be set. Allowed values are:['i=', '0=Xoutput', '1=Youtput', '2=Routput', 'Thetaoutput', '4=Aux IN1', '5=Aux IN2', '6=Aux IN3', '7=Aux IN4', '8=Xnoise', '9=Ynoise', '10=AUXOut1', '11=AuxOut2', '12=Phase', '13=Sine Out amplitude', '14=DCLevel', '15I=nt.referenceFreq', '16=Ext.referenceFreq']

property parameter_DAT4
A integer property that assigns a parameter to data channel 3(orange). This parameters can be set. Allowed values are:['i=', '0=Xoutput', '1=Youtput', '2=Routput', 'Thetaoutput', '4=Aux IN1', '5=Aux IN2', '6=Aux IN3', '7=Aux IN4', '8=Xnoise', '9=Ynoise', '10=AUXOut1', '11=AuxOut2', '12=Phase', '13=Sine Out amplitude', '14=DCLevel', '15I=nt.referenceFreq', '16=Ext.referenceFreq']

property phase
A floating point property that represents the lock-in phase in degrees. This property can be set.

property reference_externalinput
A string property that represents the external reference input. This property can be set. Allowed values are:['50OHMS', '1MEG']

property reference_source
A string property that represents the reference source. This property can be set. Allowed values are:['INT', 'EXT', 'DUAL', 'CHOP']

property reference_triggermode
A string property that represents the external reference trigger mode. This property can be set. Allowed values are:['SIN', 'POS', 'NEG', 'POSTTL', 'NEGTTL']

property screen_layout
A integer property that Sets the screen layout to trend(i=0), full strip chart history(i=1), half strip chart history(i=2), full FFT(i=3), half FFT(i=4) or big numerical(i=5).

screenshot()
Take screenshot on device The DCAP command saves a screenshot to a USB memory stick. This command is the same as pressing the [Screen Shot] key. A USB memory stick must be present in the front panel USB port.

property sensitvity
A floating point property that controls the sensitivity in Volts, which can take discrete values from 2 nV to 1 V. Values are truncated to the next highest level if they are not exact.

property sine_amplitudepreset1
Floating point property representing the preset sine out amplitude, for the A1 preset button. This property can be set.

property sine_amplitudepreset2
Floating point property representing the preset sine out amplitude, for the A2 preset button. This property can be set.

property sine_amplitudepreset3
Floating point property representing the preset sine out amplitude, for the A3 preset button. This property can be set.

property sine_amplitudepreset4
Floating point property representing the preset sine out amplitude, for the A3 preset button. This property can be set.

property sine_dclevelpreset1
A floating point property that represents the preset sine out dc level for the L1 button. This property can be set.

property sine_dclevelpreset2
A floating point property that represents the preset sine out dc level for the L2 button. This property can be set.

property sine_dclevelpreset3
A floating point property that represents the preset sine out dc level for the L3 button. This property can be set.

property sine_dclevelpreset4
A floating point property that represents the preset sine out dc level for the L4 button. This property can be set.

property sine_voltage
A floating point property that represents the reference sine-wave voltage in Volts. This property can be set.

snap(*val1='X'*, *val2='Y'*, *val3=None*)
retrieve 2 or 3 parameters at once parameters can be chosen by index, or enumeration as follows:

j enumeration parameter j enumeration parameter

0 X X output 9 YNOise Ynoise 1 Y Youtput 10 OUT1 Aux Out1 2 R R output 11 OUT2 Aux Out2 3 THeta output 12 PHAse Reference Phase 4 IN1 Aux In1 13 SAMp Sine Out Amplitude 5 IN2 Aux In2 14 LEVel DC Level 6 IN3 Aux In3 15 FInt Int. Ref. Frequency 7 IN4 Aux In4 16 FExt Ext. Ref. Frequency 8 XNOise Xnoise

Parameters

- **val1** – parameter enumeration/index
- **val2** – parameter enumeration/index
- **val3** – parameter enumeration/index (optional)

Defaults: val1 = "X" val2 = "Y" val3 = None

property strip_chart_dat1
A integer property that turns the strip chart graph of data channel 1 off(i=0) or on(i=1).

property strip_chart_dat2
A integer property that turns the strip chart graph of data channel 2 off(i=0) or on(i=1).

property strip_chart_dat3
A integer property that turns the strip chart graph of data channel 1 off(i=0) or on(i=1).

property strip_chart_dat4
A integer property that turns the strip chart graph of data channel 4 off(i=0) or on(i=1).

property theta
Reads the theta value in degrees.

property time_constant
A floating point property that controls the time constant in seconds, which can take discrete values from 10 microseconds to 30,000 seconds. Values are truncated to the next highest level if they are not exact.

property timebase
Sets the external 10 MHZ timebase to auto(i=0) or internal(i=1).

property x
Reads the X value in Volts

property y
Reads the Y value in Volts

7.40 Tektronix

This section contains specific documentation on the Tektronix instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.40.1 TDS2000 Oscilloscope

class pymeasure.instruments.tektronix.**TDS2000**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Tektronix TDS 2000 Oscilloscope and provides a high-level for interacting with the instrument

7.40.2 AFG3152C Arbitrary function generator

class pymeasure.instruments.tektronix.**AFG3152C**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Tektronix AFG 3000 series (one or two channels) arbitrary function generator and provides a high-level for interacting with the instrument.

afg=AFG3152C("GPIB::1") # AFG on GPIB 1 afg.reset() # Reset to default afg.ch1.shape='sinusoidal' # Sinusoidal shape afg.ch1.unit='VPP' # Sets CH1 unit to VPP afg.ch1.amp_vpp=1 # Sets the CH1 level to 1 VPP afg.ch1.frequency=1e3 # Sets the CH1 frequency to 1KHz afg.ch1.enable() # Enables the output from CH1

7.41 Temptronic

This section contains specific documentation on the temptronic instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.41.1 Temptronic Base Class

class pymeasure.instruments.temptronic.**ATSBase**(*adapter*, *name='ATSBase'*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

The base class for Temptronic ATSXXX instruments.

property air_temperature
Read air temperature in 0.1 °C increments.

Type float

at_temperature()

Returns `True` if at temperature.

property auxiliary_condition_code
Read out auxiliary condition status register.

Type int

Relevant flags are:

Bit	Meaning
10	None
9	Ramp mode
8	Mode: 0 programming, 1 manual
7	None
6	TS status: 0 start-up, 1 ready
5	Flow: 0 off, 1 on
4	Sense mode: 0 air, 1 DUT
3	Compressor: 0 on, 1 off (heating possible)
2	Head: 0 lower, upper
1	None
0	None

Refere to chapter 4 in the manual

clear()
Clear device-specific errors.

See *error_code* for further information.

property compressor_enable
`True` enables compressors, `False` disables it.

Type Boolean

configure(*temp_window=1*, *dut_type='T'*, *soak_time=30*, *dut_constant=100*, *temp_limit_air_low=- 60*, *temp_limit_air_high=220*, *temp_limit_air_dut=50*, *maximum_test_time=1000*)
Convenience method for most relevant configuration properties.

Parameters

- **dut_type** – string: indicating which DUT type to use
- **soak_time** – float: elapsed time in soak_window before settling is indicated
- **soak_window** – float: Soak window size or temperature settlings bounds (K)
- **dut_constant** – float: time constant of DUT, higher values indicate higher thermal mass
- **temp_limit_air_low** – float: minimum flow temperature limit (°C)
- **temp_limit_air_high** – float: maximum flow temperature limit (°C)
- **temp_limit_air_dut** – float: allowed temperature difference (K) between DUT and Flow
- **maximum_test_time** – float: maximum test time (seconds) for a single temperature point (safety)

Returns self

property copy_active_setup_file
Copy active setup file (0) to setup n (1 - 12).

Type int

property current_cycle_count
Read the number of cycles to do

Type int

property cycling_enable
CYCL Start/stop cycling.

Type bool

cycling_enable = True (start cycling) cycling_enable = False (stop cycling)

cycling_stopped()

Returns `True` if cycling has stopped.

property dut_constant
Control thermal constant (default 100) of DUT.

Type float

Lower values indicate lower thermal mass, higher values indicate higher thermal mass respectively.

property dut_mode
`On` enables DUT mode, `OFF` enables air mode

Type string

property dut_temperature
Read DUT temperature, in 0.1 °C increments.

Type float

property dut_type
Control DUT sensor type.

Type string

Possible values are:

String	Meaning
''	no DUT
'T'	T-DUT
'K'	K-DUT

Warning: If in DUT mode without DUT being connected, TS flags DUT error

property dynamic_temperature_setpoint
Read the dynamic temperature setpoint.

Type float

property enable_air_flow
Set TS air flow.

`True` enables air flow, `False` disables it

Type bool

end_of_all_cycles()

Returns `True` if cycling has stopped.

end_of_one_cycle()

Returns `True` if TS is at end of one cycle.

end_of_test()

Returns `True` if TS is at end of test.

enter_cycle()
Enter Cycle by sending `RMPC 1`.

Returns self

enter_ramp()
Enter Ramp by sending `RMPS 0`.

Returns self

property error_code
Read the device-specific error register (16 bits).

Type *ErrorCode*

error_status()
Returns error status code (maybe used for logging).

Returns *ErrorCode*

property head
Control TS head position.

Type string

`down`: transfer head to lower position `up`: transfer head to elevated position

property learn_mode
Control DUT automatic tuning (learning).

Type bool `False`: off `True`: automatic tuning on

property load_setup_file
loads setup file SFIL.

Valid range is between 1 to 12.

Type int

property local_lockout
`True` disables TS GUI, `False` enables it.

property main_air_flow_rate
Read main nozzle air flow rate in liters/sec.

property maximum_test_time
Control maximum allowed test time (s).

Type float

This prevents TS from staying at a single temperature forever. Valid range: 0 to 9999

property mode
Returns a string indicating what the system is doing at the time the query is processed.

Type string

(dynamic)

next_setpoint()
Step to the next setpoint during temperature cycling.

not_at_temperature()

Returns True if not at temperature.

property nozzle_air_flow_rate
Read main nozzle air flow rate in scfm.

property ramp_rate
Control ramp rate (K / min).

Type float

allowed values: nn.n: 0 to 99.9 in 0.1 K per minute steps. nnnn: 100 to 9999 in 1 K per minute steps.

property remote_mode
True disables TS GUI but displays a "Return to local" switch.

reset()
Reset (force) the System to the Operator screen.

Returns self

property set_point_number
Select a setpoint to be the current setpoint.

Type int

Valid range is 0 to 17 when on the Cycle screen or or 0 to 2 in case of operator screen (0=hot, 1=ambient, 2=cold).

set_temperature(*set_temp*)
sweep to a specified setpoint.

Parameters **set_temp** – target temperature for DUT (float)

Returns self

shutdown(*head=False*)
Turn down TS (flow and remote operation).

Parameters **head** – Lift head if True

Returns self

start(*enable_air_flow=True*)
start TS in remote mode.

Parameters **enable_air_flow** – flow starts if True

Returns self

property temperature
Read current temperature with 0.1 °C resolution.

Type float

Temperature readings origin depends on *dut_mode* setting. Reading higher than 400 (°C) indicates invalidity.

property temperature_condition_status_code
Temperature condition status register.

Type *TemperatureStatusCode*

property temperature_event_status
temperature event status register.

Type *TemperatureStatusCode*

Hint: Reading will clear register content.

property temperature_limit_air_dut
Air to DUT temperature limit.

Type float

Allowed difference between nozzle air and DUT temperature during settling. Valid range between 10 to 300 °C in 1 degree increments.

property temperature_limit_air_high
upper air temperature limit.

Type float

Valid range between 25 to 255 (°C). Setpoints above current value cause "out of range" error in TS.

property temperature_limit_air_low
Control lower air temperature limit.

Type float

Valid range between -99 to 25 (°C). Setpoints below current value cause "out of range" error in TS. (dynamic)

property temperature_setpoint
Set or get selected setpoint's temperature.

Type float

Valid range is -99.9 to 225.0 (°C) or as indicated by *temperature_limit_air_high* and *temperature_limit_air_low*. Use convenience function *set_temperature()* to prevent unexpected behavior.

property temperature_setpoint_window
Setpoint's temperature window.

Type float

Valid range is between 0.1 to 9.9 (°C). Temperature status register flags `at temperature` in case soak time elapsed while temperature stays in between bounds given by this value around the current setpoint.

property temperature_soak_time
Set the soak time for the currently selected setpoint.

Type float

Valid range is between 0 to 9999 (s). Lower values shorten cycle times. Higher values increase cycle times, but may reduce settling errors. See *temperature_setpoint_window* for further information.

property total_cycle_count
Set or read current cycle count (1 - 9999).

Type int

Sending 0 will stop cycling

wait_for_settling(*time_limit=300*)
block script execution until TS is settled.

Parameters **time_limit** – set the maximum blocking time within TS has to settle (float).

Returns self

Script execution is blocked until either TS has settled or time_limit has been exceeded (float).

class pymeasure.instruments.temptronic.temptronic_base.**TemperatureStatusCode**(*value*)
Temperature status enums based on IntFlag

Used in conjunction with *temperature_condition_status_code*.

Value	Enum
32	CYCLING_STOPPED
16	END_OF_ALL_CYCLES
8	END_OF_ONE_CYCLE
4	END_OF_TEST
2	NOT_AT_TEMPERATURE
1	AT_TEMPERATURE
0	NO_STATUS

class pymeasure.instruments.temptronic.temptronic_base.**ErrorCode**(*value*)
Error code enums based on IntFlag.

Used in conjunction with *error_code*.

Value	Enum
16384	NO_DUT_SENSOR_SELECTED
4096	BVRAM_FAULT
2048	NVRAM_FAULT
1024	NO_LINE_SENSE
512	FLOW_SENSOR_HARDWARE_ERROR
128	INTERNAL_ERROR
32	AIR_SENSOR_OPEN
16	LOW_INPUT_AIR_PRESSURE
8	LOW_FLOW
2	AIR_OPEN_LOOP
1	OVERHEAT
0	OK

7.41.2 Temptronic ATS525 Thermostream

class pymeasure.instruments.temptronic.**ATS525**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.temptronic.temptronic_base.ATSBase*

Represent the TemptronicATS525 instruments.

property system_current
Operating current.

7.41.3 Temptronic ATS545 Thermostream

class pymeasure.instruments.temptronic.**ATS545**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.temptronic.temptronic_base.ATSBase*

Represents the TemptronicATS545 instrument.

Coding example

```
ts = ATS545('ASRL3::INSTR')  # replace adapter address
ts.configure()  # basic configuration (defaults to T-DUT)
ts.start()  # starts flow (head position not changed)
ts.set_temperature(25)  # sets temperature to 25 degC
ts.wait_for_settling()  # blocks script execution and polls for settling
ts.shutdown(head=False)  # disables thermostream, keeps head down
```

next_setpoint()

not implemented in ATS545

set `self.set_point_number` instead

7.41.4 Temptronic ECO560 Thermostream

class pymeasure.instruments.temptronic.**ECO560**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.temptronic.temptronic_base.ATSBase*

Represent the TemptronicECO560 instruments.

copy_active_setup_file = None

7.42 TEXIO

This section contains specific documentation on the TEXIO instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.42.1 TEXIO PSW-360L30 Power Supply

class pymeasure.instruments.texio.**TexioPSW360L30**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.keithley.keithley2260B.Keithley2260B*

Represents the TEXIO PSW-360L30 Power Supply (minimal implementation) and provides a high-level interface for interacting with the instrument.

For a connection through tcpip, the device only accepts connections at port 2268, which cannot be configured otherwise. example connection string: 'TCPIP::xxx.xxx.xxx.xxx::2268::SOCKET'

For a connection through USB on Linux, the kernel is going to create a /dev/ttyACMX device automatically. The serial connection properties are fixed at 9600–8-N-1.

The read termination for this interface is Line-Feed n.

This driver inherits from the Keithley2260B one. All instructions implemented in the Keithley 2260B driver are also available for the TEXIO PSW-360L30 power supply.

The only addition is the "output" property that is just an alias for the "enabled" property of the Keithley 2260B. Calling the output switch "enabled" is confusing because it is not clear if the whole device is enabled/disable or only the output.

```
source = TexioPSW360L30("TCPIP::xxx.xxx.xxx.xxx::2268::SOCKET")
source.voltage = 1
print(source.voltage)
print(source.current)
print(source.power)
print(source.applied)
```

class ChannelCreator(*cls*, *id=None*, *prefix='ch_'*, ***kwargs*)

Bases: `object`

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The variable name (e.g. `channels`) will be used as the *collection* of the children. You may define the attribute prefix. If there are no other pressing reasons, use `channels` as variable and leave the prefix at the default `"ch_"`.

```
class SomeInstrument(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C' in 'channels'
    channels = Instrument.ChannelCreator(ChildClass, ["A", "B", "C"])
    # Two functions of different types: 'fn_power', 'fn_voltage' in 'functions'
    functions = Instrument.ChannelCreator((PowerChannel, VoltageChannel),
                                          ["power", "voltage"], prefix="fn_")
    # A channel without a prefixed attribute name, simply: 'motor'
    motor = Instrument.ChannelCreator(MotorControl, prefix=None)
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – Single value or tuple/list of ids of the children.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the variable name.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, ***kwargs*)

Add a child to this instance and return its index in the children list.

The newly created child may be accessed either by the id in the children dictionary or by the created attribute. The fifth channel of *instrument* with id "F" has two access options: `instrument.channels["F"] == instrument.ch_F`

Note: Do not change the default *collection* or *prefix* parameter, unless you have to distinguish several collections of different children, e.g. different channel types (analog and digital).

Parameters

- **cls** – Class of the channel.
- **id** – Child id how it is used in communication, e.g. *"A"*.

- **collection** – Name of the collection of children, used for the dictionary.
- **prefix** – Collection prefix for the attributes, e.g. *"ch_"* creates attribute *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- ****kwargs** – Keyword arguments for the channel creator.

Returns Instance of the created child.

property applied
Simultaneous control of voltage (volts) and current (amps). Values need to be supplied as tuple of (voltage, current). Depending on whether the instrument is in constant current or constant voltage mode, the values achieved by the instrument will differ from the ones set.

binary_values(*command*, *query_delay=0*, ***kwargs*)
Write a command to the instrument and return a numpy array of the binary data.

Parameters

- **command** – Command to be sent to the instrument.
- **query_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for `read_binary_values()`.

Returns NumPy array of values.

check_errors()
Logs any system errors reported by the instrument.

property complete
This property allows synchronization between a controller and a device. The Operation Complete query places an ASCII character 1 into the device's Output Queue when all pending selected device operations have been finished.

property current
Reads the current (in Ampere) the dc power supply is putting out.

property current_limit
A floating point property that controls the source current in amps. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual current achieved.

property error
Returns a tuple of an error code and message from a single error.

property id
Requests and returns the identification of the instrument.

property options
Requests and returns the device options installed.

property output_enabled
A boolean property that controls whether the source is enabled, takes values True or False.

property power
Reads the power (in Watt) the dc power supply is putting out.

read_binary_values(***kwargs*)
Read binary values from the device.

read_bytes(*count*, ***kwargs*)
Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – Number of bytes to read. A value of -1 indicates to read the whole read buffer.
- **kwargs** – Keyword arguments for the adapter.

Returns bytes Bytes response of the instrument (including termination).

remove_child(*child*)
Remove the child from the instrument and the corresponding collection.

Parameters **child** – Instance of the child to delete.

reset()
Resets the instrument.

shutdown()
Disable output, call parent function

property status
Requests and returns the status byte and Master Summary Status bit.

property voltage
Reads the voltage (in Volt) the dc power supply is putting out.

property voltage_setpoint
A floating point property that controls the source voltage in volts. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual voltage achieved.

wait_for(*query_delay=0*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters **query_delay** – Delay between writing and reading in seconds.

write_binary_values(*command*, *values*, **args*, ***kwargs*)
Write binary values to the device.

Parameters

- **command** – Command to send.
- **values** – The values to transmit.
- ****kwargs** (**args,*) – Further arguments to hand to the Adapter.

write_bytes(*content*, ***kwargs*)
Write the bytes *content* to the instrument.

7.43 Thermotron

This section contains specific documentation on the Thermotron instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.43.1 Thermotron 3800 Oven

pymeasure.instruments.thermotron.**thermotron3800**
alias of <module 'pymeasure.instruments.thermotron.thermotron3800' from '/home/docs/checkouts/readthedocs.org/user_builds/pymeasure/checkouts/latest/pymeasure/instruments/thermotron/thermotron38

7.44 Thorlabs

This section contains specific documentation on the Thorlabs instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.44.1 Thorlabs PM100USB Powermeter

class pymeasure.instruments.thorlabs.**ThorlabsPM100USB**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents Thorlabs PM100USB powermeter.

property energy
Energy in J.

property power
Power in W.

property wavelength
Wavelength in nm.

property wavelength_max
Maximum wavelength, in nm

property wavelength_min
Minimum wavelength, in nm

7.44.2 Thorlabs Pro 8000 modular laser driver

class pymeasure.instruments.thorlabs.**ThorlabsPro8000**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents Thorlabs Pro 8000 modular laser driver

property LDCCurrent
Laser current.

property LDCCurrentLimit
Set Software current Limit (value must be lower than hardware current limit).

property LDCPolarity
Set laser diode polarity. Allowed values are: ['AG', 'CG']

property LDCStatus
Set laser diode status. Allowed values are: ['ON', 'OFF']

property TEDSetTemperature
Set TEC temperature

property TEDStatus
Set TEC status. Allowed values are: ['ON', 'OFF']

property slot
Slot selection. Allowed values are: range(1, 9)

7.45 Toptica

This section contains specific documentation on the Toptica Photonics instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.45.1 Toptica Adapters

class pymeasure.instruments.toptica.adapters.**TopticaAdapter**(*port*, *baud_rate*, ***kwargs*)
Bases: *pymeasure.adapters.visa.VISAAdapter*

Adapter class for connecting to Toptica Console via a serial connection.

Parameters

- **port** – pyvisa resource name of the instrument
- **baud_rate** – communication speed
- **kwargs** – Any valid key-word argument for VISAAdapter

extract_value(*reply*)
preprocess_reply function which tries to extract <value> from 'name = <value> [unit]'. If <value> can not be identified the orignal string is returned.

Parameters **reply** – reply string

Returns string with only the numerical value, or the original string

7.45.2 Toptica IBeam Smart Laser diode

class pymeasure.instruments.toptica.ibeamsmart.**IBeamSmart**(*adapter*, *baud_rate=115200*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

IBeam Smart laser diode

Parameters

- **port** – pyvisa resource name of the instrument
- **baud_rate** – communication speed, defaults to 115200
- **kwargs** – Any valid key-word argument for VISAAdapter

property channel1_enabled
Status of Channel 1 of the laser. This can be True if the laser is on or False otherwise

property channel2_enabled
Status of Channel 2 of the laser. This can be True if the laser is on or False otherwise

disable()
shutdown all laser operation

enable_continous()
enable countinous emmission mode

enable_pulsing()
enable pulsing mode. The optical output is controlled by a digital input signal on a dedicated connnector on the device

property laser_enabled
Status of the laser diode driver. This can be True if the laser is on or False otherwise

property power
Actual output power in uW of the laser system. In pulse mode this means that the set value might not correspond to the readback one.

property serial
Serial number of the laser system

property system_temp
base plate (heatsink) temperature in degree centigrade.

property temp
temperature of the laser diode in degree centigrade.

property version
Firmware version number

7.46 Yokogawa

This section contains specific documentation on the Yokogawa instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.46.1 Yokogawa 7651 Programmable Supply

class pymeasure.instruments.yokogawa.**Yokogawa7651**(*adapter*, ***kwargs*)
Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Yokogawa 7651 Programmable DC Source and provides a high-level for interacting with the instrument.

```
yoko = Yokogawa7651("GPIB::1")

yoko.apply_current()                # Sets up to source current
yoko.source_current_range = 10e-3   # Sets the current range to 10 mA
yoko.compliance_voltage = 10        # Sets the compliance voltage to 10 V
yoko.source_current = 0             # Sets the source current to 0 mA

yoko.enable_source()                # Enables the current output
yoko.ramp_to_current(5e-3)          # Ramps the current to 5 mA

yoko.shutdown()                     # Ramps the current to 0 mA and disables output
```

apply_current(*max_current=0.001*, *compliance_voltage=1*)
Configures the instrument to apply a source current, which can take optional parameters that defer to the *source_current_range* and *compliance_voltage* properties.

apply_voltage(*max_voltage=1*, *compliance_current=0.01*)
Configures the instrument to apply a source voltage, which can take optional parameters that defer to the *source_voltage_range* and *compliance_current* properties.

property compliance_current
A floating point property that sets the compliance current in Amps, which can take values from 5 to 120 mA.

property compliance_voltage
A floating point property that sets the compliance voltage in Volts, which can take values between 1 and 30 V.

disable_source()
Disables the source of current or voltage depending on the configuration of the instrument.

enable_source()
Enables the source of current or voltage depending on the configuration of the instrument.

property id
Returns the identification of the instrument

ramp_to_current(*current*, *steps=25*, *duration=0.5*)
Ramps the current to a value in Amps by traversing a linear spacing of current steps over a duration, defined in seconds.

Parameters

- **steps** – A number of linear steps to traverse
- **duration** – A time in seconds over which to ramp

ramp_to_voltage(*voltage*, *steps=25*, *duration=0.5*)
Ramps the voltage to a value in Volts by traversing a linear spacing of voltage steps over a duration, defined in seconds.

Parameters

- **steps** – A number of linear steps to traverse
- **duration** – A time in seconds over which to ramp

shutdown()
Shuts down the instrument, and ramps the current or voltage to zero before disabling the source.

property source_current
A floating point property that controls the source current in Amps, if that mode is active.

property source_current_range
A floating point property that sets the current voltage range in Amps, which can take values: 1 mA, 10 mA, and 100 mA. Currents are truncted to an appropriate value if needed.

property source_enabled
Reads a boolean value that is True if the source is enabled, determined by checking if the 5th bit of the OC flag is a binary 1.

property source_mode
A string property that controls the source mode, which can take the values 'current' or 'voltage'. The convenience methods *apply_current()* and *apply_voltage()* can also be used.

property source_voltage
A floating point property that controls the source voltage in Volts, if that mode is active.

property source_voltage_range
A floating point property that sets the source voltage range in Volts, which can take values: 10 mV, 100 mV, 1 V, 10 V, and 30 V. Voltages are truncted to an appropriate value if needed.

7.46.2 Yokogawa GS200 Source

class pymeasure.instruments.yokogawa.**YokogawaGS200**(*adapter*, ***kwargs*)

Bases: *pymeasure.instruments.instrument.Instrument*

Represents the Yokogawa GS200 source and provides a high-level interface for interacting with the instrument.

property current_limit

Floating point number that controls the current limit. "Limit" refers to maximum value of the electrical value that is conjugate to the mode (current is conjugate to voltage, and vice versa). Thus, current limit is only applicable when in 'voltage' mode

property source_enabled

A boolean property that controls whether the source is enabled, takes values True or False.

property source_level

Floating point number that controls the output level, either a voltage or a current, depending on the source mode.

property source_mode

String property that controls the source mode. Can be either 'current' or 'voltage'.

property source_range

Floating point number that controls the range (either in voltage or current) of the output. "Range" refers to the maximum source level.

trigger_ramp_to_level(*level*, *ramp_time*)

Ramp the output level from its current value to "level" in time "ramp_time". This method will NOT wait until the ramp is finished (thus, it will not block further code evaluation).

Parameters

- **level** (*float*) – final output level
- **ramp_time** (*float*) – time in seconds to ramp

Returns None

property voltage_limit

Floating point number that controls the voltage limit. "Limit" refers to maximum value of the electrical value that is conjugate to the mode (current is conjugate to voltage, and vice versa). Thus, voltage limit is only applicable when in 'current' mode

CHAPTER

EIGHT

CONTRIBUTING

Contributions to the instrument repository and the main code base are highly encouraged. This section outlines the basic work-flow for new contributors.

8.1 Using the development version

New features are added to the development version of PyMeasure, hosted on GitHub. We use Git version control to track and manage changes to the source code. On Windows, we recommend using GitHub Desktop. Make sure you have an appropriate version of Git (or GitHub Desktop) installed and that you have a GitHub account.

In order to add your feature, you need to first fork PyMeasure. This will create a copy of the repository under your GitHub account.

The instructions below assume that you have set up Anaconda, as described in the *Quick Start guide* and describe the terminal commands necessary. If you are using GitHub Desktop, take a look through their documentation to understand the corresponding steps.

Clone your fork of PyMeasure `your-github-username/pymeasure`. In the following terminal commands replace your desired path and GitHub username.

```
cd /path/for/code
git clone https://github.com/your-github-username/pymeasure.git
```

If you had already installed PyMeasure using `pip`, make sure to uninstall it before continuing.

```
pip uninstall pymeasure
```

Install PyMeasure in the editable mode.

```
cd /path/for/code/pymeasure
pip install -e .
```

This will allow you to edit the files of PyMeasure and see the changes reflected. Make sure to reset your notebook kernel or Python console when doing so. Now you have your own copy of the development version of PyMeasure installed!

8.2 Working on a new feature

We use branches in Git to allow multiple features to be worked on simultaneously, without causing conflicts. The master branch contains the stable development version. Instead of working on the master branch, you will create your own branch off the master and merge it back into the master when you are finished.

Create a new branch for your feature before editing the code. For example, if you want to add the new instrument "Extreme 5000" you will make a new branch "dev/extreme-5000".

```
git branch dev/extreme-5000
```

You can also make a new branch on GitHub. If you do so, you will have to fetch these changes before the branch will show up on your local computer.

```
git fetch
```

Once you have created the branch, change your current branch to match the new one.

```
git checkout dev/extreme-5000
```

Now you are ready to write your new feature and make changes to the code. To ensure consistency, please follow the *coding standards for PyMeasure*. Use `git status` to check on the files that have been changed. As you go, commit your changes and push them to your fork.

```
git add file-that-changed.py
git commit -m "A short description about what changed"
git push
```

8.3 Making a pull request

While you are working, it is helpful to start a pull request (PR) targeting the `master` branch of `pymeasure/pymeasure`. This will allow you to discuss your feature with other contributors. We encourage you to start this pull request already after your first commit. You may mark a pull request as a draft, if it is in an early state.

Start a pull request on the PyMeasure GitHub page.

There is some automation in place to run the unit tests and check some coding standards. Annotations in the "Files changed" tab indicate problems for you to correct (e.g. linting or docstring warnings).

Your pull-request will be reviewed by the PyMeasure maintainers. Frequently there is some iteration and discussion based on that feedback until a pull request can be merged. This will happen either in the conversation tab or in inline code comments.

Be aware that due to maintainer manpower limitations it might take a long time until PRs get reviewed and/or merged. In general, review effort scales badly with PR size. Therefore, **smaller PRs are much preferred**. Try to limit your contribution to one "aspect", e.g. one instrument (or a few if closely related), one bug fix, or one feature contribution.

If you placed your contribution in a separate branch as suggested above, you can easily use your contribution in the meantime – just check out your feature branch instead of *master*.

8.4 Unit testing

Unit tests are run each time a new commit is made to a branch. The purpose is to catch changes that break the current functionality, by testing each feature unit. PyMeasure relies on pytest to preform these tests, which are run on TravisCI and Appveyor for Linux/macOS and Windows respectively.

Running the unit tests while you develop is highly encouraged. This will ensure that you have a working contribution when you create a pull request.

```
pytest
```

If your feature can be tested, unit tests are required. This will ensure that your features keep working as new features are added.

Now you are familiar with all the pieces of the PyMeasure development work-flow. We look forward to seeing your pull-request!

CHAPTER

NINE

REPORTING AN ERROR

Please report all errors to the Issues section of the PyMeasure GitHub repository. Use the search function to determine if there is an existing or resolved issued before posting.

CHAPTER

TEN

ADDING INSTRUMENTS

You can make a significant contribution to PyMeasure by adding a new instrument to the `pymeasure.instruments` package. Even adding an instrument with a few features can help get the ball rolling, since its likely that others are interested in the same instrument.

Before getting started, become familiar with the *contributing work-flow* for PyMeasure, which steps through the process of adding a new feature (like an instrument) to the development version of the source code. This section will describe how to lay out your instrument code.

10.1 File structure

Your new instrument should be placed in the directory corresponding to the manufacturer of the instrument. For example, if you are going to add an "Extreme 5000" instrument you should add the following files assuming "Extreme" is the manufacturer. Use lowercase for all filenames to distinguish packages from CamelCase Python classes.

```
pymeasure/pymeasure/instruments/extreme/
    |--> __init__.py
    |--> extreme5000.py
```

10.1.1 Updating the init file

The `__init__.py` file in the manufacturer directory should import all of the instruments that correspond to the manufacturer, to allow the files to be easily imported. For a new manufacturer, the manufacturer should also be added to `pymeasure/pymeasure/instruments/__init__.py`.

10.1.2 Add test files

Test files (pytest) for each instrument are highly encouraged, as they help verify the code and implement changes. Testing new code parts with a test (Test Driven Development) is a good way for fast and good programming, as you catch errors early on.

```
pymeasure/tests/instruments/extreme/
    |--> test_extreme5000.py
```

10.1.3 Adding documentation

Documentation for each instrument is required, and helps others understand the features you have implemented. Add a new reStructuredText file to the documentation.

```
pymeasure/docs/api/instruments/extreme/
    |--> index.rst
    |--> extreme5000.rst
```

Copy an existing instrument documentation file, which will automatically generate the documentation for the instrument. The `index.rst` file should link to the `extreme5000` file. For a new manufacturer, the manufacturer should be also linked in `pymeasure/docs/api/instruments/index.rst`.

10.2 Instrument file

All standard instruments should be child class of *Instrument*. This provides the basic functionality for working with *Adapters*, which perform the actual communication.

The most basic instrument, for our "Extreme 5000" example starts like this:

```
#
# This file is part of the PyMeasure package.
#
# Copyright (c) 2013-2023 PyMeasure Developers
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#

# from pymeasure.instruments import Instrument
```

This is a minimal instrument definition:

```
class Extreme5000(Instrument):
    """Control the imaginary Extreme 5000 instrument."""

    def __init__(self, adapter, name="Extreme 5000", **kwargs):
        super().__init__(
```

(continues on next page)

(continued from previous page)

```
            adapter,
            name,
            **kwargs
        )
```

Make sure to include the PyMeasure license to each file, and add yourself as an author to the `AUTHORS.txt` file.

There is a certain order of elements in an instrument class that is useful to adhere to:

- First, the initializer (the `__init__()` method), this makes it faster to find when browsing the source code.
- Then class attributes/variables, if you need them.
- Then properties (pymeasure-specific or generic Python variants). This will be the bulk of the implementation.
- Finally, any methods.

10.3 Your instrument's user interface

Your instrument will have a certain set of properties and methods that are available to a user and discoverable via the documentation or their editor's autocomplete function.

In principle you are free to choose how you do this (with the exception of standard SCPI properties like `id`). However, there are a couple of practices that have turned out to be useful to follow:

- Naming things is important. Try to choose clear, expressive, unambiguous names for your instrument's elements.
- If there are already similar instruments in the same "family" (like a power supply) in pymeasure, try to follow their lead where applicable. It's better if, e.g., all power supplies have a `current_limit` instead of an assortment of `current_max`, `Ilim`, `max_curr`, etc.
- If there is already an instrument with a similar command set, check if you can inherit from that one and just tweak a couple of things. This massively reduces code duplication and maintenance effort. The section *Instruments with similar features* shows how to achieve that.
- The bulk of your instrument's interface will probably be made up of properties for quantities to set and/or read out. Our custom properties (see *Writing properties* ff. below) offer some convenience features and are therefore preferable, but plain Python properties are also fine.
- "Actions", commands or verbs should typically be methods, not properties: `recall()`, `trigger_scan()`, `prepare_resistance_measurement()`, etc.
- This separation between properties and methods also naturally helps with observing the "command-query separation" principle.
- If your instrument has multiple identical channels, see XXX. TODO: write section on channel implementations

In principle, you are free to write any methods that are necessary for interacting with the instrument. When doing so, make sure to use the `self.ask(command)`, `self.write(command)`, and `self.read()` methods to issue commands instead of calling the adapter directly. If the communication requires changes to the commands sent/received, you can override these methods in your instrument, for further information see *advanced_communication_protocols*.

In practice, we have developed a number of best practices for making instruments easy to write and maintain. The following sections detail these, which are highly encouraged to follow.

10.3.1 Common instrument types

There are a number of categories that many instruments fit into. In the future, pymeasure should gain an abstraction layer based on that, see this issue. Until that is ready, here are a couple of guidelines towards a more uniform API. Note that not all already available instruments follow these, but expect this to be harmonized in the future.

Frequent properties

If your instrument has an **output** that can be switched on and off, use a *boolean property* called `output_enabled`.

Power supplies

PSUs typically can measure the *actual* current and voltage, as well as have settings for the voltage level and the current limit. To keep naming clear and avoid confusion, implement the properties `current`, `voltage`, `voltage_setpoint` and `current_limit`, respectively.

10.3.2 Managing status codes or other indicator values

Often, an instrument features one or more collections of specific values that signal some status, an instrument mode or a number of possible configuration values. Typically, these are collected in mappings of some sort, as you want to provide a clear and understandable value to the user, while abstracting away the raw data, think `ACQUISITION_MODE` instead of `0x04`. The mappings normally are kept at module level (i.e. not defined within the instrument class), so that they are available when using the property factories. This is a small drawback of using Python class attributes.

The easiest way to handle these mappings is a plain `dict`. However, there is often a better way, the Python `enum.Enum`. To cite the Python documentation,

> An Enum is a set of symbolic names bound to unique values. They are similar to global variables, but they offer a more useful `repr()`, grouping, type-safety, and a few other features.

As our signal values are often integers, the most appropriate enum types are `IntEnum` and `IntFlag`.

`IntEnum` is the same as `Enum`, but its members are also integers and can be used anywhere that an integer can be used (so their use for composing commands is transparent), but logic/code they appear in is much more legible.

```
>>> from enum import IntEnum
>>> class InstrMode(IntEnum):
...     WAITING = 0x00
...     HEATING = 0x01
...     COOLING = 0x05
...
>>> received_from_device = 0x01
>>> current_mode = InstrMode(received_from_device)
>>> if current_mode == InstrMode.WAITING:
...     print('Idle')
... else:
...     print(current_mode)
...     print(f'Mode value: {current_mode}')
...
InstrMode.HEATING
Mode value: 1
```

`IntFlag` has the added benefit that it supports bitwise operators and combinations, and as such is a good fit for status bitmasks or error codes that can represent multiple values:

```
>>> from enum import IntFlag
>>> class ErrorCode(IntFlag):
...     TEMP_OUT_OF_RANGE = 8
...     TEMPSENSOR_FAILURE = 4
...     COOLER_FAILURE = 2
...     HEATER_FAILURE = 1
...     OK = 0
...
>>> received_from_device = 7
>>> print(ErrorCode(received_from_device))
ErrorCode.TEMPSENSOR_FAILURE|COOLER_FAILURE|HEATER_FAILURE
```

`IntFlags` are used by many instruments for the purpose just demonstrated.

The status property could look like this:

```
status = Instrument.measurement(
    "STB?",
    """Measure the status of the device as enum.""",
    get_process=lambda v: ErrorCode(v),
)
```

10.4 Defining default connection settings

When implementing instruments, it's sometimes necessary to define default connection settings. This might be because an instrument connection requires *specific non-default settings*, or because your instrument actually supports *multiple interfaces*.

The *VISAAdapter* class offers a flexible way of dealing with connection settings fully within the initializer of your instrument.

10.4.1 Single interface

The simplest version, suitable when the instrument connection needs default settings, just passes all keywords through to the `Instrument` initializer, which hands them over to *VISAAdapter* if `adapter` is a string or integer.

```
def __init__(self, adapter, name="Extreme 5000", **kwargs):
    super().__init__(
        adapter,
        name,
        **kwargs
    )
```

If you want to set defaults that should be prominently visible to the user and may be overridden, place them in the signature. This is suitable when the instrument has one type of interface, or any defaults are valid for all interface types, see the documentation in *VISAAdapter* for details.

```
def __init__(self, adapter, name="Extreme 5000", baud_rate=2400, **kwargs):
    super().__init__(
        adapter,
        name,
```

(continues on next page)

(continued from previous page)

```
        baud_rate=baud_rate,
        **kwargs
    )
```

If you want to set defaults, but they don't need to be prominently exposed for replacement, use this pattern, which sets the value only when there is no entry in `kwargs`, yet.

```
def __init__(self, adapter, name="Extreme 5000", **kwargs):
    kwargs.setdefault('timeout', 1500)
    super().__init__(
        adapter,
        name,
        **kwargs
    )
```

10.4.2 Multiple interfaces

Now, if you have instruments with multiple interfaces (e.g. serial, TCPI/IP, USB), things get interesting. You might have settings common to all interfaces (like `timeout`), but also settings that are only valid for one interface type, but not others.

The trick is to add keyword arguments that name the interface type, like `asrl` or `gpib`, below (see here for the full list). These then contain a *dictionary* with the settings specific to the respective interface:

```
def __init__(self, adapter, name="Extreme 5000", baud_rate=2400, **kwargs):
    kwargs.setdefault('timeout', 1500)
    super().__init__(
        adapter,
        name,
        gpib=dict(enable_repeat_addressing=False,
                  read_termination='\r'),
        asrl={'baud_rate': baud_rate,
              'read_termination': '\r\n'},
        **kwargs
    )
```

When the instrument instance is created, the interface-specific settings for the actual interface being used get merged with `**kwargs` before passing them on to PyVISA, the rest is discarded. This way, we always pass on a valid set of arguments. In addition, any entries in `**kwargs**` take precedence, so if they need to, it is *still* possible for users to override any defaults you set in the instrument definition.

For many instruments, the simple way presented first is enough, but in case you have a more complex arrangement to implement, see whether *advanced_communication_protocols* fits your bill. If, for some exotic reason, you need a special connection type, which you cannot model with PyVISA, you can write your own Adapter.

10.5 Writing properties

In PyMeasure, Python properties are the preferred method for dealing with variables that are read or set.

10.5.1 The property factories

PyMeasure comes with three central convenience factory functions for making properties for classes: *CommonBase.control*, *CommonBase.measurement*, and *CommonBase.setting*. You can call them, however, as `Instrument.control`, `Instrument.measurement`, and `Instrument.setting`.

The *Instrument.measurement* function returns a property that can only read values from an instrument. For example, if our "Extreme 5000" has the `*IDN?` command, we can write the following property to be added after the `def __init__` line in our above example class, or added to the class after the fact as in the code here:

```
Extreme5000.cell_temp = Instrument.measurement(
   ":TEMP?",
   """Measure the temperature of the reaction cell.""",
)
```

You will notice that a documentation string is required, see *Docstrings* for details.

When we use this property we will get the temperature of the reaction cell.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.cell_temp  # Sends ":TEMP?" to the device
127.2
```

The *Instrument.control* function extends this behavior by creating a property that you can read and set. For example, if our "Extreme 5000" has the `:VOLT?` and `:VOLT <float>` commands that are in Volts, we can write the following property.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float)."""
)
```

You will notice that we use the Python string format `%g` to format passed-through values as floating point.

We can use this property to set the voltage to 100 mV, which will send the appropriate command, and then to request the current voltage:

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 0.1        # Sends ":VOLT 0.1" to set the voltage to 100 mV
>>> extreme.voltage              # Sends ":VOLT?" to query for the current value
0.1
```

Finally, the *Instrument.setting* function can only set, but not read values.

Using the *Instrument.control*, *Instrument.measurement*, and *Instrument.control* functions, you can create a number of properties for basic measurements and controls.

The next sections detail additional features of the property factories. These allow you to write properties that cover specific ranges, or that have to map between a real value to one used in the command. Furthermore it is shown how to perform more complex processing of return values from your device.

10.5.2 Restricting values with validators

Many GPIB/SCPI commands are more restrictive than our basic examples above. The *Instrument.control* function has the ability to encode these restrictions using *validators*. A validator is a function that takes a value and a set of values, and returns a valid value or raises an exception. There are a number of pre-defined validators in *pymeasure.instruments.validators* that should cover most situations. We will cover the four basic types here.

In the examples below we assume you have imported the validators.

In many situations you will also need to process the return string in order to extract the wanted quantity or process a value before sending it to the device. The *Instrument.control*, *Instrument.measurement* and *Instrument.setting* function also provide means to achieve this.

In a restricted range

If you have a property with a restricted range, you can use the `strict_range` and `truncated_range` functions.

For example, if our "Extreme 5000" can only support voltages from -1 V to 1 V, we can modify our previous example to use a strict validator over this range.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float strictly from -1 to 1).""",
    validator=strict_range,
    values=[-1, 1]
)
```

Now our voltage will raise a ValueError if the value is out of the range.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 100
Traceback (most recent call last):
...
ValueError: Value of 100 is not in range [-1,1]
```

This is useful if you want to alert the programmer that they are using an invalid value. However, sometimes it can be nicer to truncate the value to be within the range.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float from -1 to 1).

    Invalid voltages are truncated.
    """,
    validator=truncated_range,
    values=[-1, 1]
)
```

Now our voltage will not raise an error, and will truncate the value to the range bounds.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 100  # Executes ":VOLT 1"
>>> extreme.voltage
1.0
```

In a discrete set

Often a control property should only take a few discrete values. You can use the `strict_discrete_set` and `truncated_discrete_set` functions to handle these situations. The strict version raises an error if the value is not in the set, as in the range examples above.

For example, if our "Extreme 5000" has a `:RANG <float>` command that sets the voltage range that can take values of 10 mV, 100 mV, and 1 V in Volts, then we can write a control as follows.

```
Extreme5000.voltage = Instrument.control(
    ":RANG?", ":RANG %g",
    """Control the voltage range in Volts (float in 10e-3, 100e-3, 1).""",
    validator=truncated_discrete_set,
    values=[10e-3, 100e-3, 1]
)
```

Now we can set the voltage range, which will automatically truncate to an appropriate value.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 0.08
>>> extreme.voltage
0.1
```

10.5.3 Mapping values

Now that you are familiar with the validators, you can additionally use maps to satisfy instruments which require non-physical values. The `map_values` argument of *`Instrument.control`* enables this feature.

If your set of values is a list, then the command will use the index of the list. For example, if our "Extreme 5000" instead has a `:RANG <integer>`, where 0, 1, and 2 correspond to 10 mV, 100 mV, and 1 V, then we can use the following control.

```
Extreme5000.voltage = Instrument.control(
    ":RANG?", ":RANG %d",
    """Control the voltage range in Volts (float in 10 mV, 100 mV and 1 V).
    """,
    validator=truncated_discrete_set,
    values=[10e-3, 100e-3, 1],
    map_values=True
)
```

Now the actual GPIB/SCIP command is ":RANG 1" for a value of 100 mV, since the index of 100 mV in the values list is 1.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 100e-3
>>> extreme.read()
'1'
>>> extreme.voltage = 1
>>> extreme.voltage
1
```

Dictionaries provide a more flexible method for mapping between real-values and those required by the instrument. If instead the `:RANG <integer>` took 1, 2, and 3 to correspond to 10 mV, 100 mV, and 1 V, then we can replace our previous control with the following.

```
Extreme5000.voltage = Instrument.control(
    ":RANG?", ":RANG %d",
    """Control the voltage range in Volts (float in 10 mV, 100 mV and 1 V).
    """,
    validator=truncated_discrete_set,
    values={10e-3:1, 100e-3:2, 1:3},
    map_values=True
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 10e-3
>>> extreme.read()
'1'
>>> extreme.voltage = 100e-3
>>> extreme.voltage
0.1
```

The dictionary now maps the keys to specific values. The values and keys can be any type, so this can support properties that use strings:

```
Extreme5000.channel = Instrument.control(
    ":CHAN?", ":CHAN %d",
    """Control the measurement channel (string strictly in 'X', 'Y', 'Z')."""",
    validator=strict_discrete_set,
    values={'X':1, 'Y':2, 'Z':3},
    map_values=True
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.channel = 'X'
>>> extreme.read()
'1'
>>> extreme.channel = 'Y'
>>> extreme.channel
'Y'
```

As you have seen, the *Instrument.control* function can be significantly extended by using validators and maps.

10.5.4 Boolean properties

The idea of using maps can be leveraged to implement properties where the user-facing values are booleans, so you can interact in a pythonic way using `True` and `False`:

```
Extreme5000.output_enabled = Instrument.control(
    "OUTP?", "OUTP %d",
    """Control the instrument output is enabled (boolean).""",
    validator=strict_discrete_set,
    map_values=True,
    values={True: 1, False: 0},  # the dict values could also be "on" and "off", etc.␣
↪depending on the device
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.output_enabled = True
>>> extreme.read()
'1'
>>> extreme.output_enabled = False
>>> extreme.output_enabled
False
>>> # Invalid input raises an exception
>>> extreme.output_enabled = 34
Traceback (most recent call last):
...
ValueError: Value of 34 is not in the discrete set {True: 1, False: 0}
```

Good names for boolean properties are chosen such that they could also be a yes/no question: "Is the output enabled?" -> `output_enabled`, `display_active`, etc.

10.5.5 Processing of set values

The *Instrument.control*, and *Instrument.setting* allow a keyword argument *set_process* which must be a function that takes a value after validation and performs processing before value mapping. This function must return the processed value. This can be typically used for unit conversions as in the following example:

```
Extreme5000.current = Instrument.setting(
    ":CURR %g",
    """Set the measurement current in A (float strictly from 0 to 10).""",
    validator=strict_range,
    values=[0, 10],
    set_process=lambda v: 1e3*v,  # convert current to mA
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.current = 1  # set current to 1000 mA
```

10.5.6 Processing of return values

Similar to *set_process* the *Instrument.control*, and *Instrument.measurement* functions allow a *get_process* argument which if specified must be a function that takes a value and performs processing before value mapping. The function must return the processed value. In analogy to the example above this can be used for example for unit conversion:

```
Extreme5000.current = Instrument.control(
    ":CURR?", ":CURR %g",
    """Control the measurement current in A (float strictly from 0 to 10).""",
    validator=strict_range,
    values=[0, 10],
    set_process=lambda v: 1e3*v,  # convert to mA
    get_process=lambda v: 1e-3*v,  # convert to A
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.current = 3.1
>>> extreme.current
3.1
```

Another use-case of *set-process*, *get-process* is conversion from/to a `pint.Quantity`. Modifying above example to set or return a quantity, we get:

```
from pymeasure.units import ureg

Extreme5000.current = Instrument.control(
    ":CURR?", ":CURR %g",
    """Control the measurement current (float).""",
    set_process=lambda v: v.m_as(ureg.mA),  # send the value as mA to the device
    get_process=lambda v: ureg.Quantity(v, ureg.mA),  # convert to quantity
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.current = 3.1 * ureg.A
>>> extreme.current.m_as(ureg.A)
3.1
```

Note: This is, how quantities can be used in pymeasure instruments right now. Issue 666 develops a more convenient implementation of quantities in the property factories.

get_process can also be used to perform string processing. Let's say your instrument returns a value with its unit (e.g. `1.23 nF`), which has to be removed. This could be achieved by the following code:

```
Extreme5000.capacity = Instrument.measurement(
    ":CAP?",
    """Measure the capacity in nF (float).""",
    get_process=lambda v: float(v.replace('nF', ''))
)
```

The same can be also achieved by the *preprocess_reply* keyword argument to *Instrument.control* or *Instrument.measurement*. This function is forwarded to `Adapter.values` and runs directly after receiving the reply from the device. One can therefore take advantage of the built in casting abilities and simplify the code accordingly:

```
Extreme5000.capacity = Instrument.measurement(
    ":CAP?",
    """Measure the capacity in nF (float).""",
    preprocess_reply=lambda v: v.replace('nF', '')
    # notice how we don't need to cast to float anymore
)
```

10.5.7 Tweaking command strings

If you need to tweak

- the `set_command` string immediately before the value to set is inserted via string formatting (`%g` etc.), or
- the `get_command` string before sending it to the device,

use the `command_process` parameter of *control()*.

Note that there is only one parameter for both setting and getting, so the utility of this is probably limited. Note also that for adding e.g. channel identifiers, there are other, more preferable methods.

10.5.8 Checking the instrument for errors

If you need to separately ask your instrument about its error state after getting/setting, use the parameters `check_get_errors` and `check_set_errors` of *control()*, respectively. If those are enabled, the method *check_errors()* will be called after device communication has concluded.

10.5.9 Using multiple values

Seldomly, you might need to send/receive multiple values in one command. The *Instrument.control* function can be used with multiple values at one time, passed as a tuple. Say, we may set voltages and frequencies in our "Extreme 5000", and the the commands for this are `:VOLTFREQ?` and `:VOLTFREQ <float>,<float>`, we could use the following property:

```
Extreme5000.combination = Instrument.control(
    ":VOLTFREQ?", ":VOLTFREQ %g,%g",
    """Simultaneously control the voltage in Volts and the frequency in Hertz (both␣
↪float).

    This property is set by a tuple.
    """
)
```

In use, we could set the voltage to 200 mV, and the Frequency to 931 Hz, and read both values immediately afterwards.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.combination = (0.2, 931)        # Executes ":VOLTFREQ 0.2,931"
>>> extreme.combination                     # Reads ":VOLTFREQ?"
[0.2, 931.0]
```

This interface is not too convenient, but luckily not often needed.

10.5.10 Dynamic properties

As described in previous sections, Python properties are a very powerful tool to easily code an instrument's programming interface. One very interesting feature provided in PyMeasure is the ability to adjust properties' behaviour in subclasses or dynamically in instances. This feature allows accomodating some interesting use cases with a very compact syntax.

Dynamic features of a property are enabled by setting its `dynamic` parameter to `True`.

Afterwards, creating specifically-named attributes (either in class definitions or on instances) allows modifying the parameters used at the time of property definition. You need to define an attribute whose name is *<property*

name>_<*property_parameter*> and assign to it the desired value. Pay attention *not* to inadvertently define other class attribute or instance attribute names matching this pattern, since they could unintentionally modify the property behaviour.

Note: To clearly distinguish these special attributes from normal class/instance attributes, they can only be set, not read.

The mechanism works for all the parameters in properties, except `dynamic` and `docs` – see *Instrument.control*, *Instrument.measurement*, *Instrument.setting*.

Dynamic validity range

Let's assume we have an instrument with a command that accepts a different valid range of values depending on its current state. The code below shows how this can be accomplished with dynamic properties.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float).""",
    validator=strict_range,
    values=[-1, 1],
    dynamic = True,
)
def set_bipolar_mode(self, enabled = True):
    """Safely switch between bipolar/unipolar mode."""

    # some code to switch off the output first
    # ...

    if enabled:
        self.mode = "BIPOLAR"
        # set valid range of "voltage" property
        self.voltage_values = [-1, 1]
    else:
        self.mode = "UNIPOLAR"
        # note the "propertyname_parametername" form of the attribute
        self.voltage_values = [0, 1]
```

Now our voltage property has a dynamic validity range, either [-1, 1] or [0, 1]. A side effect of this is that the property's docstring should be less specific, to avoid it containing dynamically changed information (like the admissible value range). In this example, the property name was `voltage` and the parameter to adjust was `values`, so we used `self.voltage_values` to set our desired values.

10.6 Instruments with similar features

When instruments have a similar set of features, it makes sense to use inheritance to obtain most of the functionality from a parent instrument class, instead of copy-pasting code.

Note: Don't forget to update the instrument's `name` attribute accordingly, by either supplying an appropriate argument (if available) during the `super().__init__()` call, or by setting it anew below that call.

In some cases, one only needs to add additional properties and methods. In other cases, some of the already present properties/methods need to be completely replaced by defining them again in the derived class. Often, however, only some details need to be changed. This can be dealt with efficiently using dynamic properties.

10.6.1 Instrument family with different parameter values

A common case is to have a family of similar instruments with some parameter range different for each family member. In this case you would update the specific class parameter range without rewriting the entire property:

```
class FictionalInstrumentFamily(Instrument):
    frequency = Instrument.setting(
        "FREQ %g",
        """Set the frequency (float).""",
        validator=strict_range,
        values=[0, 1e9],
        dynamic=True,
        # ... other possible parameters follow
    )
    #
    # ... complete class implementation here
    #

class FictionalInstrument_1GHz(FictionalInstrumentFamily):
    pass

class FictionalInstrument_3GHz(FictionalInstrumentFamily):
    frequency_values = [0, 3e9]

class FictionalInstrument_9GHz(FictionalInstrumentFamily):
    frequency_values = [0, 9e9]
```

Notice how easily you can derive the different family members from a common class, and the fact that the attribute is now defined at class level and not at instance level.

10.6.2 Instruments with similar command syntax

Another use case involves maintaining compatibility between instruments with commands having different syntax, like in the following example.

```
class MultimeterA(Instrument):
    voltage = Instrument.measurement(get_command="VOLT?",...)

    # ...full class definition code here

class MultimeterB(MultimeterA):
    # Same as brand A multimeter, but the command to read voltage
    # is slightly different
    voltage_get_command = "VOLTAGE?"
```

In the above example, `MultimeterA` and `MultimeterB` use a different command to read the voltage, but the rest of the behaviour is identical. `MultimeterB` can be defined subclassing `MultimeterA` and just implementing the difference.

10.7 Instruments with channels

Some instruments, like oscilloscopes and voltage sources, have channels whose commands differ only in the channel name. For this case, we have *Channel*, which is similar to *Instrument* and its property factories, but does expect an *Instrument* instance (i.e., a parent instrument) instead of an *Adapter* as parameter. All the channel communication is routed through the instrument's methods (*write*, *read*, etc.). However, *Channel.insert_id* uses *str.format* to insert the channel's id at any occurence of the class attribute `Channel.placeholder`, which defaults to `"ch"`, in the written commands. For example `"Ch{ch}:VOLT?"` will be sent as `"Ch3:VOLT?"` to the device, if the channel's id is "3".

In order to add a channel to an instrument or to another channel (nesting channels is possible), create the channels with the class *ChannelCreator* as class attributes. Its constructor accepts a single channel class or list of classes and a list of corresponding ids. Instead of lists, you may also use tuples. If you give a single class and a list of ids, all channels will be of the same class.

At instrument instantiation, the instrument will add the channels accordingly with the attribute names as a composition of the prefix (default `"ch_"`) and channel id, e.g. the channel with id "A" will be added as attribute `ch_A`. Additionally, the channels will be collected in a dictionary with the same name as you used for the *ChannelCreator*. Without pressing reasons, call the dictionary `channels` and do not change the default prefix in order to keep the code base homogeneous.

In order to add or remove programatically channels, use the parent's *add_child()*, *remove_child()* methods.

```
class VoltageChannel(Channel):
    """A channel of the voltage source."""

    voltage = Channel.control(
        "SOURce{ch}:VOLT?", "SOURce{ch}:VOLT %g",
        """Control the output voltage of this channel.""",
    )

class InstrumentWithChannels(Instrument):
    """An instrument with channels."""
    channels = Instrument.ChannelCreator(VoltageChannel, ("A", "B"))
```

If you set the voltage of the first channel of above `ExtremeChannel` instrument with `inst.chA.voltage = 1.23`, the driver sends `"SOURceA:VOLT 1.23"` to the device, supplying the "A" of the channel name. The same channel could be addressed with `inst.channels["A"].voltage = 1.23` as well.

10.7.1 Channels with fixed prefix

If all channel communication is prefixed by a specific command, e.g. `"SOURceA:"` for channel A, you can override the channel's `insert_id()` method. That is especially useful, if you have only one channel of that type, e.g. because it defines one function of the instrument vs. another one.

```
class VoltageChannelPrefix(Channel):
    """A channel of a voltage source, every command has the same prefix."""

    def insert_id(self, command):
        return f"SOURce{self.id}:{command}"

    voltage = Channel.control(
        "VOLT?", "VOLT %g",
        """Control the output voltage of this channel.""",
    )
```

This channel class implements the same communication as the previous example, but implements the channel prefix in the `insert_id()` method and not in the individual property (created by `control()`).

10.7.2 Collections of different channel types

Some devices have different types of channels. In this case, you can specify a different *collection* and *prefix* parameter.

```
class PowerChannel(Channel):
    """A channel controlling the power."""

    power = Channel.measurement(
        "POWER?", """Measure the currently consumed power.""")

class MultiChannelTypeInstrument(Instrument):
    """An instrument with two different channel types."""
    analog = Instrument.ChannelCreator(
        (VoltageChannel, VoltageChannelPrefix),
        ("A", "B"),
        prefix="an_")
    digital = Instrument.ChannelCreator(VoltageChannel, (0, 1, 2), prefix="di_")
    power = Instrument.ChannelCreator(PowerChannel, prefix=None)
```

This instrument has two collections of channels and one single channel. The first collection in the dictionary `analog` contains an instance of `VoltageChannel` with the name `an_A` and an instance of `VoltageChannelPrefix` with the name `an_B`. The second collection contains three channels of type `VoltageChannel` with the names `di_0, di_1, di_2` in the dictionary `digital`. You can address the first channel of the second group either with `inst.di_0` or with `inst.digital[0]`. Finally, the instrument has a single channel with the name `power`, as it does not have a prefix.

If you have a single channel category, do not change the default parameters of *ChannelCreator* or *add_child()*, in order to keep the code base homogeneous. We expect the default behaviour to be sufficient for most use cases.

10.8 Advanced communication protocols

Some devices require a more advanced communication protocol, e.g. due to checksums or device addresses. In most cases, it is sufficient to subclass *Instrument.write* and *Instrument.read*.

10.8.1 Instrument's inner workings

In order to adjust an instrument for more complicated protocols, it is key to understand the different parts.

The *Adapter* exposes *write()* and *read()* for strings, *write_bytes()* and *read_bytes()* for bytes messages. These are the most basic methods, which log all the traffic going through them. For the actual communication, they call private methods of the Adapter in use, e.g. `VISAAdapter._read`. For binary data, like waveforms, the adapter provides also *write_binary_values()* and *read_binary_values()*, which use the aforementioned methods. You do not need to call all these methods directly, instead, you should use the methods of *Instrument* with the same name. They call the Adapter for you and keep the code tidy.

Now to *Instrument*. The most important methods are *write()* and *read()*, as they are the most basic building blocks for the communication. The pymeasure properties (*Instrument.control* and its derivatives *Instrument.measurement* and *Instrument.setting*) and probably most of your methods and properties will call them. In any instrument, `write()` should write a general string command to the device in such a way, that it understands it. Similarly, `read()` should return a string in a general fashion in order to process it further.

The getter of *Instrument.control* does not call them directly, but via a chain of methods. It calls `values()` which in turn calls `ask()` and processes the returned string into understandable values. `ask()` sends the readout command via `write()`, waits some time if necessary via `wait_for()`, and reads the device response via `read()`.

Similarly, `Instrument.binary_values` sends a command via `write()`, waits with `wait_till_read()`, but reads the response via *Adapter.read_binary_values*.

10.8.2 Adding a device address and adding delay

Let's look at a simple example for a device, which requires its address as the first three characters and returns the same style. This is straightforward, as `write()` just prepends the device address to the command, and `read()` has to strip it again doing some error checking. Similarly, a checksum could be added. Additionally, the device needs some time after it received a command, before it responds, therefore `wait_for()` waits always a certain time span.

```
class ExtremeCommunication(Instrument):
    """Control the ExtremeCommunication instrument.

    :param address: The device address for the communication.
    :param query_delay: Wait time after writing and before reading in seconds.
    """
    def __init__(self, adapter, name="ExtremeCommunication", address=0, query_delay=0.1):
        super().__init__(adapter, name)
        self.address = f"{address:03}"
        self.query_delay = query_delay

    def write(self, command):
        """Add the device address in front of every command before sending it."""
        super().write(self.address + command)

    def wait_for(self, query_delay=0):
        """Wait for some time.
```

(continues on next page)

(continued from previous page)

```
        :param query_delay: override the global query_delay.
        """
        super().wait_for(query_delay or self.query_delay)

    def read(self):
        """Read from the device and check the response.

        Assert that the response starts with the device address.
        """
        got = super().read()
        if got.startswith(self.address):
            return got[3:]
        else:
            raise ConnectionError(f"Expected message address '{self.address}', but read '
↪{got[3:]}' for wrong address '{got[:3]}'.")

    voltage = Instrument.measurement(
        ":VOLT:?", """Measure the voltage in Volts.""")
```

If the device is initialized with `address=12`, a request for the voltage would send `"012:VOLT:?"` to the device and expect a response beginning with `"012"`.

10.8.3 Bytes communication

Some devices do not expect ASCII strings but raw bytes. In those cases, you can call the `write_bytes()` and `read_bytes()` in your `write()` and `read()` methods. The following example shows an instrument, which has registers to be written and read via bytes sent.

```
class ExtremeBytes(Instrument):
    """Control the ExtremeBytes instrument with byte-based communication."""
    def __init__(self, adapter, name="ExtremeBytes"):
        super().__init__(adapter, name)

    def write(self, command):
        """Write to the device according to the comma separated command.

        :param command: R or W for read or write, hexadecimal address, and data.
        """
        function, address, data = command.split(",")
        b = [0x03] if function == "R" else [0x10]
        b.extend(int(address, 16).to_bytes(2, byteorder="big"))
        b.extend(int(data).to_bytes(length=8, byteorder="big", signed=True))
        self.write_bytes(bytes(b))

    def read(self):
        """Read the response and return the data as a string, if applicable."""
        response = self.read_bytes(2)  # return type and payload
        if response[0] == 0x00:
            raise ConnectionError(f"Device error of type {response[1]} occurred.")
        if response[0] == 0x03:
```

(continues on next page)

(continued from previous page)

```
            # read that many bytes and return them as an integer
            data = self.read_bytes(response[1])
            return str(int.from_bytes(data, byteorder="big", signed=True))
        if response[0] == 0x10 and response[1] != 0x00:
            raise ConnectionError(f"Writing to the device failed with error {response[1]}
↪")

    voltage = Instrument.control(
        "R,0x106,1", "W,0x106,%i",
        """Control the output voltage in mV.""",
    )
```

10.9 Writing tests

Tests are very useful for writing good code. We have a number of tests checking the correctness of the pymeasure implementation. Those tests (located in the `tests` directory) are run automatically on our CI server, but you can also run them locally using `pytest`.

When adding instruments, your primary concern will be tests for the *instrument driver* you implement. We distinguish two groups of tests for instruments: the first group does not rely on a connected instrument. These tests verify that the implemented instrument driver exchanges the correct messages with a device (for example according to a device manual). We call those "protocol tests". The second group tests the code with a device connected.

Implement device tests by adding files in the `tests/instruments/...` directory tree, mirroring the structure of the instrument implementations. There are other instrument tests already available that can serve as inspiration.

10.9.1 Protocol tests

In order to verify the expected working of the device code, it is good to test every part of the written code. The *expected_protocol()* context manager (using a *ProtocolAdapter* internally) simulates the communication with a device and verifies that the sent/received messages triggered by the code inside the `with` statement match the expectation.

```
import pytest

from pymeasure.test import expected_protocol

from pymeasure.instruments.extreme5000 import Extreme5000

def test_voltage():
    """Verify the communication of the voltage getter."""
    with expected_protocol(
        Extreme5000,
        [(":VOLT 0.345", None),
         (":VOLT?", "0.3000")],
    ) as inst:
        inst.voltage = 0.345
        assert inst.voltage == 0.3
```

In the above example, the imports import the pytest package, the expected_protocol and the instrument class to be tested.

The first parameter, Extreme5000, is the class to be tested.

When setting the voltage, the driver sends a message (`":VOLT 0.345"`), but does not expect a response (`None`). Getting the voltage sends a query (`":VOLT?"`) and expects a string response (`"0.3000"`). Therefore, we expect two pairs of send/receive exchange. The list of those pairs is the second argument, the expected message protocol.

The context manager returns an instance of the class (`inst`), which is then used to trigger the behaviour corresponding to the message protocol (e.g. by using its properties).

If the communication of the driver does not correspond to the expected messages, an Exception is raised.

Note: The expected messages are **without** the termination characters, as they depend on the connection type and are handled by the normal adapter (e.g. `VISAAdapter`).

Some protocol tests in the test suite can serve as examples:

- Testing a simple instrument: `tests/instruments/keithley/test_keithley2000.py`
- Testing a multi-channel instrument: `tests/instruments/tektronix/test_afg3152.py`
- Testing instruments using frame-based communication: `tests/instruments/hcp/tc038.py`

10.9.2 Device tests

It can be useful as well to test the code against an actual device. The necessary device setup instructions (for example: connect a probe to the test output) should be written in the header of the test file or test methods. There should be the connection configuration (for example serial port), too. In order to distinguish the test module from protocol tests, the filename should be `test_instrumentName_with_device.py`, if the device is called `instrumentName`.

To make it easier for others to run these tests using their own instruments, we recommend to use `pytest.fixture` to create an instance of the instrument class. It is important to use the specific argument name `connected_device_address` and define the scope of the fixture to only establish a single connection to the device. This ensures two things: First, it makes it possible to specify the address of the device to be used for the test using the `--device-address` command line argument. Second, tests using this fixture, i.e. tests that rely on a device to be connected to the computer are skipped by default when running pytest. This is done to avoid that tests that require a device are run when none is connected. It is important that all tests that require a connection to a device either use the `connected_device_address` fixture or a fixture derived from it as an argument.

A simple example of a fixture that returns a connected instrument instance looks like this:

```
@pytest.fixture(scope="module")
def extreme5000(connected_device_address):
    instr = Extreme5000(connected_device_address)
    # ensure the device is in a defined state, e.g. by resetting it.
    return instr
```

Note that this fixture uses `connected_device_address` as an input argument and will thus be skipped by automatic test runs. This fixture can then be used in a test functions like this:

```
def test_voltage(extreme5000):
    extreme5000.voltage = 0.345
    assert extreme5000.voltage == 0.3
```

Again, by specifying the fixture's name, in this case `extreme5000`, invoking `pytest` will skip these tests by default.

It is also possible to define derived fixtures, for example to put the device into a specific state. Such a fixture would look like this:

```
@pytest.fixture
def auto_scaled_extreme5000(extreme5000):
    extreme5000.auto_scale()
    return extreme5000
```

In this case, do not specify the fixture's scope, so it is called again for every test function using it.

To run the test, specify the address of the device to be used via the `--device-address` command line argument and limit pytest to the relevant tests. You can filter tests with the `-k` option or you can specify the filename. For example, if your tests are in a file called `test_extreme5000_with_device.py`, invoke pytest with `pytest -k extreme5000 --device-address TCPIP::192.168.0.123::INSTR"`.

There might also be tests where manual intervention is necessary. In this case, skip the test by prepending the test function with a `@pytest.mark.skip(reason="A human needs to press a button.")` decorator.

CHAPTER

ELEVEN

CODING STANDARDS

In order to maintain consistency across the different instruments in the PyMeasure repository, we enforce the following standards.

11.1 Python style guides

The PEP8 style guide and PEP257 docstring conventions should be followed.

Function and variable names should be lower case with underscores as needed to separate words. CamelCase should only be used for class names, unless working with Qt, where its use is common.

In addition, there is a configuration for the flake8 linter present. Our codebase should not trigger any warnings. Many editors/IDEs can run this tool in the background while you work, showing results inline. Alternatively, you can run `flake8` in the repository root to check for problems. In addition, our automation on Github also runs some checkers. As this results in a much slower feedback loop for you, it's not recommended to rely only on this.

It is allowed but not required to use the black code formatter. To avoid introducing unrelated changes when working on an existing file, it is recommended to use the darker tool instead of *black*. This helps to keep the focus on the implementation instead of unrelated formatting, and thereby facilitates code reviews. `darker` is compatible with `black`, but only formats regions that show as changed in Git. If there are conflicts between `black`/`darker`'s output and flake8 (especially related to E203), flake8 takes precedence. Use `#noqa : E203` to disable E203 warnings for a specific line if appropriate.

There are no plans to support type hinting in PyMeasure code. This adds a lot of additional code to manage, without a clear advantage for this project. Type documentation should be placed in the docstring where not clear from the variable name.

11.2 Documentation

PyMeasure documents code using reStructuredText and the Sphinx documentation generator. All functions, classes, and methods should be documented in the code using a docstring, see section *Docstrings*.

11.3 Usage of getter and setter functions

Getter and setter functions are discouraged, since properties provide a more fluid experience. Given the extensive tools available for defining properties, detailed in the sections starting with *Writing properties*, these types of properties are prefered.

11.4 Docstrings

Descriptive and specific docstrings for your properties and methods are important for your users to quickly glean important information about a property. It is advisable to follow the PEP257 docstring guidelines. Most importantly:

- Use triple-quoted strings (""") to delimit docstrings.
- One short summary line in imperative voice, with a period at the end.
- Optionally, after a blank line, include more detailed information.
- For functions and methods, you can add documentation on their parameters using the reStructuredText docstring format.

Specific to our properties, start them with "Control", "Measure" or "Set" to indicate the kind of property (this information is not visible after import). In addition, it is useful to add type and information about *Restricting values with validators* (if applicable) at the end of the summary line, see the docstrings shown in examples throughout the *Adding instruments* section.

The docstring is for information that is relevant for *using* a property/method. Therefore, do *not* add information about internal/hidden details, like the format of commands exchanged with the device.

CHAPTER

TWELVE

AUTHORS

PyMeasure was started in 2013 by Colin Jermain and Graham Rowlands at Cornell University, when it became apparent that both were working on similar Python packages for scientific measurements. PyMeasure combined these efforts and continues to gain valuable contributions from other scientists who are interested in advancing measurement software.

The following developers have contributed to the PyMeasure package:

```
Colin Jermain
Graham Rowlands
Minh-Hai Nguyen
Guen Prawiro-Atmodjo
Tim van Boxtel
Davide Spirito
Marcos Guimaraes
Ghislain Antony Vaillant
Ben Feinstein
Neal Reynolds
Christoph Buchner
Julian Dlugosch
Sylvain Karlen
Joseph Mittelstaedt
Troy Fox
Vikram Sekar
Casper Schippers
Sumatran Tiger
Michael Schneider
Dennis Feng
Stefano Pirotta
Moritz Jung
Richard Schlitz
Manuel Zahn
Mikhaël Myara
Domenic Prete
Mathieu Jeannin
Paul Goulain
John McMaster
Dominik Kriegner
Jonathan Larochelle
Dominic Caron
Mathieu Plante
Michele Sardo
Steven Siegl
```

(continues on next page)

(continued from previous page)

Benjamin Klebel-Knobloch
Markus Röleke
Demetra Adrahtas
Dan McDonald
Hud Wahab
Nicola Corna
Robert Eckelmann
Sam Condon
Andreas Maeder
Bastian Leykauf
Matthew Delaney
Marco von Rosenberg
Jack Van Sambeek
JC Arbelbide
Florian Jünger
Benedikt Moneke
Asaf Yagoda
Fabio Garzetti
Daniel Schmeer
Mike Manno
David Sanchez Sanchez
Andres Ruz-Nieto
Carlos Martinez
Scott Candey
Tom Verbeure

CHAPTER

THIRTEEN

LICENSE

CHAPTER

FOURTEEN

CHANGELOG

14.1 Upcoming version

14.1.1 New adapter and instrument mechanics

- Channel class added. Instrument.channels and Instrument.ch_X (X is any channel name) are reserved for channel implementations.
- All instruments are required to accept a `name` argument.

14.2 Version 0.11.1 (2022-12-31)

14.2.1 Adapter and instrument mechanics

- Fix broken *PrologixAdapter.gpib*. Due to a bug in *VISAAdapter*, you could not get a second adapter with that connection (#765).

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.11.0...v0.11.1

14.3 Version 0.11.0 (2022-11-19)

Main items of this new release:

- 11 new instrument drivers have been added
- A method for testing instrument communication **without** hardware present has been added, see the documentation.
- The separation between `Instrument` and `Adapter` has been improved to make future modifications easier. Adapters now focus on the hardware communication, and the communication *protocol* should be defined in the Instruments. Details in a section below.
- The GUI is now compatible with Qt6.
- We have started to clean up our API in preparation for a future version 1.0. There will be deprecations and subsequent removals, which will be prominently listed in the changelog.

14.3.1 Deprecated features

In preparation for a stable 1.0 release and a more consistent API, we have now started formally deprecating some features. You should get warnings if those features are used.

- Adapter methods `ask`, `values`, `binary_values`, use `Instrument` methods of the same name instead.
- Adapter parameter `preprocess_reply`, override `Instrument.read` instead.
- `Adapter.query_delay` in favor of `Instrument.wait_for()`.
- Keithley 2260B: `enabled` property, use `output_enabled` instead.

14.3.2 New adapter and instrument mechanics

- Nothing should have changed for users, this section is mainly interesting for instrument implementors.
- Documentation in 'Advanced communication protocols' in 'Adding instruments'.
- Adapter logs written and read messages.
- Particular adapters (*VISAAdapter* etc.) implement the actual communication.
- `Instrument.control` getter calls `Instrument.values`.
- `Instrument.values` calls `Instrument.ask`, which calls `Instrument.write`, `wait_for`, and `read`.
- All protocol quirks of an instrument should be implemented overriding `Instrument.write` and `read`.
- `Instrument.wait_until_read` implements waiting between writing and reading.
- reading/writing binary values is in the `Adapter` class itself.
- `PrologixAdapter` is now based on `VISAAdapter`.
- `SerialAdapter` improved to be more similar to `VISAAdapter`: `read`/`write` use strings, `read`/`write_bytes` bytes. - Support for termination characters added.

14.3.3 Instruments

- New Active Technologies AWG-401x (@garzetti, #649)
- New Eurotest hpp_120_256_ieee (@sansanda, #701)
- New HC Photonics crystal ovens TC038, TC038D (@bmoneke, #621, #706)
- New HP 6632A/6633A/6634A power supplies (@LongnoseRob, #651)
- New HP 8657B RF signal generator (@LongnoseRob, #732)
- New Rohde&Schwarz HMP4040 power supply. (@bleykauf, #582)
- New Siglent SPDxxxxX series Power Supplies (@AidenDawn, #719)
- New Temptronic Thermostream devices (@mroeleke, #368)
- New TEXIO PSW-360L30 Power Supply (@LastStarDust, #698)
- New Thermostream ECO-560 (@AidenDawn, #679)
- New Thermotron 3800 Oven (@jcarbelbide, #606)
- Harmonize instruments' adapter argument (@bmoneke, #674)
- Harmonize usage of `shutdown` method (@LongnoseRob, #739)

- Rework Adapter structure (@bmoneke, #660)
- Add Protocol tests without hardware present (@bilderbuchi, #634, @bmoneke, #628, #635)
- Add Instruments and adapter protocol tests for adapter rework (@bmoneke, #665)
- Add SR830 sync filter and reference source trigger (@AsafYagoda, #630)
- Add Keithley6221 phase marker phase and line (@AsafYagoda, #629)
- Add missing docstrings to Keithley 2306 battery simulator (@AidenDawn, #720)
- Fix hcp instruments documentation (@bmoneke, #671)
- Fix HPLegacyInstrument initializer API (@bilderbuchi, #684)
- Fix Fwbell 5080 implementation (@mcdo0486, #714)
- Fix broken documentation example. (@bmoneke, #738)
- Fix typo in Keithley 2600 driver (@mcdo0486, #615)
- Remove dynamic use of docstring from ATS545 and make more generic (@AidenDawn, #685)

14.3.4 Automation

- Add storing unitful experiment results (@bmoneke, #642)
- Add storing conditions in file (@CasperSchippers, #503)

14.3.5 GUI

- Add compatibility with Qt 6 (@CasperSchippers, #688)
- Add spinbox functionality for IntegerParameter and FloatParameter (@jarvas24, #656)
- Add “delete data file” button to the browser_item_menu (@jarvas24, #654)
- Split windows.py into a folder with separate modules (@mcdo0486, #593)
- Remove dependency on matplotlib (@msmttchr, #622)
- Remove deprecated access to QtWidgets through QtGui (@maederan201, #695)

14.3.6 Miscellaneous

- Update and extend documentation (@bilderbuchi, #712, @bmoneke, #655)
- Add PEP517 compatibility & dynamically obtaining a version number (@bilderbuchi, #613)
- Add an example and documentation regarding using a foreign instrument (@bmoneke, #647)
- Add black configuration (@bleykauf, #683)
- Remove VISAAdapter.has_supported_version() as it is not needed anymore.

14.3.7 New Contributors

@jcarbelbide, @mroeleke, @bmoneke, @garzetti, @AsafYagoda, @AidenDawn, @LastStarDust, @sansanda

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.10.0...v0.11.0

14.4 Version 0.10.0 (2022-04-09)

Main items of this new release:

- 23 new instrument drivers have been added
- New dynamic Instrument properties can change their parameters at runtime
- Communication settings can now be flexibly defined per protocol
- Python 3.10 support was added and Python 3.6 support was removed.
- Many additions, improvements and have been merged

14.4.1 Instruments

- New Agilent B1500 Data Formats and Documentation (@moritzj29)
- New Anaheim Automation stepper motor controllers (@samcondon4)
- New Andeen Hagerling capacitance bridges (@dkriegner)
- New Anritsu MS9740A Optical Spectrum Analyzer (@md12g12)
- New BK Precision 9130B Instrument (@dennisfeng2)
- New Edwards nXDS (10i) Vacuum Pump (@hududed)
- New Fluke 7341 temperature bath instrument (@msmttchr)
- New Heidenhain ND287 Position Display Unit Driver (@samcondon4)
- New HP 3478A (@LongnoseRob)
- New HP 8116A 50 MHz Pulse/Function Generator (@CodingMarco)
- New Keithley 2260B DC Power Supply (@bklebel)
- New Keithley 2306 Dual Channel Battery/Charger Simulator (@mfikes)
- New Keithley 2600 SourceMeter series (@Daivesd)
- New Keysight N7776C Swept Laser Source (@maederan201)
- New Lakeshore 421 (@CasperSchippers)
- New Oxford IPS120-10 (@CasperSchippers)
- New Pendulum CNT-91 frequency counter (@bleykauf)
- New Rohde&Schwarz - SFM TV test transmitter (@LongnoseRob)
- New Rohde&Schwarz FSL spectrum analyzer (@bleykauf)
- New SR570 current amplifier driver (@pyMatJ)
- New Stanford Research Systems SR510 instrument driver (@samcondon4)
- New Toptica Smart Laser diode (@dkriegner)

- New Yokogawa GS200 Instrument (@dennisfeng2)
- Add output low grounded property to Keithley 6221 (@CasperSchippers)
- Add shutdown function for Keithley 2260B (@bklebel)
- Add phase control for Agilent 33500 (@corna)
- Add assigning "ONCE" to auto_zero to Keithley 2400 (@mfikes)
- Add line frequency controls to Keithley 2400 (@mfikes)
- Add LIA and ERR status byte read properties to the SRS Sr830 driver (@samcondon4)
- Add all commands to Oxford Intelligent Temperature Controller 503 (@CasperSchippers)
- Fix DSP 7265 lockin amplifier (@CasperSchippers)
- Fix bug in Keithley 6517B Electrometer (@CasperSchippers)
- Fix Keithley2000 deprecated call to visa.config (@bklebel)
- Fix bug in the Keithley 2700 (@CasperSchippers)
- Fix setting of sensor flags for Thorlabs PM100D (@bleykauf)
- Fix SCPI used for Keithley 2400 voltage NPLC (@mfikes)
- Fix missing return statements in Tektronix AFG3152C (@bleykauf)
- Fix DPSeriesMotorController bug (@samcondon4)
- Fix Keithley2600 error when retrieving error code (@bicarlsen)
- Fix Attocube ANC300 with new SCPI Instrument properties (@dkriegner)
- Fix bug in wait_for_trigger of Agilent33220A (neal-kepler)

14.4.2 GUI

- Add time-estimator widget (@CasperSchippers)
- Add management of progress bar (@msmttchr)
- Remove broken errorbar feature (@CasperSchippers)
- Change of pen width for pyqtgraph (@maederan201)
- Make linewidth changeable (@CasperSchippers)
- Generalise warning in plotter section (@CasperSchippers)
- Implement visibility groups in InputsWidgets (@CasperSchippers)
- Modify navigation of ManagedWindow directory widget (@jarvas24)
- Improve Placeholder logic (@CasperSchippers)
- Breakout widgets into separate modules (@mcdo0486)
- Fix setSizePolicy bug with PySide2 (@msmttchr)
- Fix managed window (@msmttchr)
- Fix ListParameter for numbers (@moritzj29)
- Fix incorrect columns on showing data (@CasperSchippers)
- Fix procedure property issue (@msmttchr)

- Fix pyside2 (@msmttchr)

14.4.3 Miscellaneous

- Improve SCPI property support (@msmttchr)
- Remove broken safeKeyword management (@msmttchr)
- Add dynamic property support (@msmttchr)
- Add flexible API for defining connection configuration (@bilderbuchi)
- Add write_binary_values() to SerialAdapter (@msmttchr)
- Change an outdated pyvisa ask() to query() (@LongnoseRob)
- Fix ZMQ bug (@bilderbuchi)
- Documentation for passing tuples to control property (@bklebel)
- Documentation bugfix (@CasperSchippers)
- Fixed broken links in documentation. (@samcondon4)
- Updated widget documentation (@mcdo0486)
- Fix typo SCIP->SCPI (@mfikes)
- Remove Python 3.6, add Python 3.10 testing (@bilderbuchi)
- Modernise the code base to use Python 3.7 features (@bilderbuchi)
- Added image data generation to Mock Instrument class (@samcondon4)
- Add autodoc warnings to the problem matcher (@bilderbuchi)
- Update CI & annotations (@bilderbuchi)
- Test workers (@mcdo0486)
- Change copyright date to 2022 (@LongnoseRob)
- Removed unused code (@msmttchr)

14.4.4 New Contributors

@LongnoseRob, @neal, @hududed, @corna, @Daivesd, @samcondon4, @maederan201, @bleykauf, @mfikes, @bicarlsen, @md12g12, @CodingMarco, @jarvas24, @mcdo0486!

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.9...v0.10.0

14.5 Version 0.9 – released 2/7/21

- PyMeasure is now officially at github.com/pymeasure/pymeasure
- Python 3.9 is now supported, Python 3.5 removed due to EOL
- Move to GitHub Actions from TravisCI and Appveyor for CI (@bilderbuchi)
- New additions to Oxford Instruments ITC 503 (@CasperSchippers)
- New Agilent 34450A and Keysight DSOX1102G instruments (@theMashUp, @jlarochelle)

- Improvements to NI VirtualBench (@moritzj29)
- New Agilent B1500 instrument (@moritzj29)
- New Keithley 6517B instrument (@wehlgrundspitze)
- Major improvements to PyVISA compatbility (@bilderbuchi, @msmttchr, @CasperSchippers, @cjermain)
- New Anapico APSIN12G instrument (@StePhanino)
- Improvements to Thorelabs Pro 8000 and SR830 (@Mike-HubGit)
- New SR860 instrument (@StevenSiegl, @bklebel)
- Fix to escape sequences (@tirkarthi)
- New directory input for ManagedWindow (@paulgoulain)
- New TelnetAdapter and Attocube ANC300 Piezo controller (@dkriegner)
- New Agilent 34450A (@theMashUp)
- New Razorbill RP100 strain cell controller (@pheowl)
- Fixes to precision and default value of ScientificInput and FloatParameter (@moritzj29)
- Fixes for Keithly 2400 and 2450 controls (@pyMatJ)
- Improvments to Inputs and open_file_externally (@msmttchr)
- Fixes to Agilent 8722ES (@alexmcnabb)
- Fixes to QThread cleanup (@neal-kepler, @msmttchr)
- Fixes to Keyboard interrupt, and parameters (@CasperSchippers)

14.6 Version 0.8 – released 3/29/19

- Python 3.8 is now supported
- New Measurement Sequencer allows for running over a large parameter space (@CasperSchippers)
- New image plotting feature for live image measurements (@jmittelstaedt)
- Improvements to VISA adapter (@moritzj29)
- Added Tektronix AFG 3000, Keithley 2750 (@StePhanino, @dennisfeng2)
- Documentation improvements (@mivade)
- Fix to ScientificInput for float strings (@moritzj29)
- New validator: strict_discrete_range (@moritzj29)
- Improvements to Recorder thread joining
- Migrating the ReadtheDocs configuration to version 2
- National Instruments Virtual Bench initial support (@moritzj29)

14.7 Version 0.7 – released 8/4/19

- Dropped support for Python 3.4, adding support for Python 3.7
- Significant improvements to CI, dependencies, and conda environment (@bilderbuchi, @cjermain)
- Fix for PyQT issue in ResultsDialog (@CasperSchippers)
- Fix for wire validator in Keithley 2400 (@Fattotora)
- Addition of source_enabled control for Keithley 2400 (@dennisfeng2)
- Time constant fix and input controls for SR830 (@dennisfeng2)
- Added Keithley 2450 and Agilent 33521A (@hlgirard, @Endever42)
- Proper escaping support in CSV headers (@feph)
- Minor updates (@dvase)

14.8 Version 0.6.1 – released 4/21/19

- Added Elektronica SM70-45D, Agilent 33220A, and Keysight N5767A instruments (@CasperSchippers, @sumatrae)
- Fixes for Prologix adapter and Keithley 2400 (@hlgirard, @ronan-sensome)
- Improved support for SRS SR830 (@CasperSchippers)

14.9 Version 0.6 – released 1/14/19

- New VXI11 Adapter for ethernet instruments (@chweiser)
- PyQt updates to 5.6.0
- Added SRS SG380, Ametek 7270, Agilent 4156, HP 34401A, Advantest R3767CG, and Oxford ITC503 instrustruments (@sylkar, @jmittelstaedt, @vik-s, @troylf, @CasperSchippers)
- Updates to Keithley 2000, Agilent 8257D, ESP 300, and Keithley 2400 instruments (@watersjason, @jmittelstaedt, @nup002)
- Various minor bug fixes (@thosou)

14.10 Version 0.5.1 – released 4/14/18

- Minor versions of PyVISA are now properly handled
- Documentation improvements (@Laogeodritt and @ederag)
- Instruments now have `set_process` capability (@bilderbuchi)
- Plotter now uses threads (@dvspirito)
- Display inputs and PlotItem improvements (@Laogeodritt)

14.11 Version 0.5 – released 10/18/17

- Threads are used by default, eliminating multiprocessing issues with spawn
- Enhanced unit tests for threading
- Sphinx Doctests are added to the documentation (@bilderbuchi)
- Improvements to documentation (@JuMaD)

14.12 Version 0.4.6 – released 8/12/17

- Reverted multiprocessing start method keyword arguments to fix Unix spawn issues (@ndr37)
- Fixes to regressions in Results writing (@feinsteinben)
- Fixes to TCP support using cloudpickle (@feinsteinben)
- Restructing of unit test framework

14.13 Version 0.4.5 – released 7/4/17

- Recorder and Scribe now leverage QueueListener (@feinsteinben)
- PrologixAdapter and SerialAdapter now handle Serial objects as adapters (@feinsteinben)
- Optional TCP support now uses cloudpickle for serialization (@feinsteinben)
- Significant PEP8 review and bug fixes (@feinsteinben)
- Includes docs in the code distribution (@ghisvail)
- Continuous integration support for Python 3.6 (@feinsteinben)

14.14 Version 0.4.4 – released 6/4/17

- Fix pip install for non-wheel builds
- Update to Agilent E4980 (@dvspirito)
- Minor fixes for docs, tests, and formatting (@ghisvail, @feinsteinben)

14.15 Version 0.4.3 – released 3/30/17

- Added Agilent E4980, AMI 430, Agilent 34410A, Thorlabs PM100, and Anritsu MS9710C instruments (@TvBMcMaster, @dvspirito, and @mhdg)
- Updates to PyVISA support (@minhhaiphys)
- Initial work on resource manager (@dvspirito)
- Fixes for Prologix adapter that allow read-write delays (@TvBMcMaster)
- Fixes for conda environment on continuous integration

14.16 Version 0.4.2 – released 8/23/16

- New instructions for installing with Anaconda and conda-forge package (thanks @melund!)
- Bug-fixes to the Keithley 2000, SR830, and Agilent E4408B
- Re-introduced the Newport ESP300 motion controller
- Major update to the Keithely 2400, 2000 and Yokogawa 7651 to achieve a common interface
- New command-string processing hooks for Instrument property functions
- Updated LakeShore 331 temperature controller with new features
- Updates to the Agilent 8257D signal generator for better feature exposure

14.17 Version 0.4.1 – released 7/31/16

- Critical fix in setup.py for importing instruments (also added to documentation)

14.18 Version 0.4 – released 7/29/16

- Replaced Instrument add_measurement and add_control with measurement and control functions
- Added validators to allow Instrument.control to match restricted ranges
- Added mapping to Instrument.control to allow more flexible inputs
- Conda is now used to set up the Python environment
- macOS testing in continuous integration
- Major updates to the documentation

14.19 Version 0.3 – released 4/8/16

- Added IPython (Jupyter) notebook support with significant features
- Updated set of example scripts and notebooks
- New PyMeasure logo released
- Removed support for Python <3.4
- Changed multiprocessing to use spawn for compatibility
- Significant work on the documentation
- Added initial tests for non-instrument code
- Continuous integration setup for Linux and Windows

14.20 Version 0.2 – released 12/16/15

- Python 3 compatibility, removed support for Python 2
- Considerable renaming for better PEP8 compliance
- Added MIT License
- Major restructuring of the package to break it into smaller modules
- Major rewrite of display functionality, introducing new Qt objects for easy extensions
- Major rewrite of procedure execution, now using a Worker process which takes advantage of multi-core CPUs
- Addition of a number of examples
- New methods for listening to Procedures, introducing ZMQ for TCP connectivity
- Updates to Keithley2400 and VISAAdapter

14.21 Version 0.1.6 – released 4/19/15

- Renamed the package to PyMeasure from Automate to be more descriptive about its purpose
- Addition of VectorParameter to allow vectors to be input for Procedures
- Minor fixes for the Results and Danfysik8500

14.22 Version 0.1.5 – release 10/22/14

- New Manager class for handling Procedures in a queue fashion
- New Browser that works in tandem with the Manager to display the queue
- Bug fixes for Results loading

14.23 Version 0.1.4 – released 8/2/14

- Integrated Results class into display and file writing
- Bug fixes for Listener classes
- Bug fixes for SR830

14.24 Version 0.1.3 – released 7/20/14

- Replaced logging system with Python logging package
- Added data management (Results) and bug fixes for Procedures and Parameters
- Added pandas v0.14 to requirements for data management
- Added data listeners, Qt4 and PyQtGraph helpers

14.25 Version 0.1.2 – released 7/18/14

- Bug fixes to LakeShore 425
- Added new Procedure and Parameter classes for generic experiments
- Added version number in package

14.26 Version 0.1.1 – released 7/16/14

- Bug fixes to PrologixAdapter, VISAAdapter, Agilent 8722ES, Agilent 8257D, Stanford SR830, Danfysik8500
- Added Tektronix TDS 2000 with basic functionality
- Fixed Danfysik communication to handle errors properly

14.27 Version 0.1.0 – released 7/15/14

- Initial release

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z